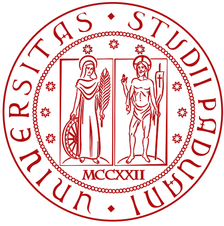


Università degli Studi di Padova

Master Thesis in Electronic Engineering

Hardware Implementation of a Spiking Neural Network for online processing of muon detectors datastream

Master Candidate

Marco Toffano

Supervisor

Prof. Daniele Vogrig

Department of Information Engineering

Co-supervisor

Prof. Andrea Triossi

Department of Physics and Astronomy

Padova, 9/7/2024

Academic Year
2023/2024

What we usually consider as impossible
are simply Engineering problems...
There's no law of Physics preventing them.

- Michio Kaku

Abstract

Spiking Neural Networks (SNNs) hold the potential to revolutionize neuromorphic computing by better emulating the human brain's natural processing capabilities, offering in turn significant advantages in terms of power efficiency and real-time performance. This thesis focuses on the detailed RTL (Register Transfer Level) modeling and FPGA (Field Programmable Gate Array) resource mapping of a general architecture for SNNs, so providing a comprehensive description of the design steps which lead to its final realization. Emphasis has been placed on the hardware-level representation of individual SNN components, specifically leveraging the Leaky Integrate-and-Fire (LIF) neuron due to its computational efficiency. Key to this approach is the efficient handling of fixed-point arithmetic to optimize resource utilization on the FPGA. Various optimization strategies, such as pipelining, loop unrolling, and resource sharing, are employed to maximize performance and minimize resource overhead. The implementation is validated through both functional simulations and synthesis results, providing insights into the design's resource utilization, timing performance, and power consumption.
To benchmark the SNN, a classification problem has been tackled submitting to the network two standard timeseries datasets.
Moreover, an innovative application for the SNN is proposed regarding the problem of filtering the noisy datastream of drift-tube-chambers muon detectors employed in fundamental research in Particle Physics. In this context, the SNN is used within the muon trigger system, for which ultra-low latency hardware implementation is required.
Finally, a comparative analysis is performed with an ASIC implementation using 130nm technology custom cell libraries. This second possibility, although more optimized under some aspects, will be shown to lack the substantial benefits in terms of reconfigurability, rapid prototyping capabilities and further integration with other systems, such as IP cores for handling communications.
A *GitHub* repository containing the most relevant code has been arranged at [39].

Sommario

Le Reti Neurali Impulsive (SNN) hanno il potenziale per rivoluzionare il calcolo neuromorfico, riuscendo ad emulare più fedelmente le capacità di elaborazione naturale del cervello umano, ed offrendo in tal modo significativi vantaggi in termini di efficienza energetica e prestazioni in tempo reale. Questa tesi si concentra sulla dettagliata modellazione a livello RTL (Register Transfer Level) e sulla mappatura delle risorse necessarie su FPGA (Field Programmable Gate Array) di un'architettura generale per le SNN, fornendo una descrizione completa dei passaggi progettuali che ne portano alla realizzazione finale. L'enfasi è stata posta sulla rappresentazione a livello hardware dei singoli componenti della SNN, utilizzando in particolare il neurone Leaky Integrate-and-Fire (LIF) per la sua efficienza computazionale. Chiave per questo approccio è la gestione efficiente dell'aritmetica in virgola fissa per ottimizzare l'utilizzo delle risorse su FPGA. Varie strategie di ottimizzazione, come il pipelining, il loop unrolling e la condivisione delle risorse, vengono impiegate per massimizzare le prestazioni e minimizzare il consumo di risorse. L'implementazione è validata attraverso simulazioni funzionali e risultati di sintesi, mostrando l'utilizzo delle risorse, il rispetto dei vincoli temporali e il consumo energetico del circuito.

Per valutare la SNN, sono stati affrontati dei problemi di classificazione sottoponendo alla rete due dataset espressi come serie temporali standard.

Viene inoltre proposta un'applicazione innovativa per le SNN, concernente il problema del filtraggio del rumoroso flusso di dati proveniente dai rivelatori di muoni delle camere a fili impiegate nella ricerca fondamentale in fisica delle particelle. In questo contesto, la SNN verrebbe utilizzata all'interno del sistema di trigger per i muoni, che impone l'utilizzo di hardware che possa garantire una latenza bassissima.

Infine, viene eseguita un'analisi comparativa con un'implementazione ASIC effettuata con librerie di celle standard basate su tecnologia a 130nm. Questa seconda implementazione, sebbene più ottimizzata sotto alcuni aspetti, manca dei sostanziali benefici in termini di riconfigurabilità, capacità di prototipazione rapida e ulteriore integrazione con altri sistemi, come processori proprietari per la gestione delle comunicazioni.

È stato predisposto un archivio contente il codice di maggior rilevanza in [39].

Contents

List of Figures xi

List of Tables xiii

List of Algorithms xvii

List of Code Snippets xvii

List of Acronyms xix

1 Introduction 1
- 1.1 Types of Spiking Neurons . . . 2
- 1.2 Data Encoding . . . 4
- 1.3 Training peculiarities . . . 6
  - 1.3.1 Backpropagation through time . . . 6
  - 1.3.2 Spike timiming dependent plasticity . . . 9
- 1.4 Advantages of SNNs . . . 10

2 Modeling and training SNNs 12
- 2.1 Inference . . . 13
- 2.2 Training via *SNNTorch* . . . 15
- 2.3 Training via *Matlab* . . . 17
- 2.4 Parameters extraction . . . 19
- 2.5 Direct conversion from pretrained ANN . . . 20

3 RTL description of SNNs 22
- 3.1 Neuron structure . . . 23
  - 3.1.1 Binary Adder Tree . . . 25
  - 3.1.2 MAC . . . 26
- 3.2 Datapath sizing . . . 28
- 3.3 Hierarchy Management . . . 29
- 3.4 Convolutional and pooling layers . . . 31
- 3.5 Softmax . . . 33
  - 3.5.1 Exponential unit . . . 35
  - 3.5.2 CORDIC divider . . . 37

3.6 Pipeline and sequential execution 40
3.7 Verification 42

4 Implementation and Benchmarking **46**
4.1 TOP entity 47
4.1.1 *AXI4 Stream* communication 48
4.2 Testing on Spiking Tactile MNIST 49
4.3 Testing on Spiking Heidelberg Digits 52
4.4 Runtime coefficients setting 55

5 Case study: drift tubes hits filtering **58**
5.1 The experiment 58
5.2 Dataset generation 60
5.3 Training challenges 62
5.4 Proposed approach for spurious hits filtering 64
5.5 Enhancing reliability 66

6 ASIC realization **68**
6.1 Preliminary steps 69
6.2 Synthesis 70
6.3 Physical design 73

7 Conclusions and Outlook **78**

A Appendix **80**
A.1 Hardware resources of KCU1500 board 81
A.2 Standard cells of faraday 130nm technological library 82
A.3 VHDL testbench for generic timeseries dataset 83
A.4 Bash script for AXI LITE registers read/write 84

References **85**

List of Figures

1.1 Spiking neuron as a compromise between biological accuracy and computational utility. . . . 2
1.2 Summary of the possible encoding schemes for SNN data. . . . 6
1.3 Recurrent representation of spiking neurons on which BPTT lies its foundation. . . 7
1.4 Synaptic weights correction mechanism of STDP. . . . 9

2.1 LIF neuron response to a random input stimulus over 200 timesteps. . . . 13
2.2 Evolution of $U[t]$ over 40 timesteps for the first neuron of each layer of a SNN with structure [16, 64, 32, 16]. . . . 15
2.3 Comparison between the 'ideal' evolution of $U[t]$ and the actual one as a result of some possible quantizations. . . . 20
2.4 ANN-to-SNN conversion principle. . . . 21

3.1 RTL model of the LIF spiking neuron. . . . 23
3.2 Vivado synthesis engine log showing removal of unused adder inputs. . . . 25
3.3 Generic-order IIR filter implemented as cascade of biquad stages. . . . 28
3.4 Datapath width optimization performed by Vivado synthesis compiler. . . . 29
3.5 Hierarchical organization of SNN entity, with simplified internal scheme of the neuron. . . . 30
3.6 Skecth of convolution meschanism with input spikes for a given kernel matrix: software algorithm and hardware circuit. . . . 31
3.7 RTL scheme of the softmax unit. . . . 33
3.8 Softmax unit internal signals widths in the specific case of *sfixed(3 downto -12)* input data. . . . 34
3.9 Xilinx's LUT6 primitive internal block scheme. . . . 35
3.10 CORDIC pseudo-rotation mechanism. . . . 38
3.11 RTL schemes of CORDIC processor. . . . 39
3.12 State diagram of Moore-type control unit for a layer of neurons in case sequential SNN execution is desired. . . . 41
3.13 State diagram of Mealy-type control unit. . . . 41
3.14 Sequential adder for neuron synaptic weights. . . . 42
3.15 Testing of 1st order pipeline neuron. . . . 43
3.16 Testing of 2nd order pipeline neuron. . . . 43

3.17 Comparison between the evolution of $U[t]$ obtained from RTL simulation and from golden reference model in *Matlab*. 43
3.18 Scope view of the evolution of a sequential neuron with 2 inputs. 44
3.19 Simulation of inference for a sequential SNN having [16, 40, 32, 16] structure. . . 45
3.20 Result of exponentiation realized as explained in Sec. 3.5.1. 45
3.21 Simulation of pipeline CORDIC divider. 45

4.1 XDMA IP core handled thorugh AXI protocol. 47
4.2 AXI4 Stream signals timing diagram for the consecutive processing of 25 timesteps of a varying 16-bit wide input signal. 49
4.3 Decoded events of the tactile sensor array for a sample corresponding to 8 digit. . 49
4.4 Block scheme of the SNN-based classifier for the STMNIST dataset. 50
4.5 STMNIST digits classification accuracy as function of the parameters resolution. 51
4.6 SHD input sample. 52
4.7 Coincidence detection using spiking neurons. 53
4.8 Block scheme of the SNN-based classifier for the SHD dataset. 54
4.9 Vivado schematic view of *delay_block* module. 55
4.10 AXI Lite register transaction timing diagram. 57

5.1 Working principle of muon drift tubes. 59
5.2 CMS cross-sectional view in correspondence of DTs. 59
5.3 DAQ system for muon events detected by drift tubes. 60
5.4 Simulated muon arrivals: hits time and crossed cells. 61
5.5 Generated hits labeling 62
5.6 Generation of the training dataset: the single samples originated from muon arrival simulation are merged together to form a unique datastream. 62
5.7 Server resource usage during SNN Matlab-based training. 63
5.8 Learning curves for SNN filter training with Code 2.7. 64
5.9 Membrane potential evolution for a [16, 64, 32, 16] net with bias accumulation effect. 65
5.10 Sketch of the structure of the online neural filter for drift tubes particle detectors. 66
5.11 Triple Modular Redundancy working principle. 67

6.1 Slack histogram in *Design Vision*. 70
6.2 Extensive XCKU115-2FLVB2104E FPGA utilization report for a [16, 40, 32, 16] SNN. 72
6.3 FPGA-based SNN power consumption. 73
6.4 Physical repartition of chip area between the layers of the network. 74
6.5 Path slack historam after clock tree synthesis. 74
6.6 Final layout of [16, 40, 32, 16] SNN with 1st order reset-by-subtraction LIF neurons. 75
6.7 Power report of the final layout for the SNN ASIC. 76
6.8 Physical view of the circuit realized employing the FPGA of KCU1500 board. . . 77
6.9 Close-up look at SNN circuit realizations. 77

List of Tables

1.1 Overview of alternative spiking neurons models to LIF. 4

3.1 Lookup table input address mapping to reproduce $f(x) = e^x$ using as x the 4 MSBs of a sfixed(3 downto -12) type. 37
3.2 Lookup table input address mapping to reproduce $f(x) = e^x$ using as x the successive 4 bits of a sfixed(3 downto -12) type. 37

4.1 Description of AXI4-Stream signals. 48
4.2 Description of AXI-Lite signals. 56

6.1 Area report of SNN ASIC. 71
6.2 Power report summary. 71
6.3 Resource employed by FPGA-based SNN. 72

A.1 Primary hardware resources of the UltraScale XCKU115-2FLVB2104E FPGA. . . . 81
A.2 Main types of standard cells in the *Faraday* UMC-130nm technological library . . 82

List of Algorithms

1 Backpropagation Through Time (BPTT) 8
2 Spike-Timing-Dependent Plasticity (STDP) 10

List of Code Snippets

2.1 Matlab function for implementing LIF neuron signal elaboration. Actually, this function simulates an entire layer of neuron, as it accepts a vector as input x and a matrix for weights w. 13
2.2 Matlab function of LIF neuron (entire layer, actually), accepting a tensor as input data ($n_{channels} \times n_{timesteps} \times n_{samples}$). 14
2.3 Matlab function for general-structure SNN simulation. 14
2.4 Python code snippet for single hidden layer SNN definition, using *Sequential()* construct. 15
2.5 Python function executing SNN inference. 16
2.6 Training loop within SNNTorch. 16
2.7 Matlab code snippet for SNN training using BPTT. 18
3.1 LIF neuron (1st order) VHDL code snippet. 24
3.2 VHDL code snippet for cascade summation of an arbitrary number of input sfixed(INT downto -FRAC) signals. 26
3.3 Full VHDL code of generic MAC entity. 27
3.4 VHDL code snippet implementing max pooling exploiting OR reduction operator. 32
3.5 VHDL code snippet for average pooling of a group of spikes. 32
4.1 VHDL code of the TOP entity. 47
4.2 VHDL Function for determining the index of the element with the greatest value in an array. 50
4.3 VHDL process realizing a generic-length shift register minimizing the use of FFs in favour of BRAM. 54
6.1 Impure VHDL function used to read network coefficients from file. 69

List of Acronyms

ANN: Artificial Neural Network

ASIC: Application Specific Integrated Circuit

AXI: Advanced eXtensible Interface

BPTT: BackPropagation Through Time

BRAM: Block Random Access Memory

BX: Bunch Crossing

CE: Cross-Entropy

CERN: Conseil Européen pour la Recherche Nucléaire - European Council for Nuclear Research

CLB: Configurable Logic Block

CMS: Compact Muon Solenoid

CORDIC: Coordinate Rotation Digital Circuits

CU: Control Unit

CTS: Clock Tree Synthesis

DAQ: Data AQuistion (system)

DCLS: Dilated Convolution with Learnable Spacings

DFM: Design For Manufacturability

DSP: Digital Signal Processor/Processing

DRC: Design Rules Checking

DT: Drift Tube

FF: Flip Flop

FSM: Finite State Machine

FPGA: Field-Programmable Gate Array

HDL: Hardware Description Language

IIR: Infinite Impulse Response

I/O: Input/Output

IP: Intellectual Property

LHC: Large Hadron Collider

LIF: Leaky Integrate and Fire

LUT: LookUp Table

LVS: Layout-Vs-Schematic

MAC: Multiply-ACcumulate

MSE: Mean Square Error

OBDT: On-detector Board for Drift Tubes

PLL: Phase-locked loop

QAT: Quantization Aware Training

RNN: Recurrent Neural Network

ROM: Read-Only Memory

RTL: Register Tranfer Level

SHD: Spiking Heidelberg Digits (dataset)

SGD: Stochastic Gradient Descent

SNN: Spiking Neural Network

STDP: Spike Timing Dependent Plasticity

TDC: Time-to-Digital Converter

VHDL: VHSIC Hardware Description Language

XDMA: Xilinx Direct Memory Access

1

Introduction

This work is intended to illustrate the development of a circuital implementation of a generic neural network architecture based on spiking neurons. As a first step, a brief introduction to the topic of Spiking Neural Networks (SNNs) will be assessed in this chapter, comparing them with their well-established “classical” Artificial Neural Networks (ANNs) counterparts. In particular, some of the peculiar aspects of the training procedure will be shown.
Then, in chapter 2, the approach used to simulate the network inference phase, as well as for network training on timeseries datasets, is going to be presented from a practical standpoint, making use of commonly available tools such as Matlab and Python libraries. Next, in chapter 3, the core of this thesis is found, namely the actual register-transfer-level description of the spiking neural network through VHDL code: a step-by-step analysis will be carried out, emphasizing the choices made to keep the project the most modular and generic possible.
To follow, in chapter 4, the actual implementation on KCU1500 harware evaluation board is discussed, highlighting how the RTL model of the network has been physically mapped on the available processing blocks and which strategies have been applied to favour the use of elements such as LUTs and BRAM w.r.t. other less abundant blocks. After that, the network will be used to realise a classifier and will be tested on two distinct natively-spiking datasets, analyzing its effectiveness as function of the network structural complexity and parameters quantization.
Another very interesting proposed use case in the realm of particle detectors is described in chapter 5, showing how a “direct” application of the net for instant-by-instant signal filtering has revealed not to perform well enough, for which a more complex architecture involving multiple identical SNN-based classifiers was adopted to realize a smart neural filter for muon hits in drift tube chambers.
Finally, chapter 6 presents an ASIC implementation of a SNN, comparing it to its equivalent FPGA-implemented version. Some conclusions and a few points for future investigations are then discussed in chapter 7.

1.1 Types of Spiking Neurons

Classical neurons and spiking neurons fundamentally differ in their operation and applications. Classic neurons output continuous values and operate based on an activation function. The output y of a classic neuron is computed as:

$$y = \varphi(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b) \tag{1.1}$$

where φ is a nonlinear activation function such as the sigmoid, hyperbolic tangent (tanh), or rectified linear unit (ReLU), w_i are the input weights, x_i are the inputs, and b is the bias. These neurons have an instantaneous response to their inputs and rely on differentiable activation functions to facilitate optimization during the training process. Classic neurons are the backbone of many artificial neural network architectures, including feedforward, convolutional, and recurrent neural networks (RNNs), which are extensively used in tasks like image recognition and natural language processing.

In contrast, spiking neurons produce discrete events known as spikes or action potentials. A spiking neuron integrates incoming signals over time, and when its membrane potential exceeds a certain threshold, it generates a spike. The behavior of spiking neurons can be represented by models such as the leaky integrate-and-fire (LIF):

$$\tau_m \frac{dU(t)}{dt} = -U(t) + RI(t) \tag{1.2}$$

where $U(t)$ is the membrane potential, τ_m is the membrane time constant, R is the membrane resistance, and $I(t)$ is the input current. A spike is emitted when $U(t)$ exceeds a threshold U_{thr}.

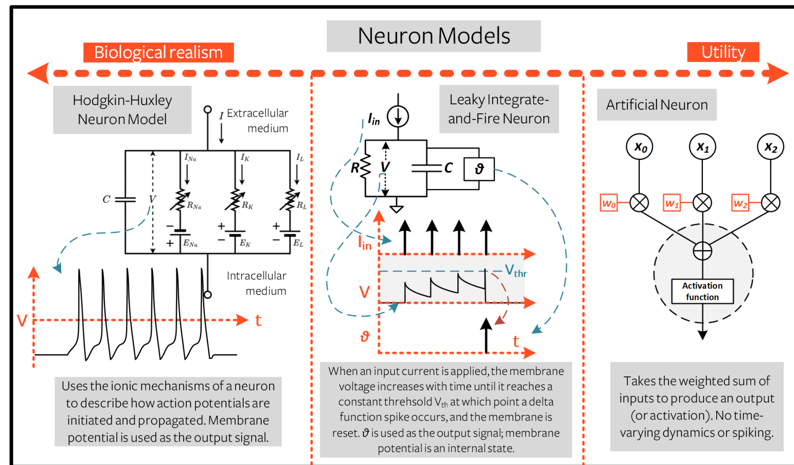


Figure 1.1: Spiking neuron as a compromise between biological accuracy and computational utility. Taken from [33].

The origin of Eq. 1.2 stems from observing that biological neurons communicate through electrical signals, largely influenced by the ions movements across the neuronal membrane, creating a voltage known as the membrane potential, pictorically presented in Fig. 1.1.

These ionic currents can be split into capacitive and resistive components[1]. Equation 1.2 governs $U(t)$ based on the principle of conservation of charge, obtained by equating the total incoming current to the rate of change of charge on the membrane capacitor:

$$C_m \frac{dU(t)}{dt} = I_{\text{total}}(t), \tag{1.3}$$

where the total current comprises ionic currents (I_{ion}) and external currents (I_{ext}). The ionic current through leak channels can be further described by Ohm's law as

$$I_{\text{ion}}(t) = g_L(U(t) - E_L), \tag{1.4}$$

where g_L is the leak conductance and E_L is the equilibrium potential for the leak current. Substituting $I_{\text{ion}}(t)$ into the membrane equation yields:

$$C_m \frac{dU(t)}{dt} = -g_L(U(t) - E_L) + I_{\text{ext}}(t). \tag{1.5}$$

Dividing by C_m, introducing the membrane time constant $\tau_m = \frac{C_m}{g_L}$ and assuming $E_L = 0$ for simplicity, Eq. 1.5 finally becomes

$$\frac{dU(t)}{dt} = -\frac{U(t)}{\tau_m} + \frac{I_{\text{ext}}(t)}{C_m}. \tag{1.6}$$

The threshold condition triggers a spike when $U(t) \geq U_{thr}$, immediately followed by the reset condition $U(t) = U_{reset}$, which may correspond to zero or could alternatively set by the current value minus that of the threshold. For numerical simulation, the model needs to be discretized w.r.t. time[2], and requires subsequent iteration of the membrane potential, threshold check, and evaluation of whether reset should be applied at each time step. Ultimately, the LIF model simplifies neuronal dynamics, capturing integration and spiking behavior derived from membrane physics and ionic currents. A glimpse of other more complicated models is reported in Table 1.1.

It is worth here to introduce the 2nd order version of the LIF neuron, precisely as defined in [33], where an additional dynamic variable, namely the synaptic current $I_{syn}(t)$, is introduced alongside the membrane potential $U_{mem}(t)$. The model is described by the following system of differential equations:

$$\begin{cases} \tau_{mem} \frac{dU_{mem}(t)}{dt} & = -U_{mem}(t) + R(I_{ext}(t) + I_{syn}(t))) \\ \tau_{syn} \frac{dI_{syn}(t)}{dt} & = -I_{syn}(t) + wU_{mem}(t) \end{cases} \tag{1.7}$$

Note that here two time constants are employed: τ_{mem} and τ_{syn}, allowing for more complex neural dynamics and modeling capabilities, and w is a proportionality constant emulating the strength of a synaptic interconnection.

[1]Inside human brain, in fact, the lipid bilayer acts as a capacitor, while ion channels offer non-negligible resistance to ions flow.

[2]Discretization employing Forward Euler method is shown in Eq. 2.3 at the beginning of chapter 2, while an alternative one employing bilinear transfmor is later shown in Sec. 3.1.2.

Neuron Type	Equation/Model	Description
Integrate-and-Fire (IF)	$\frac{dV(t)}{dt} = \frac{I(t)}{C_m}$	Simplest model capturing the basic mechanism of membrane potential integration and spike generation without the leakage term.
Hodgkin-Huxley (HH)	$C_m \frac{dV}{dt} = I - \bar{g}_{\text{Na}} m^3 h (V - E_{\text{Na}}) - \bar{g}_{\text{K}} n^4 (V - E_{\text{K}}) - \bar{g}_l (V - E_l)$ $\frac{dm}{dt} = \alpha_m (1 - m) - \beta_m m$ $\frac{dh}{dt} = \alpha_h (1 - h) - \beta_h h$ $\frac{dn}{dt} = \alpha_n (1 - n) - \beta_n n$	Detailed biophysical model replicating the ionic currents through the neuron membrane that generate action potentials.
Izhikevich Model	$\frac{dV}{dt} = 0.04V^2 + 5V + 140 - u + I$ $\frac{du}{dt} = a(bV - u)$	Combines biologically realistic spiking behavior with computational efficiency. Suitable for large-scale simulations.
FitzHugh-Nagumo Model	$\frac{dv}{dt} = v - \frac{v^3}{3} - w + I$ $\frac{dw}{dt} = 0.08(v + 0.7 - 0.8w)$	Simplified version of the Hodgkin-Huxley model focusing on qualitative features of neurons like excitability and spike generation.
Adaptive Exponential Integrate-and-Fire (AdEx)	$C_m \frac{dV(t)}{dt} = -g_L (V - E_L) + g_L \Delta_T \exp\left(\frac{V - V_T}{\Delta_T}\right) = -u + I$ $\tau_w \frac{du}{dt} = a(V - E_L) - u$	Extends the LIF model by adding an adaptation mechanism and an exponential term for membrane potential dynamics.

Table 1.1: Overview of alternative spiking neurons models to LIF.

1.2 Data Encoding

Spiking Neural Networks leverage the temporal dimension of neural activity to process information, which requires encoding input data into spike trains. This encoding process is crucial as it influences the effictiveness and accuracy of the network, whose outputs also need to be somehow decoded. Several encoding methods have been developed to represent continuous-valued inputs as spike trains that can ultimately be processed by SNNs, such as the ones presented in [2], [43] and [13]. The most prominent of them are presented in the following.

Rate coding is one of the simplest and most commonly used encoding schemes. In rate coding, the intensity of a stimulus is represented by the firing rate of a neuron. The neuron emits spikes at a frequency proportional to the input signal's magnitude. Mathematically, if $I(t)$ is the input signal, the firing rate r is given by: $r = f(I(t))$, where $f(\cdot)$ is a function mapping input intensity to spike rate. The spike train can be generated using a Poisson process where spikes occur with a probability directly proportional to the desired firing rate. While rate coding is easy to implement and interpret, it may not capture the full richness of temporal dynamics of the input data. Nevertheless, the vast majortiy of the examples employing SNNs which are found in literature

are based on this principle.

An alternative approach is that of *temporal encoding*, which grasps the precise timing of spikes to encode information. Unlike rate coding, where only the spike count within a time window is of interest, temporal coding considers the exact timing of individual spikes. This method can convey more information per single spike, making it a more efficient encoding scheme. Examples of temporal coding include:

- **Latency Coding**, in which the temporal delay between the onset of a stimulus and the time of the first spike encodes the input magnitude, where shorter latencies correspond to stronger stimuli.
- **Phase Coding**, aligning instead spikes with a periodic signal, where the phase shift of spikes relative to a reference oscillation captures the desired information.

One challenge with temporal coding is its sensitivity to spike timing precision, which surely needs more computational effort during the training phase and may require high-resolution timing mechanisms to be implemented in hardware.

Another interesting approach is that of *population coding*, which makes use of the collective activity of a group of neurons to represent information. Each neuron in the population responds to a specific range of the input signal, and the overall population activity pattern encodes the input. This method can be more robust to noise and can capture a broader range of input samples compared to single-neuron rate or temporal coding. An example formula for population coding might involve a Gaussian tuning curve where each neuron's firing rate r_i is a function of the input signal $I(t)$ and the neuron's preferred stimulus I_i:

$$r_i = \exp\left(-\frac{(I(t) - I_i)^2}{2\sigma^2}\right) \tag{1.8}$$

where σ determines the tuning curve width.

Rank order coding then relies on the order in which neurons fire to encode information. The relative timing or sequence of spikes across a set of neurons represents the input stimulus. This method is particularly useful when the precise timing between spikes matters less than the sequence of activation. Finally, *delta modulation* also exists, in which the production of spikes is regulated by the change in magnitude of the input data.

Each of the presented encoding methods, graphically summarized in Fig. 1.2, has its own advantages and trade-offs. The choice of encoding scheme depends on the specific application and constraints of the SNN. For instance, rate coding may be preferred for its simplicity and ease of implementation, whereas temporal or rank order coding might be chosen to exploit the temporal dynamics for richer information representation. In this work, rate encoding has been employed for most of the considered network use cases, as will be later discussed in chapters 4-5. Nonetheless some examples were also developed encoding input and output data according to their latency w.r.t. a properly set starting instant, even though some training challenges arose due to the extremely sparse nature of the considered dataset.

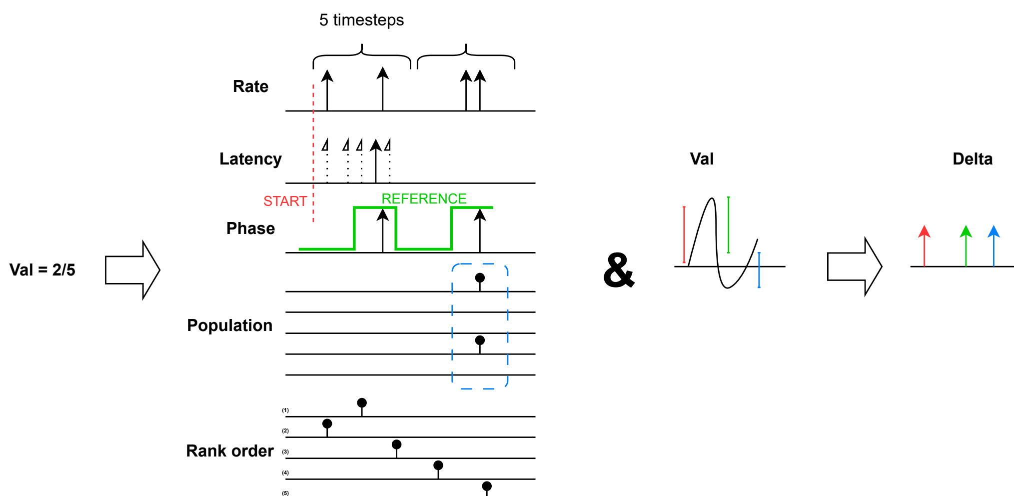


Figure 1.2: Summary of the possible encoding schemes for SNN data, most of which are depicted on the left side. In rate coding, 2 spikes are produced with random timing every 5 time instants, while for latency type the spike is the second to last w.r.t. the red start point. For phase type, a constant shift of 2/5 of the period is kept w.r.t. the green reference signal. Then, population counts the number of simultaneous spikes on different channels at the instant of interest and rank order takes as "winner" the second channel, which is the first to spike. Finally, a sketch of the underlying principle of *delta* modulation is also shown in the right half, namely that a spike gets produced every time there is a sufficiently big amplitude variation of the original signal.

1.3 Training peculiarities

Adapting training techniques for SNNs involves rethinking how information is processed and learned. Training SNNs involves adjusting synaptic weights to improve the network's performance on specific tasks. This process typically leverages the unique temporal dynamics of spikes, guiding the network to accurately process and respond to time-dependent inputs. Modern approaches often employ biologically inspired learning rules and optimization techniques that incorporate the discrete nature of spikes. A technique of the former type is discussed in Sec. 1.3.2, while one of the latter type is immediately presented in Sec. 1.3.1. In their own way, these methods ensure efficient adaptation and synchronization of neuron firing patterns, enabling SNNs to learn from temporal data and achieve robust performance.

1.3.1 Backpropagation through time

The Backpropagation Through Time (BPTT) technique was introduced in 1990 by Paul Werbos in [45]. It is an extension of the backpropagation algorithm used for training recurrent neural networks (RNNs), which works by unrolling the RNN through time and then using backpropagation to calculate gradients through this "extended" version of the network, allowing for the adjustment of weights (and other parameters) based on the temporally extended sequences of data. The necessity of BPTT for SNNs arises from the inherent temporal dependencies in these networks. In fact, each neuron's spike not only influences its immediate neighbours but also

affects neural activity over subsequent time steps. Traditional feedforward networks lack this temporal aspect and cannot therefore capture the dynamic behavior of SNNs. BPTT addresses this by unrolling the network across many time steps, effectively transforming it into a deep feedforward network where each layer corresponds to the network's state at a specific time step. A graphical depiction of this underlying principle is reported in Fig. 1.3.

