People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

University M'Hamed BOUGARA – Boumerdès

Institute of Electrical and Electronic Engineering

Department of Electronics

Project Report Presented in Partial Fulfilment of

the Requirements of the Degree of

'MASTER'

In: **Electronics**

Option: **Computer Engineering**

Title:

Design and Implementation of Spiking Neural Networks on FPGA for Event-Based Spatio-Temporal Applications

Presented By:

- **BOUMERZOUG Nadhir**
- **ZERRARI Dhia Elhak**

Supervisor:

- **Prof. CHERIFI Dalila**

Co-Supervisor:

- **Prof. KHOUAS Abdelhakim**

Registration Number: 2023/2024

Abstract

Inspired by the intricacies of real biological neural systems, Spiking Neural Networks (SNNs) represent an advanced type of artificial neural network. SNNs operate with discrete spikes, closely mimicking the way neurons communicate in the human brain. This unique method of information processing not only enhances the computational efficiency of SNNs but also opens up new possibilities for developing low-power neural network systems. In this work, we proposed a generic hardware design of an SNN based on Field-Programmable Gate Arrays (FPGA). The proposed design was implemented and tested with the event-based benchmark dataset “Neuromorphic-MNIST”, and managed to achieve a low power consumption and latency, while requiring very minimal hardware resources, all this for an evaluated accuracy.

Keywords
Spiking Neural Networks, Neuromorphic Computing, Spatio-Temporal Pattern, FPGA, RTL design, VHDL

Acknowledgments

First and foremost, we would like to extend our deepest gratitude to God Almighty who provided us with his blessing and the opportunity to successfully conclude our project.

In the successful accomplishment of our final year project titled "Design & Implementation of Spiking Neural Networks on FPGA for Spatio-temporal Applications", we would like to express our deepest gratitude to our supervisor, Prof.CHERIFI Dalila. We are immensely grateful for her valuable advices regarding our research and future career prospects. Her willingness to assist us, steadfast encouragement and continuous support were instrumental in making this project a reality. We deeply appreciate her patience, dedication, and unwavering belief in our abilities.

We would also like to express our appreciation to our co-supervisor, Prof. KHOUAS Abdelhakim for his significant contributions. His expertise and willingness to provide assistance have greatly enhanced the quality of this work.

We would like to thank BOUANANE Mohamed Sadek, for his insightful guidance and help throughout the project duration.

Lastly but not least, we are immensely thankful to our parents and friends for their unwavering support in completing this report. We would like to extend our gratitude to all the individuals who have supported us throughout this journey.

Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
ANNs	Artificial Neural Networks
AER	Address Event Representation
ASIC	Application-specific integrated circuit
BRAM	Block RAM
CNNs	Convolutional Neural Networks
CUBA-LIF	Current-Based LIF
DRAM	Dynamic RAM
DSP	Digital Signal Processor
DVS	Dynamic Vision Sensors
FIFO	First-In-First-Out
FPGAs	Field Programmable Gate Arrays
FSM	Finite State Machine
GPU	Graphics Processing Unit
GPIO	General-Purpose Input/Output
HDL	Hardware Description Language
HDMI	High-Definition Multimedia Interface
HPC	High-Performance Computing
IF	Integrate-and-Fire
JTAG	Joint Test Action Group
LIF	Leaky Integrate-and-Fire
LUT	Look Up Table
MAC	Multiply-and-ACcumulate operation
MNIST	Mixed National Institute of Standard Technology
N-MNIST	Neuromorphic MNIST
PCI	Peripheral Component Interconnect
PLLs	Phase-Locked Loops
RAM	Random Access Memory
RNNs	Recurrent Neural Networks
RTL	Register-Transfer-Level

SCNN	Spiking Convolutional Neural Networks
SNNs	Spiking Neural Networks
SoPC	System-on-Programmable-Chip
SRAM	Static RAM
TTFS	Time-To-First-Spike
UART	Universal Asynchronous Receiver / Transmitter
VHDL	VHSIC HDL

General Introduction

Artificial Neural Networks (ANNs) have garnered significant attention from academia and industry in the last ten years. The abundance of publicly available data and the increased processing power of modern computers have led to a significant increase in interest in ANNs because they have made ANN training and inference very efficient. Particularly deep neural networks have proven to be very effective at tasks like image classification, making them standout instruments in machine learning and artificial intelligence research. However, there are still a lot of obstacles to overcome when it comes to real-time processing on edge devices, like those found in autonomous cars. These difficulties result from traditional hardware implementations' high energy consumption and high cost, which are ill-suited for the computational requirements of artificial neural networks. In contrast, the human brain performs complex cognitive tasks efficiently, using only a few watts of power. Inspired by this biological efficiency, Spiking Neural Networks (SNNs) have emerged as a promising alternative, aiming to achieve energy-efficient machine intelligence by mimicking the brain's neural mechanisms. Despite their potential, there remains a considerable gap in understanding the differences and impacts of various spiking neuron models, particularly concerning their hardware implementation and performance in real-time applications.

This project seeks to address this gap by studying the effect of membrane leakages in different spiking neuron models, which vary in their levels of biological abstraction, on a spatio-temporal classification task. By analyzing metrics such as power consumption, latency, spiking activity, and resource allocation, we aim to provide a comprehensive evaluation of these models' performance on digital hardware. To facilitate this comparative analysis, we propose a generic and efficient digital hardware design for SNN inference. Our focus on inference, rather than training, is driven by the fact that training is highly resource-intensive and not well-suited for hardware implementation. Training a neural network requires substantial computational resources, but this process is only performed once to determine the optimal weights and biases. Once trained, the model can be efficiently deployed for real-time inference on hardware, making this approach more feasible for practical applications.

This report is organized as follows: Chapter 1, introduces the field of neuromorphic computing, discussing the motivations and challenges that have led to the exploration of SNNs. Chapter 2, provides an overview of existing methodologies for SNN inference and introduces the spiking neuron models used in our project. Chapter 3 details the design and implementation of the various building blocks of the SNN, focusing on their application to the handwritten digit classification task. The design is then evaluated in Chapter 4, where various metrics were collected from two experimental setups.

Chapter 1

Overview on Neuromorphic Computing

1.1 Introduction

As the need for Artificial Intelligence (AI) applications continues to grow, a critical challenge arises: power consumption. Traditional computing architectures, such as Von Neuman and Harvard architectures, while capable of handling AI tasks, are known for their massive power consumption. This power hunger becomes most noticeable when working with spatio-temporal data, for instance, speech recognition, image detection, and video processing. Furthermore, these applications demand immense computational resources, often pushing conventional hardware to its limits and beyond. Compounding this issue is the slowdown of Moore's law, the principle that has guided semiconductor technology for decades. As transistor sizes approach physical limits, the time of exponential growth in computing power seems to be coming to an end. As a result, the search for alternative computing paradigms has received a lot of attention from scholars in the field in recent decades. Neuromorphic computing, inspired by the structure and function of the human brain, offers a promising approach into addressing the dual concerns of power consumption and resource utilization. By emulating the very complicated connection and interchange of neurons in the human brain, neuromorphic computing offers the potential to revolutionize AI models. These novel models utilize bio-inspired neurons to perform complex tasks with enough higher efficiency. Moreover, they hold the promise of developing new AI Accelerators that are not only less power-hungry but also more suitable at handling spatio-temporal data, known as "Neuromorphic Chips". Field Programmable Gate Arrays (FPGAs), with their reconfigurable hardware architecture, afford us with the flexibility to design and implement bio-inspired based AI accelerators. Their low power demand, reconfigurability, and parallelism makes them an ideal platform for prototyping and deploying neuromorphic AI models.

1.2 Rethinking Computation

It is noticeable the rapid advancement in computing systems for the last decades. However, such systems are not really efficient in terms of several factors, highlighting the need to explore new computing primitives to address evolving demands and challenges in the digital landscape.

1.2.1 Power Hungry Embedded AI Systems

The growing use of Artificial Intelligence in embedded systems has resulted in a notable increase in power consumption. For instance, the GPT-3 transformer model, which boasts 175 billion parameters, necessitates a minimum of 350GB of GPU (Graphics

Processing Unit) memory just for inference. This translates to the need for approximately 8 Nvidia A6000 GPUs, which are among the most advanced deep learning-oriented High-Performance Computing (HPC) devices available, such a setup consumes around 2400W of power. This is also noticeable in autonomous applications like surveillance cameras, robots, and drones that rely on machine learning models for their operation. The energy efficiency of these systems is important as they depend most of the time on limited-capacity batteries and solar cells.

Consequently, the energy scalability of Machine Learning is becoming a pressing concern within both scientific circles and the general public. This energy issue is particularly acute in the domain of embedded systems, where power efficiency is paramount.

1.2.2 Moore's Law is dead

Moore's Law, named after Intel co-founder Gordon Moore, posits that the number of transistors on a microchip doubles approximately every two years [1]. This has led to an extraordinary 3,500-fold increase in processor speeds over 30 years—from 1 MHz to 5 GHz. In stark contrast, innovations in architecture only achieved about a 50-fold improvement in the same period. It is worth mentioning that Moore's law is not a law of physics; it is more an empirical observation. Although, this principle has driven the exponential growth of the computer science industry for over half a century, transistors are approaching atomic scales, with the smallest ones commercially available being only 3 nanometers wide, the pace of miniaturization has slowed. For instance, it took Intel five years to progress from 14-nanometer to 10-nanometer technology, rather than the two years predicted by Moore's Law. This has led some, including MIT Professor Charles Leiserson, to declare that Moore's Law has been effectively over since at least 2016 [2].

The growing computational hunger of AI models poses a set of significant challenges, including increasing global carbon emissions and the escalating privatization of AI research. In response to these challenges, the semiconductor industry has been investigating future advancements and finding solutions to the death of Moore's Law.

1.2.3 Memory bottleneck

The Von Neumann bottleneck is a fundamental limitation of the traditional computer architecture design. However, it is important to note that this bottleneck is not just a hardware issue. It is also closely tied to the way software is designed and executed. In the traditional Von Neumann architecture, instructions and data are stored in the same memory and processed sequentially [3]. This means that even if we could infinitely increase the speed of our hardware, the sequential nature of software execution could still limit the overall system performance.

1.3 Biological Neuron

The human brain is among the most intricate and complex entities known. It comprises approximately 100 billion neurons and nearly 40 trillion synapses, rendering it one of the most challenging structures to study. Each neuron can form thousands of synaptic connections, resulting in an incredibly dense and interconnected network. This complexity is further enhanced by the diversity of neuronal types, with estimates suggesting the presence of over tens of thousands of distinct types, each contributing uniquely to the brain's functionality.

The brain's ability to perform a vast array of functions with remarkable efficiency is a subject of great scientific interest. Despite its complexity, it operates with minimal power consumption, utilizing approximately 20 watts only. This efficiency is partly due to the brain's sophisticated mechanisms for energy conservation and optimization in neural processing.

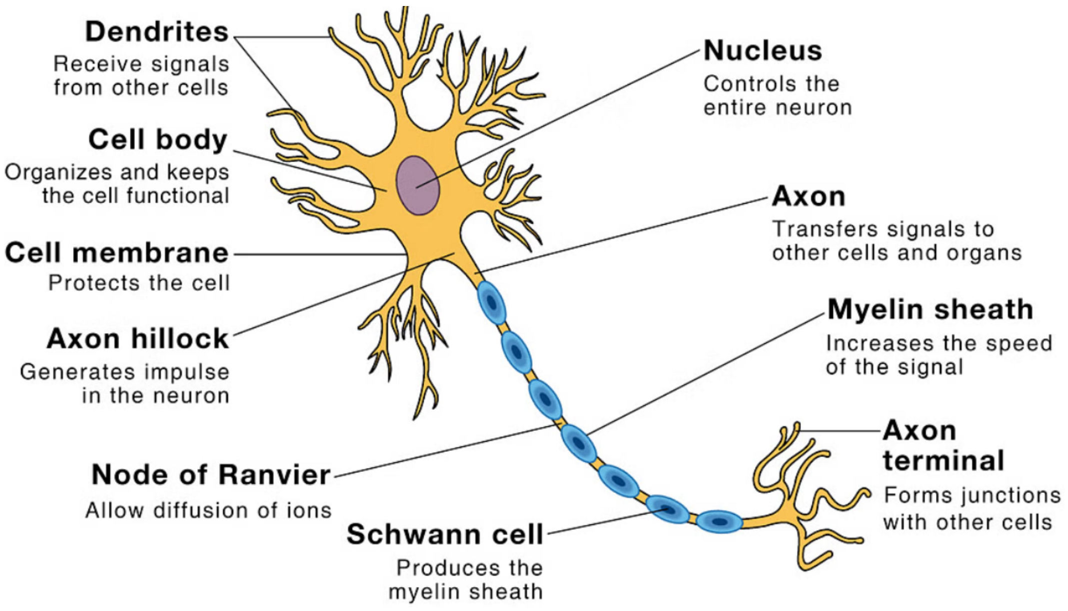
Dendrites
Receive signals from other cells
Cell body
Organizes and keeps the cell functional
Cell membrane
Protects the cell
Axon hillock
Generates impulse in the neuron
Node of Ranvier
Allow diffusion of ions
Schwann cell
Produces the myelin sheath
Nucleus
Controls the entire neuron
Axon
Transfers signals to other cells and organs
Myelin sheath
Increases the speed of the signal
Axon terminal
Forms junctions with other cells

Figure 1.1: The anatomy of a biological neuron [4].

A biological neuron, also known as a nerve cell, is the fundamental building block of the nervous system. Neurons are specialized cells responsible for transmitting and processing information throughout the body, enabling complex functions such as sensation, thought, movement, and homeostasis. They achieve this through electrochemical signaling, which involves the generation and propagation of electrical impulses and the release of neurotransmitters.

Figure 1.1 depicts a simplified anatomy of the biological neuron. Each neuron consists of three main parts: the cell body (soma), dendrites, and an axon. The cell body contains the nucleus and other organelles essential for the neuron's metabolic activities. Dendrites are branched extensions that receive signals from other neurons and convey them towards the cell body. The axon is a long, slender projection that transmits electrical impulses away from the cell body to other neurons, muscles, or glands. The axon often ends in a series of terminal branches, each of which forms synaptic connections with target cells.

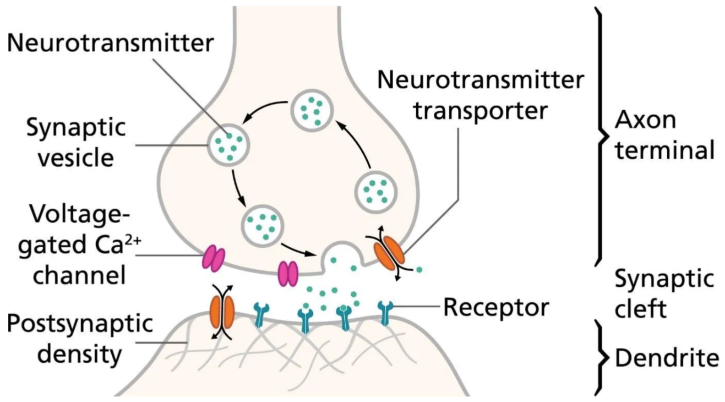

Figure 1.2: Schema of synaptic transmission [4].

As shown in Figure 1.2, Neurons communicate through synapses, specialized junctions where the axon terminal of one neuron comes into close proximity with the dendrite or cell body of another. When an electrical impulse, or action potential, reaches the synaptic terminal, it triggers the release of neurotransmitters into the synaptic cleft. These chemical messengers bind to receptors on the post-synaptic neuron, initiating a response that can either excite or inhibit the generation of a new action potential.

1.4 Human Visual System

The human visual system, particularly the retina, is an exemplar of efficient information encoding, reducing input from approximately 125 million photoreceptors to output through just 1 million ganglion cells. This compression is achieved by organizing photoreceptors into receptive fields of various sizes, each connected to a ganglion cell. The structure of these fields, with center and surround cells, allows ganglion cells to convey spatial contrast by comparing the differential firing rates within their receptive field. Ganglion cells can fire independently and maintain a spontaneous firing rate, ensuring continuous transmission of visual information via the optic nerve to the brain. This inherent efficiency and specialization in contrast extraction have inspired the field of neuromorphic engineering, which aims to mimic these biological processes in artificial systems. Pioneered by Carver Mead in the late 1980s, neuromorphic engineering gained significant momentum with Misha Mahowald's creation of the first silicon retina, designed to emulate the human retina's center-surround receptive fields. Subsequent advancements by Tetsuya Yagi and Tobi Delbrück led to the development of refined temporal contrast sensors, known as event cameras.

1.5 Event-Based Vision Sensors

Current cameras acquire frames by reading the brightness value of all pixels at the same time at a fixed time interval, the frame rate, regardless of whether the recorded information has actually changed.

Trying to take inspiration from the way our eyes encode information, neuromorphic cameras capture changes in illuminance over time for individual pixels corresponding to one retinal ganglion cell and its receptive field.

If light increases or decreases by a certain percentage, one pixel will trigger what's called an event, which is the technical equivalent of a cell's action potential. One event will have a timestamp, x/y coordinates and a polarity depending on the sign of the change. Pixels can fire completely independently of each other, resulting in an overall firing rate that is directly driven by the activity of the scene. It also means that if nothing moves in front of a static camera, no new information is available hence no pixels fire apart from some noise. The absence of accurate measurements of absolute lighting information is a direct result of recording change information. This information can be refreshed by moving the camera itself, much like a microsaccade [5].

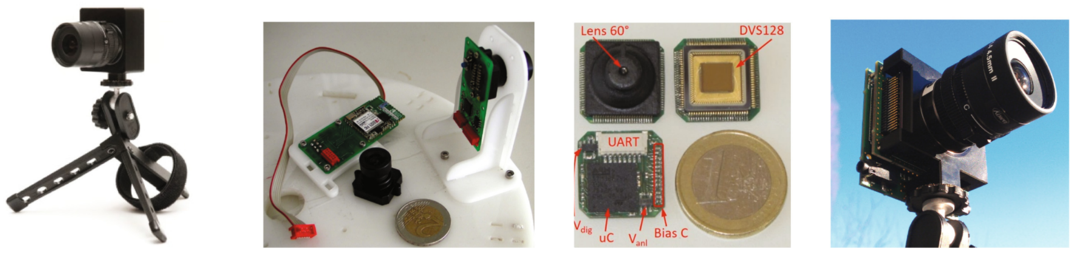

Figure 1.3: Different types of Dynamic Vision Sensors [6].

1.6 AI Generations

Our brains, with their billions of neurons firing in intricate patterns, have long been a source of wonder and inspiration. Traditional Artificial Neural Networks (ANNs) have attempted to replicate its complexity, but they often fall short in terms of energy efficiency and real-time processing. This yields in introducing the third generation of neural networks, a more bio-inspired neuron based network architecture, known as Spiking Neural Networks (SNNs). Unlike conventional neural networks, which rely on continuous firing rates, SNNs communicate through discrete spikes, similar to the way neurons in our brains function. This biologically mimicking approach holds the potential to revolutionize computing, offering unparalleled efficiency and capabilities [7].