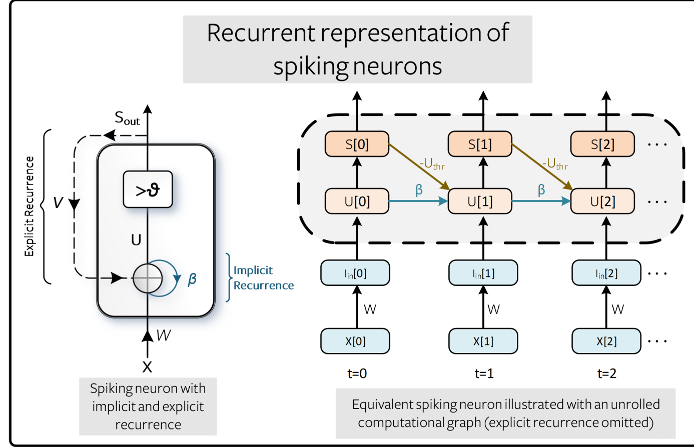


Figure 1.3: Recurrent representation of spiking neurons on which BPTT lies its foundation. Taken from [9].

BPTT unfolds the recurrent structure of an SNN over time. At each time step, the network's state is represented by its neurons' membrane potentials and spike outputs. During the forward pass, the algorithm calculates the network's response to a sequence of inputs, storing the states and spikes at each time step. The loss function, typically involving the difference between the expected and actual spike patterns, is then evaluated.

In the backward pass, BPTT computes the gradients of the loss function with respect to the network's parameters. This involves backpropagating the error through the unfolded network layers, taking into account the temporal dependencies, as explained in [9]. The gradients are then used to update the synaptic weights, minimizing the loss and improving the network's performance over time. Mathematically, the parameter updates are ruled by the gradient descent equation:

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \mathscr{L}}{\partial \theta_t} \tag{1.9}$$

where θ_t represents the network parameters (synaptic weights and time constants) at time step t, η is the learning rate, and $\mathscr{L}$ is the loss function. The error gradients $\frac{\partial \mathscr{L}}{\partial \theta_t}$ are computed by backpropagating the loss through time:

$$\frac{\partial \mathscr{L}}{\partial \theta_t} = \sum_{\tau=t}^{T} \frac{\partial \mathscr{L}}{\partial h_\tau} \frac{\partial h_\tau}{\partial \theta_t} \tag{1.10}$$

where h_τ represents the hidden states of the network at time step τ. A review of the view of the main steps of BPTT is provided in Alg. 1.

One significant challenge in training SNNs with such a method is related to the *non-differentiability* of the spiking function. The hard thresholding nature of spikes[3] means that traditional gradient-based optimization methods are not directly applicable. To address this, a common solution is to replace the non-differentiable spike function with a smooth, differentiable approximation during the backward pass only. Common surrogate functions include the so-called "fast" sigmoid or some kind of piecewise linear function, which approximate the spiking behavior well enough to allow gradient-based optimization.

During training, the forward pass uses the actual spiking dynamics, while the backward pass uses the surrogate gradient to compute the error gradients. This approach effectively "tricks" the optimization process into working with the non-differentiable spike events.
The application of BPTT with surrogate gradients allows SNNs to be trained on a wide range of tasks, including time-series prediction, classification, and even control systems. The temporal dynamics captured by SNNs make them particularly well-suited for tasks involving sequence learning and real-time processing. This is, in fact, the approach that will be considered for training the networks to be mapped on hardware devices.

Algorithm 1 Backpropagation Through Time (BPTT)

Initialize network parameters θ
Initialize learning rate α
Initialize truncation parameter T
Initialize initial hidden state h_0
for each training sequence $(x_1, x_2, \dots, x_N)$ **do**
  Forward Pass:
  for $t = 1$ to N **do**
    $h_t \leftarrow f(h_{t-1}, x_t; \theta)$
    $\hat{y}_t \leftarrow g(h_t; \theta)$
    Compute loss $L_t \leftarrow \ell(\hat{y}_t, y_t)$
  end for
  Backward Pass:
  Initialize gradient accumulators: $\nabla_\theta L \leftarrow 0$
  Initialize gradient at final time step: $\delta_N \leftarrow \frac{\partial L_N}{\partial h_N}$
  for $t = N$ to 1 **do**
    if $t \geq T$ **then**
      $\delta_{t-T} \leftarrow \delta_{t-T} + \frac{\partial L_t}{\partial h_{t-T}}$
    end if
    Accumulate gradients: $\nabla_\theta L \leftarrow \nabla_\theta L + \frac{\partial L_t}{\partial \theta}$
    Propagate gradients through time: $\delta_{t-1} \leftarrow \delta_t \cdot \frac{\partial h_t}{\partial h_{t-1}}$
  end for
  Update parameters: $\theta \leftarrow \theta - \alpha \nabla_\theta L$
end for

[3]Mathematically represented with an Heaviside step function.

1.3.2 Spike timiming dependent plasticity

The tecnique discussed until this point regards supervised learning, still, a well-established technique also exists for *unsupervised learning*, namely Spike-Timing-Dependent Plasticity (STDP). This is a biologically inspired learning rule that adjusts the synaptic weights in spiking neural networks (SNNs) based on the precise timing of spikes between pre- and post-synaptic neurons. The core idea is that the synaptic modification is influenced by the relative timing of spikes: if a pre-synaptic neuron fires shortly before a post-synaptic neuron, the synaptic weight is typically increased, in which case the process takes the name of long-term potentiation (LTP), whereas if the pre-synaptic neuron fires shortly after the post-synaptic neuron, the synaptic weight is decreased (long-term depression, LTD). This temporal learning rule is mathematically formulated to capture these dynamics, in fact, according to [19] the weight change (Δw) in STDP needs to be calculated as:

$$\Delta w = \begin{cases} A_+ e^{-\Delta t/\tau_+}, & \text{if } \Delta t > 0 \\ -A_- e^{\Delta t/\tau_-}, & \text{if } \Delta t < 0 \end{cases}$$

where $\Delta t = t_{\text{post}} - t_{\text{pre}}$ is the time difference between the post-synaptic and the pre-synaptic spike. The parameters A_+ and A_- are positive constants that determine the maximum magnitude of potentiation and depression, respectively, whereas the time constants τ_+ and τ_- control the rate of exponential decay for potentiation and depression.

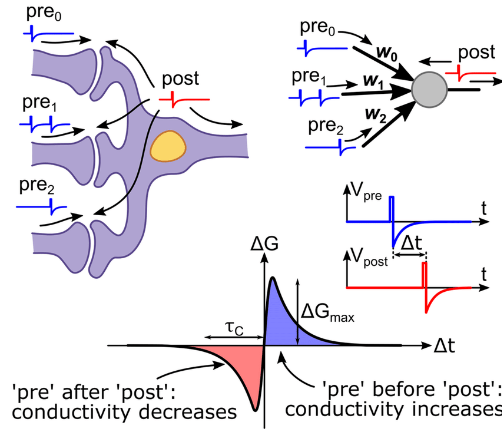


Figure 1.4: Synaptic weights correction mechanism of STDP. Taken from [35].

When $\Delta t > 0$, i.e. the pre-synaptic neuron fires before the post-synaptic one, the synaptic weight w is increased, conversely, when $\Delta t < 0$, w is decreased. This equation reflects the principle that the closer in time pre-spike and post-synaptic spikes are, the greater will be the increase in synaptic strength.

The combined learning window of STDP is often visualized as an asymmetric window function that integrates both the LTP and LTD components, as depicted in Fig. 1.4. This window function indicates that synaptic modification is highly sensitive to the exact temporal relationship between pre- and post-synaptic spikes, fostering an intrinsic mechanism for temporal pattern learning. The parameters A_+, A_-, τ_+, and τ_- can be tuned to reflect biological observations

or to meet specific computational goals. In practice, these parameters are set based on the desired learning dynamics and the specific application domain. By carefully choosing these parameters, STDP can be harnessed for various unsupervised learning tasks such as pattern recognition [44], feature extraction [10] and anomaly detection [3]. Regardless of how promising this approach has demonstrated to be, it has not been considered for training the physically-realized network to the tasks required by this work, since a supervised learning approach has revealed to be mandatory.

Algorithm 2 Spike-Timing-Dependent Plasticity (STDP)

```
Initialize parameters: synaptic weights w_ij, decay constant τ, potentiation &
depression time constants τ+ & τ−
Initialize hyperparameters: learning rate α
for each pair of pre-synaptic neuron i and post-synaptic neuron j do
    Initialize spike timing trace s_i ← 0 and s_j ← 0
end for
for each time step t do
    for each pre-synaptic neuron i do
        if neuron i fires at time t then
            s_i ← s_i + 1
        end if
    end for
    for each post-synaptic neuron j do
        if neuron j fires at time t then
            s_j ← s_j + 1
            for each pre-synaptic neuron i do
                Δt ← t_j − t_i
                if Δt > 0 then
                    w_ij ← w_ij + α · exp(−Δt/τ+)
                else
                    w_ij ← w_ij − α · exp(Δt/τ−)
                end if
            end for
        end if
    end for
    for each pre-synaptic neuron i and post-synaptic neuron j do
        s_i ← s_i · exp(−1/τ)
        s_j ← s_j · exp(−1/τ)
    end for
end for
```

1.4 Advantages of SNNs

Spiking Neural Networks offer several advantages over traditional artificial neural networks, some of which are:

- Event-Based Processing: Unlike ANNs that process data in a continuous manner, SNNs operate in an event-driven manner where information is processed only when there is a spike event. This can lead to improved efficiency and reduced computational costs, especially in tasks that involve sparse or asynchronous data.

- Temporal Dynamics: SNNs naturally capture the timing and sequence of spike events, allowing them to encode and process temporal information more effectively than traditional neural networks. This makes them well-suited for tasks involving time-sensitive data and dynamic patterns.
- Energy Efficiency: The event-driven nature of SNNs can lead to significant energy savings, especially in applications where power consumption is a critical factor, as suggested by [20]. Its spike-based communication in fact allows them to be temporally dynamic and energy-efficient, making them suitable for processing sequential and time-dependent data. This makes SNNs promising candidates for low-power neuromorphic hardware implementations and edge computing devices.
- Robustness to Noise: According to [29], SNNs exhibit inherent fault tolerance and robustness to noise due to their spiking nature. They can handle noisy inputs and partial information more effectively, making them suitable for harsh environments with uncertain or variable data.
- Sparse Connectivity: SNNs often employ sparse connectivity patterns, where neurons are only connected to a subset of other neurons. This sparse connectivity can lead to more efficient information processing and improved scalability in large-scale neural networks.

These advantages make spiking neural networks a compelling choice for tasks that require efficient and temporally sensitive processing of data. While there are challenges in training and optimizing SNNs, ongoing research and developments are addressing these issues, paving the way for broader adoption of spiking neural networks in various applications. With this work the event-based processing capabilities of SNNs have been explored, leading to the considerations later reported in chapter 5.

2

Modeling and training SNNs

After the brief description of SNNs assessed in the previous chapter, it is time to direct the reader's attention to how to practically deal with these networks, i.e. how to model and track the evolution of its internal nodes and, most importantly, how to train them for a specific purpose.

In order to derive a finite difference equation (FDE) describing the LIF neuron internal state, the Forward Euler discretization has been applied, for which the derivative of a generic continuous function (of time t) can be approximated as

$$\frac{dy}{dt} = f(t, y) \approx \frac{y(t + \Delta t) - y(t)}{\Delta t} = \frac{y[t_{n+1}] - y[t_n]}{t_{n+1} - t_n} \tag{2.1}$$

from which the following iterative expression for the future value of y is obtained:

$$y[t_{n+1}] = y[t_n] + hf(t_n, y(t_n)) \tag{2.2}$$

where $h = t_{n+1} - t_n$. Writing the discrete time instant directly as t, for which the distance between two consecutive samples is simply $h = 1$, and applying Eq. 2.2 to Eq. 1.2 finally yields the following expression for the LIF neuron membrane potential

$$U[t+1] = \beta U[t] + WX[t+1] - S[t]U_{thr} \text{ , where } S[t] = \begin{cases} 1, \text{ if } U[t] > U_{thr} \\ 0, \text{ otherwise} \end{cases} \tag{2.3}$$

This equation, upon further addition of a constant bias term b, will be used to describe the network both in software, for monitoring and training purposes, as well as in hardware. A discrete-time version of Eq. 1.7 can also be easily obtained employing a set of two FDEs, and it is directly reported in Sec. 3.1.2 when discussing the HDL code for a second order LIF neuron.

2.1 Inference

Matlab has been used to reproduce the inner workings of the SNN. To begin with, the function that was written to reproduce the behaviour of a LIF neuron is reported in Code 2.1. As a first step, the output state is determined by comparing the current membrane potential value (passed as input of the function) and the defined threshold. Secondly, the new value of the potential, i.e. $U[t+1]$ of Eq. 2.3, is calculated and brought to the output with the previous one. The same code can actually be used to simulate an entire layer of neurons of the same type, thanks to the use of elementwise products (`.*`) operated on the array defining the decay constant and the one containing each neuron's membrane potential, between the threshold of each neuron in the layer and output the spike values, and finally with the matrix product between the input vector and the weight matrix describing all the layer interconnections.

```
1 function [spk, mem] = LIF_neuron_original (mem, x, w, beta, threshold)
2     spk = (mem > threshold);
3     mem = beta.*mem + (x'*w)' - spk.*threshold;
4 end
```

Code 2.1: Matlab function for implementing LIF neuron signal elaboration. Actually, this function simulates an entire layer of neuron, as it accepts a vector as input x and a matrix for weights w.

An example of simulation of a LIF neuron based on Code 2.1 is reported in Fig. 2.1. In this case the neuron features a single input, allowing to observe the evolution of $U[t]$ as a function of the state of the only input. Every time the membrane potential exceeds the threshold, $U_{thr} = 1$ the state of the output is brought high. Note how the exponential decay of $U[t]$ takes about 9 timesteps to fall below 5% of the starting value, since $\beta = 0.7$.

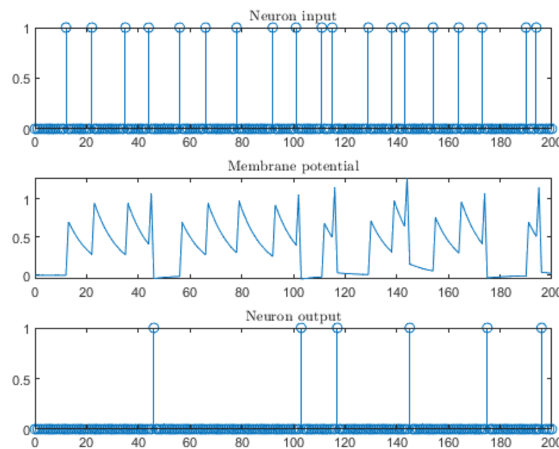


Figure 2.1: LIF neuron response to a random input stimulus over 200 timesteps. Here $W = 0.7$, $b = 0, U_{thr} = 1, \beta = 0.7$.

Since the training of the network will be operated on mini-batches of input samples, as later discussed in Sec. 2.2-2.3, a new version of the `LIF_neuron` funtion was realized, exploiting *Mat-*

lab vectorization capabilities to immediately and efficiently perfom the inference of an entire layer of neurons over multiple sets of stimuli simultaneously. In order to do so a tensor multiplication function was defined, after having previously reshaped the `w` tensor to the adequate dimensions, as shown in Code 2.2.

```
function [next_spk, next_mem] = LIF_neuron (curr_mem, x, w, b, beta, threshold)
    next_spk = (curr_mem > threshold);
    w = permute(w, [2, 1, 3]);
    next_mem = beta.*curr_mem + multiply3Dmat(w, x) + b - next_spk.*threshold;
end
```

Code 2.2: Matlab function of LIF neuron (entire layer, actually), accepting a tensor as input data ($n_{channels} \times n_{timesteps} \times n_{samples}$).

Finally, to simulate the inference of a SNN with generic structure, cell arrays have been used to store the arrays of network parameters, as well as spikes and membrane potential of each neuron. Each of this arrays has, in fact, its own dimension, depending on the corresponding layer size and position within the net. Indeed, a loop is perfomed on the layer index starting from the second[1] up to the last one and for each layer the `LIF_NEURON` function is applied for all the timesteps in sequence, as described by Code 2.3.

```
function [spk_data, mem_data] = SNN(in_data, layer_dim, w, b, beta, thr)
    x = in_data;
    n_layer = length(layer_dim); n_timesteps = size(in_data, 2); n_samples = size(in_data, 3);
    spk_data{1} = x; % The output of the input neurons is simply the input data
    mem_data{1} = [];
    for j = 2:1:n_layer
        w_prime = repmat(w{j-1}, 1, 1, n_samples);  % Extending dimensions to allow
        b_prime = repmat(b{j-1}, 1, 1, n_samples);  % matrix calculations all at once,
        beta_prime = repmat(beta{j},1,1,n_samples); % instead of looping through
        thr_prime = repmat(thr{j}, 1, 1, n_samples); % samples
        out_spk = zeros(layer_dim(j),n_timesteps,n_samples); % Preallocating the size
        out_mem = zeros(layer_dim(j),n_timesteps,n_samples); % of output matrices
        for i = 1:1:n_timesteps
            [spk, mem] = LIF_neuron(out_mem(:, i, :), x(:, i, :), w_prime(:, 1, :), ...
            ... b_prime(:, 1, :), beta_prime(:, 1, :), thr_prime(:, 1, :));
            out_mem(:, i+1, :) = mem; out_spk(:, i+1, :) = spk;
        end
        x = out_spk(:, 2:end, :);
        spk_data{j} = x; mem_data{j} = out_mem(:, 2:end, :);
    end
end
```

Code 2.3: Matlab function for general-structure SNN simulation.

An example of network evolution is reported in Fig. 2.2. Specifically, the membrane potential of the first neuron of each layer is shown. Note how the sudden drops in value of the blue curve, corresponding to $U[t]$ of the first neuron of the last layer, are caused by that neuron producing a spike. The exponential decay then does not tend to settle around zero in absence of external stimulation, since a noteworthy bias term is added at every new timestep.

[1]The first layer does not need any membrane potential to be calculated, as its outputs spikes are simply given by the input stimulus to the network.

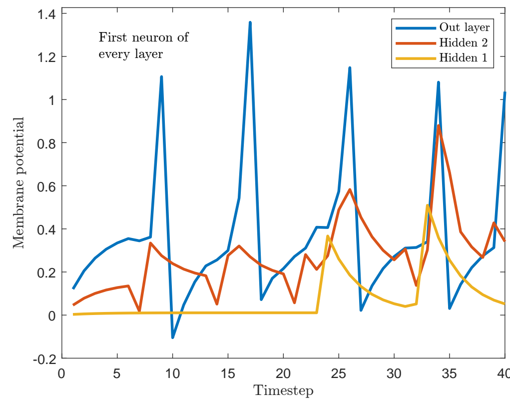


Figure 2.2: Evolution of $U[t]$ over 40 timesteps for the first neuron of each layer of a SNN with structure [16, 64, 32, 16].

2.2 Training via *SNNTorch*

Over the past few years many tools for spiking neural network simulation and training have been developed, and among all of them *SNNTorch* [33] has been chosen to first approach SNN training. As the name suggests, it is a Python library designed to facilitate the use of spiking neural networks within the PyTorch ecosystem. After installing the framework and importing the training dataset, the definition of a SNN is done by instantiating a `snn.Leaky()` object, which implements a layer of LIF neurons and which is combined with the other normally used blocks for ANNs, such as linear layers, implementing feedforward layer interconnections, as well as 1d/2d convolutions and max/average pooling operations. Several types of spiking neurons are present, the already mentioned one and `snn.Synaptic()`, implementing 2nd order LIF, being the most useful. Additionally, `spike_grad` needs to be assigned accordingly to the type of surrogate gradient function to apply. An example of network definition is reported in Code 2.4. Since neurons chained together in `nn.Sequential()` expect only one value, the `init_hidden` flag is used to initialize the hidden states as instance variables to be processed in the background. The final layer is not bound by this constraint, and can return multiple tensors: "`output = True`" enables the final layer to return the hidden states in addition to the spikes, allowing to monitor the overall evolution of the net and to eventually apply loss functions directly to $U[t]$. It is also possible make β and U_{thr} of LIF layers learnable parameters by enabling the corresponding flag, and by passing a *numpy* array containing the parameter initialization for each neuron.

```
1 net = nn.Sequential(nn.Linear(num_inputs, num_hidden)
2                     snn.Leaky(beta=0.7, spike_grad=surrogate.fast_sigmoid(), init_hidden=True),
3                     # Additional layers may be added e.g.
4                     # nn.Conv2d(in_ch, out_ch, kernel_size),
5                     # nn.MaxPool2d(kernel_size),
6                     # nn.Flatten(),
7                     nn.Linear(num_hidden, num_outputs),
8                     snn.Leaky(beta=0.7, spike_grad=surrogate.ATan(), init_hidden=True)
9                     ).to(device)
```

Code 2.4: Python code snippet for single hidden layer SNN definition, using *Sequential()* construct.

Then, network inference on the given input stimuli is carried out by looping `net(In_stimulus)` execution over all the timesteps in sequence, recording the value of spikes and membrane potentials (if needed) at each instant. These will be used to evaluate the necessary gradients for network parameters learning.

```
def forward_pass(net, data, num_steps):
  spk_rec = [] # record spikes over time
  utils.reset(net)  # reset/initialize hidden states for all LIF neurons in net
  for step in range(num_steps): # loop over time
      spk_out, mem_out = net(data) # one time step of the forward-pass
      spk_rec.append(spk_out) # record spikes
      mem_rec.append(mem_out) # record potentials
  return torch.stack(spk_rec, mem_rec)
```

Code 2.5: Python function executing SNN inference.

Finally the type of loss, e.g. `snntorch.functional.mse_count_loss()`, which calculates the mean square error on the accumulated spike count over the number simulation timesteps, or `ce_rate_loss()`, which instead provides the cross-entropy error as $e(x) = -\sum_x p(x)log\,(p(x))$, and the employed optimizer, e.g. `torch.optim.Adam()`, are set up. With that, all the preliminary steps are completed, and the training loop can be set up as per Code 2.6.

```
for epoch in range(num_epochs):
    for i, (data, targets) in enumerate(iter(train_loader)):
        data = data.to(device)
        targets = targets.to(device)
        net.train()
        spk_rec = forward_pass(net, data, num_steps) # forward-pass
        loss_val = loss_fn(spk_rec, targets) # loss calculation
        optimizer.zero_grad() # null gradients
        loss_val.backward() # calculate gradients
        optimizer.step() # update weights
        loss_hist.append(loss_val.item()) # store loss
        acc = SF.accuracy_rate(spk_rec, targets) # check accuracy on a single batch
        acc_hist.append(acc) # keep track of accuracy evolution
```

Code 2.6: Training loop within SNNTorch.

The `adam` optimizer, introduced by Diederik P. Kingma and Jimmy Ba in 2014, implements by itself an adaptive adjustment of the learning rate for each parameter in the model based on the history of gradients calculated for that parameter. This helps the optimizer converge faster and more accurately than fixed learning rate methods like SGD. Nonetheless, *overfitting* may still occur, for which reason an *early-stopping mechanism* may be introduced to avoid this phenomenon, for instance by exiting the minibatch loop if no improvement on the accuracy of the evaluation dataset is found after a significant number of iterations, or whether the corresponding global loss has continued to diminish instead of being more or less stuck to the same value.

A significant advantage of *SNNTorch* w.r.t. its "competitors" is that it natively embeds many different loss function metrics, neuron models, specifically designed spiking layers, as well as surrogate gradient functions, allowing more flexibility in designing custom SNN architectures.

2.3 Training via *Matlab*

Python-based frameworks allow to use the `autodiff` function to perform backpropagation. However, for a training procedure purely based on *Matlab* all the partial derivatives making up the corrective factors for the various parameters need to be expressed analytically, as outlined in [25]. Taking as an example a simple feedforward network with one hidden layer composed by a single 1st order neuron, the correction quantities are given by:

$$\frac{\partial \mathcal{L}}{\partial W^{(2)}} = \frac{\partial \mathcal{L}}{\partial y^{(2)}} \frac{\partial \tilde{y}^{(2)}}{\partial U^{(2)}} \frac{\partial U^{(2)}}{\partial W^{(2)}} = \frac{\partial \mathcal{L}}{\partial y^{(2)}} \frac{\partial \tilde{y}^{(2)}}{\partial U^{(2)}} y^{(1)} \tag{2.4}$$

$$\frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial y^{(2)}} \frac{\partial \tilde{y}^{(2)}}{\partial U^{(2)}} \frac{\partial U^{(2)}}{\partial y^{(2)}} \frac{\partial \tilde{y}^{(1)}}{\partial U^{(1)}} \frac{\partial U^{(1)}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial y^{(2)}} \frac{\partial \tilde{y}^{(2)}}{\partial U^{(2)}} W^{(2)} \frac{\partial \tilde{y}^{(1)}}{\partial U^{(1)}} x^{(1)} \tag{2.5}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = \frac{\partial \mathcal{L}}{\partial y^{(2)}} \frac{\partial \tilde{y}^{(2)}}{\partial U^{(2)}} \frac{\partial U^{(2)}}{\partial b^{(2)}} = \frac{\partial \mathcal{L}}{\partial y^{(2)}} \frac{\partial \tilde{y}^{(2)}}{\partial U^{(2)}} 1 \tag{2.6}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(1)}} = \frac{\partial \mathcal{L}}{\partial y^{(2)}} \frac{\partial \tilde{y}^{(2)}}{\partial U^{(2)}} \frac{\partial U^{(2)}}{\partial y^{(2)}} \frac{\partial \tilde{y}^{(1)}}{\partial U^{(1)}} \frac{\partial U^{(1)}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial y^{(2)}} \frac{\partial \tilde{y}^{(2)}}{\partial U^{(2)}} W^{(2)} \frac{\partial \tilde{y}^{(1)}}{\partial U^{(1)}} 1 \tag{2.7}$$

$$\dots$$

where e.g. $\frac{\partial \mathcal{L}}{\partial W^{(2)}} = (y^{(2)} - \hat{y}^{(2)})$ for MSE (Mean Square Error) loss function $\mathcal{L} = \frac{1}{2}\sum_i (y_i - \hat{y}_i)^2$, where $\hat{y}$ is the expected output. For surrogate gradient calculation two functions have been employed: the fast sigmoid function, given by `dS_over_dU = 1./(1+k.*abs(Uod)).^2` and the arctangent deribvative, `dS_over_dU = 1/pi./(1+(pi*mem_pot).^2)`, where $U_{od} = U[t] - U_{thr}$ is the so-called overdrive potential, and k becomes another hyperpameter.
For a generic number of hidden layers, the following recurrent term comes up:

$$x^{(k)} = \begin{cases} \frac{\partial \mathcal{L}}{\partial y^{(k)}} \odot \frac{\partial \tilde{y}^{(k)}}{\partial U^{(k)}} \text{ , if } k = n \\ \left(W^{(k)} \times x^{(k+1)}\right) \odot \frac{\partial \tilde{y}^{(k)}}{\partial U^{(k)}} \text{ , for } k \text{ from } n-1 \text{ downto } 2 \end{cases} \tag{2.8}$$

where the $\odot$ symbol is used to indicate Hadamard (or elementwise) product. From this equation the gradients w.r.t. the learnable paramaters yield

$$\frac{\partial \mathcal{L}}{\partial W^{(k-1)}} = y^{(k-1)} \times \left(x^{(k)}\right)^T \text{ , } \frac{\partial \mathcal{L}}{\partial b^{(k-1)}} = x^{(k)} \text{ , } \frac{\partial \mathcal{L}}{\partial U^{(k)}_{thr}} = -y^{(k)} \odot x^{(k)} \text{ , } \frac{\partial \mathcal{L}|_t}{\partial \beta^{(k)}|_t} = U^{(k)}|_{t-1} \odot x^{(k)}|_t \ \forall t \tag{2.9}$$

At this point one should apply the BPTT algorithm, due to the recurrent definition of the LIF neurons' membrane potential. For each traing quantity q the actual value to be multiplied by the learning rate and later substracted to q itself to apply gradient descent is given by:

$$\frac{\partial \mathcal{L}}{\partial q} = \sum_t \frac{\partial \mathcal{L}|_t}{\partial q} = \sum_t \sum_{s \leq t} \frac{\partial \mathcal{L}|_t}{\partial q|_s} \frac{\partial q|_s}{\partial q} = \sum_t \sum_{s \leq t} \frac{\partial \mathcal{L}|_t}{\partial q|_s} = \sum_t \left\{ \frac{\partial \mathcal{L}|_t}{\partial q|_t} + \sum_{s \leq t-1} \frac{\partial \mathcal{L}|_t}{\partial q|_s} \right\} \tag{2.10}$$

since q is constant over time.

The just completed mathematical digression brings to the `train_SNN` *Matlab* function, the most significant part of which is reported in Code 2.7. Commands for parameters update exploiting stochastic gradient descent (SGD) are not here reported for brevity, but they simply implement Eq. 1.9, where the correction factor is calculated as the mean of Eq. 2.10 terms over each minibatch. Four `learn_param` flags are then used to select which quantities to learn.

```
for l = 1:1:n_epochs % needs to be executed sequentially
        order_permutation = randperm(size(train_data,3)); % Randomly permuting the order of training examples
        train_data = train_data(:, :, order_permutation); exp_out_data = exp_out_data(:, :, order_permutation);
    for i = 1:1:n_minibatch % needs to be executed sequentially
            count = count + 1;
            % Forward pass: input-to-output network execution
            samples = (i-1)*minibatch_size+1:1:i*minibatch_size;
            [spk_data, mem_data] = SNN(train_data(:, :, samples), net_structure, w, b, beta, thr); % Built-in parallel implementation

            % Backward pass
            % Evaluate the surrogate gradient of the loss function w.r.t. net outputs
            dL_over_dy = loss_gradient(exp_out_data(:, :, samples), spk_data{end}, "MSE");
            % Evaluate surrogate gradient of step function for each layer of neurons
            dy_over_dU = cell(n);
            parfor k = 2:1:n % Paralleling for loop execution
                dy_over_dU{k} = surrogate_gradient(mem_data{k}, thr{k}, fast_sigmoid_k);
            end
            % Backpropagation related calculus: chain rule for derivatives
            x = cell(1, n-1); % Preallocating memory to gain speed
            x{n} = dL_over_dy.*dy_over_dU{n}; % initial condition of common backpropagation term
            for k = n-1:-1:2 % needs to be executed sequentially
                x{k} = (multiply3Dmat(w{k},x{k+1})).*dy_over_dU{k};
            end
            x{1} = zeros(1, size(train_data, 2), 1);
            extraction_interval = 2:1:size(train_data, 2);
            U_t_minus_one = cell(1, n);
            U_t_minus_one(2:end) = extract_value_at_t(mem_data(2:end), extraction_interval);
            y = spk_data;
            for k = n:-1:2
                U_t_minus_one{k} = cat(2, U_t_minus_one{k}, zeros(net_structure(k), 1, minibatch_size));
                dL_over_dW{k-1} = multiply3Dmat(y{k-1}, permute(x{k}, [2 1 3]));  dL_over_db{k-1} = x{k};
                 dL_over_dUthr{k} = -y{k}.*x{k}; dL_over_dbeta{k} = U_t_minus_one{k}.*x{k};
            end
            [...]
            % Back propagation through time
            for t = 1:1:size(train_data, 2) % needs to be executed sequentially
                immediate_W = dL_over_dW; immediate_b = dL_over_db; immediate_Uthr = dL_over_dUthr; immediate_beta = dL_over_dbeta;
                tprime = 1;
                prior_W = scalar_cell_by_cell(0, immediate_W); prior_b = scalar_cell_by_cell(0, immediate_b);
                prior_Uthr = scalar_cell_by_cell(0, immediate_Uthr); prior_beta = scalar_cell_by_cell(0, immediate_beta);
                while tprime < t
                    temp_x = elementwise_cell_by_cell(extract_value_at_t(x, t), ext_beta);
                    temp_W = matrix_cell_by_cell(extract_value_at_t(y(1:end-1), tprime), transpose_cell_by_cell(temp_x(2:end)));
                    temp_b = temp_x(2:end); temp_beta = scalar_cell_by_cell(0, prior_beta);
                    temp_Uthr = elementwise_cell_by_cell(extract_value_at_t(scalar_cell_by_cell(-1, y), tprime), temp_x);
                    if tprime >=2
                        temp_beta = elementwise_cell_by_cell(extract_value_at_t(mem_data, tprime-1), temp_x);
                    end
                    tprime = tprime + 1;
                    prior_W = sum_cell_by_cell(prior_W, temp_W); prior_b = sum_cell_by_cell(prior_b, temp_b);
                    prior_Uthr = sum_cell_by_cell(prior_Uthr, temp_Uthr); prior_beta = sum_cell_by_cell(prior_beta, temp_beta);
                    delta_W = sum_cell_by_cell(immediate_W, prior_W); delta_b = sum_cell_by_cell(immediate_b, prior_b);
                    delta_Uthr = sum_cell_by_cell(immediate_Uthr, prior_Uthr); delta_beta = sum_cell_by_cell(immediate_beta, prior_beta);
                end
                % Now apply gradient descent ...
            end
```

Code 2.7: Matlab code snippet for SNN training using BPTT.