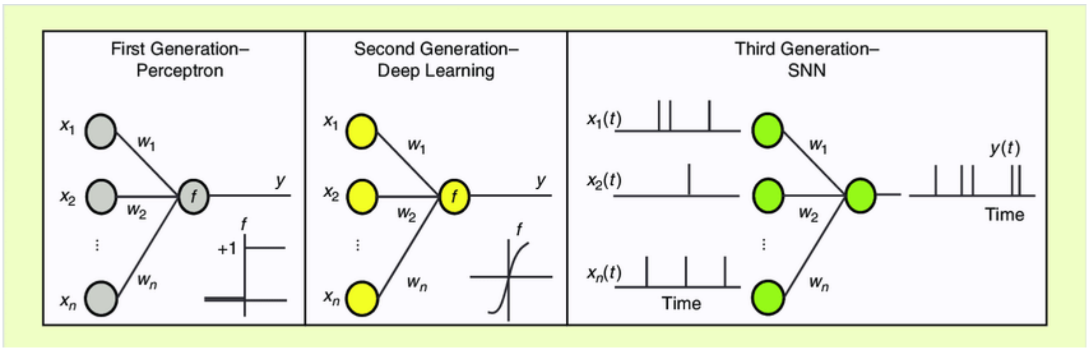

Figure 1.4: The three generations of neural networks [8].

1.7 Summary

In this chapter, we discussed the challenges of traditional computing architectures in handling AI tasks, especially regarding power consumption and efficiency. We introduce neuromorphic computing as a promising solution, inspired by the human brain, to address these issues. The chapter covers the limitations of Moore's law, the biological basis of neurons, and the potential of neuromorphic chips and event-based vision sensors in advancing AI technology.

Chapter 2

Overview of SNN for Inference Methodologies

2.1 Introduction

Existing research suggests that ANNs still outperform SNNs in terms of accuracy [9], but the bio-plausibility, power efficiency and small hardware footprint [10] of SNNs continue to attract researchers and manufacturers interested in bringing AI to edge devices, where power consumption and chip area come at a premium.

Hardware circuits differ significantly from software execution, thus several components need to be adapted or even redesigned when tackling the challenge of transitioning from a software proof-of-concept, to a digital hardware implementation. Hardware design emphasizes attention to detail, whether it is the implementation of basic math operators, or the integration of large scale multi-chip systems.

2.2 Artificial Neuron

Artificial neurons are the basic building blocks of artificial neural networks, which are computational models loosely inspired by the human brain. These neurons, also known as nodes or units, are designed to simulate the way biological neurons process and transmit information. In an artificial neuron, multiple input signals are received, each associated with a weight that signifies its importance. The neuron computes a weighted sum of these inputs and then applies an activation function to produce an output. This output is then passed on to subsequent neurons in the network. The most common activation functions include the sigmoid function, hyperbolic tangent (tanh), and Rectified Linear Unit (ReLU). These functions introduce non-linea-rity into the model, enabling the network to learn complex patterns and make sophisticated predictions. Artificial neurons are the fundamental components of various neural network architectures, such as feedforward neural networks, convolutional neural networks (CNNs), and recurrent neural networks (RNNs), which are applied in fields like image recognition, natural language processing, and autonomous systems.

The design of artificial neurons was primarily pragmatic, iterations upon artificial neurons and ANNs sought only to improve inference performance with little regard to bio-plausibility. ANNs meet or even exceed human capability in specific classification tasks, but the field is facing constant hurdles in terms of power consumption and chip area. Many researchers branched out towards mimicking the internal processes of the brain, in hopes of better replicating its energy efficiency and well-rounded performance over innumerable tasks.

2.3 Spiking Neuron Models

Spiking neurons represent a more biologically realistic model of neural activity compared to traditional artificial neurons. Inspired by the behavior of biological neurons, spiking neurons communicate through discrete events known as spikes or action potentials. Instead of continuously varying outputs, a spiking neuron emits a spike when its membrane potential reaches a certain threshold. This event-based communication mirrors the way neurons in the brain operate, making spiking neural networks (SNNs) a powerful tool for modeling biological neural processes.

Many researchers in the field of neurophysiology sought, and continue to seek, to identify and quantify the exact processes of the human brain. Significant progress has been made in the area of modelling individual neurons, with many models being proposed based on bio-plausibility, performance, ease of implementation and other criteria.

2.3.1 Hodgkin-Huxley Model

The Hodgkin-Huxley (HH) model is a foundational mathematical framework that describes how neurons generate and propagate action potentials (electrical signals). Developed by Alan Hodgkin and Andrew Huxley in 1952, the model is based on experimental data from the giant axon of the squid and has been highly influential in the field of neurophysiology. It represents the neuron as an electrical circuit with specific components that mimic the biological behavior of ion channels and membranes.

The Hodgkin-Huxley model comprises a set of nonlinear differential equations that describe the dynamics of the membrane potential V_m in response to ionic currents. These equations are derived from the equivalent electrical circuit of a neuron demonstrated in Figure 2.1.

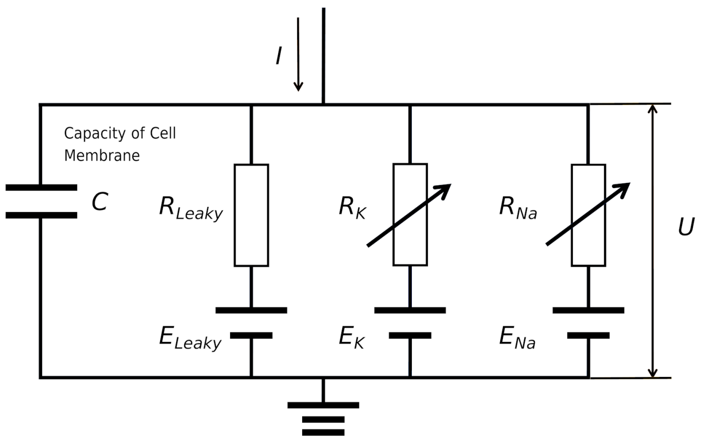

Figure 2.1: Equivalent electrical circuit of the Hodgkin-Huxley neuron [11].

The neuron is modeled as a capacitor C that takes in the sum of the synaptic currents I, in parallel with a leak resistor R_{leaky} in order to model the exponential decay of membrane potential U over time. When the potential across the membrane capacitor reaches a threshold value, the neuron fires an output spike and enters the refractory period where its potential drops to the refractory level, often well below the typical resting potential. The model also considers the ionic currents resulting from potassium R_K and sodium R_{Na} ions permeating into and out of the membrane [12].

The total current I flowing across the membrane is the sum of the capacitive current and the ionic currents, hence it can be described by equation 2.1:

$$I = C_m \frac{dV_m}{dt} + I_{Na} + I_K + I_{Leaky} \tag{2.1}$$

Where:

- C_m is the membrane capacitance per unit area.
- dV_m is the rate of change of the membrane potential.
- I_{Na}, I_K, and I_{Leaky} are the ionic currents through sodium, potassium, and leak channels, respectively.

Despite its age, the HH model remains one of the most biologically plausible neuron models to date. However, its accuracy comes at the cost of its implementation complexity. Many spiking neural networks implement a subset of the HH model instead, while still achieving favorable results.

2.3.2 Leaky Integrate-and-Fire (LIF)

The Leaky Integrate-and-Fire or LIF neuron is a simplification of the Hodgkin-Huxley model, where the effects of the ionic currents are neglected due to their apparent insignificance.

The LIF model reduces the neuron down to just the membrane capacitor and parallel leak resistance, and features similar refractory period mechanics. The differential equation of the LIF neuron model is then given by:

$$\tau_{\text{mem}} \frac{\mathrm{d}U(t)}{\mathrm{d}t} = -\left(U(t) - U_{\text{rest}}\right) + RI(t) \tag{2.2}$$

Where τ_{mem} is the membrane's RC time constant.

The circuit parameters were shown experimentally to differ significantly between neurons, even those taken from the same biological samples. The values of the circuit parameters affect the neuron's tendency to spike, and hence alters its output characteristics. Hence, the training process of a spiking neural network involves finding the best

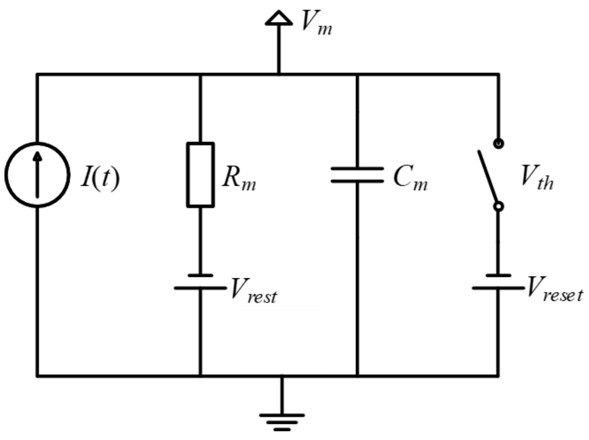

Figure 2.2: Equivalent circuit of the LIF neuron
[13].

circuit parameters, or "weights", for each neuron in the network in order to achieve accurate predictions or classifications.

In the context of discrete algorithmic design, both in software and in hardware, it is more convenient to consider the difference equation of the LIF neuron instead [14]:

$$I_i^{(l)}[t] = \sum_j W_{ij}^{(l)} S_j^{(l-1)}[t-1] + \sum_j V_{ij}^{(l)} S_j^{(l)}[t-1] \tag{2.3}$$

$$U_i^{(l)}[t] = (\beta U_i^{(l)}[t-1] + I_i^{(l)}[t]) \times (1 - S_i^{(l)}[t-1]) \tag{2.4}$$

Since for a Feed-Forward Neural Network architecture (FFNN), the reccurency term is removed, hence further simplification are made to result in equations 2.5 and 2.6

$$I_i^{(l)}[t] = \sum_j W_{ij}^{(l)} S_j^{(l-1)}[t-1] \tag{2.5}$$

$$U_i^{(l)}[t] = (\beta U_i^{(l)}[t-1] + I_i^{(l)}[t]) \times (1 - S_i^{(l)}[t-1]) \tag{2.6}$$

Where W_{ij}^l is the weight associated with the particular spike input j of a neuron i in a layer l of a neural network. β is the exponential decay factor of the membrane potential.