Further improvements to the code allow for concurrent processing of all minibatches samples also during training, thanks to the use of *Matlab*'s *parallel processing toolbox*. This results particularly helpful in reducing the training processing time, as well as allowing for an immediate update of parameters using SGD, for which the mean value of the corrections obtained for each training samples needs to be calculated. Furthermore, not disposing of any built-in optimizer such as `torch.optim.Adam()`, a *weight decay* mechanism has also been included, namely $L2$ regularization for interconnection weights and $L1$-type for all the other parameters.

2.4 Parameters extraction

Once the network has been trained following some of the presented methods, its defining parameters, represented as 32-bit precision floating point numbers, need to be adapted to a finite resolution fixed point representation that can be handled by an FPGA. In order to do so, the following sequence of *Matlab* commands has been employed for each constant:

1. `p_quant = fi(p, sign, nbit, frac)` is used to effectively quantize the original number, where `sign` is set to 0 for unsigned types and to 1 for signed ones[2], `nbit` is the total number of bits used for representation and `frac` tells how many of these need to be allocated for the fractional part.

2. `p_hex = hex(p_quant)` then returns the hexadecimal string representing the resulting `nbit` precision number, which can then be saved to a file using the `writematrix()` command or can directly be employed for initialization a constant in VHDL.

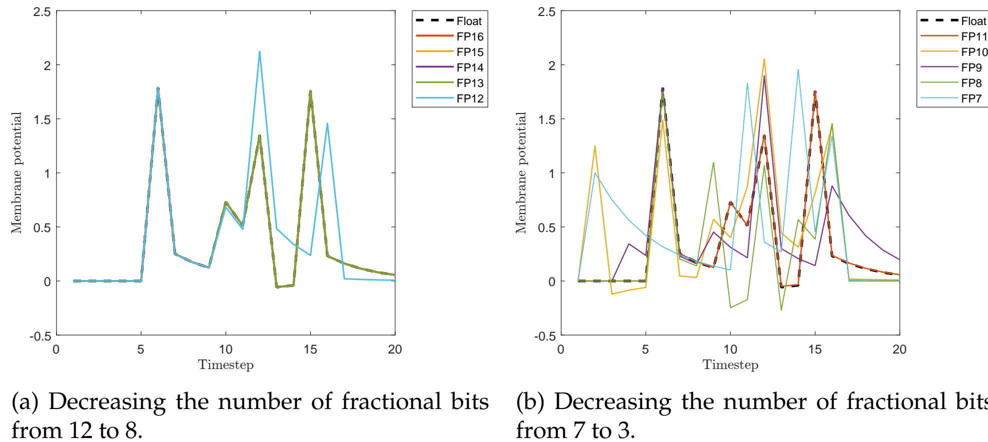
This procedure is of course valid also for Python-obtained parameters, which can be passed to *Matlab* by directly exporting them into a `.mat` file using the `savemat()` method from `scipy` library. Moreover, the opposite procedure can be also be put into action to analyze or debug the hexadecimal codes provided by a logic state analyzer. In order to do so a `quantizer` object resembling the type of data encoding has to be defined[3], which is then passed as argument to the `hex2num()` function together with the hexadecimal code of interest.

The number of integer and fractional bits must be carefully assigned so not to disrupt the equivalence between the *Matlab* model and the physically realized network, which must be operated under the rules of fixed point arithmetic. An example of the impact of quantization on the network parameters is reported in Fig. 2.3, in which 4 bits are allocated for the integer part (one of which is needed for sign encoding) and the remaining fractional part width is made to vary between 12 and 3. Clearly, as the representation resolution is lowered, more and more discrepancies appear between the expected and actual evolution of the network. In Fig. 2.3, for instance, even by using only 13 bits in total for parameters encoding, 9 of which are reserved to the fractional part, no macroscopic differences are found between the two. As resolution is further lowered, indeed, the two plots tend to diverge. A counterexample, in this sense, is however given by the red curve of Fig. 2.3-(b), labeled as *FP11*, which seems to "recover" the quantization error accumulated by the previous representation featuring an extra fractional bit. This shows how no general analysis on the effects of quantization is possible, as the actual network behaviour is totally dependent of the network structure, on the actual values assumed by the parameters and the operating conditions under which inference is performed, i.e. the input stimulus and current state of all neurons. Moreover, the analysis just shown is carried out in *Matlab*, where the various operations between `fi` objects cause their data width to be adjusted,

[2]In this case, by default, *Matlab* uses 2's complement for representing the integer part. It is nonetheless possible to change this specification to 'signed magnitude' by setting the corresponding property.

[3]An example of quantizer declaration is the following: *q = quantizer([16, 10], 'ufixed')*, which instantiates an unsigned fixed number with a total length of 16 bit, of which 10 are reserved to the fractional part. By default, *sfixed* type is used, and requires no additional string to be parsed.

and no explicit resizing command has been placed inside the LIF_NEURON function where assignment for the next value of the membrane potential is performed[4]. This differs from the actual implementation discussed in chapter 3, where the value of $U[t]$ is resized to the same resolution for each of the neurons composing the network. Therefore, the lesson learned here is that one should always apply the best possible quantization compatible with the available hardware resources so to avoid unaccounted errors.



(a) Decreasing the number of fractional bits from 12 to 8.

(b) Decreasing the number of fractional bits from 7 to 3.

Figure 2.3: Comparison between the 'ideal' evolution of $U[t]$ and the actual one as a result of some possible quantizations.

An alternative, smarter, way to overcome the problem of finding the correct data width is the direct quantization of the network evolution during inference and the subsequent application of the so-called Quantization-aware training (QAT) to deploy the SNN on resource-constrained devices. *SNNTorch* and other tools offer some support in this sense, however training convergence is not trivially obtainable with a limited amount of bits, and for this reason no example based on this technique will be shown. Further advancements for easing covergence of QAT of SNNs have nonetheless recently been presented in [8]. Also in this second case, however, an investigation on how PyTorch libraries apply quantization is needed, in order to evaluate whether the outcome of the training procedure will exactly resemble the behaviour of the SNN circuit or if it is possible to adapt QAT to the same rules used by HDL description.

2.5 Direct conversion from pretrained ANN

As a last point, a method for direct ANN to SNN conversion is now reported, even tough it has not been taken under consideration for experimental analysis. First, an already trained ANN, typically with ReLU activation functions, is used as the starting point. Each layer in the ANN is mapped to a corresponding layer in the SNN. The ReLU activation function in the ANN can be directly interpreted in the SNN by representing neural activation levels as spiking rates, as explained in [6]. A caveat is that the neurons reset must be imposed to zero for this procedure

[4]i.e. at line 4 of Code 2.2.

to work. To ensure robust conversion, the weights W of the ANN are normalized as:

$$\hat{W} = \frac{W}{\max(|W|)}$$

where the denominator is the maximum absolute value of the weights in a given layer. Inputs to the SNN are typically encoded in a rate-based or temporal coding scheme. Based on the discussion of Sec. 1.2, in rate encoding input values x are converted into spike rates as follows:

$$r_i = \frac{f_{\max} \cdot x_i}{\max(x)}$$

where r_i is the spike rate for input x_i, and $f_{\max}$ is the maximum firing rate. Each neuron in the SNN then integrates incoming spikes and generates an output spike train when the membrane potential U exceeds a certain threshold U_{thr}. Mathematically, this can be expressed as:

$$U(t) = \sum_i \hat{W}_{ij} \cdot S_i(t)$$

where $S_i(t)$ is the spike train from presynaptic neuron i, and $\hat{W}_{ij}$ is the synaptic weight between neurons i and j. As usual, a spike is then generated when $U(t) \geq U_{thr}$. A critical step in the conversion process is tuning the membrane potential thresholds and the synaptic weights to match the firing rate dynamics, as sketched in Fig. 2.4. Weight scaling is used to align the firing rates with the activation levels of the corresponding neurons in the ANN:

$$\hat{W}' = \alpha \cdot \hat{W}$$

where α is a scaling factor chosen to balance the dynamic range of the weights and the threshold.

Finally, the performance of the converted SNN is validated against the original ANN by running inference on test data and comparing the outputs. Iterative fine-tuning of the weights $\hat{W}$ and thresholds U_{thr} may be necessary to optimize spiking behavior and ensure the converted SNN performs comparably to the ANN in terms of accuracy and efficiency.

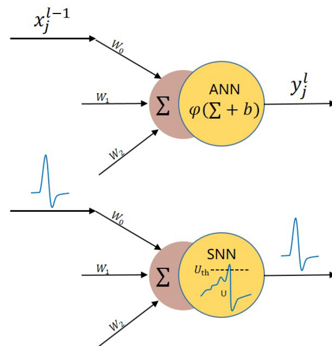


Figure 2.4: ANN-to-SNN conversion principle. Taken from [21].

3

RTL description of SNNs

In this chapter the structure of a generic spiking neural network will be constructed step by step using a series of VHDL entities, the main of which are reported in Fig. 3.5. In particular, the 2008 standard of VHDL has been used due to its enhanced fixed-point arithmetic support (through the inclusion of the `IEEE.fixed_pkg` library), better handling of generics and unconstrained arrays, and finally for the native `hex_read` function for direct acquisition of net coefficients from file. Of course, all of this was done by first ensuring that Xilinx Vivado[1], namely the software used for programming the FPGA, fully supported VHDL-2008 during RTL elaboration, simulation, synthesis and implementation of the design.

Let's first state that the aforementioned package defines two main types, both of which are unconstrained arrays of `std_logic` elements, called `sfixed` and `ufixed`. The former represents signed fixed-point numbers using 2's-complement notation, whilst the latter encodes unsigned numbers using a simple magnitude notation. Note that the range of the type uses `integer`, not `natural` as with `signed` or `unsigned`. This means that the range can include negative indices, which are used to represent numbers' fractionary part. Therefore, the range of a fixed-point number is defined by the number of bits used in its array representation and by the offset of the binary point. Throughout this section, the following convention will be used when referring to fixed-point data (i.e. network parameters and neruons' state variables) representation format:

- NBIT = total number of bits
- FRAC = number of bits reserved for fractional part
- INT = number of bits reserved to the integer part

Note that for `sfixed` type the last two terms do not simply add up to the first, but follow the equality NBIT = FRAC + INT + 1, where the extra bit is needed to take the sign of the numbers into account. More information about operators, conversion functions etc. may be found in [26].

[1]Specifically, *Vivado* 2022.1 was used for project development.

3.1 Neuron structure

The fundamental processing block of a SNN is the LIF neuron, whose proposed RTL scheme is reported in Fig. 3.1. Formally, the `NEURON` entity does not only account for the evolution of membrane potential and related output spikes generation as outlined in Eq. 2.3, but also includes the synapses entering into it, i.e. the interconnections from the preceding layer (or the input of the network) to the neuron under consideration. Classically, indeed, one needs to distinguish between linear layers (summing up the products of the input variables with the interconnections weights) and non-linear layers, which are actually what allows the network to learn a specific task. This distinction results clear by actually looking at how neural networks are normally defined e.g. in PyTorch[2]. From the hardware implementation perspective, however, this distinction has been dropped. The main reason for this choice is that the general transformation applied from the weight matrix W to the input signal vector X, which yields the synaptic current feeding the neuron's membrane potential and that in general needs to be implemented as a collection of products between two scalar quantities, now just becomes, in fact, a selective sum of some of the weights when X is supposed to assume just the two discrete values 0 and 1. This justifies the presence of the "mask" block in Fig. 3.1, which has exactly the function of deciding whether the stored value of a given interconnnection strenght should be passed to the summing stage or whether a zero should go instead. Therefore, this block has been obtained by looping over a `for ... generate` construct a given number of 2-input muxes, in which the selection input is simply the input spike to a given connection and the two data inputs are simply a zero and the correspondent connection weight.

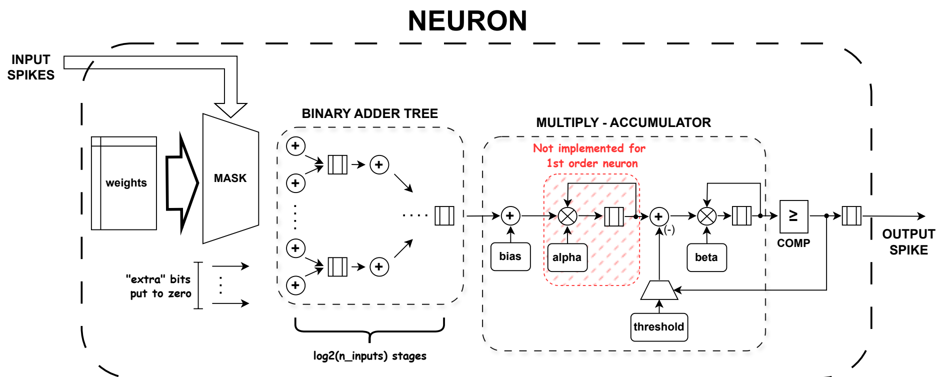


Figure 3.1: RTL model of the LIF spiking neuron. The red-coloured region inside the MAC block is present only in case the 2nd order model is considered.

After the weight have been masked, a summation needs to be performed. To do that in a hardware-efficient way an adder tree can be employed, perfoming at each stage the addition of two `sfixed(INT downto -FRAC)` terms at a time. After that a Multiply-and-ACcumulate (MAC) unit hadles the evolution of the membrane potential and finally a comparator to the defined threshold outputs the state of the neuron in response to a given set of input stimuli. Further

[2]Refer to Sec. 2.2.

details about this fundamental blocks are provided in Sec. 3.1.1 and 3.1.2.

Code 3.1 reports a snippet of a simplified version of a 1st order, reset-by-substraction LIF neuron as an example[3]. As it is evident, the structural modeling approach was here employed to mimick the underlying RTL model. The entity here shown is self-sustaining, as revealed by the fact that a control unit is also present with the aim of coordinating the interactions between the various blocks. Moreover, for debugging purposes, the output membrane potential is brought at the output in order to monitor the evolution of the internal state of the neuron and compare it, for example, with a *Matlab* simulation given the same operating conditions, as will be later presented at the end of this chapter.

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.all;
3  use STD.TEXTIO.all; use IEEE.STD_LOGIC_TEXTIO.all;
4  use IEEE.FIXED_PKG.all;
5  use WORK.SNN_PACK.all;
6
7  entity NEURON is
8    generic(N_INPUT                                          : natural;
9            WEIGHT_FILE, BIAS_FILE, THR_FILE, DECAY_FILE : string);
10   port(CLK, RESET, IN_VALID      : in  std_logic;
11        SPIKE_IN                   : in  std_logic_vector (0 to N_INPUT-1);
12        SPIKE_OUT, OUT_VALID       : out std_logic;
13        MEM_POT_OUT                : out sfixed(INT downto -FRAC));
14 end NEURON;
15
16 architecture STRUCT of NEURON is
17     constant weight      : T_DATA(0 to N_INPUT-1) := init_ram_hex(WEIGHT_FILE, N_INPUT);
18     constant bias        : sfixed(INT downto -FRAC) := init_ram_hex(BIAS_FILE, 1);
19     constant thr         : sfixed(INT downto -FRAC) := init_ram_hex(THR_FILE, 1);
20     constant decay       : sfixed(INT downto -FRAC) := init_ram_hex(DECAY_FILE, 1);
21     signal to_be_summed : T_DATA(0 to N_INPUT-1);
22     signal sum           : sfixed(INT downto -FRAC);
23     signal thr_mask, update_pot, res_pot, res_out: std_logic;
24 begin
25
26     MASK: entity WORK.W_MASK
27         generic map (MSIZE => N_INPUT)
28         port map (IN_DATA  => weight, SEL => spike_in, OUT_DATA => to_be_summed);
29
30     ADD: entity WORK.ADDER
31         generic map (ASIZE => N_INPUT)
32         port map (IN_DATA  => to_be_summed, CLK => CLK, OUT_DATA => sum);
33
34     MAC: entity WORK.MAC
35         port map (W_SUM => sum, B => bias, D => decay, THR => thr,
36                   RESET => res_pot, THR_MASK => thr_mask, STATE_UPDATE => update_pot,
37                   CLK => CLK, MEM_POT => MEM_POT_OUT);
38
39     P: process(CLK) -- Out spike comparator process (+ threshold subtraction masking)
40     begin
41         if CLK'event and CLK = '1' then
42             if RES_OUT = '1' then
43                 SPIKE_OUT <= '0';
44             elsif (MEM_POT_OUT >= THR) then
45                 SPIKE_OUT <= '1'; thr_mask <= '0' when RES_POT = '0' else '1';
```

[3]As stated in the abstract, the full code related to the project can be found at `https://github.com/marcotoffano/SNN_Thesis.git`.

```vhdl
46             else
47                 SPIKE_OUT <= '0'; thr_mask <= '1';
48             end if;
49         end if;
50     end process;
51 
52     CU : entity WORK.CU
53         generic map (N_INPUT => N_INPUT)
54         port map (START => IN_VALID, RESET => RESET, CLK => CLK,
55                   UPDATE_POT => update_pot,  RES_POT => res_pot,
56                   Y => SPIKE_OUT, THR_MASK => thr_mask, DATA_VALID => OUT_VALID);
57 
58 end STRUCT;
```

Code 3.1: LIF neuron (1st order) VHDL code snippet.

3.1.1 Binary Adder Tree

For a given number of terms to be added together, the actual number of inputs of this structure is equal to 2^n, where $n = \lceil log_2 (n_{inputs}) \rceil$ is the number of stages composing the adder. Inserting a register after each summation helps reducing timing problems and allows for pipeline operation. Unused bits are simply grounded and are therefore subsequently removed by the optimizer during synthesis phase, as shown by the Vivado compilation log snippets of Fig. 3.2.

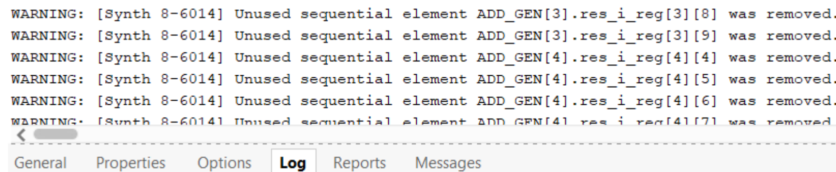


Figure 3.2: Vivado synthesis engine log showing removal of unused adder inputs. Removal of subsequent adder and registers block is also performed, but not here reported.

The VHDL code needed to generate this structure is composed of a `for ... generate` loop over the number of summing stages n, which at every iteration instantiates a clocked process performing the needed amount of assignments using another `for` loop, as outlined in Code 3.2. An "oversized" two-dimesional array is defined so to contain all the signals along the adder path[4]. As the generate construct proceeds to the later stages of the adder more and more registers are left unused and will be removed as well during compilation. The VHDL fashion for 2D array instantiation consists of the following lines of code:

```vhdl
type res_i_t is array (0 to 2**N-1) of sfixed(INT+N downto -FRAC);
type res_t is array (0 to N) of res_i_t;
signal res_i: res_t;
```

Note how, in order to avoid overflow (or underflow) as partial sums get accumulated, the number of bits allocated for the integer part of all the intermediate signals has been extended by a certsin amount n, owing to the fact that fixed point logic imposes an extra bit on every

[4]A similar approach will be also used in Sec. 3.3 for storing network parameters before distributing them over to the corresponding neurons.

two-terms addition of data having the same size. Actually, only the final result will need all of the extra bits, while in the previous stages additional resources will be freed once again during synthesis.

```
ADD_GEN: for k in 1 to N generate
    ADD_PROC: process (CLK)
    begin
        if CLK'event and CLK = '1' then
            for i in 0 to ((2**N)/(2*k))-1 loop
                res_i(k)(i) <= resize(res_i(k-1)(i*2) + res_i(k-1)(i*2+1),INT+N,-FRAC);
            end loop;
        end if;
    end process;
end generate;
```

Code 3.2: VHDL code snippet for cascade summation of an arbitrary number of input sfixed(INT downto -FRAC) signals.

3.1.2 MAC

The LIF neuron discrete-time state equation (Eq. 2.3) needs at this point to be implemented. In order to do so, each time a valid input is submitted to the network (corresponding to the `STATE_UPDATE` signal going high) a sum between the cumulative value of the "active" connections weights, the bias constant and membrane potential value at the previous instant needs to be executed. Moreover, an additional term comes into play after the output neuron fires, namely the threshold value, which actually needs to be substracted to the previous quantity. This, of course, in case reset-by-substraction is desired. Otherwise, when reset-to-zero mechanism is needed e.g. in order to apply the direct ANN-SNN conversion method presented in Sec. 2.5, the $U[t-1]$ term is dropped and the new value of $U[t]$ simply becomes the sum of $\sum_i w_i$ and b. An RTL description of such behaviour is obtained using Code 3.3.
Even better, the proposed `MAC` entity allows for further generalization of the neuron model. Depending, in fact, on the value of the `NEUR_TYPE` generic parameter, a second accumulation stage may be added so to implement the discrete time version of Eq. 1.7, which, in particular, can be expressed as

$$\begin{cases} I_{syn}[t+1] = \alpha I_{syn}[t] + WX[t+1] \\ U_{mem}[t+1] = \beta U_{mem}[t] + I_{syn}[t+1] - Reset[t] \end{cases} \tag{3.1}$$

In Code 3.3, I_{syn} corresponds to `sum1`, while U_{mem} is mapped to `sum2`. The partial products are evaluated outside of the clocked process, where instead the non-blocking assignments performing the conditional sum between the various terms reside. In case `NEUR_TYPE` = "1ord" the first stage is "skipped" in the `else ... if` construct and it is not therefore physically implemented.

```
1  entity MAC is
2    generic (NEUR_TYPE : string(1 to 4); RES_TYPE  : string(1 to 4));
3    port (W_SUM                              : in sfixed(INT downto -FRAC);
4          B, D1, D2, THR                     : in sfixed(INT downto -FRAC);
5          RESET, THR_MASK, STATE_UPDATE, CLK: in std_logic;
6          SYN_POT, MEM_POT                   : out sfixed(INT downto -FRAC));
7  end MAC;
8
9  architecture BHV of MAC is
10   signal sum1, sum2, prod1, prod2 : sfixed(INT downto - FRAC);
11 begin
12
13   P : process (CLK, RESET)
14   begin
15     if RESET = '1' then -- Asynchronous reset
16       if NEUR_TYPE = "1ord" then
17         sum2 <= (others => '0');
18       elsif NEUR_TYPE = "2ord" then
19         sum1 <= (others => '0'); sum2 <= (others => '0');
20       end if;
21     elsif CLK'EVENT and CLK = '1' then
22       if NEUR_TYPE = "1ord" then
23         if STATE_UPDATE = '1' then
24           if THR_MASK = '0' then
25             if RES_TYPE = "zero" then
26                 sum2 <= (others => '0');
27             elsif RES_TYPE = "subt" then
28                 sum2 <= resize(W_SUM+B+prod2-thr, INT, -FRAC);
29             end if;
30           else
31             sum2 <= resize(W_SUM+B+prod2, INT, -FRAC);
32           end if;
33         end if;
34       elsif NEUR_TYPE = "2ord" then
35         if STATE_UPDATE = '1' then
36           sum1 <= resize(W_SUM+prod1, INT, -FRAC);
37           if THR_MASK = '0' then
38             sum2 <= resize(sum1+B+prod2-thr, INT, -FRAC);
39           else
40             sum2 <= resize(sum1+B+prod2, INT, -FRAC);
41           end if;
42         end if;
43       end if;
44     end if;
45   end process P;
46
47   prod1   <= resize(sum1*D1, INT, -FRAC) when NEUR_TYPE = "2ord" else (others => '0');
48   prod2   <= resize(sum2*D2, INT, -FRAC);
49   SYN_POT <= sum1 when NEUR_TYPE = "2ord" else (others => '0');
50   MEM_POT <= sum2;
51
52 end BHV;
```

Code 3.3: Full VHDL code of generic MAC entity.

Recent literature suggests that as for ANNs, also the recurrent version of SNNs should be explored. Recurrency may be associated with the output spikes or with the input spikes as well. At this juncture, it is interesting to note that apart from the generic parameterization of the MAC entity and its nonlinear reset mechanism[5], the MAC unit is effectively acting as infinite impulse

[5]Which only acts on the last on the last stage of the filter, i.e. the one preceding the output.

response filter (IIR) as suggested by [27], whose feedback coefficients correspond to α and β of Eq. 3.1, respectively. The feedforward coefficients, in this case, are instead both set to zero, since no recurrent action of the input is present in the equation defining the hidden variable that regulates the output spiking activity, i.e. membrane potential. If the following more general formula is however considered for potential evolution:

$$U[t+1] = \beta U[t] + X[t+1] - R[t]U_{thr} + \gamma Y[t] + \delta X[t] \tag{3.2}$$

a non-zero feedforward term δ coming from the input side is also present.
The same authors of [27] also propose an IIR modeling for the neuron which does not necessary correlate with the deep learning concept of recurrent networks, but rather has to do with how neuron dynamics is brought to the discrete time domain in the first place. A feedforward term surely appears when, for instance, the *Tustin* time-discretization method is applied, for which the following approximation holds for the differential terms of Eq. 1.2 and 1.7:

$$\frac{dx(t)}{dt} \approx \frac{x[t+1] - x[t-1]}{2 \cdot \Delta t} \tag{3.3}$$

Regardless of what the origin for the presence of the feedforward terms is, a generalization to an arbitrary number of coefficients is possible, and a commonly employed approach for efficient generic-order IIR filter realization is that of cascading several digital biquad filters, as depicted in Fig. 3.3. The decision on what form to use for the implementation of the single stages needs then to be pondered considering the overall availability of resources.

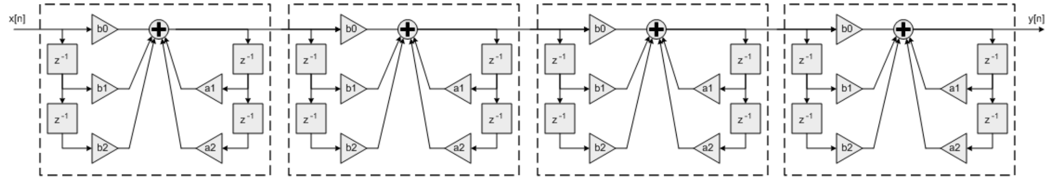


Figure 3.3: Generic-order IIR filter implemented as cascade of biquad stages.

3.2 Datapath sizing

Two possible approaches have been experimented for datapath sizing. In the first one:

- Weight summation within the `ADDER` entity is perfomed without loss of precision till the last stage, were the output is finally truncated back to `sfixed(INT downto -FRAC)`.
- The $U[t-1] \times d$ multiplication is also immediately truncated to the size of the single operands. In fact, since d is by definition a value less than unity, the loss of precision only affects digits with the lowest significance: one has to understand if the resulting error can be tolerated.
- Finally, the conditional sum between $\sum_i w_i, b, -U_{thr}$ and $U[t-1] \times d$ is resized to `sfixed(INT downto -FRAC)` as well, to be comprared with the equally defined U_{thr} constant.

When comparing with the golden reference network inference model in *Matlab*, slight differences in the exact values of the membrane potential are found. In particular, if no input stimulus is presented to the neuron, $U[t]$ tends to the LSB instead of actually settling to zero.
It is important to specificy that in order not to saturate, and consequenly cause data overflow, at any stage of the neuron datapath the resolution for network parameters has been set considering the worst possible case for max and min value of $U[t]$ state variable and then by choosing a sufficently high number of INT bits so not to exceed that value plus the threshold. Even after having kept track of the most extremes $U[t]$ values, it is good to take a safety margin on that, so to ensure a correct network operation even in conditions different from the nominal ones, specifically in case of greater incoming spiking activity, which implies a greater value needs to be accumulated in the MAC. The FRAC precision, however, remains totally arbitrary.

The compiler will, in the end, optimize all the worst-case extra (and unneeded) resources, as shown by the info messages reported in Fig. 3.4.

```
INFO: [Synth 8-3333] propagating constant 0 across sequential element (\GEN[5].NEUR/ADD /\ADD_GEN[1].res_i_reg[1][3][7] )
INFO: [Synth 8-3886] merging instance 'GEN[5].NEUR/ADD/ADD_GEN[1].res_i_reg[1][4][-11]' (FDS) to 'GEN[5].NEUR/ADD/ADD_GEN[
INFO: [Synth 8-3886] merging instance 'GEN[5].NEUR/ADD/ADD_GEN[1].res_i_reg[1][4][-10]' (FDS) to 'GEN[5].NEUR/ADD/ADD_GEN[
INFO: [Synth 8-3886] merging instance 'GEN[5].NEUR/ADD/ADD_GEN[1].res_i_reg[1][4][-9]' (FDS) to 'GEN[5].NEUR/ADD/ADD_GEN[1
INFO: [Synth 8-3886] merging instance 'GEN[5].NEUR/ADD/ADD_GEN[1].res_i_reg[1][4][-7]' (FDS) to 'GEN[5].NEUR/ADD/ADD_GEN[1
INFO: [Synth 8-3886] merging instance 'GEN[5].NEUR/ADD/ADD_GEN[1].res_i_reg[1][4][4]' (FDS) to 'GEN[5].NEUR/ADD/ADD_GEN[1]
```

Figure 3.4: Datapath width optimization performed by Vivado synthesis compiler.

If, instead, an approach sticking closer to fixed point arithmetic rules is wanted, then:

- Weight summation output is kept at increased resolution: *INT*+$log_2(n)$ *downto -FRAC*.
- The $U[t-1] \times d$ multiplication is kept to max resolution: *sfixed(INT*×2+1 *downto -FRAC*×2*)*, too.
- The conditional sum between $\sum_i w_i$, b, $-U_{thr}$ & $U(t-1) \times d$ is also set to max resolution, i.e. *sfixed*($\max\{INT \times 2 + 1, INT + log_2(n)\} + k\ downto - FRAC \times 2$), where k is the additional number of bits needed for subsequent accumulation and may be chosen to be the $log_2(n_{timesteps})$ or set to a more realistic value, possibly observing how many spikes are likely to arrive in sequence.
- Finally, the stored membrane potential, which has to be compared with threshold, to be multiplied by d and to be sent as output of the neuron, is resized back to *INT downto -FRAC*.

Also in this case some optimizations will be perfomed by Vivado compiler, but the overall resource utilization results greater, since higher precision levels are kept for the fractional part.

3.3 Hierarchy Management

A sketch of the project structure is depicted in Fig. 3.5. Neurons exhibiting the same number of input terms are grouped together to form a `LAYER` entity, and layers with different characteristics are cascaded with one another to form the final network. Since all the neurons are made from the same fundamental blocks, a control unit ensuring correct data processing has been placed for each layer, even though it has been omitted in Fig. 3.5. The only common signals

between different layers are clock and reset, while all the others depend on the layer's position within the net.

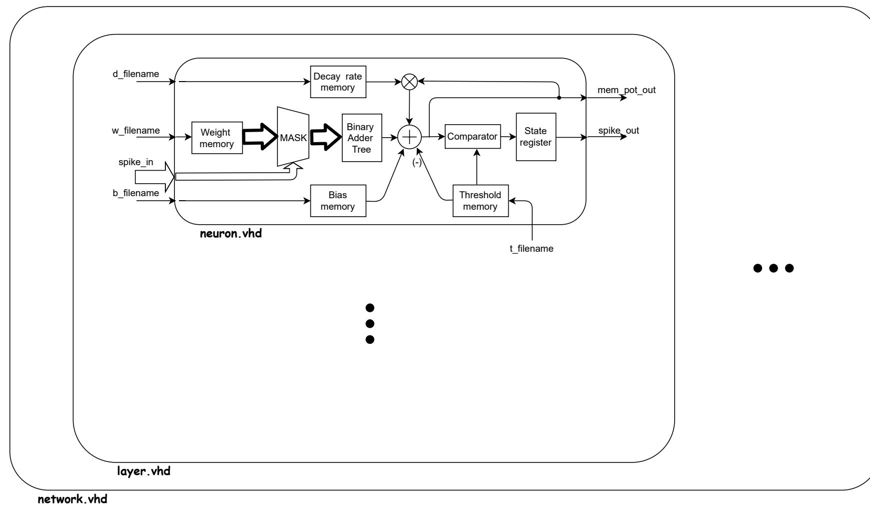


Figure 3.5: Hierarchical organization of SNN entity, with simplified internal scheme of the neuron.

The input signals to each layer are sent to all of its neurons, and different ways of retrieving the coefficients for each neuron have been studied:

1. Reading the values of w_i, b, t and d directly from a file, whose name is passed as generic `string` type parameter of the `NEURON` entity. Four different files were generated for each neuron[6], namely one per parameter type. In this way no constraint is imposed to how many values a file should contain, but the number of parameters to read[7] is imposed by another generic constant at neuron level (`N_INPUT`). This is the approach of Code 3.1.

2. Alternatively, the parameters of each neuron are defined as input of the network and for each layer a uniquely-sized array need to be defined to reproduce the network hierarchy. Oversized array types can also be defined so not to modify the code for every structural change of the net, in which case the compiler will take care of removing all the unused elements. It is even possible to define a new type collecting together all the arrays of layer parameters, which can be set directly as input of the `NETWORK` entity. This approach wil be used in Sec. 4.4 to allow coefficients reprogrammability. Moreover, this strategy also has the advantage of being retrocompatible, not exploiting, in fact, any feature of VHDL 2008, and will therefore be exploited also in chapter 6 when compiling the design with *Synopsys PRESTO*.

3. Finally, an attempt was done to simplify the previous approach by generating (in *Matlab*) a very long hexadecimal string containing all the network parameters in sequence,

[6]Assuming first order LIF model is used, otherwise an extra file is needed to store the additional time constant.

[7]Which corresponds to the actual number of stored values. The writematrix() *Matlab* function was sequentially used to this scope, feeding it with a truncated part of the array containing the parameters of a layer, namely that representing a specific neuron.

following the order dictated by the project hierarchy. This string is defined as a global constant of `std_logic_vector` type in the package, and differently-sized portions of it are then passed within the `NETWORK` entity as inputs for the various layers. The range of indexes corresponding to each layer are retrieved by exploiting the array defining the network structure. Then, inside the `LAYER` entity other divisions of the parameters array took place to finally bring to each neuron the required parameters. Even tough the approach seemed promising, Vivado compiler faced some difficulties in correctly synthesizing the coefficients assignment, probably due to execution memory overutilization related to the huge length of the parameters bit vector.

3.4 Convolutional and pooling layers

Another type of layer that may be valuable also for spiking neural networks is the convolutional one, due to its ability to efficiently process spatially structured data. By incorporating convolutional layers into SNNs, it is possible to grasp the advantages of spatially localized receptive fields and weight sharing, which reduces the number of parameters and computational complexity. This enhances the network's ability to detect spatial patterns and hierarchies in the input data, similarly to traditional artificial neural networks. Moreover, the translation invariance property of convolutional layers can contribute to the robustness and generalization capabilities of SNNs, making them effective for tasks such as image and video recognition, where recognizing patterns irrespective of their position in the input space is crucial.

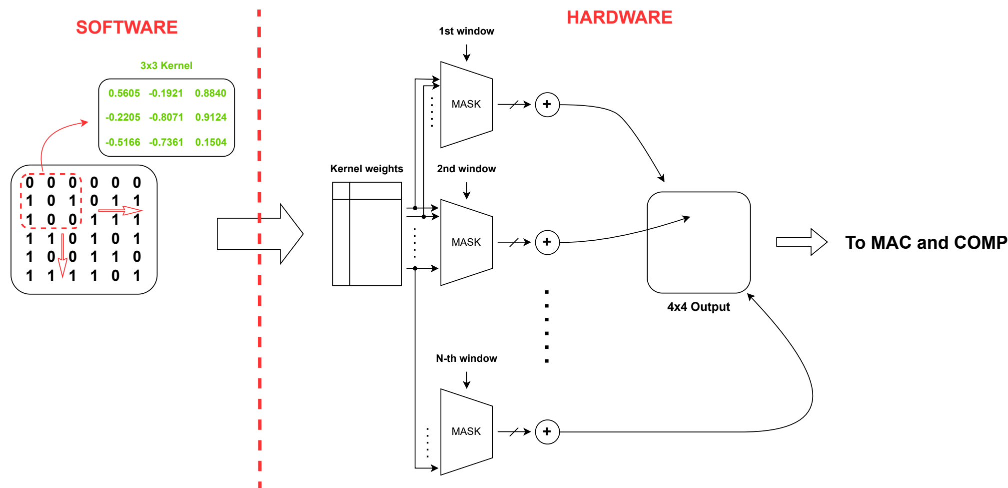


Figure 3.6: Skecth of convolution meschanism with input spikes for a given kernel matrix: software algorithm and hardware circuit.

For this reason an hardware implementation of a generic convolutional layer is here proposed, specifically in the case where the inputs of the convolutional layer are not the membrane potentials of the previous layer of neurons, but rather their output spikes. As in the case of fully-connected layers of Sec. 3.1, this assumption leads to a strong simplification of the architecture. In fact, for one-dimensional convolution, implemented using `conv1d()` method in PyTorch, sim-

ply using a set of `MASK` blocks for each available kernel yields the final result. The same reasoning also applies for 2D convolution (used in most image processing tasks), with some extra complexity being associated for correctly addressing the input channels to the corresponding kernel weight mask, expressed graphically in Fig. 3.6. In both cases the output, which would originally be composed of m sets of 1D or 2D arrays of numbers, is squeezed out to a mono-dimensional vector of coefficents, which are then taken as input of a MAC block to introduce spiking dynamics. On the network definition and training side, this dimensional compression is carried out with the `Flatten()` method in PyTorch and with consecutive applications of the `squeeze()` function in *Matlab*.

```
1 architecture BHV of MAX_POOL is
2     constant DIM: natural := N_INPUT/N_OUTPUT;
3 begin
4 GEN:     for i in 0 to OUT_SPIKE'high generate
5              OUT_SPIKE(i) <= or IN_SPIKE_VEC(i*DIM to (i+1)*DIM-1);
6          end generate;
7 end BHV;
```

Code 3.4: VHDL code snippet implementing max pooling exploiting OR reduction operator.

```
1  function count_ones(slv : std_logic_vector) return natural is
2      variable n_ones : natural := 0;
3  begin
4      for i in slv'range loop
5          if slv(i) = '1' then
6              n_ones := n_ones + 1;
7          end if;
8      end loop;
9      return n_ones;
10 end function count_ones;
11 begin
12 GEN:     for i in 0 to OUT_SPIKE'high generate
13              OUT_SPIKE(i) <=  '1' when count_ones(slv => IN_SPIKE_VEC(i*DIM to (i+1)*DIM-1)) > DIM/2
14                               else '0';
15          end generate;
16 end BHV;
```

Code 3.5: VHDL code snippet for average pooling of a group of spikes.

In ANNs, convolutional layers are often followed by pooling layers, which perform a reduction of the number of connections to the subsequent layer of the network. Codes 3.4-3.5 show how simple it is to implement maximum and average pooling layers when the inputs are only 0s or 1s. In case of max pooling, in particular, VHDL 2008 allows to use a "reduction" operator[8] instead of having to loop through the all the inputs to evaluate their logical state. This was not however possible for the other pooling time, which required the use of a process, then incapsulated into the `count_ones` function of Code 3.5. By then evaluating if the spike count is greater or equal than half of number of analyzed channels, the desired result is obtained.

[8]Truthfully, the same simplification can be obtained with the 1993 standard of VHDL by including the *std_logic_misc* package, which provides the *or_reduce* function.

3.5 Softmax

One of the most used types of output layers in neural networks employed for multi-class classification purposes is the so-called *softmax*, which performs the following mathematical operation over an input vector $X = \{x_1, x_2, ..., x_N\}$:

$$\{f(X)\}_i = \frac{e^{x_i}}{\sum_{k=1}^{N} e^{x_k}} \tag{3.4}$$

Indeed, the softmax of a given value within a lager set of values returns the probability of that value being the most relevant, i.e. begin associated with the class having the highest probability of being the correct one.

Even though this operation is widely used in neuromorphing computing perfomed on CPUs and GPUs, it is not trivial to implement it on programmble logic devices due to its computational complexity. It results clear from Eq. 3.4 that for each of the output values of the softmax, the calculation can be decomposed in three fundamental steps:

1. Calculate the exponential of each of the input values.
2. Sum up all the obtained values.
3. Divide each e^x term by their sum.

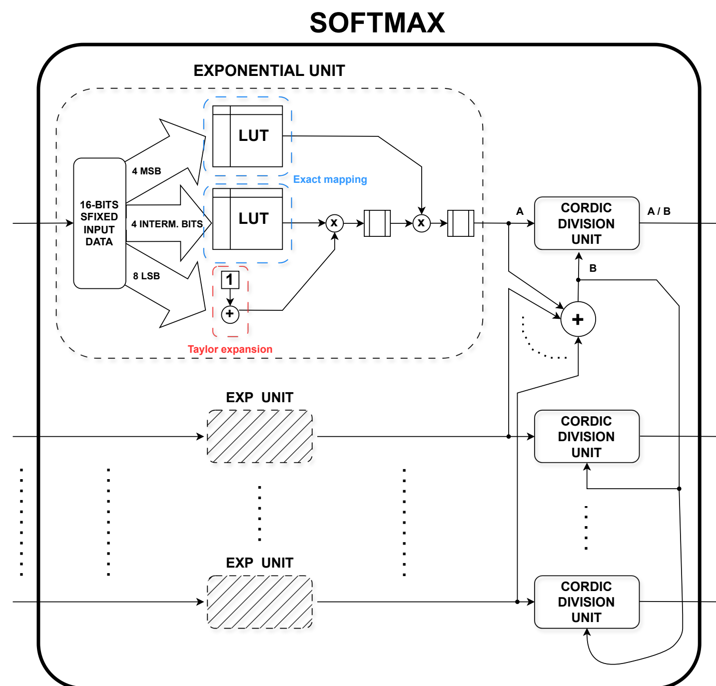


Figure 3.7: RTL scheme of the softmax unit.

The proposed circuit for this task is shown in Fig. 3.7, where also the structure of the exponential unit analised in Sec. 3.5.1 is highlighted. More details about the CORDIC-based division

unit will also be presented in Sec. 3.5.2. Putting aside for a moment the details of what the inner workings of such entities are, let us rather focus on the timing constraints imposed by this circuit. By construction, the exponential unit takes 2 T_{clk} to produce the result, then in the best possible scenario an additional T_{clk} is lost for summing all the exponential terms together[9], and finally an additional arbitrary number of clock strokes is needed by the CORDIC divider, depending on the desired precision of the output data, which is generally limited, since it is only representing a probability value[10]. Indeed, this last stage adds significant latency to the structure. Fortunately, all of the fundamental elements of the processing unit are natively pipeline or can be parallelized at the expense of additional resource usage, as it is the case for the CORDIC processor.

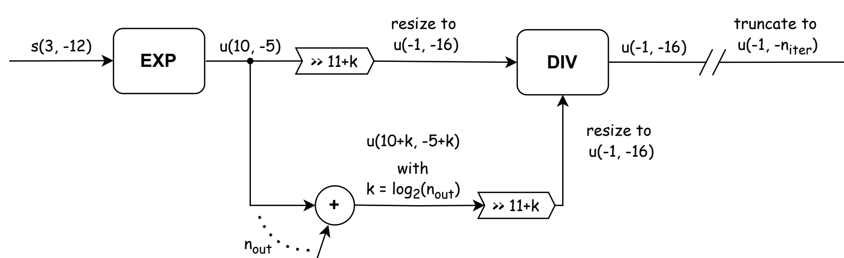


Figure 3.8: Softmax unit internal signals widths in the specific case of *sfixed(3 downto -12)* input data.

For what concerns the signals width along the various processing steps, the case of an input data of type `sfixed(3 downto -12)` is considered. First of all, the result of exponentiation of such a signal yields a purely positive number, so the type can be changed from `sfixed` to `ufixed`. Secondly, as further discussed in Sec. 3.5.1, a total of 11 bits are needed to correctly encode the exponential unit output when the input is at its maximum, so that if the total number of bits is kept constant[11] (at 16) the actual data type needs to be set as `ufixed(10 downto -5)`. Then, a given amount of this values is summed together, so that their sum will require an additional number of bits which is given by the logarithm of the number of terms on which to calculate the softmax. Finally, the CORDIC divider needs both the divider and the divisor to be of the same type, which will be the same used for the division result. Therefore, as this value is in any case smaller than one, the same totally fractional 16-bit unsigned fixed point number type has to be employed for all CORDIC unit ports. Indeed, both inputs are divided by the same quantity so to become purely fractional numbers and an efficient way of doing this is to simply right shift them by the number of bits representing the integer part of the biggest of the two, and finally resizing

[9]For standard classification datasets such as MNIST and its variations, only 10 output classes are present so a sigle adder summing 10 signals at a time is likely not to create any timing bottleneck till a resonably high clock frequency. Nonetheless, if the number of classes starts growing a more efficient structure should be considered, such as the one discussed in Sec. 3.1.1.

[10]Moreover, the aim of softmax function is exactly that of clearly separating what is the most likely outcome from all the rest, so few bits of fractional precsion are only needed, and, obviously, no space for the integer part has to be allocated.

[11]In this way some precision is lost at the output, notwithstanding there is no input variation for which the corresponding difference on the output side smaller than 2^{-5} can be appreciated.

the result to `ufixed(-1, -16)`[12]. The actual precision on the divison result finally depends on the number of processing steps granted to the CORDIC processor, meaning the actual result can be truncated to contain only the first n_{iter} bits. Visually, Fig. 3.8 best summarizes the just discussed signal bus width manipulation.

3.5.1 Exponential unit

Here a revision of the architecture proposed in [11] is considered with the aim of reproducing the application of the exponential function to the membrane potential of of the neurons of the output layer. A "naive" way of doing this operation on an FPGA is to simply employ a LUT associating to any possibile combinations of the bits composing the input signal the corresponding exponential value. This is the general method used to produce any function once the input data range is known. That is also how non-trivial combinatorial functions get realised on FPGAs, instead of using logic gates.

There is however a caveat to consider: the number of input combinations of the lookup table goes like $2^{n_{bit}}$, therefore using even a modest amount of bits for representing information may lead to an explosion of the resources to be employed. Moreover, LUTs with many input addresses are realized combining multiple smaller sized LUTs which are readily available in an FPGA. In particular, LUT6 blocks are normally present, featuring 6 single-bit inputs and one output as shown in Fig. 3.9, and can be used a an asynchronous 64-bit ROM with 6-bit addressing. Nevertheless, the same primitive can also be exploited to realize 4-to-1 muxes, by considering 4 of the input bits as "data" inputs and the remaining 2 bits as "select" lines. A LUT of generic length is then realized by cascading many LUT6 logic levels together, where the first stage is composed of a series a concurrently-operated ROMs and all the others are just needed to ensure proper multiplexing of outputs coming from the first stages. A simple example of this procedure can be seen in Fig. 3.9, where the standard LUT6 block is realized as a combination of two LUT5 and one 2-to-1 mux, whose selection bit is the 6th bit, which does not fit inside LUT5 modules.

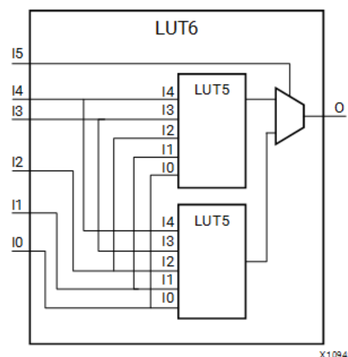


Figure 3.9: Xilinx's LUT6 primitive internal block scheme.

[12]Internally to the CORDIC unit, the two inputs will be aligned to the same type of the angle signal z, which needs to be an *sfixed(0 downto -15)*, as its value is given by the algebraic sum of fractional powers of 2, starting from $2^0 = 1$ downto 2^{-15}, as later explained in Sec. 3.5.2.

As presented in [32], the following formula can be used to estimate the LUT6 utilization for a given input data size[13] bigger then six:

$$n_{LUTs} = \frac{2^{n_{inputs}}}{64} + \sum_{k=1}^{\lceil log_4(n_{inputs}-6) \rceil} \left\lceil \frac{2^{n_{inputs}-6}}{4^k} \right\rceil = \sum_{k=0}^{\lceil log_4(n_{inputs}-6) \rceil -1} \left(2^{n_{inputs}-6-2k} \right) + 1 \tag{3.5}$$

where the first term accounts for the LUTs used as ROMs, while the other gives the number needed to realize the $2^{n_{inputs}-6}$-to-1 multiplexer that finally yields the result.
For input data having $n_{bit} = 16$ of resolution, the total resource usage to implement a general function whose result has, as well, 16-bit of resolution amounts to $n_{LUTs} = 16 \times 1365 = 21840$, corresponding to 4.12% utilization of total LUTs[14] present on the FPGA empoyed for circuit implementations of chapters 4-5.

In order to avoid such congestion, the following property of exponentials has been exploited:

$$e^x = e^{a+b+c} = e^a \cdot e^b \cdot e^c \tag{3.6}$$

Therefore, by subdividing the input into the sum of three terms, namely:

a) the first four bits, representing the integer part when employing `sfixed(3 downto -12)`

b) the second four bits, representing the most significant bits of the fractional part

c) the last eight (less significant) bits

and calculating the exponential separately for each of them, it is possible to obtain the final value by multiplying the resulting terms. Moreover, for the last 8 bits it may not be necessary to actually use a lookup table. In fact, if a sufficiently large number of fractional bits is used, those last eight bits represent a quite small quantity, for which it makes sense to approximate the result using the Taylor expansion $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} \approx 1+x$. Doing so, a very simple addition, two 4-input-bits lookup tables (for a total of 16 possible combinations) whose mapping is reported in Table 3.1-3.2, and two multiplications yield the desired processing. Note how the exponentiation of the integer part of the input data requires a total of $log_2(8914) = 11$ bits not to overflow, therefore to ease hexadecimal code reading from file in VHDL, the `ufixed(10 downto -5)` encoding scheme has been used.

[13] The actual number of synthesized LUTs may differ, due to the application of resourse sharing mechanisms by Vivado compiler, which may decide to substitute groups of LUT6s with some kind of mux (e.g. MUXF7, MUXF8, MUXF9) or underutilizted LUT6s with smaller LUTs (e.g. LUT3).

[14] Consult Appendix A.1 for further information on total XCKU115-2FLVB2104E FPGA resources.

Input		Output	
Address (hex)	Value (dec)	Code (hex)	Value (dec)
0	0	0020	1
1	1	0057	2.7188
2	2	00EC	7.375
3	3	0283	20.0938
4	4	06D3	54.5938
5	5	128D	148.4062
6	6	326E	403.4375
7	7	8914	1096.625
{8, 9, A, B}	{-8, -7, -6, -5}	0000	0
C	-4	0001	0.0313
D	-3	0002	0.0625
E	-2	0004	0.1250
F	-1	000C	0.3750

Table 3.1: Lookup table input address mapping to reproduce $f(x) = e^x$ using as x the 4 MSBs of a sfixed(3 downto -12) type, whose possible values are the integers in the range $[-8, 7]$.

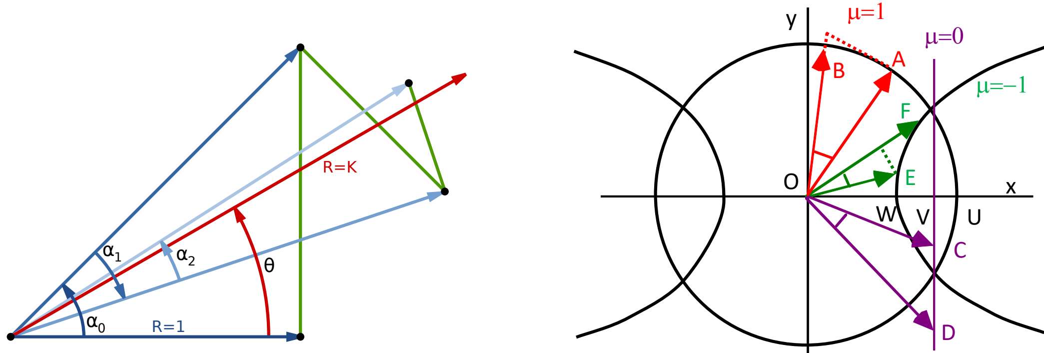
Input		Output	
Address (hex)	Value (dec)	Code (hex)	Value (dec)
0	0	40	1
1	0.0625	44	1.0625
2	0.125	49	1.1406
3	0.1875	4D	1.2031
4	0.25	52	1.2812
5	0.3125	57	1.3594
6	0.375	5D	1.4531
7	0.4375	63	1.5469
8	0.5	6A	1.6562
9	0.5625	70	1.75
A	0.625	78	1.875
B	0.6875	7F	1.9844
C	0.75	87	2.1094
D	0.8125	90	2.25
E	0.875	9A	2.4062
F	0.9375	A3	2.5469

Table 3.2: Lookup table input address mapping to reproduce $f(x) = e^x$ using as x the successive 4 bits of a sfixed(3 downto -12) type, whose possible values are reported in the second column. Note that no sign encoding is present since a purely fractional part of the data is considered.

3.5.2 CORDIC DIVIDER

The COordinate Rotation DIgital Computer (CORDIC) is a simple and efficient algorithm to calculate trigonometric, hyperbolic and even linear functions. Originally developed by Jack Volder in 1959, CORDIC is particularly suited for hardware implementations as it primarily uses shift-add operations instead of multiplications. The CORDIC algorithm works in either rotation mode or vectoring mode and can be used to compute a wide range of functions including sines, cosines, exponentials, logarithms, multiplications, and, interestingly, also divisions. To understand its mathematical background, it is sufficient to consider that the standard CORDIC algorithm employs iterative approximations of vector rotations in a plane to achieve the desired angle. The angles used in each iterative step are given by $\alpha_k = \arctan(2^{-k})$, where k is the iteration index. The sequence of these angles α_k ensures convergence towards the desired end angle. A graphical depiction of this process is presented in Fig. 3.10-(a). Additionally, the process can

be extended to so-called "linear" rotations and "hyperbolic" rotations, which are reported as well in Fig. 3.10-(b). In the former modality, in particular, the algorithm scales one of the vector components by a constant factor, as exemplified by the purple vectors, whereas in the latter one the predefined angles correspond to the hyperbolic tangent of fractional power of two. However, repeated iterations for certain steps (commonly labeled as "extra passes") are often needed to ensure convergence.



(a) Example of circular pseudo-rotation. The desired angle θ is obtained by rotating the original vector (of unit magnitude) by the angles α_0, α_1, etc. in sequence. Note how the direction of rotations is adjusted so to converge to θ. After all the rotations have been applied, the resulting vector has amplitude $K = \prod_{i=0}^{n} \sqrt{1+2^{-2i}}$, where n is the number of iterations of the algorithm.

(b) Generalization to linear ($\mu = 0$) and hyperbolic ($\mu = -1$) rotations. The starting and ending point of the different rotations applied to the red, purple and green vector are identified by $\{A, C, E\}$ and $\{B, D, E\}$, respectively. The ending point actually given by pseudo-rotations, which for $\mu = \{1, -1\}$ modifies the expected rotated vector magnitude, is plotted with a discontinuous line. Note how for $\mu = -1$, the CORDIC amplitude is reduced w.r.t its purely-rotational counterpart.

Figure 3.10: CORDIC pseudo-rotation mechanism.

Iterative equations of generalized CORDIC (see [1]) are given by:

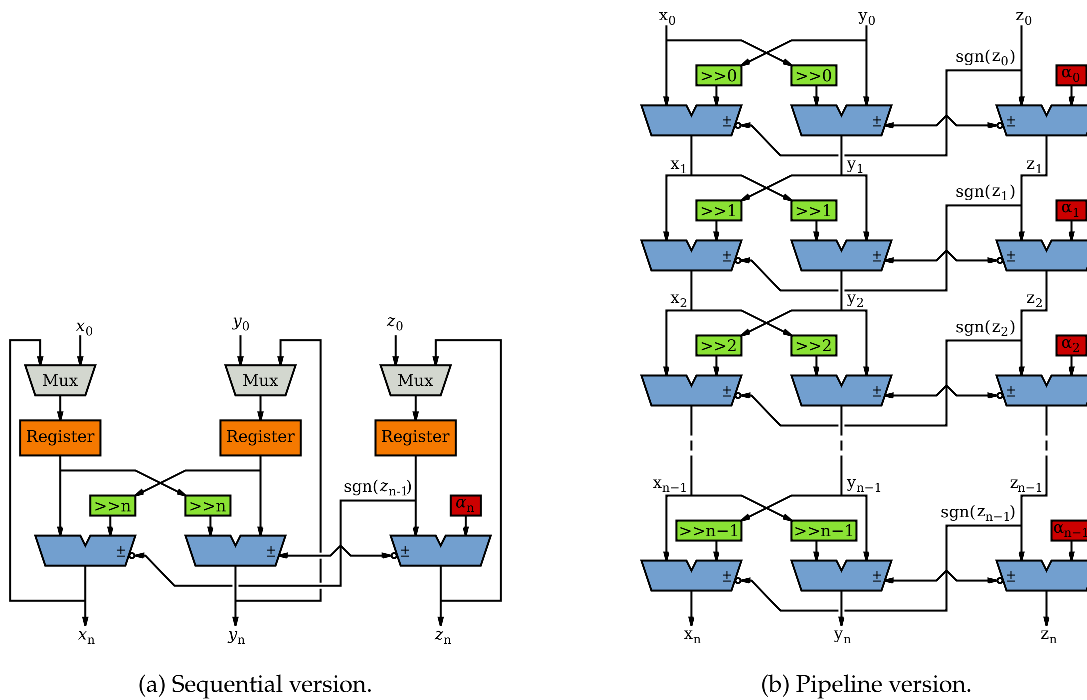
$$\begin{cases} x_{k+1} = x_k - y_k \mu d_k 2^{-k} \\ y_{k+1} = x_k d_k 2^{-k} + y_k \\ z_{k+1} = z_k - d_k \alpha_k \end{cases}, \text{ where } (\mu, \alpha_k) = \begin{cases} \left(1, \arctan(2^{-k})\right) \text{ , for circular mode} \\ \left(0, 2^{-k}\right) \text{ , for linear mode} \\ \left(-1, arctanh(2^{-k})\right) \text{ , for hyperbolic m.} \end{cases} \tag{3.7}$$

For division, operation in the vectoring mode is required, where the goal is to make the starting vector align with the x-axis through successive rotations. In particular, given two numbers x and y, the objective is to compute $z = \frac{y}{x}$. The CORDIC algorithm initializes these numbers as vector components and applies a sequence of predefined rotational angles to nullify the y-component, ensuring that the x-component converges to the vector's magnitude, and the rotation angle converges to the arctangent of $\frac{y}{x}$. Using CORDIC for division involves several steps:

- Initialize the vectors with the given values: $x_0 = x$, $y_0 = y$, and $z_0 = 0$.
- Then, for each iteration k, we compute the rotational direction $d_k = \text{sign}(y_k)$.
- Subsequently, we update the vector components as per Eq. 3.7, with $\mu = 0$ and $\alpha_k = 2^{-k}$.

- After n iterations, the x-component approximates the magnitude, and the accumulated angle z_n approximates $\arctan\left(\frac{y}{x}\right)$. For division, the result is given by $\frac{y}{x} \approx \sum_{i=0}^{n-1} d_i 2^{-i}$.

Finally, the convergence of the CORDIC algorithm hinges on the iterative reduction of the y-component. Each rotation decreases the magnitude of the y-component by a factor related to the powers of two, ensuring that, after sufficient iterations, the y-component becomes negligible. The number of iterations n determines the precision of the result. Convergence is rapid and depends on the desired accuracy, with more iterations yielding greater precision. The only limitation concerns the input data range, which should be such that $\frac{y}{x} \leq 1$. Since, however, the division to perform yields by definition a quantity less than unity as a result, this is not problematic at all. Let's finally note that for circular and hyperbolic rotation a scaling factor needs to be applied to the final results due to the original vector being also stretched out at each rotation. This is not the case for linear rotations, which do not need the final result to be rescaled.



(a) Sequential version.

(b) Pipeline version.

Figure 3.11: RTL schemes of CORDIC processor. Registers between stages are omitted in (b).

Two hardware implementation of CORDIC exist, both reported in Fig. 3.7. The iterative version makes use of a read-only memory (ROM) storing the rotation angles α_k, three muxes used to select the data on which to work on, and three externally-selectable adders/substractors blocks. What is missing are the two *barrel shifters*, used to produce a divison by a generic power of two, and of course the registers put in between to store the result of elaboration at a given iteration k. Finally a control unit is needed to manage iteration counting and angle sign flag generation. The alternative pipeline version uses instead fixed shift registers and physical repetition of all the other resources but the muxes.

3.6 Pipeline and sequential execution

The SNN entity as described till this point is natively suited for pipeline operation, due to the presence of multiple registers within each module. Indeed, coordination between the different blocks of a neuron is obtained as follows:

1. The input spikes are always fed to the adder, no matter the state of the `IN_VALID` signal. If power consumption results critical in the final design, gating the adder inputs to this signal may help reducing energy consumption.

2. The `IN_VALID` signal needs to be propagated to the MAC so to enable accumulation. Therefore, the `STATE_UPDATE` signal of Code 3.1.2 follows the same trend of the former, just delayed of n clock cycles, where n refers to the number of stages of the binary adder tree.

3. After membrane potential has been correctly set, comparison to the threshold is done (again, with no check on the readiness of the value of $U[t]$) and after a T_{clk} the result is available, and the `OUT_VALID` flag should go high. Indeed, this signal is a version of `STATE_UPDATE` forward-shifted in time by `NEUR_ORDER`+1 samples.

4. Finally, if $Y[t]$ = '1', the neuron fired due to its membrane potential overcoming the fixed threshold and so within the MAC a substraction of the threshold needs take place. Practically, this means the `THR_MASK` signal, which is normally at '1', now settles at a low logic value.

Additionally, handling `RESET` simply means propagating it to the MAC and output spike register[15]. Assuming all this operations are handled by a control unit at layer level, this will simply consist of a series of shift registers involving the aforementioned signals.

For sequential execution, the CU should instead follow the state diagram of Fig. 3.12. The same sequence of operations is carried out, with the only difference lying in the fact that this time if new data arrives before the completion of the previous execution, the unfulfilled calculations are discarded and new data processing is started over. A different strategy may have been adopted, namely that of ignoring new data until processing of a neuron module is finished.
The fact that the "refresh rate" of every neuron input and output happens at a fraction of the clock frequency implies by all means that under this conditons the architecture can be classified as *multi-cycle*. This translates to the fact that timing constraints for input and output nodes of the network can be relaxed, for instance by imposing a `set_multicycle_path` directive from one layer's outputs to the inputs of the following one in the constraint (`.xdc`) file of the entity. This can be particularly beneficial during physical mapping on hardware resources where obtaining a positive slack time for each and every signal path may be critical. By telling the optimizer that some paths need less attention than others, more effort can be put where timing was not likely to be closed in the first place.

[15]Being the output clocked, this can be reset independently of the value of the membrane potential (if needed for some reason), since a FDRE primitive will be physically used for storing that value.

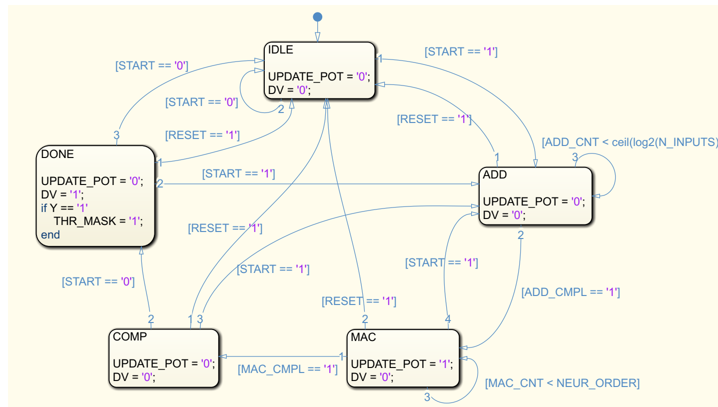


Figure 3.12: State diagram of Moore-type control unit for a layer of neurons in case sequential SNN execution is desired.

The proposed control unit realizes a Moore finite state machine (FSM), however a Mealy-type one could also have been used for the scope. When considering which of the two models to adopt, it is worth observing that from a discrete logic and HDL perspective:

- Mealy machines generally have fewer states, since they change their output based on their current input and present state, rather than just on the present state. However, fewer states don't ensure an easier implementation from coding perspective.
- Moore machines may be safer to use, because they change output states at each clock edge[16], however Mealy machines are faster, as, even tough synchronous, their output is directly dependent on the inputs, and lacks the intrinsic latency typical of Moore FSMs.

Knowing all of this, Moore model has been taken into consideration to ensure FSM predictability. Nonetheless, an alternative Mealy FSM is proposed in Fig. 3.13. Its equivalence w.r.t. the other one has been checked with the help of *Simulink Stateflow* tool.

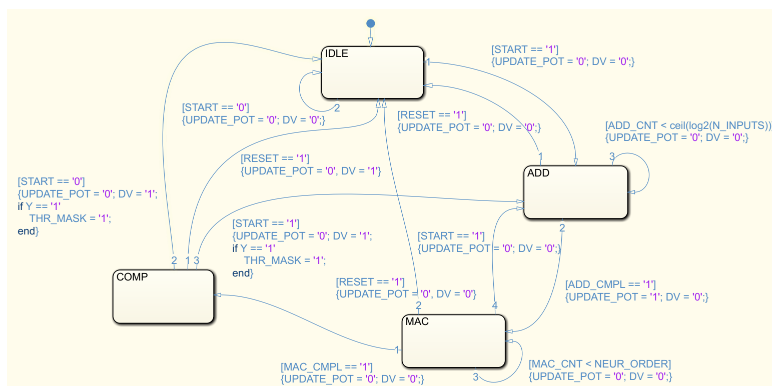


Figure 3.13: State diagram of Mealy-type control unit. Note how fewer states are present, even though the same signals timing and output spike latency are mantained.

[16] under the assumption a clocked process is used for present and next state assignments, which is the case.

In case a sequential execution is wanted, hoping to reduce resource utilization and relax some timing constraints when the update frequency of the inputs is a submultiple of the clock frequency, another implementation of the adder should be preferred. In fact, once the max arrival rate of input data is known, it is sufficient to ensure that every neuron terminates the elaboration in that time. So, knowing that one clock cycle is spent for clocking the output comparator and that another one or two clock cycles are spent within the MAC (depending whether the first order or the second order model is wanted), the timing budget for the adder is obtained. By simply splitting the sum of the coefficients between m of them at a time, were m is the total number of input divided by the number s of necessary clock cycles to complete the elaboration, only a fraction of the resources is needed w.r.t. the binary tree structure[17]. A block scheme of such a structure is reported in Fig. 3.14. A similar strategy may also be applied for convolutional layer, if employed, by only including a fraction of the masking blocks for the coefficients of a given kernel and allowing different selection windows to pass sequentially.

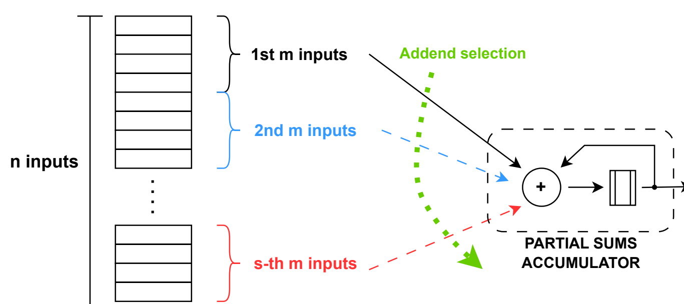


Figure 3.14: Sequential adder for neuron synaptic weights.

3.7 Verification

In this section several snapshots from the Vivado simulator will be included, with the aim of showing the simulated behaviour of the most relevant blocks discussed within this chapter.

At first, the evolution of the output spike and membrane potential of the first and second order versions of a pipeline-operated neuron are shown in Fig. 3.15-3.16. The neuron is stimulated with a random sequence of 16 spikes over 10 timesteps (`SPIKE_IN` signal) and it is subsequently compared with its golden reference *Matlab* model in Fig. 3.17-(a) and 3.17-(b). With the employed quantization, no visible difference emerges with respect to their theoretical counterparts. Here, the quantization of parameters was done considering just 3 bits for the integer part[18] (included the sign bit) and 14 bits for the fractional one. However, a visible difference in Fig. 3.17-(a) can only be observed when just 4 fractional bits are used, as highlighted by the red curve. If the residual quantization error can be tolerated, an 8-bit encoding may be applied to the neuron coefficients so to save hardware resources.

[17]Which, in the end, requires a number of adders equal to the number of inputs minus one: $n_{add} = \sum_{i=0}^{N-1} 2^i = N - 1$.
[18]In fact, the maximum registered value of $U[t]$ for the given stimulus does not exceed 3 as integer part, as evident in Fig. 3.17-(a).

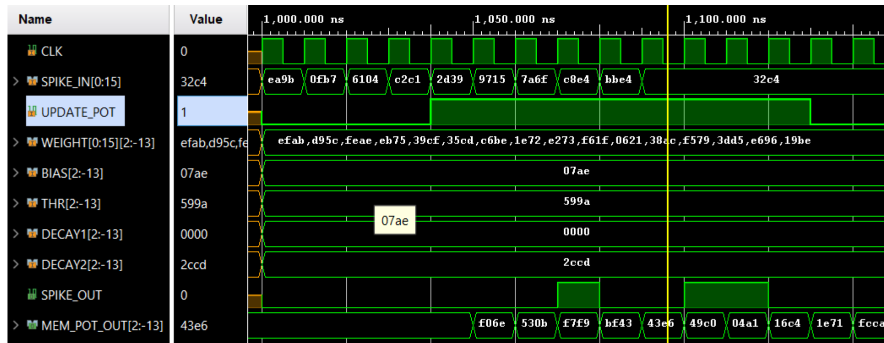


Figure 3.15: Testing of 1st order pipeline neuron.

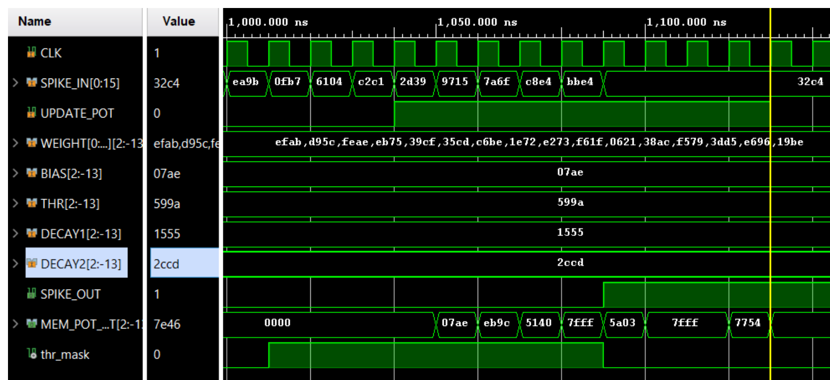
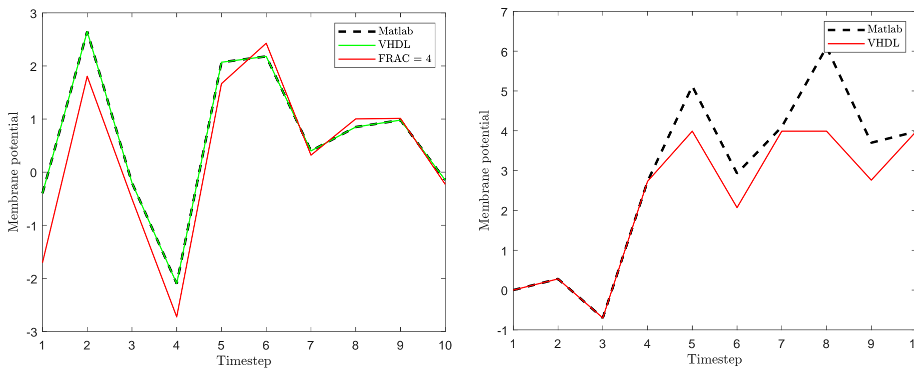


Figure 3.16: Testing of 2nd order pipeline neuron.



(a) 1st order neuron.

(b) 2nd order neuron.

Figure 3.17: Comparison between the evolution of $U[t]$ obtained from RTL simulation and from golden reference model in *Matlab*.

Note, instead, how the second order model causes $U[t]$ to saturate, due the extra accumulation operated by the `d1` time constant, which gets however clamped to the maximum representable as as by default setting of the `overflow_style` flag to "`fixed_saturate`" within the

VHDL `resize()` function[19] used at lines 37-40 of Code 3.3 to assign to the membrane potential value. Here an additional bit for the integer part should have therefore been employed.

A sequentially-operated neuron is then reported in Fig. 3.18. Here all the relevant quantities have been encoded with *real settings* matching the data definition, i.e. `sfixed(5 downto -10)`, to provide an immediate sense to the monitored values. Moreover, $U[t]$ has been plotted in "analog" style so to highlight the exponential decay and the polarity inversion that happens when the threshold is exceeded. Clearly, the values in between two actual updates of $U[t]$, happening the instant before `VALID_OUT` goes high, are only a linear interpolation useful to track visually the evolution of the internal variable. The state of the CU is also shown in the bottom part of Fig. 3.18. Since the simulated neuron only has 2 inputs, as `IN_VALID` goes up the current state remains to ADD for one only T_{clk}, then transitions to MAC, staying here just one instant as the neuron is of 1st order type, then to COMP and finally to DONE, before getting back to IDLE. Note how the `UPDATE_POT` flag goes high when the CU is in MAC state and how the `THR_MASK` signal goes down when the need of U_{thr} substraction from $U[t]$ arises.

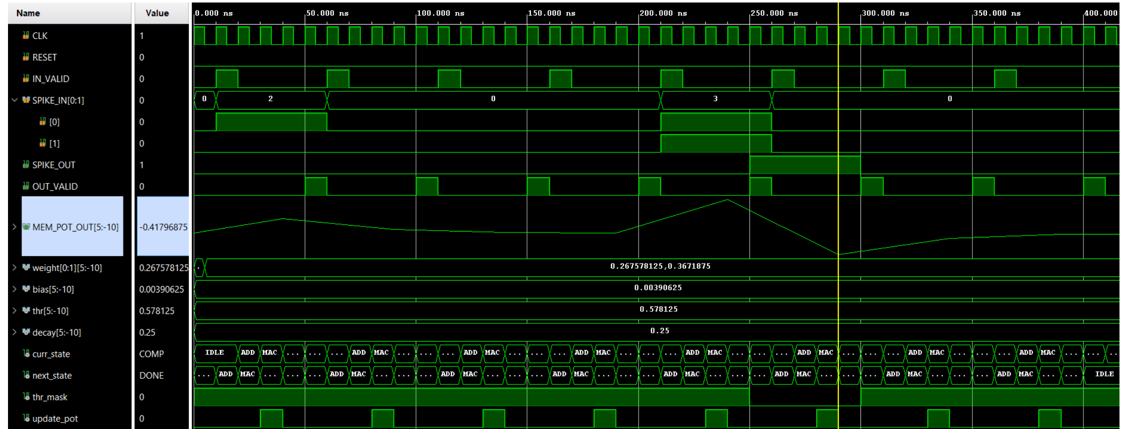


Figure 3.18: Scope view of the evolution of a sequential neuron with 2 inputs. Here T_{clk} = 10 ns.

An example of inference for a sequential network is then reported in Fig. 3.19. The output spikes correspond to the prediction obtained by using *Matlab* function of Code 2.4. The latency of the network, composed of 1st order neurons arranged according to the scheme [16, 40, 32, 16], is given by $\Delta t = \sum_i t_{layer}(i) = \sum_{i=1}^{n_{layers}-1} \left[log_2\left(n_{inputs}(i)\right) + 3\right] - 1 = 23\ T_{clk}$. The processing time of each layer can also be seen as the corresponding output spikes (plus their respective validation signal) is reported ordered from top to bottom of the scope.

Finally, exponentiation and division units used in *softmax* are verified in Fig. 3.20-3.21.

[19]Further clarifications on the handling of fixed point types in VHDL 2008 can be obtained by consulting [26] and [5].

Figure 3.19: Simulation of inference for a sequential SNN having [16, 40, 32, 16] structure.

Figure 3.20: Result of exponentiation realized as explained in Sec. 3.5.1. Intermediate results are also shown, together with the LUTs output mapping reported in Tab. 3.1-3.2.

Figure 3.21: Simulation of pipeline CORDIC divider. Here T_{clk} = 100 ns and the type of handled data is *sfixed(3 downto -12)*.

Since the CORDIC divider has been implemented in pipeline fashion, it is possible to have a glimpse of all the intermediate results, which are stored in the `Y` and `Z` signal arrays, as `X` is not involved for division. Specifically, it can be observed how `Y` tends to zero, as the hex values after each iteration stick closer and closer to ± LSB (`0001` and `ffff`, respectively), while `Z` approchaes the actual value of the quotient.

4

Implementation and Benchmarking

After having discussed in detail the fundamental blocks composing a general SNN architecture in chapter 3, it is now time to focus on the actual implementation of the circuit on a programmable logic device, namely the AMD KCU1500 hardware evaluation board, and to its functional testing on common machine learning tasks. In order to do so, *Xilinx Vivado* has been used, sticking to the following key design steps:

- **Design entry**, the initial step where the digital design is created using either schematic capture or HDL coding. The source files are then added to the project. *Vivado* provides extensive code editors and graphical interfaces to facilitate this process.
- **Synthesis**, in which the HDL code is translated to a netlist, i.e. a gate-level representation of the design abstracted from the high-level functional description. *Vivado*'s synthesis tool optimizes this netlist for area, speed, and power based on specified constraints.
- **Implementation**: the synthesized netlist undergoes implementation, which includes three main phases: translation, mapping, and place-and-route. Translation converts the synthesized netlist into a format suitable for further processing. Mapping assigns the logic to specific components within the FPGA, and place-and-route determines the physical layout by positioning elements and routing the interconnections between them.
- **Bitstream Generation**: after successful implementation, a bitstream is produced. The bitstream is a binary file that can be loaded onto an FPGA to configure it with the desired design. This file contains all the necessary information to control the FPGA's logic blocks and interconnects.
- **Verification**, to be carried out throughout the design realization process. *Vivado* provides simulation tools for functional verification at the HDL level. Post-implementation, timing and power analysis, as well as hardware debugging through the integrated logic analyzer (ILA), ensure the design meets all requirements.

4.1 TOP ENTITY

In order to communicate with the XCKU115 model FPGA on which the network is implemented, a *Xilinx direct memory access* (XDMA) IP core, whose symbol from *IP integrator* catalog is reported in Fig. 4.1, was included inside the TOP entity. In this way, it was possible to exploit *PCI express* ports of KCU1500 to send the input spikes from the host machine to the card and collect the output spikes moving in the opposite direction. In particular, this is done by sending 256 bit long packets according to *AXI stream* protocol, whose aspects are further discussed in Sec. 4.1.1. During testing only a fraction of this payload has been used, all the remaining bits being set to zero. A counter has then been used to keep track of data transmission from host to card and vice versa, properly setting the various flag signals for the AXI stream protocol.

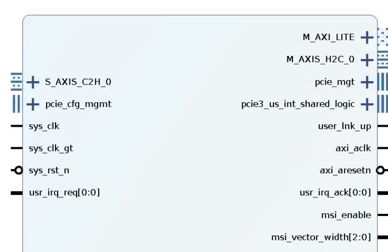


Figure 4.1: XDMA IP core handled thorugh AXI protocol.

A simplified version of the code for the TOP entity is reported in Code 4.1. An `in_data_valid` bit starts the net execution by feeding the input spikes to the first layer binary adder tree and then, after a given number of `user_clk` periods, the validity of the output is retrieved as the `out_data_valid` is enabled. The `sys_clk` signal dictating the board operation is different from the `user_clk` regulating the execution of network inference, whose frequency is 2.5 times greater owing to an internal PLL setting. Independently of the specific design to be implemented, resource occupation of the XDMA core is in any case not significantly impacting the design when used only for online validation of the network[1], as data just pass through the FPGA I/Os.