2.3.3 Integrate-and-Fire (IF)

The Integrate-and-Fire or IF model further optimizes the spiking neuron down to its most fundamental operation: the accumulation of weights, and firing a spike if a threshold is reached. It is represented by the following difference equations:

$$I_i^{(l)}[t] = \sum_j W_{ij}^{(l)} S_j^{(l-1)}[t-1] \tag{2.7}$$

$$U_i^{(l)}[t] = (U_i^{(l)}[t-1] + I_i^{(l)}[t]) \times (1 - S_i^{(l)}[t-1]) \tag{2.8}$$

2.4 Spike Train Encoding

To make use of SNNs, information needs to be encoded into, or decoded from spike trains. There exist multiple approaches to modulate information onto spike trains, the primary contenders being rate coding and temporal coding, specifically time-to-first-spike (TTFS) encoding. In rate coding, the number of spikes registered over a period of time is proportional to the intensity of the stimulus, while in TTFS more emphasis is placed on the exact time that a given spike was registered, where earlier spikes correspond to stronger stimuli. Rate coding tends to outperform temporal coding accuracy wise, especially in smaller datasets [15]. However, temporal coding continues to be an active topic of research, such as in the work of Mostafa [16] where temporal coding proves to be a potential avenue towards differentiable SNNs that can be trained using conventional gradient-descent techniques.

2.5 SNN Topology

The basic architecture of typical SNNs closely mimics that of conventional ANNs, with the only major difference being the structure of each neuron. In fact, a common method of SNN training is the ANN-to-SNN conversion method, where an equivalent ANN is constructed and trained, from which the resulting weights are quantized and deployed to the target SNN [9]. Most SNNs consist of an input layer, an output layer and one or more hidden layers, with certain implementations making use of additional convolutional and pooling layers as is the case in spiking convolutional neural networks or SCNNs [15, 17].

2.5.1 Input Layer

The input layer neurons feed directly into the synapses of neurons in the first hidden layer, and thus the input layer serves more as an interface to distribute incoming spike events without performing any processing. Each neuron in the input layer represents a tangible feature in the input data, for example the intensity of a specific pixel in an image. The generation of input layer spikes occurs either in an event-based sensor, or using encoders that convert data from conventional sensors into spike trains. In most digital hardware implementations, the input layer is a simply a buffer (queue), constructed using either logic elements or block RAM, that the input spike trains are latched onto before being analyzed further in the hidden layer.

The transmission of sample data to the neural network can severely affect its latency, i.e. the time it takes for the network to guess the correct output after being fed an input sample. Multiple interfaces exist for connecting FPGAs either to computers or even to sensors producing live data, ranging from simple chip-to-chip serial communication protocols such as UART, to high-speed network-based interfaces such as Etherent, to direct communication with a computer's resources using low-level busses such as PCI Express.

2.5.2 Hidden Layer

The hidden layer neurons perform feature extraction, which is the identification of relevant abstract attributes in the incoming data in order to make a prediction or classification. Feature extraction is achieved through a linear combination of input data and weights obtained during the training process.

In conventional ANNs, the combination of input data with the weights of all the neurons in the layer can be mathematically interpreted as a matrix multiplication, the latter itself is a series of vector dot products performed over the various rows and columns of inputs and weights respectively. Finally, at the heart of the dot product is the multiply-and-accumulate operation (MAC) which multiplies corresponding elements in each vector and adds up each result in a running summation. While hardware circuits generally benefit from increased concurrency by performing multiple tasks in parallel, multiplication is considered a relatively expensive operation. Care must be taken when designing hardware multipliers by balancing multiple constraints such as latency, chip area, power consumption and portability of designs. The complexity of the design is exasperated when attempting to implement more parallel multipliers.

In contrast, the binary nature of spike inputs in an SNN allows us to completely bypass this problem, reducing the MAC operation down to a conditional accumulation of weights, based on the existence or lack of a spike at a given weight index [18]. The design of such accumulation process is trivial, especially when coupled with equally simple numerical representations. In an SNN, the accumulated weights are added to the membrane potential of the neuron, each neuron then makes a decision on whether to spike or not in the current time-step based on whether the new membrane potential crosses a preset threshold. The membrane potential of spiking neurons is reset to zero, it can also be reset to a level below zero in order to emulate the refractory period present in biological neurons.

When IF neurons are used, the new membrane potential is stored directly to some

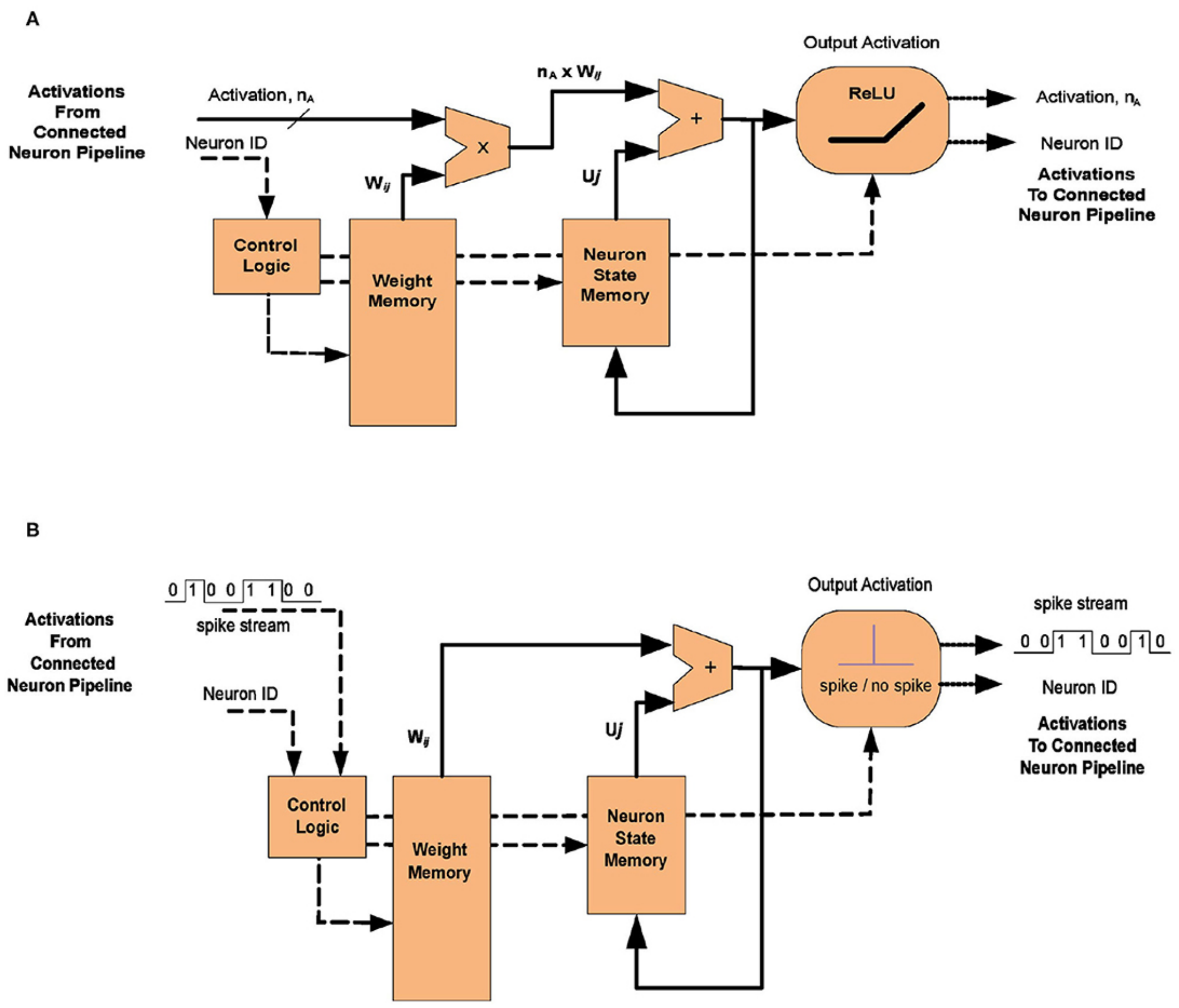

Figure 2.3: (A) Typical Artificial neuron pipeline (B) An equivalent Spiking neuron system [18].

form of state memory, often implemented in block RAM, so that it can be used for spike calculation in the next time-step. LIF neurons operate in mostly the same principle, with an extra added step of multiplying by a decay factor β before memory write-back. The decay factor depends on the time-step, hence in variable time-step systems a look-up table is often used to fetch values from an exponential function. Some optimizations were proposed, such as approximating the exponential decay with an arithmetic right shift, which was implemented in the modified LIF neuron by Reddy at al. [19].

Hidden layer neurons typically feature a large number of input connections, a numerical weight value is assigned to each input as part of the training process. The storage of the weight matrix is an important design decision with impacts on both performance and power consumption. The naive approach would be to encode the weights directly using distributed RAM (i.e. within the logic fabric), this allows multiple access of the stored weights and can improve parallelism in theory, bringing the architecture closer to memory-in-compute levels of latency. But this approach is not scalable for non-trivial neural network. Instead, either on-chip block RAM or off-chip DRAM are used to store the weights, granting much higher flexibility when it comes to neuron counts.

Multiple SNN implementations have experimented with different neuron counts and even multiple successive hidden layers, such as in the work of Sankaran et al. [20] which reveals that increasing neuron counts in the hidden layers of an SNN can help improve accuracy, but with diminishing returns relative to significant increases in resource utilization. For image classification tasks, some SNNs incorporate additional convolutional layers like in the work of Zhang et al. [17], where input spikes are grouped into 2D spike maps and then convolved by a fixed kernel, the results of this operation are further aggregated into pooling layers, which down-sample the dimensions of input data while retaining important information.