```
1  entity SNN is
2      port(PCI_EXP_RXP, PCI_EXP_RXN : in  std_logic_vector(7 downto 0);
3           SYS_CLK_P, SYS_CLK_N     : in  std_logic;
4           SYS_RST_N                : in  std_logic;
5           PCI_EXP_TXP, PCI_EXP_TXN : out std_logic_vector(7 downto 0));
6  end SNN;
7
8  architecture TOP of SNN is
9      -- signals declaration
10 begin
11
12     xdma_i : xdma_0 -- Instatiation XDMA core component declared in SNN_PACK.vhd
13     port map( -- ...
14             );
```

[1]If, on the other hand, the XDMA core is used as an hardware accelerator, then a considerable amount of resources should be dedicated to it.

```
15      m_axis_h2c_tready_0 <= '1';
16      spike_in <= m_axis_h2c_tdata_0(200-1 downto 0);
17
18      NETWORK_INST : entity WORK.NETWORK
19        port map (CLK => user_clk, RESET => user_resetn,
20                  SPIKE_IN => spike_in, IN_VALID => snn_valid,
21                  SPIKE_OUT_VEC => spike_out, OUT_VALID => out_valid);
22
23      snn_valid <= m_axis_h2c_tvalid_0;
24      s_axis_c2h_tkeep_0 <= (others=>'1') when  snn_valid = '1' else (others=>'0');
25      s_axis_c2h_tdata_0 <= x"00000000_000000" & spike_out;
26      s_axis_c2h_tvalid_0 <= out_valid;
27      new_sample <= snn_valid and s_axis_c2h_tready_0;
28      s_axis_c2h_tlast_0 <= '1' when cnt_last = unsigned(packet_lenght)-1 else '0';
29
30  -- ... Process for packet counter and secondary components instantiation (buffers, ect.)
31
32  end TOP;
```

Code 4.1: VHDL code of the TOP entity.

4.1.1 *AXI4 Stream* COMMUNICATION

The AXI4-Stream protocol is part of the Advanced eXtensible Interface (AXI) family of protocols developed by ARM, designed specifically for high-speed data transmission. Introduced in the 2010s as part of the AMBA 4 specification, it addresses the need for a highly efficient way to handle continuous data flows without the overhead of addressing, making it particularly suitable for applications such as video transmission and data acquisition systems. This protocol facilitates the unidirectional flow of data from a single source to a single destination, ensuring high throughput and low latency. It operates efficiently without addressing overhead and supports data streaming with widths ranging from 8 to 1024 bits. To understand the functionality of this protocol, the main signals used to control the data flow and their respective functions are summarized in Table 4.1.

Signal	Direction	Description
TDATA	Source to Destination	The primary data signal carrying the payload from the source to the destination.
TKEEP	Source to Destination	Byte qualifier signal indicating valid bytes within the data stream.
TSTRB	Source to Destination	Write strobe signal for the corresponding byte in TDATA; used for marking bytes that are part of the transfer.
TLAST	Source to Destination	Indicates the last data word in a stream packet.
TREADY	Destination to Source	Handshake signal asserting that the destination is ready to receive data.
TVALID	Source to Destination	Handshake signal asserting that the source is sending valid data on the TDATA bus.
TID	Source to Destination	Identifies the data stream, useful for maintaining transfer order for interleaved streams.
TDEST	Source to Destination	Specifies the destination for the data stream packet, allowing dynamic packet routing.
TUSER	Source to Destination	Optional user-defined sideband information accompanying the stream data.

Table 4.1: Description of AXI4-Stream signals.

The AXI4-Stream protocol operates on a handshake mechanism where a data transfer occurs only when both TVALID and TREADY are asserted, ensuring valid data is present and can be

accepted. The absence of addressing overhead simplifies the protocol, making it optimal for continuous data streams. The TKEEP and TSTRB signals provide additional flexibility by indicating which bytes within the data are valid, which is particularly useful for handling variable-length data packets. Moreover, the TLAST signal is essential for delineating packet boundaries within a stream, facilitating proper segmentation and reassembly. An example of data trasmission for packets featuring 16 channels for a total of 25 timesteps is reported in Fig. 4.2.

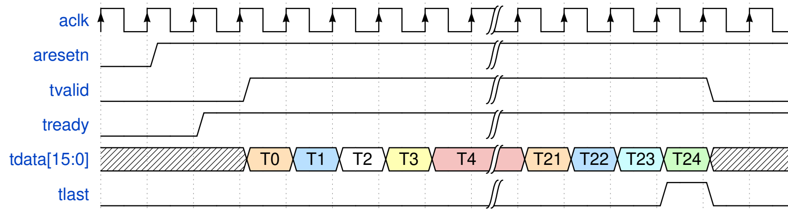


Figure 4.2: AXI4 Stream signals timing diagram for the consecutive processing of 25 timesteps of a varying 16-bit wide input signal.

4.2 Testing on Spiking Tactile MNIST

The Spiking Tactile MNIST (ST-MNIST) is a novel dataset introduced in [31] and designed to capture the tactile information associated with natural writing. It consists of handwritten digits from 0 to 9, which were generated by 23 human subjects using a 100-taxel biomimetic event-based tactile sensor array. This means that the dataset records the motion and pressure dynamics of the participants' handwriting, providing a more detailed and realistic representation of tactile interactions compared to traditional vision-based datasets. A total of 6953 samples from 23 participants ere collected, a glimpse of which is available in Fig. 4.3. Each sample is saved as a `.mat` file which contains an array named spiketrain. This array has a size of $101 \times n$, where n is the total number of events in the sample. The increase and decrease of the pressure exerted on the taxels are denoted by 1 and −1 in the spiketrain, respectively, with 0 for no event. The last row of spiketrain provides the event timestamps (in seconds).

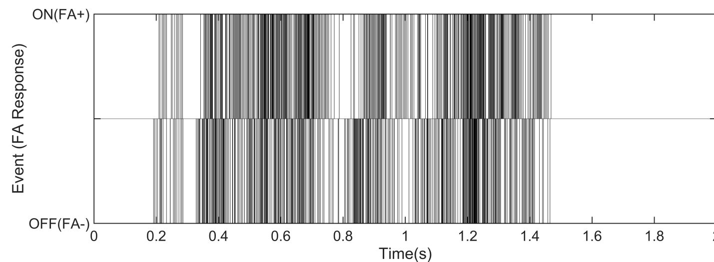


Figure 4.3: Decoded events of the tactile sensor array for a sample corresponding to 8 digit. Taken from [31].

Training of the network was done via *SNNTorch* adapting the *Colab* notebook prepared at [42]. The dataset was split as: 80% for training, 10% for evaluation and 10% for testing. A sim-

ple feedforward network of has been employed for the task, featuring the following pyramidal structure: [200, 128, 64, 32, 10], in which all parameters have been made learnable. Indeed, since the network has been mapped on the FPGA using the first method explained in Sec. 3.3, a total of 936 files for its parameter have been prepared, experimenting different quantization options. The loss metric for gradient descent was set to MSE on the spike count at each output channel, namley the `snntorch.functional.mse_count_loss()` method.

Physically realizing an image classifier involves counting the number of output spikes for each output class within a given number of timesteps, taking then the class with the maximum count as the prediction operated by the net, as shown in Fig. 4.4. Since input data is not strictly binary but ternary, (1, 0, -1), a modification to the input layer was applied, applying a sign inversion block after the 1/0 selection of the coefficients, which, thanks to the `FIXED_PKG` package can be simply realized with the following assignment:

```
y <= resize(-x, INT, -FRAC);
```

In this way, two different sets of input were provided to the net: a 200-bit long vector encoding if a spike was present or not, and a second 200-bit long vector for distinguishing if the incoming pulse is positive or negative.

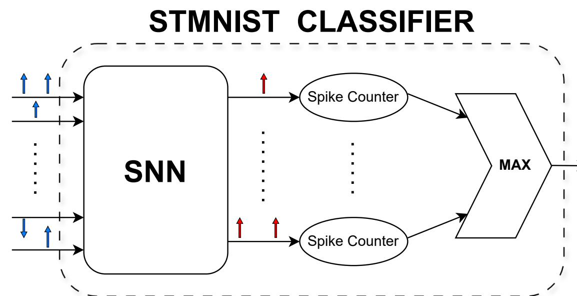


Figure 4.4: Block scheme of the SNN-based classifier for the STMNIST dataset.

Finally, to determine the correct output class the index corresponding to the output neuron with the highest spike count should be returned, and this is done by simply performing the following continuous assignment:

```
DIGIT_OUT <= to_unsigned(integer(maxindex(count)), DIGIT_OUT'length);
```

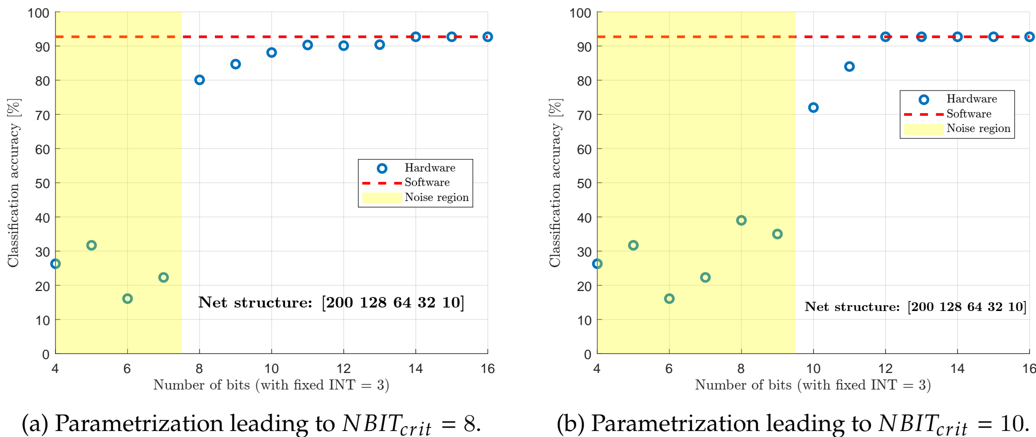
where the circuit generating that maximum index is described by means of the function reported in the following code snippet:

```
1   function maxindex(a: spk_count_log) return natural is
2       variable index : natural;
3       variable foundmax : unsigned(COUNT_SIZE-1 downto 0);
4   begin
5       for i in 0 to a'high loop
6           if a(i) > foundmax then
7               index := i; foundmax := a(i);
8           end if;
```

```
9            end loop;
10           return index;
11       end function;
```

Code 4.2: VHDL Function for determining the index of the element with the greatest value in an array.

In the following, some empirical observations regarding the SNN-based classifier are presented. No general conclusions can be drawn however since a strong training parameters dependency has been observed. By slightly changing the structure of the network or even by settling on modestly different coefficients for the net, some of the points of the following analysis are also changed. Changing test data also affects results in a non-trivial way. For example, Fig. 4.5 shows how two different parametrizations of w, b, t and d affect the final classification accuracy. The rough conclusion one can take is that since no *quantization-aware methodologies* have been considered during training, when quantizing the network parameters by a significant amount it is difficult to predict what is going to happen. It is likely to think that those "steps" between the experimentally found accuracies and the "theoretical" ones in full precision are due to an increasing number of interconnections dramatically changing their values: as the number of "failing" links increases, more and more error is accumulated, disrupting the network functionality.



(a) Parametrization leading to $NBIT_{crit} = 8$.

(b) Parametrization leading to $NBIT_{crit} = 10$.

Figure 4.5: STMNIST digits classification accuracy as function of the parameters resolution. A general descending trend may be identified, however the exact trimming point for network breakdown is strongly dependent on the employed test dataset and parameter configuration.

For what concerns timing closure, with a 12-bits coefficients resolution an upper limit of 80 MHz was reached, whereas with 16-bits the maximum clock frequency could not exceed 50 MHz. Regarding FPGA utilization, instead, in these both extreme cases over 80% LUTs where employed, and a total of 234 DSPs (1 per neuron), corresponding to 4.2% of the totally available slices. However, for sufficiently low f_{clk}, e.g. 10 MHz when using 12-bits parameters, it has been observed how the optimizer of *Vivado* preferred not to use DSPs in favour of generic combinational logic elements realized through LUTs and BRAM. The rationale for this choice is likely to be that DSPs are tought for 'fast' operation, and since other resources are still available, it is advantageous to use less specialized hardware.

4.3 Testing on Spiking Heidelberg Digits

The Heidelberg spiking digits (SHD) is a comprehensive dataset that has been developed for classification purposes. The dataset includes a wide range of audio stimuli that cover various speech patterns and characteristics, providing a diverse set of inputs for classification models. The data is structured in a way that captures the temporal information of the audio signals, allowing for the analysis of spike timing and patterns within the dataset. More specifically, it consists of approximately ten thousands high-quality recordings of spoken digits ranging from zero to nine in English and German language[2], therefore 20 output classes need to be arranged for the SNN. An example of the input stimulus for the network is shown in Fig. 4.6. Both spatial and temporal binning have been applied to the original dataset. For each sample, in fact, SHD dataset provides the activation of 700 channels over an interval of 0.7 seconds, with a time resolution of 1 μs, and has been reduced to 64 channels with an update period of 10 ms. This "pruning" procedure has been adapted from the *Python* script presented in [41].

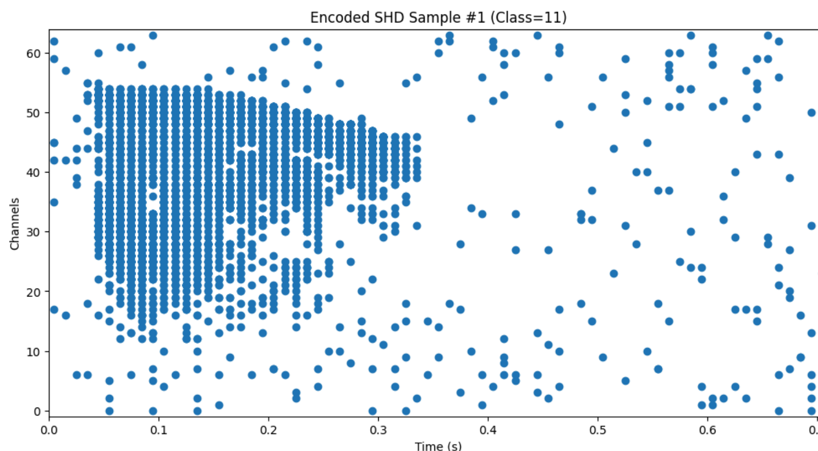


Figure 4.6: SHD input sample. The number of channels has been reduced by about 10 times w.r.t. the raw sensor acquisition. Original samples have been averaged out together, as well, remaining with a total of 70 samples only.

In order to decide the architecture to be used, the scoreboard on proposed nets for SHD classification available at [34] was consulted. Here several candidates propose their architectures and report the obtained accuracy on the dataset. In the end, the classifier was realized following the work presented in [14] and [15]. In the proposed *dilated convolution with learnable spacings* (DCLS) SNN architecture, spike timing is utilized by incorporating delays between layers using 1D convolutions across time. These delays allow for spikes to reach neurons at different times, influencing the spike arrival times and enhancing the network's expressivity. Spiking neurons function as coincidence detectors by responding more strongly to synchronous input spikes, where the spike arrival times coincide, rather than asynchronous spikes, as depicted in Fig. 4.7.

[2]More information on the dataset and they way it was aquired are found in [7].

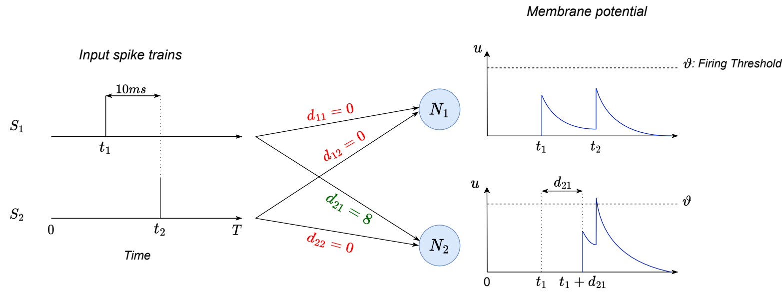


Figure 4.7: Coincidence detection using spiking neurons. Here two neurons with the same positive synaptic weight values are shown. Both spikes from spike train S1 and S2 will reach N2 quasi-simultaneously, resulting in N2 emitting a spike, while N1 has no reaction. Taken from [14].

Network training on SHD dataset was carried out by simply adapting the official DCLS-SNN implementation available at [38]. A different framework from the already introduced ones was used, namely *Spikingjelly*, which is based on *TensorFlow* instead of *Pytorch*. An accuracy of 93% was reached by training a DCLS-SNN composed of two hidden layers of 128 and 32 LIF neurons of the 1st order, respectively, each with a different decay time constant. More precisely, each feedforward layer is implemented using a DCLS module where each synaptic connection is modeled as a 1D temporal convolution with one Gaussian kernel element[3], followed by batch normalization, LIF layer and dropout. The readout layer finally consists of 20 LIF neurons with an infinite threshold[4] and the generic output i at a given time instant t is given by the softmax of the i-th neuron membrane potential at time t. The final output of the model after T time-steps is defined as summation of all the softmax values at each time instant. Cross-entropy was applied on this quantity as a loss function for backpropagation.

The only difference of this new design w.r.t. the previous structure is the addition of a variable delay block in front of every input of each neuron, plus some adjustments of the control unit so to account for the incerased latency due to delay blocks. In fact, an additional T_{max} clock cycles are needed for each layer processing, where T_{max} is the longest delay to be applied in the entire layer, i.e. the size of the longest shift register. Moreover, the softmax unit of Sec. 3.5 is now put after the output neurons and an actual accumulator (not just a binary counter) needs to be placed as well. The bit length of the accumulator register is increased of $\lceil log_2 (n_{timesteps}) \rceil$ w.r.t. that of the softmax-generated probability. Finally, after that, the index corresponding to the maximum value is again taken the output prediction. A graphical description of this new entity is shown in Fig. 4.8.

[3]Please refer to [14] for an in-depth explanation of the training procedure for neuron connections delays.
[4]Meaning that membrane potential is only accumulated, and no output spiking mechanism is included in the final set of LIF entities.

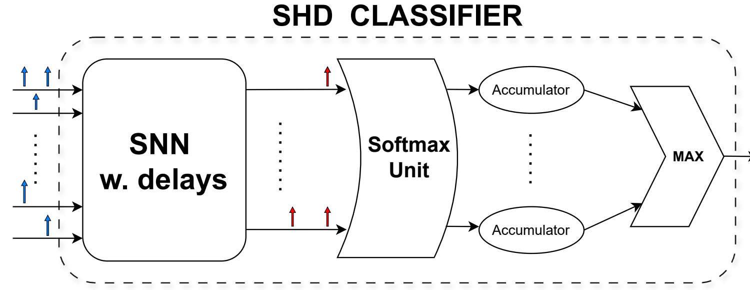


Figure 4.8: Block scheme of the SNN-based classifier for the SHD dataset.

If for the pipeline control unit a "classical" shift register implementation using flip flops could be accepted, given the undoubtedly overall low utilization of this components (a bunch of them for each layer), for introducing the delays in [15] the FFs utilization could be much greater, i.e. proportional to the number of neurons, to the number of inputs of each layer and also to average delay to be applied to the input spikes. A hardware description favouring the use of BRAM instead of FFs, without the use of explicit directives for *Vivado* compiler, is proposed in Code 4.3, following the approach of [17]. A given number of this optimized shift registers (in the same quantity as the inputs of the layer) are grouped together, each with its own custom length defined by the generic parameter `SR_DEPTH`, to form the `delay_block` entity, as shown by Fig. 4.9.

```
1  process(CLK)
2  begin
3    if CLK'event and CLK='1' then
4      if RES = '1' then
5        res_counter <= 0; DELAYED_SPIKE <= '0';
6      else
7        r <= r(r'high-1 downto r'low) & SPIKE;
8        if res_counter = SR_DEPTH-1 then
9          DELAYED_SPIKE <= r(r'high);
10       else
11         res_counter <= res_counter + 1; DELAYED_SPIKE <= '0';
12       end if;
13     end if;
14   end if;
15 end process;
16
17 end architecture BHV;
```

Code 4.3: VHDL process realizing a generic-length shift register minimizing the use of FFs in favour of BRAM.

The aim of batch normalization is that of ease training: by keeping track of the mean and variance of consecutive weights and biases adjustments along loss function gradient descent, it is not necessary anymore to use small learning rates or fancy learning rate scheduling techniques. Batch normalization has not been performed in hardware, but rather at coefficient quantization level. Mathematically, in fact, the contribution of a weight-normalization layer is the following:

$$[WX(t-d)+b]\cdot\mu+\sigma = \underline{W\mu}\cdot X(t-d)+\left(\underline{b\mu+\sigma}\right) = W'X(t-d)+b' \tag{4.1}$$

Therefore, after training has been completed it is simply necessary to rescale the linear layer coefficients, as clarified in [16]. Droput layers are also included in DCLS Python model, but are not used during inference phase and have therefore been omitted in HDL. In fact, dropout is a regularization technique used during the training phase to prevent overfitting, and consists in randomly setting a fraction of the input units to zero at each update step, effectively creating a different "thinned" network each time, which helps in making the model more robust and generalizable. However, during inference (when the model is making predictions on new data), dropout is typically turned off. Instead of randomly dropping units, the weights of the network are scaled to account for the dropped units during training. This ensures that the entire model is used for making predictions, and the outputs are more stable.

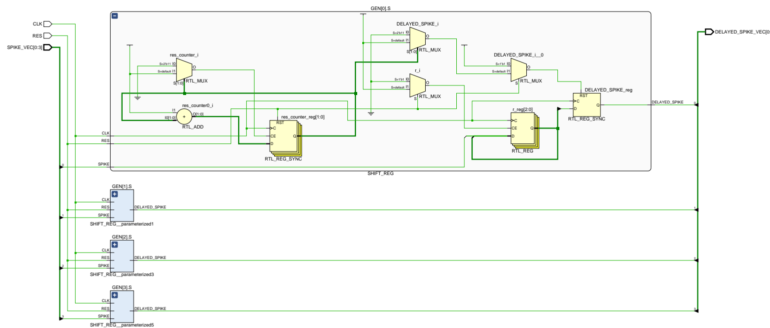


Figure 4.9: Vivado schematic view of *delay_block* module.

Finally, some tests have been assessed aiming at determining the impact of the total number of timesteps on network accuracy. It has nonetheless been oberserved that no significance influence on the final accuracy is found by reducing the dataset temporal binning, the only difference lying in the larger number of training epochs necessary to reach a given performance level[5].

4.4 Runtime coefficients setting

As a final improvement to the processing system, the possibility of changing the values of the network defining parameters without needing to recompile the design was taken into consideration. The aim of this was to allow the system to become more versatile and potentially serve purposes different from the originally designated ones, even once fixed into place for operation. An *AXI-Lite* memory IP was manually instantiated within *Vivado* to ensure this functionality, connecting it (configured in slave operation) to the DMA IP, and porting out of the entity its registers, so to feed them to the network for its parameters definition. The AXI-Lite protocol was introduced to provide a simplified version of the full AXI protocol for low-throughput control register access scenarios. AXI-Lite reduces the overhead associated with the full AXI protocol by omitting certain features, thereby simplifying its implementation in systems where only basic read/write operations are required. This streamlined protocol is ideal for register access in

[5]An accuracy level of 93% has been considered as the arrival point for network training.

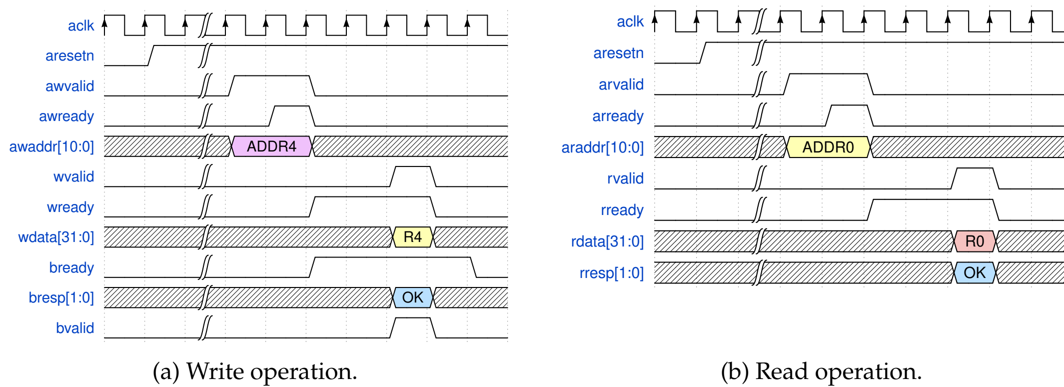
peripheral devices or simple memory-mapped interfaces. To understand the functionality of AXI-Lite, a summary of the main signals used in the protocol and their respective functions in Tab. 4.2. Not every one of these signals will be used in the final circuit, though, and the corresponding component ports will be left in the `open` logical state.

Signal	Direction	Description
AWADDR	Master to Slave	Write address. Specifies the address of the target location for the write operation.
AWPROT	Master to Slave	Write protection type. Indicates the privilege level and security status of the write transaction.
AWVALID	Master to Slave	Write address valid. Indicates that the address and control information are ready to be accepted by the slave.
AWREADY	Slave to Master	Write address ready. Indicates that the slave is ready to accept the address and control information.
WDATA	Master to Slave	Write data. Contains the data to be written to the target location.
WSTRB	Master to Slave	Write strobes. Indicates which byte lanes of the data bus are valid during the write operation.
WVALID	Master to Slave	Write valid. Indicates that the write data is ready to be accepted by the slave.
WREADY	Slave to Master	Write ready. Indicates that the slave can accept the write data.
BRESP	Slave to Master	Write response. Provides feedback on the status of the write transaction.
BVALID	Slave to Master	Write response valid. Indicates that the feedback on the status of the write transaction is available.
BREADY	Master to Slave	Write response ready. Indicates that the master has accepted the write response.
ARADDR	Master to Slave	Read address. Specifies the address of the target location for the read operation.
ARPROT	Master to Slave	Read protection type. Indicates the privilege level and security status of the read transaction.
ARVALID	Master to Slave	Read address valid. Indicates that the address and control information are ready to be accepted by the slave.
ARREADY	Slave to Master	Read address ready. Indicates that the slave is ready to accept the address and control information.
RDATA	Slave to Master	Read data. Contains the data read from the target location.
RRESP	Slave to Master	Read response. Provides feedback on the status of the read transaction.
RVALID	Slave to Master	Read valid. Indicates that the read data and status are available.
RREADY	Master to Slave	Read ready. Indicates that the master can accept the read data and status.

Table 4.2: Description of AXI-Lite signals.

The AXI-Lite protocol operates using a handshake mechanism where each transaction consists of address and data phases for both read and write operations. A transaction begins when the master asserts the `AWVALID` or `ARVALID`, indicating valid address information. The slave responds with `AWREADY` or `ARREADY`, acknowledging readiness to accept this information. For write transactions, the master sends the data along with `WSTRB` signals, and for read transactions, the slave returns the data when it asserts `RVALID`. The simpler design of AXI-Lite reduces resource usage and implementation complexity, making it ideal for low-throughput, low-latency control applications in modern digital systems. Simple timing examples of AXI LITE register reading and wrinting are provided in Fig. 4.10-(b) and 4.10-(a), respectively.

For the ultimate implementation of the circuit, the `FALSE_PATH` constraint needs to be set `from` AXI LITE registers to the `NETWORK` entity. This relaxation condition holds since network coefficients are known to be kept constant during network inference.



(a) Write operation.

(b) Read operation.

Figure 4.10: AXI Lite register transaction timing diagram.

The fact that coefficients can be reprogrammed allows for training the network performing the inference phase in hardware, while the more complex backpropagation phase can be handled by floating point arithmetic on a host machine. With such kind of networks, in which parallelization of execution instances is limited by the inherent temporal dependency of each layer with the previous one, there can still be a substantial benefit in reducing the effort and time spent in calculating all network quantities during the feedforward phase. In this case, however, no value-dependent optimization is possible on the network coefficients, meaning the overall resource utilization for the same design will be somewhat greater, in particular for FFs used to implement the cofficient registers and LUTs/CLBs/LUTRAM for additions operations. Also the multiplications performed by each neuron become more complex operations, since now they are performed between two signals, and not between one signal and one constant scalar, as in the previous case. This implies a different setting of DSPs, when in use, or of additional logic when implemented “discretely”, e.g. at lower clock frequencies. Indeed, this is the price to pay to have reprogrammable coefficients.

5

Case study: drift tubes hits filtering

Owing to their event-driven nature, the application of spiking neural networks to datasets natively expressed as timeseries seems promising. After having discussed "standard" classification tasks reproposing input data as time varying quantities in the previous chapter, it is now high time to discuss a more specific task, for which low latency hardware implementation is needed. Specifically, an application to the elaboration of data coming out of particle detectors placed inside the Large Hadron Collider (LHC) at CERN will be shown. Particle Physics experiments at CERN require low latency hardware for data processing since:

- particle collisions occur at extremely high rates, often millions of times per second (e.g. at 40 MHz) and efficient and prompt capture of interesting events (such as rare particle decays) is essential before readout buffers fill up and new events are overwritten;
- experiments use complex multi-level trigger systems to filter out uninteresting events in real-time and retain only the most relevant data for detailed analysis, therefore low latency is crucial to ensure the triggers operate swiftly and accurately, minimizing the risk of discarding significant events.

5.1 The experiment

The drift tube (DT) chambers used in the Compact Muon Solenoid (CMS) experiment at CERN are a type of gaseous particle detector designed to track the trajectory of charged particles, depicted in Fig. 5.1. They consist of long tubes filled with a gas, typically a mixture of argon and carbon dioxide, which are usually arranged in parallel with each other. When a charged particle passes through the drift tube chamber, it ionizes the gas along its path, creating electrons and positive ions. The electric field within the chamber drifts these electrons towards a wire in the center of the tube, where they are collected. By measuring the time it takes for the electrons to reach the wire, the position of the charged particle along the tube can be precisely determined.

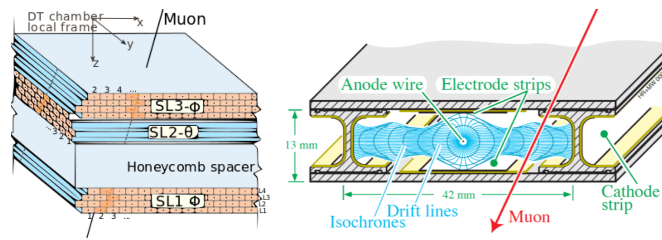


Figure 5.1: Working principle of muon drift tubes. Taken from [24].

DTs are particularly well-suited for detecting muons due to their penetrating power and ability to travel through dense materials. They are used in conjunction with other detectors in the CMS experiment to reconstruct the paths of particles produced in high-energy collisions, such as resistive plate chambers (RPCs) and cathode strip chambers (CSCs). They are arranged in a specific geometry to maximize their efficiency in detecting muons. These chambers are typically organized in multiple layers or "stations" around the collision points where particles are produced. Layers are placed parallel to the beamline, allowing for the measurement of the particle's position in both the transverse and longitudinal directions, as outlined by Fig. 5.2. Finally, two types of chambers exist: Phi (Φ), measuring the azimuthal angle around the beam axis and Theta (Θ), measuring instead the polar angle w.r.t. it.

There are approximately 250 drift tube chambers distributed throughout the detector. These chambers are placed in different regions of the CMS detector to provide efficient coverage for detecting and tracking charged particles. A "superlayer" in the context of particle detectors, then refers to a group of individual detection layers or modules that are stacked together to increase the coverage and precision of particle tracking. The proposed application for a SNN architecture is as a filter for the event detections from a group of cells within a superlayer. Precisely, a block of 4×4 cells is considered, as it is expected to be sufficiently extended in the transversal direction to "catch" an impinging muon for sufficiently small angles of incidence. This is particularly true for MS2 (muon station) and MS3, in which the trajectory of particles is almost linear, due to the change in direction of the applied magnetic field, whose intensity is of about 3.8 T.

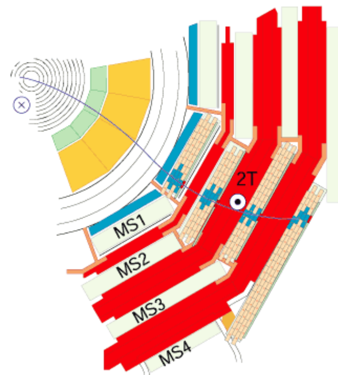


Figure 5.2: CMS cross-sectional view in correspondence of DTs. Note how the trajectory of charged particles generated during collisions causes multiple cells to fire (coloured in blue) within each superlayer.

The DAQ and triggering system is currently under upgrade as the requirements in terms of trigger rate are going to exceed the capabilities of current electronics once the new High Luminosity (HL) LHC will enter into operation[1]. *On-Board Electronics for Drift Tubes* (OBDT) board represents the main part of the electronics that will be deployed in the upgrade of the detector. The OBDT board is built using a *Microsemi* Polarfire MPF300 FPGA. The time digitization of up to 240 channels is performed inside this FPGA through a deserialization method. Each input signal is sampled by a 640 MHz DDR deserializer, and then 0 to 1 transitions are detected in the parallel array of sampled data. The detected transitions are converted to a digital value which includes the coarse count in steps of the LHC bunch crossing (BX), that is 25 ns, and a fine bin size with a least significant bin of 1/32 of a BX. That is, the least significant bin of the TDC (Time-to-Digital Converter) is 0.78125 ns. Further details about the DAQ system are found in [28], however the provided information is sufficient to follow the discussion presented in the following sections.

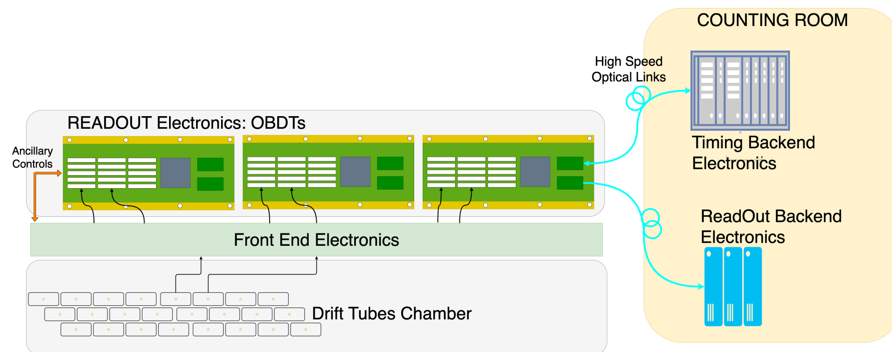
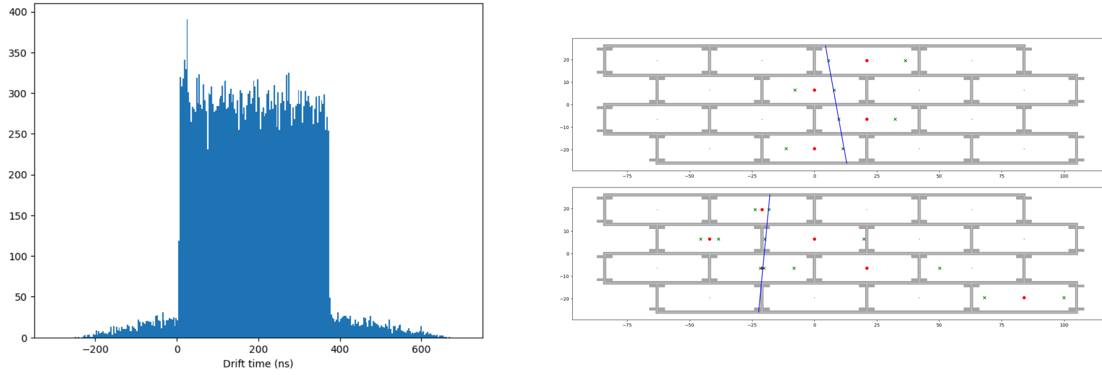


Figure 5.3: DAQ system for muon events detected by drift tubes. Taken from [4].

5.2 Dataset generation

The starting point for approaching the filtering problem with SNN was the work presented by Migliorini et al. in [22], showing a demonstrator for a muon triggering system with a set of fast ReLU-based ANNs. In this case an ANN was devoted to the task of filtering the spurious hits due to noise originating from DTs operating environment. The filtering action was performed on the "coarse" time measurement, i.e. the BX count, and the TDC value of the firing cells was then taken into account to determine the absolute time of passage of a particle by applying the well-established mean-timer technique explained in [12], which exploits geometrical relations associated to the staggering of cells in adjacent layers. Since however different formulas hold for muons arriving from opposite direction w.r.t. the normal of a given cell, another network was used for left/right disambiguation, and an additional one was futher employed as a first preprocessing step with the aim of grouping hits together.

[1]A comprehensive presentation of the upgrades taking place within the CMS detector in relation to HL-LHC are reported in [37].



(a) Histogram of muon drift times in DTs. The simulation was carried out for 10000 events.

(b) Cross-sectional view of cells struck by muons: 4 true hits case and 3 true + 3 noise hits scenario.

Figure 5.4: Simulated muon arrivals: hits time and crossed cells.

The spiking version of the just mentioned filtering network requires a binary input for each drift tube status reading, instead of an integer value indicating the count w.r.t. the last LHC orbit[2] starting instant. For direct ANN-SNN conversion using the coefficients rescaling method of Sec. 2.5 the BX count provided by the OBDTs should be converted to the spike domain using firing rate encoding and then wait for a sufficiently large number of instants so to have the output predictions ready. A smarter approach is that of reducing the BX reading to the last n instants, instead of considering the entire orbit as a reference. It would be even more interesting to directly use the arrival timeslot of muons to encode the BX value within the time window of interest. Stated differently, temporal encoding (with reversed magnitude assignment w.r.t. what explained in Sec. 1.2, meaning the later the spike arrives, the bigger its associated value) can be efficiently applied to input data, and, indeed, this is the approach that has been used. More precisely, a Python code was provided[3] to simulate Poisson arrivals of muons at a given time, and taking into account geometric relations of gas-filled chambers, the expected drift time of excited electrons, i.e. cells firing times, were obtained. A 40 bin suddivision was then applied, taking into account that:

- muon impact was set to happen at a randomly distributed angle between $\pm 10°$ w.r.t. normal axis;
- muon arrival time t_0 was set to the 20th BX, where 1 BX $= \frac{1}{f_{clk}^{LHC}} = 25$ ns
- considering a 1 μs ($=$ 40 BX) time interval, all the cell firing times appearing in the drift time histogram of Fig. 5.4-(a) are taken into account, even for remote events located at the tails of the distribution.

The inefficiencies of DTs due to aging were also simulated, randomly removing some of the hits occurring on the cells along the muons tracks. Then, uniformly distributed noise was finally

[2]In the LHC, a single orbit is completed approximately every 89.3 μs. In fact, the LHC operates at a radio frequency of 40.078 MHz, which means that there are about 3564 bunch crossings per orbit.
[3]A **huge** thanks goes to Dr. Migliorini, Ph.D.

added to the samples. For each hit, a label was assigned to identify it as a true event or as an artifact to be discarded. For a number of true events in between 3 and 4 (occurring on groups of adjacent cells, as ordered according to official CMS naming rules), only the extra noisy hits are discarded as shown in Fig. 5.5, whereas if the true hit count is lower or equal to 2, also the true events need to be discarded as the mean timer requires at least three points to work properly.

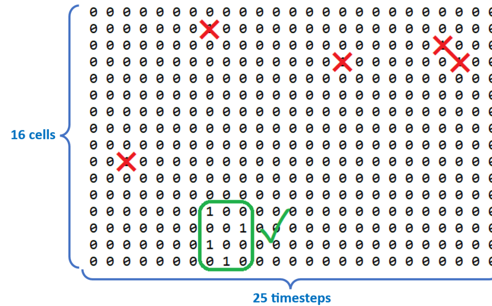


Figure 5.5: Generated hits labeling. Here only 25 timesteps have been simulated for brevity.

5.3 Training challenges

The BX arrival time for each channel for each 40-timesteps long simulation outcome is then converted to a spike train in which a 1 is generated at the proper time, being all the rest of the stream left to zero. In order to account for the events that could occur at the 'boundaries' of the muons drift time range, all samples have been assembled together to form a very long timestream, as graphically shown in Fig. 5.6. The actual stimulus for the SNN is then retrieved by considering the content of a 16-timesteps long window sliding over the entire dataset. The length of the window has been chosen to such value since almost all muon events take less than 400 ns (= 16 BX) to drift the deposited charge inside the chamber towards the central wire. This will be the starting value fore the decay time of LIF neurons, effectively regulating the "temporal memory" of the network.

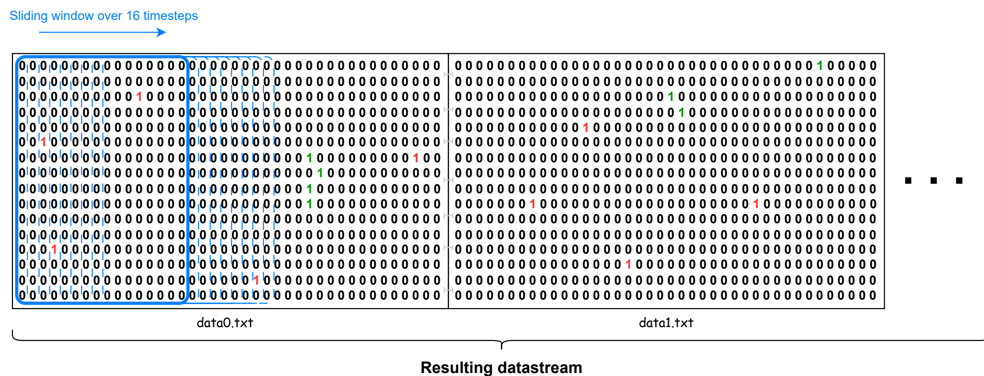


Figure 5.6: Generation of training dataset: the single samples originated from muon arrival simulation are merged together to form a unique datastream. Then, each training sample for the SNN is obtained by sliding a window of 16 timesteps over all the available time instants.

After this kind of data-augmentation procedure, the network has initially been trained with the *Matlab* script of Sec. 2.3 so to exactly match the values of each channel at every time instant. The resulting samples were feeded to the net sequentially, and MSE was employed as loss function for the output value mismatch. A fixed latency of 16 samples was then imposed between the current output value and the desired output value, even tough some tests were also performed decreasing this delay. A glimpse on the training status log in Matlab shell is shown in Fig. 5.7 for a simple [16, 100, 16] feedforward net, together with the overall utilization of the multicore CERN machine on which the program was running. Indeed, the procedure is quite resource intensive (even for a simple dataset) w.r.t. a standard 'static' dataset training of an ANN.

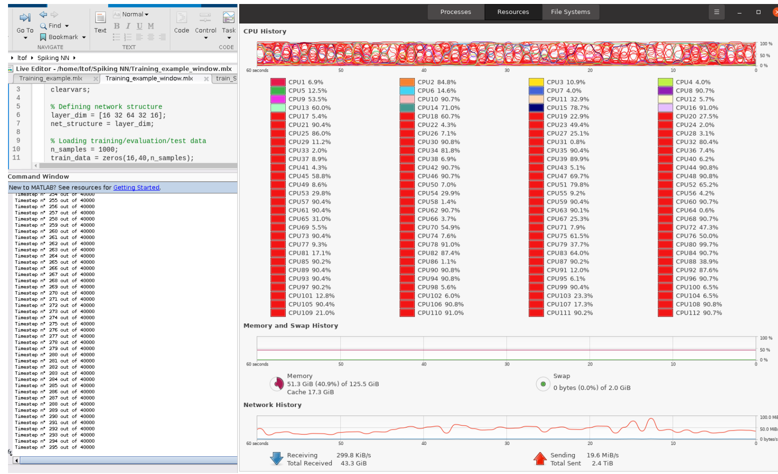


Figure 5.7: Server resource usage during SNN *Matlab*-based training. Thanks to the use of *parallel computing toolbox* almost all cores of the employed machine are involved into calculations.

Actually, to limit the effects of the strong imbalance of the input, the following modification was introduced on the classic mean square error, so to penalize the occurrence of false negatives:

$$\begin{cases} \mathscr{L} = \frac{1}{2}(y-\hat{y})^2 \cdot (1+K\hat{y}) \\ \frac{d\mathscr{L}}{dy} = (y-\hat{y}) \cdot (1+K\hat{y}) \end{cases} \text{, where } K \text{ is the penalty term.} \tag{5.1}$$

Fig. 5.8 reports the trend of train loss, train accuracy and evaluation accuracy for a [16, 40, 32, 16] net when 1000 events are considered, having set $K = 9$, meaning the error associated with false negatives is 10 times grater than that of false positives. Even though the final accuracy is nearly unitary, the actual *F1-score*[4] on the predicted output is very low, since the network has the tendency to "filter out" all input spikes if training is not stopped early enough, as the resulting total error is in any case very small, and it difficult to direct gradient descent toward the absolute minimum. As an alternative, the use of MSE applied to the spiking time difference w.r.t. the expected output has also been tried, however no fundamental change was observed.

[4]The F1 measure for binary classification problems is given by $F1 = 2 \cdot \frac{P \times R}{P+R}$, where $P = \frac{TruePositives}{TruePositives+FalsePositives}$ is the so-called *precision*, and $R = \frac{TruePositives}{TruePositives+FalseNegatives}$ is the *recall*.

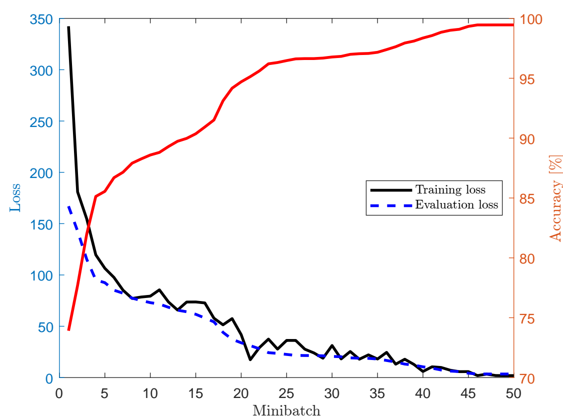


Figure 5.8: Learning curves for SNN filter training with Code 2.7. Note how training and evaluation losses follow the same trend, indicating that overfitting is not occurring until the last minibatches.

5.4 Proposed approach for spurious hits filtering

Given the effectiveness of rate-based output codes methods for SNN training, as presented in chapter 4, the same approach of counting the number of output spikes produced in a given time lapse has also been applied to the muon hit filtering problem, with the aim of solving the problem of total input cancellation. In order to do so, for each sample (16 channels $\times$ 16 instants) the network outputs a positive prediction on one or more channels that have experienced the passage of a muon. The prediction is ready after the last set of input stimuli has passed, i.e. at the 16th time step, as the output state of a channels is determined by comparing the spike count with a given threshold (to be properly set after training).

From the machine-learning point of view, the problem has changed from a multi-class task (as for the examples of chapter 4), to a multi-label classification problem. *SNNTorch* has been used to adjust the previously developed training algorithm following the indications found in [40]. As an additional step, the values to which the spike counters should be compared for each output channels were defined based on the spikes produced on the test dataset: the compromise best separating firing and non-firing cells was chosen. A relatively small net with structure [16, 20, 20, 16] has revealed sufficient for the task, gasping a testing accuracy of about 83%. This value is not good enough to justify the use of such a complex network w.r.t. to other simpler denoising algorithms, however further exploration of other architectures may yield better results. The practical advantage of using such a tiny deep SNN, is that the resulting filter circuit fits inside the available FPGA.

Particular care was taken in the initialization of parameters before starting training. In fact, the still very low input spikign activity does not allow the training to converge to the desired results. The main idea to overcome the sparsity of the input was that of allowing a non-negligible

bias to be applied at each neuron summing node, instead of being initially set to zero[5]. In this way the network is forced to into an "awake" state featuring a mild neuronal activity: some spikes are produced even without external simulation. An example of that is shown in Fig. 5.9, which, in particular, details the evolution of the neurons' membrane potentials. Note how a sudden drop in $U[t]$ indicates that a spike occurred in the interested neuron. If no hits arrive the spike count at the output is too low to overcome the threshold to classify the channel as affected by the impact of a muon. On the other hand, if some of the detectors fire, the evolution of the network is perturbed and may increase or decrease the spiking count of some channels.

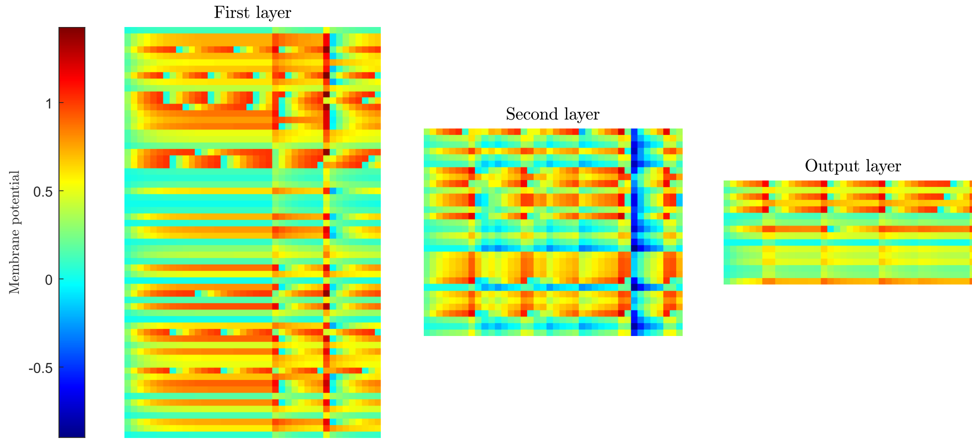


Figure 5.9: Membrane potential evolution for a [16, 64, 32, 16] net with bias accumulation effect.

From hardware perspective, in order to achieve *online* filtering as with a single SNN, multiple SNN-based classifiers are needed to operate in parallel, as depicted in Fig. 5.10. These networks are identical in every aspect, and they are operated with on $T_{BX} = 25$ ns of difference w.r.t. the adjacent one. The output of the filter is finally obtained by choosing the results of the comparison operations for the network that has concluded its execution, i.e. for which `OUT_VALID` = '1' in that instant. This multiplexing action is needed so to ensure no incorrect partial results are send to the output.

The total latency of the network, considering a $f_{clk} = 240$ MHz, is given by:

$$T_{latency} = 16 \cdot T_{BX} + T_{net} + T_{count} + T_{comp} + T_{mux} = (96+22+1+1+1) \cdot T_{clk} = 121 \cdot T_{clk} \approx 20 \cdot T_{BX} \quad (5.2)$$

Resource utilization instead yields about: 53% of total LUTs, 19% of FFs, 16% of DSPs, 47% of CARRY8 units and a negligible consume of LUTRAM and MUX7.

[5]which is the standard way of initializing biases, as stated in [25].

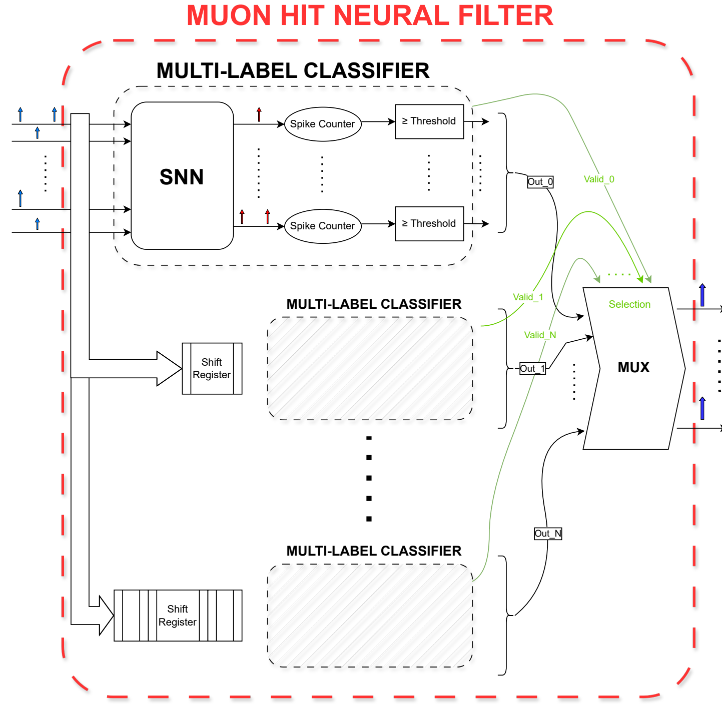


Figure 5.10: Sketch of the structure of the online neural filter for drift tubes particle detectors.

5.5 Enhancing reliability

Given the harsh working environment in which the just described system is to be operated, a brief digression on a common technique employed in radiation-subject environments such as CERN for limiting as much as possibile the appearance of soft errors when running a circuit for data acquisition or process control is mandatory. If, in fact, the design is to be implemented on the FPGAs mounted on the OBDTs, and not on those of the backend electronics, which are located inside the counting room and experience much lower radiation levels, some kind of mitigation for radiation-induced errors must be applied. The most straightforward approach is employ *triple modular redundancy* (TMR). This technique simply consists in producing three copies of the same entity and executing the same operation on each separate module in parallel, evaluating thereafter the result by applying a majority voting operation on produced results. In this way, if a just one single event upset occurs the result of the operation will be unaffected, while at least two errors are needed to trigger a wrong result, as depicted in Fig. 5.11. Mathematically, the probability of all three modules operate correctly is given by $P_3 = R^3$, while that of two modules only operating correctly and one failing by $P_2 = 3 \cdot R^2 \cdot (1 - R)$, where R represents the reliability of a single module. Thus, the reliability of the TMR system can be expressed as:

$$R_{TMR} = P(\text{all 3 modules working}) + P(\text{2 modules only working}) = R^3 + 3R^2(1 - R) \tag{5.3}$$

Expanding and simplifying, one finally gets:

$$R_{TMR} = 3R^2 - 2R^3 = R^2(3 - 2R) \tag{5.4}$$

This formula shows that the TMR system offers enhanced reliability compared to a single module, as long as the base reliability R is sufficiently high ($R \geq \frac{1}{2}$). By triplicating the critical components and using majority voting, TMR effectively mitigates single-point failures, improving the overall system's fault tolerance.

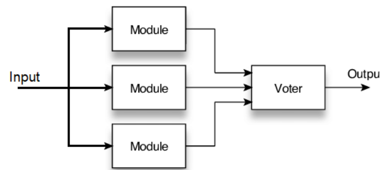


Figure 5.11: Triple Modular Redundancy working principle.

Moreover, thanks to the sparse nature of the network structure, an unwanted bit flip is likely to occurr on one only of the many processing block, either in the memory storing the coefficients or within the registers storing partial sums and accumulated potentials. The most tragic effect could be related to $U[t]$ overflowing or simply passing the threshold, for which an unwanted spike gets generated. If the number of spurious spike is contained, however, the influence of the final result may not be so relevant, depending totally on the network parametrization. In fact, SNNs have been suggested to be potentially less sensitive to radiation-induced soft errors like bit flips and stuck-at-value errors compared to conventional ANNs[6]. This hypothesis stems from the fact that SNNs operate on the principle of spiking activity and event-based processing, which could potentially offer inherent fault tolerance properties. Since information is encoded in the timing and frequency of spikes, rather than in continuous values of activations as in ANNs, it may allow for more robustness against certain types of errors since individual spikes can carry discrete and invariant information. Additionally, the asynchronous and event-driven nature of SNNs may help in localizing errors to specific spikes or synapses without affecting the entire network. However, the entire working of the network may be compromised if the syncronization between the various blocks is lacking. This depends on a few signals generated by one control unit for each layer. If an error occurs at control unit level all the neurons of a given layer may fall into an erroneous state. Therefore, if some kind of assurance is wanted without triplicating the resource utilization of the design, a good compromise could be to just triplicate the CU of each layer which could safeguard from fatal errors. The introduction of a majority voting operation will indeed slightly increase the latency of the network.

If hardware resources are not of particular concern, the entire design could be triplicated, in which case a fast approach to generate the new datastream for the FPGA could be to used CERN's *TMRG toolset* [18], which assists one in the process of creating digital designs immune to single event upsets by exactly exploiting TMR.

[6]Relative to the topic, a report by NASA on radiation tolerance and mitigation for meuromorphic mrocessors is available at [30].

6 ASIC realization

Once a satisfying version of the VHDL code describing the network has been produced and meticulously tested, mapping the design to a given technology is a choice based purely on which resources are available to the designer. Chapters 4-5 proposed some FPGA-based implementations of the spiking neural network model previously discussed in chapter 3. In the following sections, the synthesis on *Faraday*'s standard cell libraries[1] based on *UMC-130nm* CMOS process is going to be presented. A concise overview of the steps leading to the final realization of the ASIC (Application Specific Integrated Circuit) layout of Fig. 6.6 will be drawn up as well, as the various subtleties of the mapping process are addressed.

In order to implement the design, the following software tools have been employed:

- *Synopsys VCS Compiler*, used to analyze the design[2] entities and compile them (using the `vcs` command, with all the desired options set) for further testing through the use of DVE interface.
- *Synopsys Design Compiler (DC)*, and specifically Design Vision, its graphical user interface, utilized to re-analyze the code, elaborate the compiled RTL models and finally sythesize the design using the aforementioned technological libraries. Several reports about the synthesized design can be requested at this time, the content of which is discussed in Sec. 6.2.
- *Cadence Innovus Implementation System*, to finally transform the netlist obtained from *DC* into the physical layout of the chip to be produced, taking care of components placement and signal[3] routing, as well as power routing. Several optimization runs were needed in order to ensure signal and power integrity of the design, especially after the *clock tree synthesis*.

[1]The primitive cells of such library are reported in Appendix A.2.

[2]This is done by applying the *vhdlan -vhdl08* command, followed by the list of the relative paths at which the various entities of the design are placed, listed starting from the leaf modules and going up to the TOP one.

[3]In particular, clock signal routing, which is fundamental for the circuit to properly operate.

6.1 PRELIMINARY STEPS

A first obstacle that was encountered when transporting the design files from *Vivado* to *Design Compiler* is that the latter does not accept VHDL 2008 as language for its input files. Thus, the code has been adapted so to maintain the same functionality while being totally written according to VHDL '93 fashion. *DVE* has then been used to re-check the behaviour of the HDL code after the aforementioned modifications.

Nevertheless, it was fundamental to somehow include the `IEEE.fixed_pkg` library and all its features for fixed point types handling, which are ubiquitous in the design and cannot therefore be easily substituted by `signed` types[4]. This was done through the inclusion of VHDL-2008 compatibility libraries, following the instructions reported in [36].
Apart from this aspect, some minor modifications were applied so to ensure no unconstrained arrays were actually employed in the design. Practically, this meant that within the newly created `NETWORK_SYNTH` entity, each layer needed to have its own data type collecting all the parameters of the enclosed neurons. Furthermore, PRESTO compiler used for standard cell mapping did not support impure functions, such as the one reported in Code 6.1, which was used to read the parameters from their correspondent files in the original design. Therefore, the second approach of Sec. 3.3 was employed to set the network coefficients, namely that of grouping them together into a single array at layer level and then perform an assignment to a package-defined constant.