2.5.3 Output Layer

The final layer in a neural network is the output layer, with neuron counts corresponding to the number of possible classification outcomes, the output of each neuron represents a likelihood that the associated classification decision is the correct one. For example, in digit recognition tasks, ten neurons are used, each neuron corresponding to a specific digit ('0' through '9'). Contrary to ANNs, which typically employ some form of a "softmax" function to extract outcome probabilities, SNNs were combined with a variety of output decoding techniques, often corresponding to the encoding scheme used to generate the input spike trains in the first place. For rate coded SNNs, the most likely output is that which corresponds to the highest frequency spike trains, while in TTFS SNNs, the first output neuron to spike is considered the winning output.

The design of output layer neurons tends to be identical to that of hidden layer neurons, but one alternative approach involves the use of non-spiking neurons on the output layer. These neurons accumulate their weights onto the membrane potential as is the case with the aforementioned hidden layer neurons. However, their membrane potential is allowed to grow beyond the spiking threshold without any reset. This approach is not present in biological neurons, but allows for easier training as membrane potential growth features less discontinuities than spike activations. The winning neuron in this approach is simply that with the highest peak recorded membrane potential.

2.6 Address Event Representation

The spike-train communication between neurons within an SNN can be modelled by digital systems with relative ease. Rather than receiving multi-valued (bit vector) inputs from pre-synaptic neurons, spiking neurons operate with single-bit inputs, each input can only be either 0 (no spike) or 1 (spike). The vast reduction in the required data lines not only reduces gate count and routing cost, but also opens the door to alternative techniques of spike transmissions.

Many event-based sensors have adopted the Address Event Representation (AER) protocol, where instead of assigning a unique bit input coming from each neuron, the AER protocol transmits the address or index of the pre-synaptic neuron that spiked at any given moment, ignoring any neurons that did not spike [21]. In effect, this allows the neuromorphic system to simply skip weight calculations for non-spiking neurons, vastly improving efficiency and bio-plausibility. The on-demand nature of neuromorphic systems corresponds to reduced switching in an equivalent digital system, a key strategy in reducing power consumption. Finally, the AER protocol can be scaled up to arbitrary neuron counts with a mere logarithmic increase in the required bit width for transmission, making larger neuron layers more feasible.

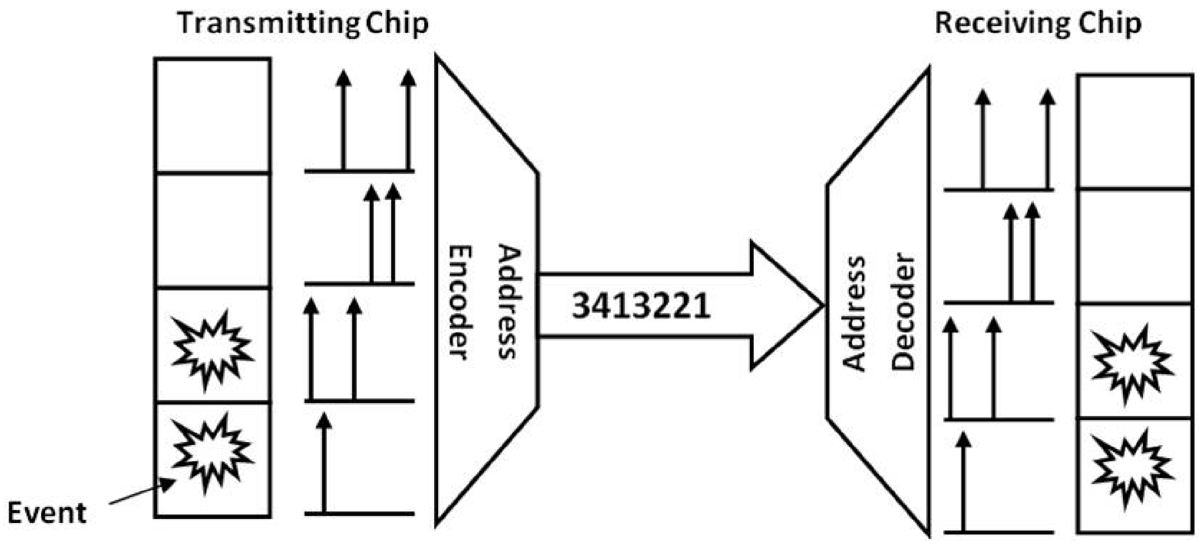

Figure 2.4: Address Event Representation for chip-to-chip communication. [22]

2.7 Spatio-Temporal Data

The inner workings of the brain are directly influenced by the spiking-nature of our various senses, including our sense of sight as it is more event-oriented in nature rather than behaving like a conventional digital camera. Even when passively observing a static scene, the subtle timing differences in spike events registered in the eyes are of utmost significance to the human vision system. Replicating the brain's processing prowess necessitates the interpretation of the world's data the way a brain would experience it.

2.7.1 Conventional MNIST

The MNIST (Modified National Institute of Standards and Technology) dataset is a collection of 28x28 pixel grayscale images depicting hand-written digits, it was first proposed and used by LeCun et al. [23] as a filtered variant of the US NIST dataset. The dataset is widely used for testing various digit recognition techniques, including both DSP algorithms and machine-learning-based approaches.

The MNIST dataset only offers a spatial component and lacks any temporal information, but through its popularity among the research community, it serves as a benchmark for evaluating ML vision models under development, and comparing their accuracy against existing models. This implored researchers in the area of neuromorphic computing to adapt the dataset into spatio-temporal (event-based) formats, with different approaches being explored across the literature.

2.7.2 Event-based MNIST

The early development of neuromorphic computing systems was stunted by a shortage of neuromorphic datasets to train and test on. A common approach to bypass this problem is to generate that data synthetically by converting conventional datasets into event-based datasets, such as the case of Sequential MNIST [24]. However, the performance disparity between ANNs and SNNs is often attributed to the disadvantage that SNNs face by processing data that is not natively suitable for them [25]. Hence, development efforts were directed towards the native generation of neuromorphic datasets, these datasets were often open-sourced in hopes of reinvigorating SNN research.

Event-based (spiking) datasets remain relatively sparse, but the technology behind them continues to develop. A recent development by See et al.[26] involved the use of tactile (touch) events rather than vision, culminating in the spiking-tactile-MNIST dataset. Pixels from MNIST samples were replicated by touch events recorded on a ″taxel″ (tactile pixel) array constructed using a piezo-resistive thin film, a relatively common material used in pressure and strain gauge sensors. This approach aims to improve the training of high-precision robots, but the time domain sparsity of touch events could increase latency for computer vision systems beyond acceptable levels.

Advancements in event-based camera sensors led to the creation of natively event-based image datasets such as MNIST-DVS [27], where traditional MNIST samples are moved on a computer monitor, and recorded by a fixed dynamic vision sensor into spike trains. This approach was further developed for NMNIST [28].

2.7.3 Neuromorphic-MNIST

The Neuromorphic MNIST, or N-MNIST, is a neuromorphic vision dataset containing MNIST samples recorded using a dynamic vision sensor. The dataset can be considered to be an evolution of MNIST-DVS, but its experimental setup was significantly altered.

MNIST samples were displayed on a computer monitor, but they were left static. Instead, the DVS itself is moved against the screen in a periodic manner using a custom-built pan-tilt mechanism [28]. This configuration more accurately imitates micro-saccades in the human eye, which are involuntary micro-movements believed to counteract visual decay when fixating on static objects [29]. This approach also eliminates certain artifacts in the data due to the monitor's refresh behavior.

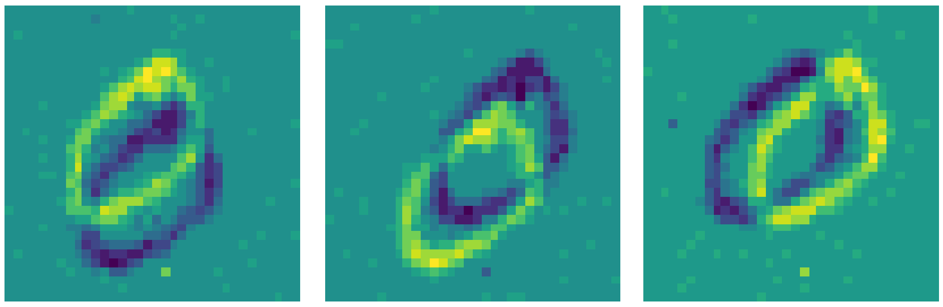

Figure 2.5: Digit 0 sample from the N-MNIST dataset.

Each sample in NMNIST is composed of three saccades, each saccade lasts about 100 milliseconds and consists of spike recordings of DVS pixel activation and deactivation events, with a spiking pixel resolution of 34 by 34. Figure 2.5, shows a sample of digit 0, after the accumulation of events for each of the three saccades. Each event is encoded as an (x, y, t, p) tuple, where x and y are the image coordinates of the event, t is the timestamp at which the event occurred (where time starts at the beginning of the current sample recording), and p is the polarity of the event: ”On” events are denoted by 1, while ”off” events are denoted by a 0. Pixels that did not experience a change in the current timestamp are not recorded, significantly reducing the required bandwidth for visual data transmissions.

2.8 Numerical Representation

Digital hardware ultimately deals only in zeroes and ones, and the basic laws of boolean algebra. To do conventional math, it needs to be framed in the context of binary values. The framing or representation of real numbers is of particular nuance, as many representational systems exist, each with their own advantages and hurdles.