```vhdl
impure function init_ram_hex (filename: string; ram_depth: natural) return T_DATA is
      file text_file : text open read_mode is filename;
      variable text_line : line;
      variable temp : std_logic_vector(N-1 downto 0);
      variable ram_content : T_DATA(0 to ram_depth-1);
    begin
      for i in 0 to ram_depth - 1 loop
        readline(text_file, text_line);
        hread(text_line, temp);
        ram_content(i) := to_sfixed(temp, INT, -FRAC);
      end loop;

      return ram_content;
    end function;
```

Code 6.1: Impure VHDL function used to read network coefficients from file.

In this way it was possible to synthesize a "test" network with structure [16, 40, 32, 16]. The upper limit for timing closure for such a net was reached at 320 MHz, i.e. at a clock frequency one third greater than its FPGA counterpart, for which the optimizer of *Vivado* was not able to push synthesis to more than 240 MHz. Hereafter, the results of the implementation of the aforementioned net realized with standard cells and mapped on an XCKU115-2FLVB2104E FPGA are reported, where f_{clk} has been set to 240 MHz for both cases, in order to allow for a comparison to be made between the two implementations. The circuit netlist obtained after synthesis will then be used to generate the physical layout later discussed in Sec. 6.3.

[4]Which would also require the fractional point to be "manually" aligned during arithmetic operations.

6.2 Synthesis

After design compilation is terminated, the first thing to do is to control whether the synthesized circuit is correct by launching the `check_design` command, and, if this is the case, the next mandatory step is to ensure the timing for the given constraints on the clock signal frequency and input/output delay is met. In digital circuits, in particular, the so-called "slack" refers to the difference between the required time and the actual time taken for a signal to propagate from one point to another within the circuit. Considering setup and hold time requirements, it can be further divided into:

- $Setup_{slack} = T_{clk} - t_{CQ} - t_{setup} - t_{prop}$, where t_{CQ} is the time interval between the clock edge and the moment the output of the FF can be considered stable and valid and t_{prop} is the propagation delay of the combinational logic.
- $Hold_{slack} = t_{actual} - t_{hold}$, where t_{actual} is the actual duration the data remains stable after the clock edge.

Positive slack means that the signal arrives earlier than required, which implies that there is a safety margin in the timing. This is typically a desirable situation as it indicates that the circuit can operate reliably within the specified timing parameters. Conversely, negative slack means that the signal arrives later than required, indicating a timing violation. This is a critical issue as it suggests that the circuit will not function correctly at the intended clock frequency, and corrective measures must be taken, such as optimizing the design or reducing the clock speed. On the other hand, zero slack means that the signal arrival time exactly matches the required time, which is indicative of optimal timing but leaves no margin for variation or uncertainty, that may be problematic in the subsequent place and route phase. The slack histogram for the design is reported in Fig. 6.1, where no path yields a negative slack and therefore timing is met. The most critical paths of the design are located at HIDDEN2/NEUR_10/MAC/ and, more specifically, within the multiplier, adder and final register used for storing the value of the membrane potential.

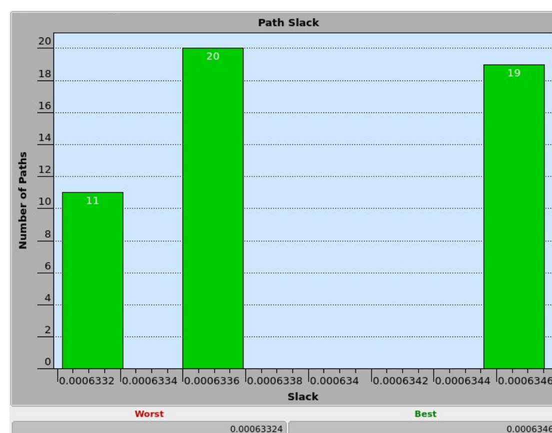


Figure 6.1: Slack histogram in *Design Vision*. Slack (on the x-axis) is reported in ns.

The `report_timing` command also precisely lists which kind of standard cells are critical, upon which the following are found: MAC_38_DW02_mult_0_DW02_mult_37, FA1DHD, XOR2 EHD, MAC_38_DW01_add_5, AO12EHD and many others.

The report on the circuit area is shown in Tab. 6.1. About 70% of the total area is occupied by combinational logic, whereas 29.7% by sequential elements and only the remaining 0.3% for buffer[5]/inverter area for signal distribution across the various blocks of the design. From the total area it is possible to estimate the actual size of the final chip: considering that a core utilization factor of 0.7 has been used, the side length of the square silicon die of Sec. 6.3 for the SNN is found to be $l \approx \sqrt{\frac{4.86\ mm^2}{0.7}} = 2.635\ mm$.

Metric	Value
Number of ports	413269
Number of nets	802674
Number of cells	339774
Number of combinational cells	277337
Number of sequential cells	56239
Number of buf/inv	29931
Number of references	4
Combinational area	3414187.5 μm^2
Buf/Inv area	127847.7 μm^2
Noncombinational area	1445479.7 μm^2
Total cell area	4859667.2 μm^2

Table 6.1: Area report of SNN ASIC.

A brief report of SNN power consumption are reported in Tab. 6.2. The total dynamic power is found to be $P_{dyn} = \frac{1}{2}\sum_i \alpha_i C_{L,i} V_{dd}^2 f_{clk} \approx 192$ mW, which is dissipated for the most part within the clock networks inside the cells. Since the operating voltage is fixed by technological specs to 1.2 V, and the logical activity α and capacitance C_L of a given node are dependent on the circuit structure, the primary way to diminish power consumption is to simply reduce the clock frequency, if at all possible. To improve the energy consumed by the circuit, *clock gating* could also be employed, which involves selectively turning off the power supply to certain portions of a circuit when they are not in use, thereby reducing leakage power. This could be beneficial since the SNN may remain inactive for relatively long periods of time.

Metric	Value
Global Operating Voltage	1.2 V
Cell Internal Power	187.8006 mW (98%)
Net Switching Power	4.7302 mW (2%)
Total Dynamic Power	192.5308 mW (100%)
Cell Leakage Power	1.5928 mW

Table 6.2: Power report summary.

[5]Buffers are used to strengthen signals, drive large loads, or restore signal integrity over long interconnects. They do not change the logic value but provide amplification or isolation.

The fundamental processing blocks employed by XCKU115-2FLVB2104E FPGA for the exact same circuit are reported in Fig. 6.3. Clearly, a DSP slice is used to perform the multiplication inside the MAC entity of every neuron, as confirmed by the fact that a total of $n = 40+32+16 = 88$ neurons are present in the sample net. A great number of flip flops is then used, but FPGAs have plenty of them, so this is not likely to present an issue even for a growing number of nodes. More critical is instead the usage of lookup tables, which are needed to implement combinational logic for arithmetic operations and are used as memory for the various coefficients. A careful look at Fig. 6.2, showing directly *Vivado*'s utilization report, better details the blocks usage following the design hierarchy: apart the single DSP per neuron, the number of CLB LUTs and F7 muxes, mainly related to the `W_MASK` block, registers, and CARRY8 (employed in `ADDER` and `MAC`) has been optimized w.r.t. the actual value of the parameters.
Finally, input and output pins are simply physically limited to a given amount: if the design requires more than the available ones, it will need to be incapsulated into some kind of "wrapper" entity that has to take care of demultiplexing the inputs and, conversely, multiplexing the outputs of the design. An ASIC does not suffer from such a limitation in principle, even though an increased pin count is likely to increase routing congestion and might therefore decrease the chance of meeting timing requirements under the same constraints.

Resource type	Used	Available	Utilization [%]
LUT	48293	663360	7.28
FF	35437	1326720	2.67
DSP	88	5520	1.59
IO	39	702	5.56

Table 6.3: Resource employed by FPGA-based SNN.

Name	CLB LUTs (663360)	CLB Registers (1326720)	CARRY8 (82920)	F7 Muxes (331680)	CLB (82920)	LUT as Logic (663360)	DSPs (5520)
NETWORK	48293	35437	5844	292	10142	48293	88
HIDDEN1 (LAYER)	16642	8856	2618	286	4187	16642	40
HIDDEN2 (LAYER_parameterized0)	24704	20446	2357	1	5760	24704	32
OUTLAYER (LAYER_parameterized1)	6641	6135	707	5	1667	6641	16
GEN[0].NEUR (NEURON_paramete	393	362	41	0	142	393	1
GEN[1].NEUR (NEURON_paramete	535	388	45	0	189	535	1
GEN[2].NEUR (NEURON_paramete	465	436	48	1	131	465	1
GEN[3].NEUR (NEURON_paramete	419	396	45	1	139	419	1
GEN[4].NEUR (NEURON_paramete	483	437	48	0	141	483	1
GEN[5].NEUR (NEURON_paramete	401	353	40	0	144	401	1

Figure 6.2: Extensive XCKU115-2FLVB2104E FPGA utilization report for a [16, 40, 32, 16] SNN.

An overview of power consumption for the FPGA-based design is also reported in Fig. 6.3. The absorbed current is split between the 3.342 A of the 0.95 V rail of internal processing blocks and an additional 237 mA for the auxiliary 1.8 V rail. Almost the same amount of static power is consumed by both supply rails, namely $P_{stat} \approx 430$ mW, while the remaining part is due to internal nodes switching activity. Note, finally, how a non-negligible amount of power is lost on clock signal distribution and how DSPs, even if present in the lowest amount, are efficiently performing a complex operation as the product between two 16-bit values is.

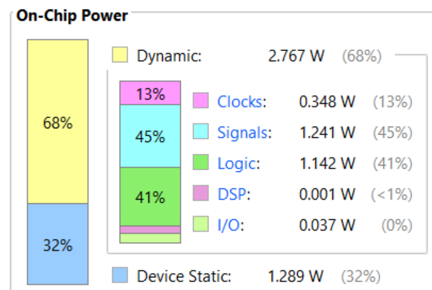


Figure 6.3: FPGA-based SNN power consumption.

Evidently, a great saving of power is accomplished by using a custom ASIC architecture. In fact, FPGA power consumption is generally greater due to several inherent design factors:

- the additional logic and interconnects necessary to allow deisgn flexibility, that lead to increased switching activity and interconnect overhead;
- the presence of programmable resources, such as LUTs and configuration memory, that further elevate base-level losses.

Additionally, while both options can make use of advanced process technologies, and in this case the FPGA is favoured with its 20 nm minimum feature size, ASICs still benefit from specific low-power optimizations performed during standard cells design phase, which inevitably result in 20 times lower power consumption.

6.3 Physical design

The ultimate jump to the lowest possible level of abstraction occurs by transforming the gate-level netlist obtained at the end of synthesis phase into a geometrical description of the physically placed transistors, called layout. Precisely, physical design consists of the following steps:

1. Floor planning, defining the available silicon area, divided between core and boundary area (for I/O pads), fixed to 10 μm for each side, and creating tracks for cell placement.
2. Design partitioning, i.e. the act of subdiving complex circuits into their fundamental blocks, with the aim of minimizing the required interconnects and the associated delay.
3. Power planning, in which rings of large metal interconnections[6] surrounding the core and the main individual blocks are placed, together with stripes of metal connections running horizontally and vertically so to form a grid prodiving power to all the design elements.
4. Cell placement, i.e. defining the (x, y) coordinates of each standard cell, which also adjusts floor and power planning so to fit timing requirements. The so-called "amoeba" view from *Innovus* is reported in Fig. 6.4, which shows which parts of the die have been assigned to each of the entities instantiated inside the TOP file i.e. NETWORK_SYNTH. In FPGAs,

[6]At least two levels of metal have to be used, one for VDD and one for GND.

instead, placement requires allocating each programmable cell to a logic block (e.g. CLB, FDRE, MUX7, etc.) and define the content of its configuration memory. This information will be encoded inside the bitstream generated by *Vivado* for FPGA programming.

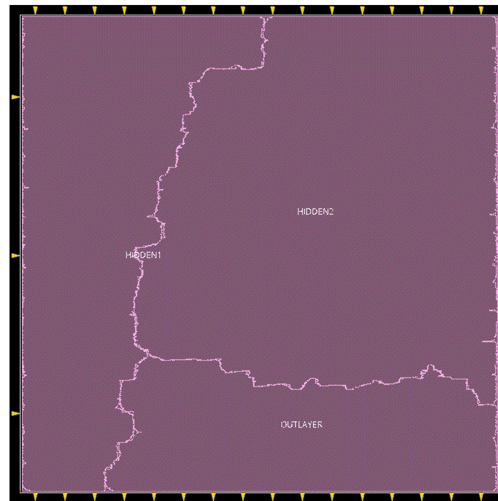


Figure 6.4: Physical repartition of chip area between the layers of the network.

5. Clock tree synthesis (CTS), generating the distribution network of the clock signal, constituted by metal interconnections and buffers. Different approaches may be used to ensure the *skew* of `CLK` between different part of the circuit remains within acceptable limits. The resulting path slack histogram after CTS is reported in Fig. 6.5. Since many negative slacks were present, several optimization rounds have been launched before passing to the next step, until timing was met with a discrete margin.

6. Routing, initially performed at a global level for outlining an approximate route for each net, and then in a more detailed manner to the actual geometric layout of the regions. In this phase, physically near blocks may be compacted together to decrease the overall area of the chip.

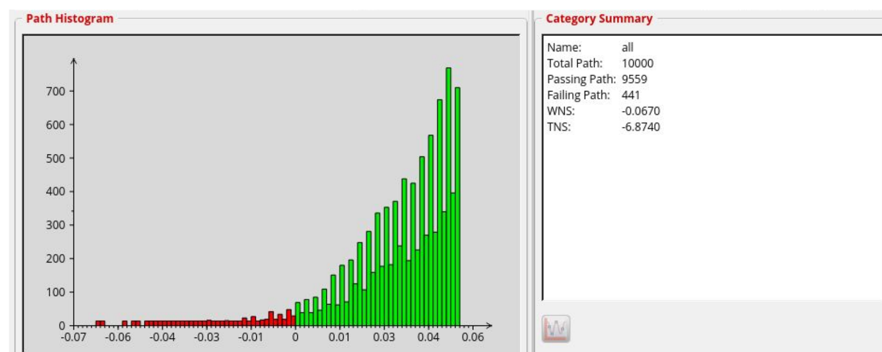


Figure 6.5: Path slack historam after clock tree synthesis.

7. Design-for-Manufacturability (DFM) enhancements for improving the yield of the fabrication process, e.g. the addition of inactive metal segments[7], used to minimize the variations of metal layers thickness, which could be responsible for parasitic capacitance change and ultimately timing inconsistencies.

8. Physical verification, which is subdivided between *design rules checking* (DRC), looking for any violations of rules such as minimum spacing/width/enclosure, min/max area, antenna violations, and other geometrical rules to which the mask for photolithography should stick to, and *layout-vs-schematic* (LVS), which extracts an equivalent netlist from the layout and compares it with the gate-level netlists originally generated by the logic synthesis step.

Fig. 6.6 shows a picture of the resulting layout after all the aforementioned steps have been performed, plus multiple post-route optimization runs. Different colours are used for the eight metal layers of the UMC 130 nm process. The pins found on the sides of the chip are equally spaced and placed simmetrically w.r.t. its centre: their position must be defined in a `.io` file according to a specific syntax.

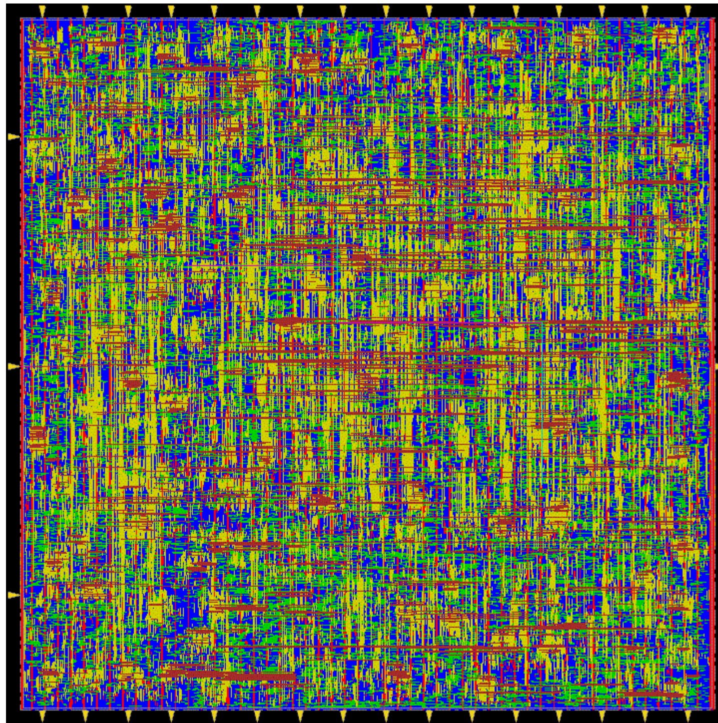


Figure 6.6: Final layout of [16, 40, 32, 16] SNN with 1st order reset-by-subtraction LIF neurons. The 16 input pins are located on the top side, while the output pins reside on the lower one. On the left, instead, the RESET, CLK and IN_VALID pins can be found, whereas the OUT_VALID pin is placed on the right. Chip side length is about 2.6 mm.

[7]called metal *fill* patterns.

The average amount of power consumed by the final circuit is two and a half times greater w.r.t. the one obtained in Sec. 6.2, as reported in Fig. 6.7, but still way smaller than that of the FPGA-based SNN. There are several reason for this fact:

- During logical synthesis, the power estimation does not accurately account for the additional capacitance due to the interconnections between various components. After place and route, instead, the actual physical layout is known, and the power consumption due to the wiring (especially for long lines) can be substantial.
- The proximity and placement of cells can affect leakage currents, and the physical distances between wires can lead to crosstalk and subsequently higher coupling capacitances. Moreover, IR voltage drop due to resistive paths in power delivery networks might not be fully estimated during logical synthesis, further increasing the total leakage current. The power analysis tool of *Innovus* allows one to find out how relevant the impact of this phenomenon is and also to seek where there excessive power dissipation is likely to cause electromigration.
- More clock buffers and inverters may be needed to meet timing requirements (and also for CTS), adding to the overall dynamic and leakage power consumption.
- Place and route might introduce logic duplication, retiming, or pipelining to meet timing, which can increase the overall switching activity and thus power consumption. Changes made to fix DRC or LVS errors can introduce additional elements as well, affecting power dissipation.