2.8.1 Floating-point numbers

Floating point numbers are a way to represent real numbers in computing, allowing for the representation of a wide range of values with varying degrees of precision. The IEEE 754 standard is the most widely adopted method for encoding these numbers, ensuring consistency and accuracy across different computer systems and applications. This standard specifies the format for floating-point arithmetic, detailing how numbers are stored in binary using three components: the sign, the exponent, and the mantissa (or significand). By allocating specific bits to each of these components, IEEE 754 allows for the representation of both very large and very small numbers, maintaining precision within a defined range. Additionally, the standard outlines rules for rounding, handling exceptions such as overflow and underflow, and defining special values like NaN (Not a Number) and infinity.

IEEE754 provides a great deal of flexibility and features used across different domains. But a full implementation of the exact standard that accommodates such features is a monumental task typically assigned solely to venerable hardware IP design companies. Such complex standard will also tend to consume more hardware resources and have higher latencies compared to regular integer arithmetic.

2.8.2 Fixed-point numbers

Fixed-point representation is an alternative numerical format for performing arithmetic using fractional numbers without the need for dedicated floating-point hardware.

The principle of fixed-point representation is to interpret a regular two's complement binary number, which would typically represent an integer, with a virtual decimal point. This decimal point creates an inherent shift of the powers of two represented by the individual bits, meaning that a portion of the integral range is traded for that of a fractional range represented by negative powers of two. In this report, we default to the Q notation for denoting fixed-point representations, first defined by Texas Instruments [30]. The Q notation uses the format "Q(n-S).S" where n is the number of bits in the entire binary string, and S is the number of bits allocated for the fractional part.

An n-bit binary string in the integer two's complement format can represent any whole number in the range of $\{-2^{n-1} \mathinner{..} 2^{n-1}-1\}$, with a step size between any two consecutive numbers (i.e. the resolution) of the smallest available power of two in the representation, that being $2^0 = 1$. When fixed-point format is employed, the aforementioned range is right shifted by the chosen number of bits allocated for the fractional part. The right shift is equivalent to a division by 2^S, resulting in a new representational range of $\left\{-2^{n-S-1} \mathinner{..} 2^{n-S-1}-2^{-S}\right\}$, with a minimum resolution of 2^{-S}. For a concrete demonstration, an example of unsigned 8-bit integer binary and unsigned Q4.4 fixed-

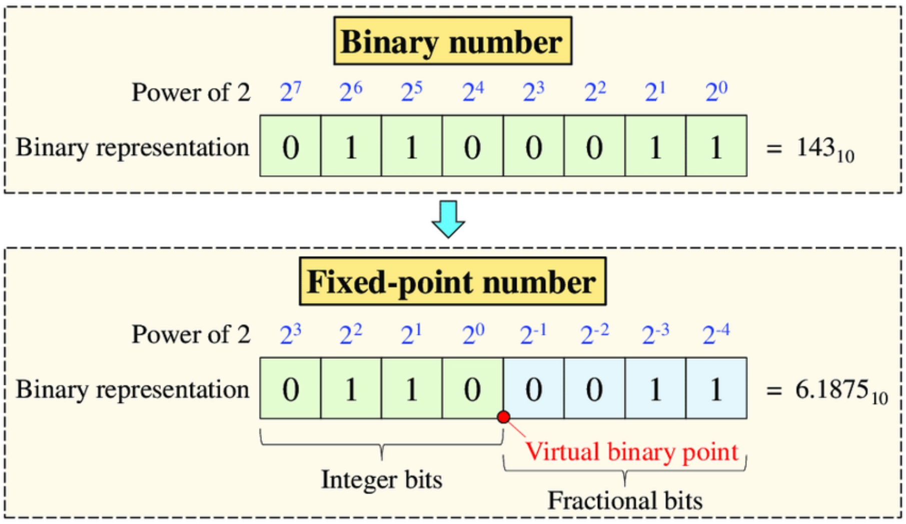

Figure 2.6: Example of fixed-point number representation [31].

point representations is shown in Figure 2.6.

The primary advantage of using fixed-point format is the fact that the decimal point is entirely virtual. There is no additional encoding overhead, and well-established integer arithmetic circuits (both unsigned and two's complement signed) can be reused without any modification, save for the re-scaling of any constants being used, resulting in faster performance, lower power consumption and smaller chip area compared to a floating-point system for a given clock frequency. The main disadvantage of fixed-point format is the limited range of representation compared to floating-point formats, in which as their name suggests the decimal point is free to move to better accommodate the numerical data being processed, with decent resolution and encoding efficiency across a wide spectrum of magnitudes.

2.9 Summary

In this Chapter, we described the different techniques and methodologies used in the topology of Spiking Neural Networks for inference. We also introduced a widely used spatio-temporal benchmark dataset; Neuromorphic-MNIST. In the next chapter, we will derive and explain our proposed design for the implementation of this neural network type on a hand written digit classification task.

Chapter 3

Design and Implementation of a Hardware SNN

3.1 Introduction

The implementation of SNNs on hardware is still a relatively immature field, the literature showed that there is often still points that can be improved upon [32]. We hope to address some trouble points with our own SNN design, which aims to be compact, power efficient and highly portable and adaptable. This chapter goes into detail about the design of the spiking neuron itself, and all the needed infrastructure and supporting logic that is used to ensure fault-free context-switching and high-accuracy inference capabilities.

3.2 Design of SNN Architecture

The key design choice of our SNN is the use of one physical neuron for each layer (hidden and output layers). Through the use of time-division multiplexing and context-switching, a single neuron circuit can be used to perform the needed calculations for all the neurons in the neural layer. This neuron shall be contained within a ″wrapper″, which keeps track of the network neuron that is currently executing, and managing its state data accordingly, the wrapper in effect represents one layer in the neural network. We also designed an output encoder that converts the output layer's weight accumulation activity into tangible inference predictions. We will also define an abstraction that represents the entire neural network, whose role is to assign time slots to the network layers to ensure that the processing occurs in the correct order and with no shared resource contention.

3.2.1 Neuron Model

The neuron model is the fundamental processing block of our SNN. Two neuron designs were conceived, an IF neuron and a LIF neuron, where the latter is functionally an extension of the former. Given the simplicity of the accumulation process, more emphasis was placed on the spike-train/weight-procurement pipeline.

- **IF Neuron**

  The primary function of the designed Integrate-and-Fire Neuron is to accumulate weights and generate an output spike. Initially, the neuron subsystem loads the membrane potential from the previous time frame $U_i^{(l)}[t-1]$. It then accumulates the weights $W_{ij}^{(l)}$ of the received spike indices for the current time frame, in addition to the previously loaded potential. For hidden layer neurons, an output spike is generated when the accumulated potential exceeds the set threshold potential, before resetting to zero. Simultaneously, a valid signal is asserted to indicate the completion of neuron processing, enabling the transition to the next neuron and update of the state table. For non-firing readout layer neurons, the action potential is used as an output instead. The architecture of the IF neuron is shown in Figure 3.1, it should be noted that for the case of the IF neuron, β is taken as one and no multiplication is required.

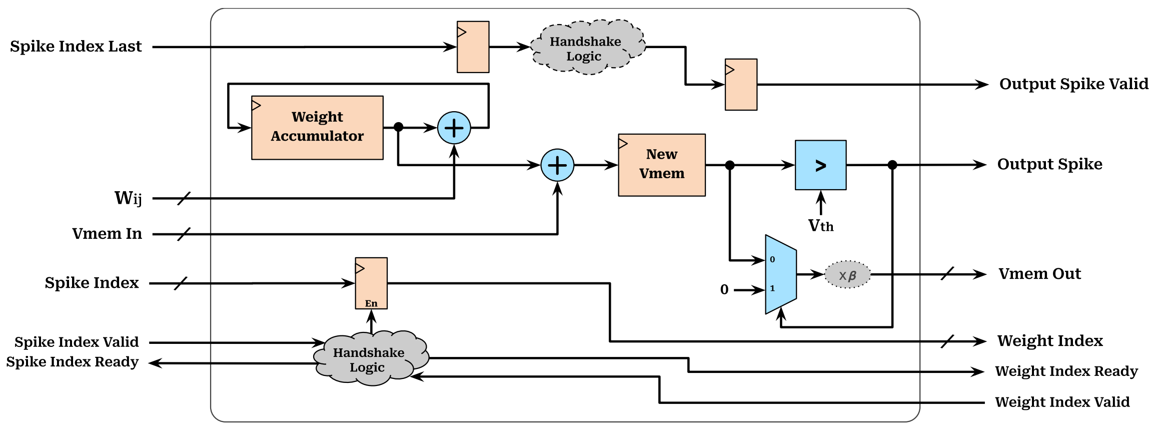

Figure 3.1: IF/LIF neuron model architecture.

The ready-valid handshake is used throughout the design, the key principle of the handshake is that data transfer can only occur when the sender has valid data, and the receiver is ready to receive said data at the same time. This allows for composable interface design with minimum assumptions. The efficiency of handshakes is improved by including a one element buffer to help smooth out spike index and weight transfers, and avoid data stalls. The architecture of the handshake logic cloud introduced in this neuron model design shown in Figure 3.1, is demonstrated in Figure 3.2. The same handshake circuit is used for the rest of different modules design.

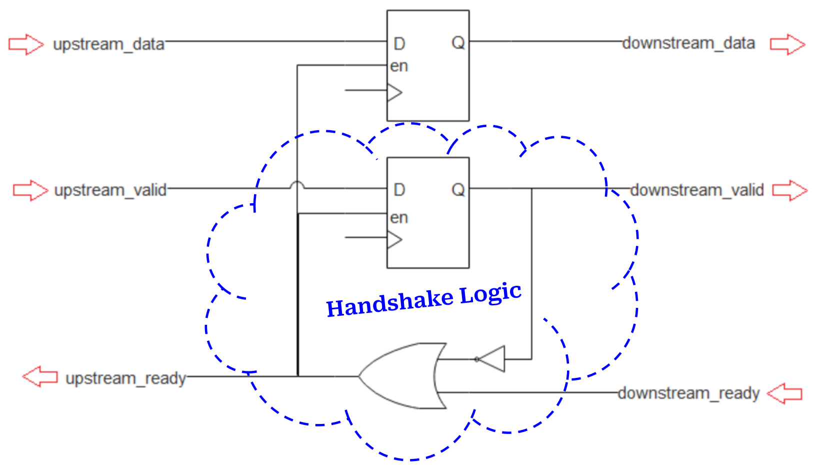

Figure 3.2: Handshake logic architecture.

- **LIF Neuron**

With the establishment of the IF neuron design, we turned our focus to the LIF neuron. The LIF neuron features a near identical architecture to that of the IF neuron, except for an added multiplier that multiplies the resulting membrane voltage output by a decay factor β.

The β factor was designed as a constant. To accommodate this, the input spike indices are grouped into frames such that they have an equal temporal spacing between successive frames, eliminating the need for implementing an exponential function block in hardware as we only require one specific value for our test dataset. We also refrained from utilizing bit-shift techniques that approximate exponentiation in order to have more control over the exact value of β.

3.2.2 Spike Index Table Design

The spike index table is the backbone of our spike-stream oriented architecture. It serves as a buffer that stores spike events occurring in a given layer in the address-event representation (AER) format, meaning that it instead of referencing all the neurons in the layer, it is meant to store only the indices (addresses) of the neurons that spiked in the previous time frame.

The table will provide a ready-valid interface for reading and writing spike indices. The indices are read in the same order they were written, which would suggest a First-In-First-Out (FIFO) principle of operation. But our table differs from a typical FIFO memory in that the elements that are read are not discarded; once all the indices in the current frame have been read, the table will roll over and begin re-transmitting the same set of indices from the beginning. This table was designed to complement the neuron model, which itself is designed to stop reading spikes events if it detects the ″last input spike″ flag. Evidently, the table will need provide said flag, indicating the end of a frame. The table will also need a control signal to entirely clear the table in preparation for a new frame.

Our proposed design takes the FIFO inspiration and tweaks it to fit our requirements. It consists of storage RAM, a memory read pointer, a size register and additional glue logic. The size register is incremented on every index write and is used to check that the table's contents do not exceed the set capacity, and it also doubles as a memory write pointer as it always points to the next available RAM address. The read pointer is similarly incremented on every index read, but when the read pointer coincides with the write pointer, the read pointer resets back to zero in order to restart the reading order. A summary of the architecture is shown in Figure 3.3.

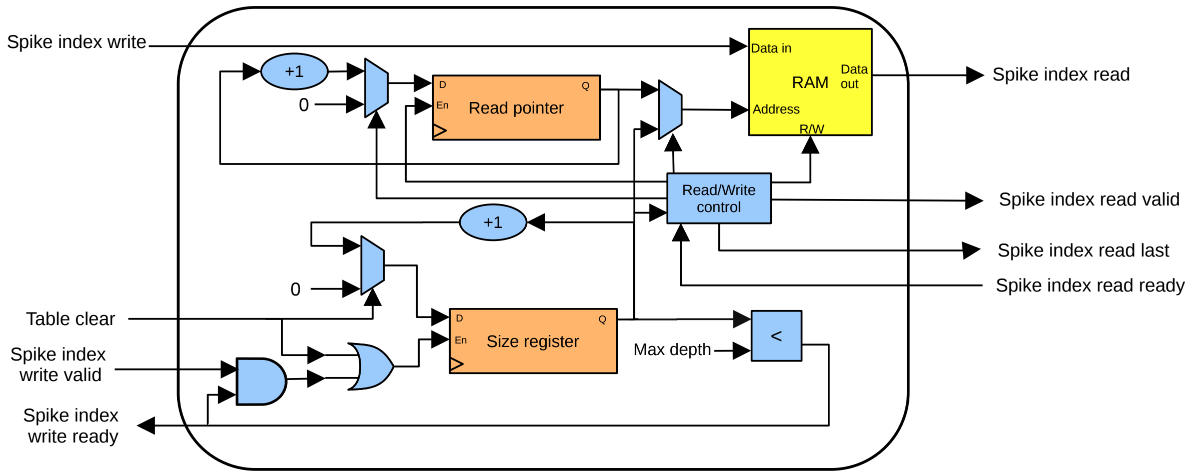

Figure 3.3: Spike index table architecture.

3.2.3 Neuron Wrapper Design

The neuron wrapper module encloses the bare neuron model, and implements additional control logic for loading and storing neuron state. The wrapper loosely represents a neuron layer in our architecture, as it contains a state table for storing the membrane potential voltages of all neurons in the layer. A "neuron index" is used to identify and keep track of the "virtual" neuron that is currently performing operations. This allows the re-use of one instance of the neuron model to perform activation calculations for every virtual neuron in the layer.

The wrapper takes in a stream of spikes provided by a spike index table, the aforementioned spikes are a result from the output activations of the preceding layer. The weight RAM interface passes through the neuron wrapper to the neuron model, and the output spike ports in turn pass from the model to the rest of the system through the wrapper. All interfaces on the neuron wrapper's ports follow the handshake principle in accordance with the rest of the design. The output spike handshake is of particular interest as it can be used to stall the wrapper's execution flow if necessary.

The input spike stream goes through the neuron model, which accumulates the appropriate weights. The neuron only specifies the weight index corresponding to a particular spike input, while the wrapper selects the exact memory region in which the current neuron's weights are mapped, this is accomplished by using the neuron index as the "high-order" component on the weight RAM address bus, the wrapper also sets the upper address bits to match the designated layer. At the end of output calculation, the neuron asserts "spike out" valid. The wrapper then stores the resulting membrane potential in the appropriate state table location. Simultaneously, the wrapper also checks whether the downstream spike consumers are ready to sample the spiking event, before incrementing the neuron index register and resetting the neuron model. It is crucial that the output spike handshake is detected before attempting to increment

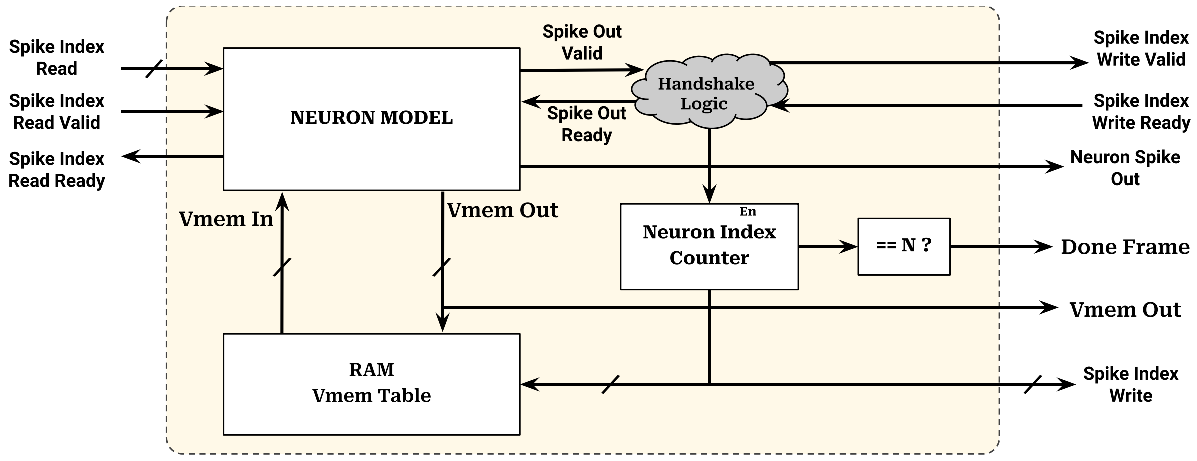

Figure 3.4: Neuron wrapper block diagram.

the neuron index, as the spiking event is transmitted not by the spiking bit, but by the index of the neuron that spiked, in accordance of the address-event representation protocol. Through the cyclical action of the index table, the same spike stream is processed by the following virtual neurons. This continues until all neurons in the layer are finished with their output activation calculations, upon which the neuron wrapper asserts the "done frame" signal, indicating that it is ready to move on and process the next stream of input spikes.

3.2.4 Output Encoding Module Design

The designed maximum membrane potential-based output encoding module compares the output action potentials of all readout layer neurons and selects the maximum value. The inferred output digit is simply the index of the neuron corresponding to this maximum value.

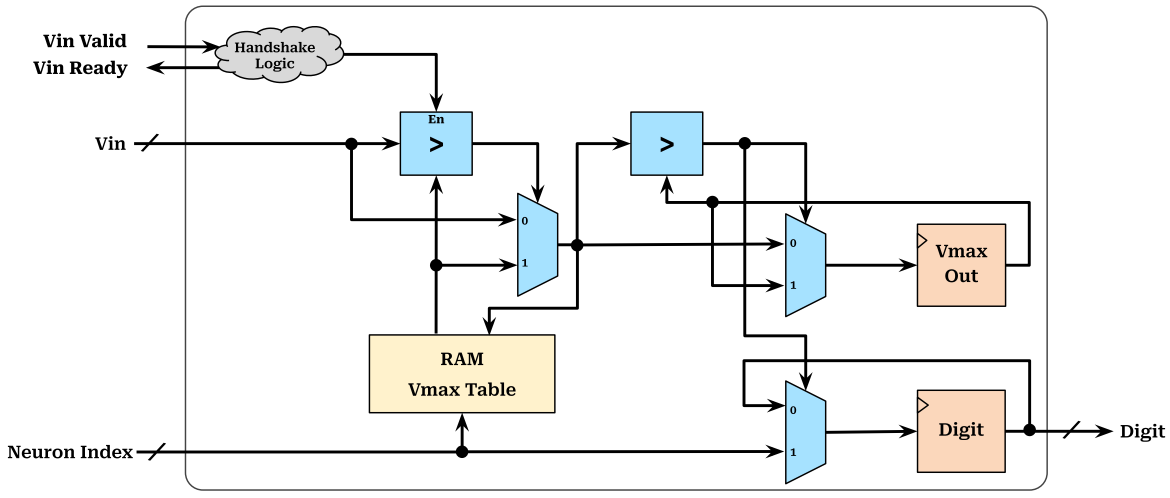

Figure 3.5: Output Encoding Module Diagram

3.2.5 Neural Network Design

The aforementioned building blocks of our design could only perform their tasks if they are connected correctly, with careful consideration to the control of processing flow starting from the input frame all the way to the predicted output. The required interconnections and glue logic are primarily implemented within the network module which as the name implies, represents the overall neural network and thus encapsulates the neuron layers, the index tables in between them and the output decoder.