```
Total Power
-----------------------------------------------------------------------------------------
Total Internal Power:      355.35550268            69.2240%
Total Switching Power:     152.84616859            29.7747%
Total Leakage Power:         5.14010186             1.0013%
Total Power:               513.34177313
-----------------------------------------------------------------------------------------


Group                           Internal   Switching     Leakage       Total  Percentage
                                Power      Power         Power         Power  (%)
-----------------------------------------------------------------------------------------
Sequential                         221.8       15.05       1.047       237.8       46.33
Macro                                  0           0           0           0           0
IO                                     0           0           0           0           0
Combinational                        118       90.36       4.028       212.4       41.38
Clock (Combinational)              15.58       47.44     0.06557       63.09       12.29
Clock (Sequential)                     0           0           0           0           0
-----------------------------------------------------------------------------------------
Total                              355.4       152.8        5.14       513.3         100
-----------------------------------------------------------------------------------------
```

Figure 6.7: Power report of the final layout for the SNN ASIC.

For comparison, the place and route of the same SNN implemented on the XCKU115-2FLVB2-104E FPGA is shown in Fig. 6.8. Here, in accordance with Tab. 6.3, a small portion of the device overall area is occupied for the 88 neuron operation. Only one of the two *Super Logic Regions* (SLR) is involved, and among that only some of the XY resource slices are used[8]. Furthermore, a zoomed view showing the fundamental cells of both implementation is reported in Fig. 6.9. While on the ASIC the blocks are closely packed and interconnections are distributed all around, connections between different block in FPGAs are constrained by switchboxes, namely a grid of

[8]And they are only used partially, as well, since this is the configuration providing better timing closure.

programmable switches that link various input and output lines, with their configuration being part of the bitstream loaded onto the FPGA. Different topologies, such as full crossbar or diagonal, offer various trade-offs in terms of area, delay, and flexibility, ultimately affecting the routing efficiency and ease of implementation within the FPGA.

Fig. 6.9-(b) also depicts a SLICEM (Slice with Memory) block, which not only supports basic logic functions but also includes additional capabilities for memory and arithmetic operations. In fact, each SLICEM consists of several Lookup Tables (LUTs) that can be configured as either logic resources or compact distributed RAMs. Knowing all of this, it results clear why a greater amount of power is consumed by the FPGA whatsoever the design to be implemented: due to the fixed resource overhead allowing a general block to be realized!

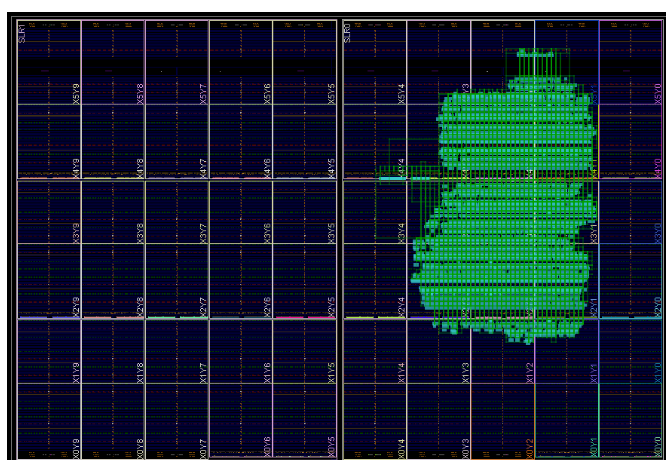
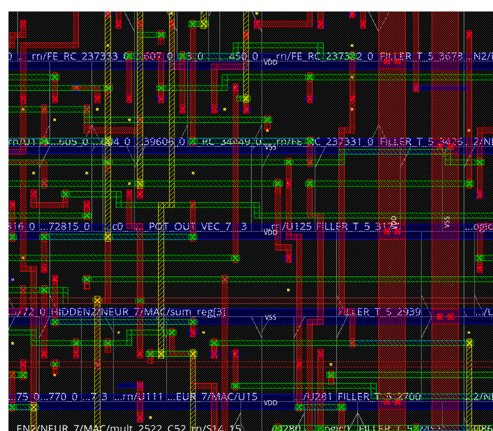
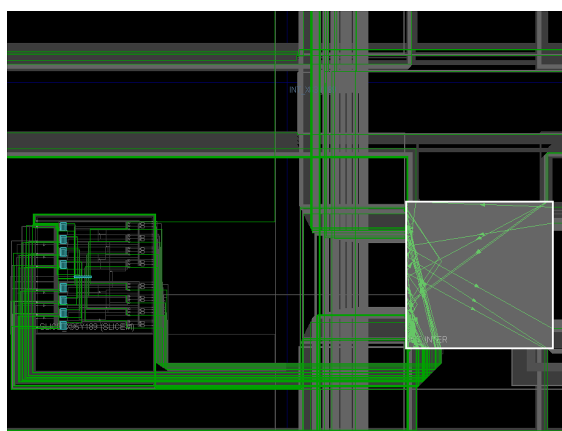


Figure 6.8: Physical view of the circuit realized employing the FPGA of KCU1500 board. The elementary processing blocks are drawn in blue, while the green lines show the interconnetions.



(a) ASIC layout showing standard cells (with their names) and power grids (thicker lines).



(b) SLICEM configurable logic block (on the left) and switchbox (on the right).

Figure 6.9: Close-up look at SNN circuit realizations.

Conclusions and Outlook

This work has just scratched the surface of the emerging topic of SNNs, showing one of many possible architectures and hardware implementations for emulating the spiking nature of neuronal dynamics. Nonetheless, it fulfilled the main objective of the thesis, i.e. demonstrating the feasibility of transporting SNNs, which enjoy full support of many software tools, to low latency devices, potentially pushing the use of this kind of neural networks towards more demanding environments, such as that of DAQ and triggering for particle Physics experiments, in which the timescale of the relevant information to be detected is comparable with the internal operating period of the involved devices. Moreover, implementation results reveal how the actual footprint of a SNN even of considerable dimensions is not critical on XCKU115-2FLVB2104E FPGA. For growing number of nodes an analysis should be assessed to understand of how much quantization can be tolerated for physically representing the neurons' coefficents, in order to find the best compromise between fidelity in network behaviour (w.r.t. the *Matlab*/*Python* golden reference model), resources utilization[1] and maximum operating frequency.

Further points for future investigations could be:

- Performance comparison between ANN-converted nets and BPTT-trained SNNs, given the same data encoding, operating conditions and amount of allocated physical resources on the target device. A first study case could be the net proposed in [22], and available at [23].
- Exploration of higher-order LIF neuron models and recurrent SNNs architectures both from the training side, which may require writing some custom code, and from the evaluation side, exploiting the general IIR implementation cited in Sec. 3.1.2.
- Employment of quantization aware training, also exploring alternative frameworks to *SNN Torch*, so to verify whether an effective network could be realized also at lower numeric

[1]Some resources must be allocated as well for "secondary" but absolutely necessary blocks, such as the IP cores taking care of data exchange between the target hardware device and an host machine, as discussed in chapter 4.

precision (and, in turn, minor resource utilization).

- In case any the previous point succeeds in enhancing the classification accuracy of the muon hit filter, it could be of interest to recast the design implementation on the Ultrascale FPGA mounted on the OBDT board introduced in Sec. 5.1. Moreover, the extension to blocks of cells spanning over two or more DT superlayers could be explored so to limit the effect of the chamber's ageing and still achieve a good detection efficiency. In this case, though, the target FPGA is the Ultrascale used in the backend electronics[2], since an OBDT only "covers" a single superlayer. Additional effort is likely to be put into network training phase (i.e. defining architecture, setting initial parameters values, etc.), as the extended event view is even sparser than for a block of cells inside a single superlayer.
- Combining hardware inference performed on FPGA with backprogation phase in software so to speed up SNN training. This last point could be particularly interesting as the quantization of network inference is not simulated by clipping the values on higher resolution variables, but is truly carried out also for the various arithmetic operations.

[2]Refer to Fig. 5.3.

Appendix

A.1 Hardware resources of KCU1500 board

The KCU1500 development board, designed around the *Xilinx* Kintex UltraScale XCKU115-2FLVB2104E FPGA, provides a broad array of hardware resources optimized for advanced digital design applications. The FPGA consists of key elements, including configurable logic blocks (CLBs), dedicated hardware multipliers, and an extensive assortment of I/O pins, which collectively enable the implementation of highly complex digital circuits. A detailed overview of the primary hardware resources is provided in Table A.1.

Resource Type	Quantity	Description
CLBs	663360	Each CLB is made up of 8 LUTs and 16 flip-flops.
LUTs	530000	Serving as the primary method for implementing combinational logic. Each LUT can be configured as either a 6-input single-output LUT or two 5-input single-output LUTs.
Flip-Flops	1327680	Implementing sequential logic and state machines.
Block RAM	75.9 Mb	Including 36 Kb dual-port memory blocks, suitable for implementing efficiently large memory arrays and FIFOs.
DSP Slices	5520	Each DSP is equipped with a 27x18 multiplier, adder, and pre-adder, which are essential for high-speed arithmetic operations.
I/O Pins	676	Facilitating extensive external interfacing, which is critical for connecting the FPGA to other hardware components and peripherals.
GTY Transceivers	32	Supporting data rates up to 32.75 Gbps, enabling high-speed serial communication for applications such as PCIe Gen3/Gen4, and 100G Ethernet.
Clock Management		MMCM and PLL blocks supporting flexible clock generation and distribution for rigorous timing requirements within the FPGA.

Table A.1: Primary hardware resources of the UltraScale XCKU115-2FLVB2104E FPGA.

In addition to the FPGA resources, KCU1500 offers a set of ancillary hardware resources designed to enhance system integration and maximize performance, such as a PCI Express (PCIe) Gen3 x16 interface that provides a high-bandwidth pathway for data exchange between the FPGA and host systems, supporting up to 128 Gbps aggregate bandwidth. The board also features 8 GB of DDR4 SDRAM connected through a 64-bit bus for ample storage and fast access, and a QSPI flash for FPGA bitstream storage and configuration, ensuring reliable startup. Networking capabilities are then enhanced by 1 and 10 gigabit Ethernet (GbE) ports with dedicated MAC and PHY layers for robust and low-latency communication. For high-speed I/O, the KCU1500 includes SMA connectors for applications requiring precise timing and high bandwidth. USB interfaces include USB 3.0 and 2.0, supporting versatile communication pathways for configuration, debugging, and data transfer, with USB 3.0 supporting speeds up to 5 Gbps. The board's clocking resources feature programmable oscillators and PLLs for versatile clock management, with multiple clock inputs and outputs. Power management includes multiple voltage regulators and monitoring systems for stable power delivery. Debugging and monitoring are facilitated through an integrated JTAG interface, onboard LEDs, push-buttons, and DIP switches, providing ease of development, troubleshooting, and real-time monitoring.

A.2 Standard cells of faraday 130nm technological library

The Faraday 130nm technological library is built on a robust process technology that offers a balanced combination of performance, power efficiency, and cost-effectiveness. The process utilizes advanced doping techniques to optimize the electrical characteristics of transistors, including precise control over channel doping to achieve desired threshold voltages and minimize leakage currents. Furthermore, the process supports up to 7 layers of metal interconnects, providing the necessary routing capabilities for complex integrated circuits. The metal layers are composed of aluminum or copper, with the top layers generally reserved for power and ground distribution to minimize resistive losses.

Table A.2: Main types of standard cells in the *Faraday* UMC-130nm technological library

Cell Category	**Description**
Logic Gates	The library includes a variety of logic gates such as AND, OR, NAND, NOR, XOR, and XNOR gates. Specific examples include 2-input AND (AND2), 6-input AND (AND6), 2-input OR (OR2), and 3-input XOR (XOR3). These gates are available in multiple drive strengths to accommodate different load conditions and performance requirements.
Flip-Flops and Latches	A range of flip-flops and latches are provided, including D flip-flops like FD1S, JK flip-flops like FJK1, and SR latches. Both edge-triggered and level-sensitive varieties are included. These cells are crucial for constructing sequential logic circuits and storing state information within digital systems.
Buffers and Inverters	The library offers multiple types of buffers and inverters such as INV, BUFX2, and IB1, necessary for signal buffering, propagation delay control, and logic level inversion. These cells are available in various drive strengths to ensure signal integrity across different stages of the circuit.
Adders and Arithmetic Units	Arithmetic functions are supported by cells such as half-adders (HA1), full-adders (FA1), and more complex units like 4-bit Ripple Carry Adder (RCA4) and 16-bit Multiplier (MULT16). These cells are essential for performing arithmetic operations in digital systems.
Multiplexers and Demul-tiplexers	Multiplexers (MUX) like MUX2x1 and demultiplexers (DEMUX) such as DEMUX1x2 are available to facilitate the selection and routing of data within a circuit. These cells support configurable data paths and are fundamental in designing efficient data control mechanisms.
Decoders and Encoders	The library includes decoders such as DEC2x4 for converting encoded data into a specific format and encoders like ENC8x3 for the opposite function. These cells are vital in addressing memory and implementing control logic.
Special Function Cells	Special function cells include clock gating cells (CG1), power management cells (PM1), and scan cells (SCAN_FF) for design-for-test (DFT) purposes. These cells enhance power efficiency, enable clock domain crossing, and facilitate testing and verification of the circuitry.

The gate oxide thickness is approximately 2.1 nm, which ensures reliable transistor operation while maintaining control over short-channel effects. The minimum feature size of 130nm allows for the fabrication of transistors with sufficient drive strength while maintaining good yields, and the typical operating voltage for circuits designed in this process technology is 1.2V, with options for lower voltage operation to reduce power consumption.

A.3 VHDL testbench for generic timeseries dataset

```
entity NETWORK_tb is
end NETWORK_tb;

architecture BHV of TB_NETWORK is
    constant N_INPUT  : integer := 8; constant T_CLK    : time := 10 ns; constant T_LATENCY:
     integer := 4;

    signal CLK, RESET, IN_VALID, OUT_VALID: std_logic := '0';
    signal SPIKE_IN      : std_logic_vector (0 to N_INPUT-1) := (others => '0');
    signal SPIKE_OUT_VEC: std_logic_vector (0 to N_INPUT-1);

    component NETWORK is
        port (CLK             : in  std_logic;
              RESET           : in  std_logic;
              SPIKE_IN        : in  std_logic_vector (0 to N_INPUT-1);
              IN_VALID        : in  std_logic;
              SPIKE_OUT_VEC   : out std_logic_vector (0 to N_INPUT-1);
              OUT_VALID       : out std_logic);
    end component;

begin

    DUT: NETWORK
        port map (CLK           => CLK,
                  RESET         => RESET,
                  SPIKE_IN      => SPIKE_IN,
                  IN_VALID      => IN_VALID,
                  SPIKE_OUT_VEC=> SPIKE_OUT_VEC,
                  OUT_VALID     => OUT_VALID);

    P: process
    begin
        while true loop
            CLK <= '0'; wait for T_CLK/2; CLK <= '1'; wait for T_CLK/2;
        end loop;
        wait;
    end process;

    P_IN: process
    begin
        -- Initialize Signals
        RESET <= '1'; wait for 20 ns;    -- Wait for global reset to finish
        RESET <= '0'; SPIKE_IN <= (others => '0');
        IN_VALID <= '0';

        wait for T_CLK*5; -- Wait a few clock cycles

        -- Submit SPIKE_IN samples every T_LATENCY clock periods
        for i in 0 to 15 loop
            SPIKE_IN <= data(i);  -- Retrieving proper stimulus from package-defined constant.
            IN_VALID <= '1'; wait for T_CLK;
            IN_VALID <= '0'; -- starts data processing
            wait for (T_LATENCY - 1) * T_CLK; -- Wait for T_LATENCY-1 Tclk
        end loop;

        wait;
    end process;

end BHV;
```

A.4 Bash script for AXI LITE registers read/write

```
1  #!/bin/bash
2
3  # Function to write to an AXI Lite register and verify the write
4  # Arguments:
5  # 1 - Address of the AXI Lite register
6  # 2 - Value to write to the register
7  write_and_verify_axi_lite_register() {
8      local address=$1
9      local value=$2
10
11     # Use devmem to write the value to the address
12     echo "Writing value 0x$(printf '%08X' $value) to AXI Lite register at address 0x$(printf '%08X'
         $address)"
13     devmem $address 32 $value
14
15     # Read back the value
16     read_value=$(devmem $address 32)
17     echo "Read back value 0x$(printf '%08X' $read_value) from address 0x$(printf '%08X' $address)"
18
19     # Check if the read back value is the same as the written value
20     if [[ $read_value -eq $value ]]; then
21         echo "Verification successful: Value at address 0x$(printf '%08X' $address) is 0x$(printf
        '%08X' $read_value)"
22     else
23         echo "Verification failed: Expected 0x$(printf '%08X' $value) but read 0x$(printf '%08X'
        $read_value)"
24         exit 1
25     fi
26 }
27
28 # Example of script usage:
29 # Define the base address of your AXI Lite interface
30 AXI_LITE_BASE_ADDR=0x400
31
32 # Define the address offsets of the registers
33 REGISTER_0_OFFSET=0x00
34 REGISTER_1_OFFSET=0x04
35
36 # Define the values to write to the registers
37 VALUE_0=0x12345678
38 VALUE_1=0x9abcdef0
39
40 # Write to the registers and verify
41 write_and_verify_axi_lite_register $((AXI_LITE_BASE_ADDR + REGISTER_0_OFFSET)) $VALUE_0
42 write_and_verify_axi_lite_register $((AXI_LITE_BASE_ADDR + REGISTER_1_OFFSET)) $VALUE_1
43
44 echo "AXI Lite register write and verification completed."
```

References

[1] Ray Andraka. "A survey of CORDIC algorithms for FPGA based computers". In: *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*. FPGA '98. Monterey, California, USA: Association for Computing Machinery, 1998, pp. 191–200. ISBN: 0897919785. DOI: 10.1145/275107.275139. URL: https://doi.org/10.1145/275107.275139.

[2] Daniel Auge et al. "A survey of encoding techniques for signal processing in spiking neural networks". In: *Neural processing letters/Neural Processing Letters* 53.6 (July 2021), pp. 4693–4710. DOI: 10.1007/s11063-021-10562-2. URL: https://doi.org/10.1007/s11063-021-10562-2.

[3] Dennis Bäßler, Tobias Kortus, and Gabriele Gühring. "Unsupervised anomaly detection in multivariate time series with online evolving spiking neural networks". In: *Machine Learning* 111.4 (Apr. 2022), pp. 1377–1408. ISSN: 1573-0565. DOI: 10.1007/s10994-022-06129-4. URL: https://doi.org/10.1007/s10994-022-06129-4.

[4] M. Bellato et al. "Radiation hardness and quality validation of the on-detector electronics for the CMS Drift Tubes upgrade". In: *Journal of Instrumentation* 19.06 (June 2024), p. C06001. DOI: 10.1088/1748-0221/19/06/C06001. URL: https://dx.doi.org/10.1088/1748-0221/19/06/C06001.

[5] David Bishop. *Fixed Point Package User's Guide by David Bishop (dbishop@vhdl.org)*. URL: https://freemodelfoundry.com/fphdl/Fixed_ug.pdf.

[6] Tong Bu et al. *Optimal ANN-SNN Conversion for High-accuracy and Ultra-low-latency Spiking Neural Networks*. 2023. arXiv: 2303.04347 [cs.NE].

[7] Benjamin Cramer et al. "The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.7 (2022), pp. 2744–2757. DOI: 10.1109/TNNLS.2020.3044364.

[8] Jason K. Eshraghian et al. *Navigating Local Minima in Quantized Spiking Neural Networks*. 2022. arXiv: 2202.07221.

[9] Jason K. Eshraghian et al. "Training Spiking Neural Networks Using Lessons From Deep Learning". In: *Proceedings of the IEEE* 111.9 (2023), pp. 1016–1054. DOI: 10.1109/JPROC.2023.3308088.

[10] Kazuhisa Fujita. "Spatial Feature Extraction by Spike Timing Dependent Synaptic Modification". In: *Neural Information Processing. Theory and Algorithms*. Ed. by Kok Wai Wong, B. Sumudu U. Mendis, and Abdesselam Bouzerdoum. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 148–154. ISBN: 978-3-642-17537-4.

[11] Yue Gao, Weiqiang Liu, and Fabrizio Lombardi. "Design and Implementation of an Approximate Softmax Layer for Deep Neural Networks". In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9180870.

[12] F. Gasparini et al. "Bunch crossing identification at LHC using a mean-timer technique". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 336.1 (1993), pp. 91–97. ISSN: 0168-9002. DOI: https://doi.org/10.1016/0168-9002(93)91082-X. URL: https://www.sciencedirect.com/science/article/pii/016890029391082X.

[13] Wenzhe Guo et al. "Neural Coding in Spiking Neural Networks: A Comparative Study for Robust Neuromorphic Systems". In: *Frontiers in Neuroscience* 15 (2021). ISSN: 1662-453X. DOI: 10.3389/fnins.2021.638474. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2021.638474.

[14] Ilyass Hammouamri, Ismail Khalfaoui-Hassani, and Timothée Masquelier. *Learning Delays in Spiking Neural Networks using Dilated Convolutions with Learnable Spacings*. 2023. arXiv: 2306.17670 [cs.NE].

[15] Ilyass Hammouamri, Ismail Khalfaoui-Hassani, and Timothée Masquelier. "Learning Delays in Spiking Neural Networks using Dilated Convolutions with Learnable Spacings". In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=4r2ybzJnmN.

[16] Johann Huber. "Batch normalization in 3 levels of understanding - Towards Data Science". In: (Jan. 2023). URL: https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338.

[17] Jonas Julian Jensen. *8 ways to create a shift register in VHDL*. July 2023. URL: https://vhdlwhiz.com/shift-register/.

[18] S. Kulis. "Single Event Effects mitigation with TMRG tool". In: *Journal of Instrumentation* 12.01 (2017), p. C01082. URL: http://stacks.iop.org/1748-0221/12/i=01/a=C01082.

[19] Robert Legenstein, Christian Naeger, and Wolfgang Maass. "What Can a Neuron Learn with Spike-Timing-Dependent Plasticity?" In: *Neural Computation* 17.11 (Nov. 2005), pp. 2337–2382. ISSN: 0899-7667. DOI: 10.1162/0899766054796888. eprint: https://direct.mit.edu/neco/article-pdf/17/11/2337/816258/0899766054796888.pdf. URL: https://doi.org/10.1162/0899766054796888.

[20] Edgar Lemaire et al. "An Analytical Estimation of Spiking Neural Networks Energy Efficiency". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2023, pp. 574–587. ISBN: 9783031301056. DOI: 10.1007/978-3-031-30105-6_48. URL: http://dx.doi.org/10.1007/978-3-031-30105-6_48.

[21] Cuixia Li et al. “IC-SNN: Optimal ANN2SNN Conversion at Low Latency”. In: *Mathematics* 11.1 (2023). ISSN: 2227-7390. DOI: 10.3390/math11010058. URL: https://www.mdpi.com/2227-7390/11/1/58.

[22] M. Migliorini et al. “Muon trigger with fast Neural Networks on FPGA, a demonstrator”. In: *Journal of Physics: Conference Series* 2374.1 (Nov. 2022), p. 012099. ISSN: 1742-6596. DOI: 10.1088/1742-6596/2374/1/012099. URL: http://dx.doi.org/10.1088/1742-6596/2374/1/012099.

[23] Mmiglio. *GitHub - 40MHz/SmallModelFirmware*. URL: https://github.com/40MHz/Small%20ModelFirmware.

[24] *Muon Drift Tubes | CMS Experiment*. May 2024. URL: https://cms.cern/detector/detecting-muons/muon-drift-tubes.

[25] Michael A. Nielsen. *Neural Networks and Deep Learning*. 2018. URL: http://neuralnetworks%20anddeeplearning.com/.

[26] A. Rushton and an O’Reilly Media Company Safari. *Vhdl for Logic Synthesis, Third Edition*. John Wiley & Sons, 2011. URL: https://books.google.it/books?id=yRU_zQEACAAJ.

[27] Sai Sanjeet et al. “IIR Filter-Based Spiking Neural Network”. In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2023, pp. 1–5. DOI: 10.1109/ISCAS46773.2023.10182209.

[28] Javier Sastre Alvaro. *The OBDT board: A prototype for the Phase 2 Drift Tubes on-detector electronics*. Tech. rep. Geneva: CERN, 2020. URL: https://cds.cern.ch/record/2797780.

[29] Sarah Ali El Sayed. *Fault tolerance in hardware spiking neural networks*. Oct. 2021. URL: https://theses.hal.science/tel-03681910v2.

[30] Johann Schumann. *Radiation tolerance and mitigation for neuromorphic processors*. Jan. 2022. URL: https://ntrs.nasa.gov/citations/20220013182.

[31] Hian-Hian See et al. “ST-MNIST - The Spiking Tactile MNIST Neuromorphic Dataset”. In: *CoRR* abs/2005.04319 (2020). arXiv: 2005.04319. URL: https://arxiv.org/abs/2005.04319.

[32] Sahil Singh. *LUT in FPGA: A Brief understanding of FPGA Resources [2023]*. Feb. 2024. URL: https://www.logic-fruit.com/blog/fpga/lut-in-fpga/.

[33] *snnTorch Documentation — snntorch 0.9.1 documentation*. URL: https://snntorch.readthe%20docs.io/en/latest/index.html#.

[34] *Spiking Heidelberg digits and Spiking Speech Commands – Zenke Lab*. URL: https://zenkelab.org/resources/spiking-heidelberg-datasets-shd/.

[35] P. Stoliar et al. “Spike-shape dependence of the spike-timing dependent synaptic plasticity in ferroelectric-tunnel-junction synapses”. In: *Scientific Reports* 9 (Nov. 2019). DOI: 10.1038/s41598-019-54215-w.

[36] *Synopsys — VHDL-2008 Support Library 1.0.0 documentation*. URL: https://fphdl.readthe%20docs.io/en/docs/synopsys.html.

[37] *The Phase-2 Upgrade of the CMS Muon Detectors*. Tech. rep. This is the final version, approved by the LHCC. Geneva: CERN, 2017. URL: https://cds.cern.ch/record/2283189.

[38] Thvnvtos. *GitHub - Thvnvtos/SNN-delays: Official implementation of "Learning Delays in Spiking Neural Networks using Dilated Convolutions with Learnable Spacings" [ICLR2024]*. URL: https://github.com/Thvnvtos/SNN-delays.git.

[39] Marco Toffano. *Hardware Implementation of a Spiking Neural Network for online processing of muon detectors datastream*. https://github.com/marcotoffano/SNN_Thesis. 2024.

[40] Pierian Training. *Multi-Label image classification in PyTorch: A guide - Pierian training*. Apr. 2023. URL: https://pieriantraining.com/multilabel-image-classification-in-pytorch-a-guide/.

[41] *Training an audio classification task using Torch — Rockpool 2.7 documentation*. URL: https://rockpool.ai/tutorials/rockpool-shd.html.

[42] *Training on ST-MNIST with Tonic + snnTorch Tutorial — snntorch 0.9.1 documentation*. URL: https://snntorch.readthedocs.io/en/latest/tutorials/tutorial_stmnist.html.

[43] *Tutorial 1 - Spike Encoding — snntorch 0.9.1 documentation*. URL: https://snntorch.readthe%20docs.io/en/latest/tutorials/tutorial_1.html.

[44] Alex Vigneron and Jean Martinet. "A critical survey of STDP in Spiking Neural Networks for Pattern Recognition". In: *2020 International Joint Conference on Neural Networks (IJCNN)*. 2020, pp. 1–9. DOI: 10.1109/IJCNN48605.2020.9207239.

[45] P.J. Werbos. "Backpropagation through time: what it does and how to do it". In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560. DOI: 10.1109/5.58337.