The input layer consists of a spike index table, which then connects to the hidden layer represented by a neuron wrapper. The resulting spikes from hidden layer neurons are stored by their neuron indices to a spike index table, where the neuron index is written only if a spike is registered at the current time frame. This leads to the output layer wrapper which performs additional processing and has a neuron count equal to the number of output classes. Instead of a spike index table, an output encoder is instead connected to the output layer in order to determine which output class was predicted by the neural network. The block diagram of the neural network is shown in 3.6.

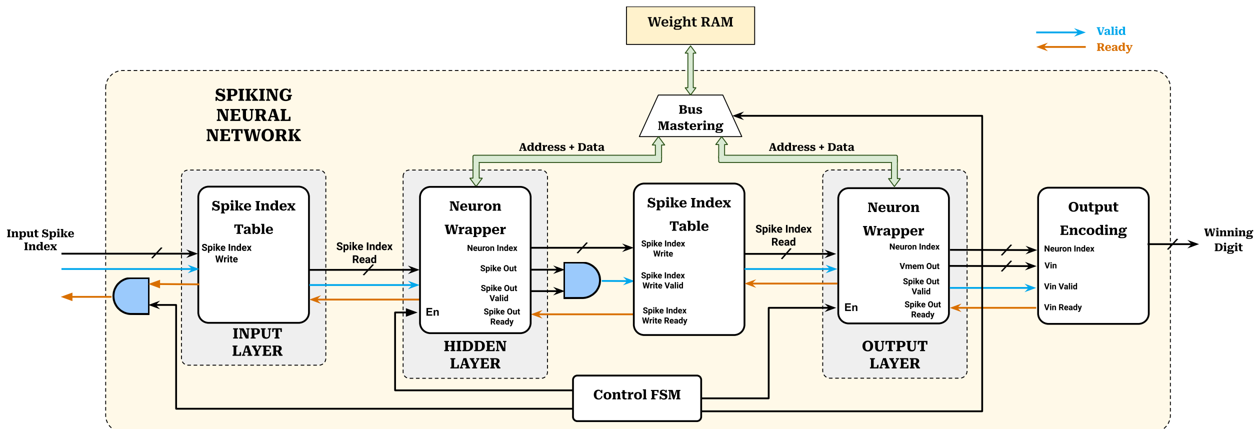

Figure 3.6: Network diagram.

The network entity governs the execution flow of the inner modules using a finite state machine or FSM, which ensures the correct sequence of processing steps between the different layers.

3.3 Implementation

After establishing the basic design of the hardware SNN and its components, we will go through the details of implementing each component, and present the reasoning behind our design choices along the way.

3.3.1 Targeted Application

To validate our SNN design, we settled on a handwritten digit classification task, specifically adopting the Neuromorphic-MNIST benchmark dataset. This initial assessment aimed to ensure the functionality and reliability of our SNN implementation.

The samples of N-MNIST consist of pixel activation and deactivation events recorded on a $34x34$ grid, leading to a total dimension of 2312 elements, which maps directly to a neural network with an input layer of 2312 neurons. Said network will naturally require 10 output neurons, each neuron corresponding to an output digit class.

As a continuation of previous SNN work done on the same dataset by Bouanane et al [14], we opted to use one hidden layer of 200 neurons, which allowed us to reuse the weights the authors obtained from training, and helped us focus on the hardware inference. The aforementioned research also came to the conclusion that using a value of $\beta = 0.967$ for the decay factor of LIF neurons produces the most ideal results in the N-MNIST dataset.

3.3.2 Tools and Equipment

- **FPGA Board**

  The Sipeed Tang Nano 4K is a development board based on the Gowin Little-Bee GW1NSR-4C FPGA. It includes an on-board JTAG programmer and debugger, camera interface, HDMI port, a 27MHz crystal oscillator and a whole host of components needed for the proper functioning of the FPGA chip.

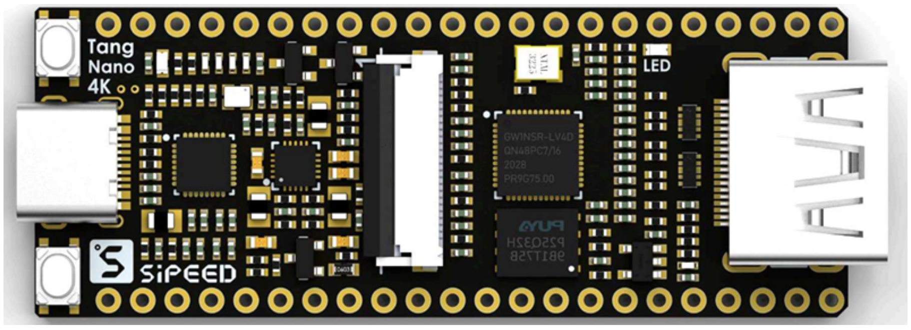

Figure 3.7: Picture of the Sipeed Tang Nano 4K FPGA development board.

- **Gowin LittleBee GW1NSR-4C**

  The Gowin LittleBee series is a family of low-cost FPGA chips based on LUT4 logic elements, with LUT counts ranging from about a thousand to over 20 thousand look-up tables. Gowin FPGAs typically feature a variety of additional components packaged within the same chip. The GW1NSR-4C in particular features 4608 LUTs, 256Kbits of integrated flash memory for non-volatile bitstream storage, 180Kbits of block SRAM, two phase-locked loops (PLLs) for clock frequency multiplication, an ARM Cortex M3 processor core for System-on-Programmable-Chip (SoPC) applications, dedicated hardware multipliers (referred to as "DSP resources"), as well as an 8MiB array of HyperRAM memory.

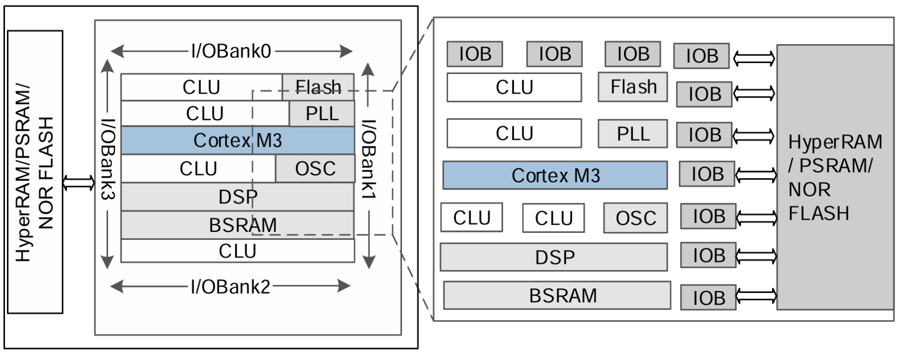

Figure 3.8: Architecture overview of the GW1NSR-4C [33].

- **HDL Simulation & Synthesis**

  To perform HDL synthesis for our FPGA, we used the vendor provided tool named Gowin EDA. It provides basic code editing and synthesis for a variety of VHDL and Verilog standards. However, it unfortunately does not provide a simulator.

  The absence of a simulator was rectified with the combination of open-source software: GHDL for VHDL-2008 simulation, and GTKWave for displaying simulation waveforms. Both tools are invoked through the terminal using documented commands and parameters, these commands were grouped into scripts for more efficient development.

  The bulk of code editing was done using Visual Studio Code thanks to the TerosHDL extension, which provides VHDL syntax highlighting and linting, as well as helpful documentation features like the generation of port diagrams for entities.

- **Software Libraries**

  Given the popularity of Python in machine learning circles, various libraries and frameworks have been developed for that purpose. We used the PyTorch library to process the pre-trained weights and test the inference of our network. We also used the Tonic [34] library to obtain and process the Neuromorphic-MNIST dataset.

3.3.3 Planning and Preliminary Testing

Before advancing to hardware experiments, it was crucial to first evaluate the model through software emulation. This step was essential to ascertain the accuracy and effectiveness of our network design. The emulation process provided a controlled environment to conduct preliminary tests and identify potential issues without the complexities introduced by hardware constraints.

A significant part of this preliminary analysis involved converting the neural network's weights. These weights, obtained through training, were initially in a 32-bit floating-point format. For implementation on an FPGA, these weights needed to be converted to a fixed-point format, a format which the neuron model in particular is designed based on. Determining the best fixed-point format for our application was the core objective of building the software model.

- **Dataset Wrangling**

  The N-MNIST dataset is based on individual spike events, each sample consists of a series of (x, y, t, p) tuples with no apparent image-like structure. Thus, we performed some pre-processing as shown in Figure 3.9. The discrete spike events of each sample are sorted into "bins" using Tonic's frame transform. This results in two 34x34 frames, one representing a history of pixel-on events in a given time-window, and the other represents a history of off-events in the same window. The frames are combined and then flattened to match the input structure of the neural network. Finally, the spike indices are extracted from the flattened frames and stored as ASCII binary strings into a text file, we pre-pended the spike indices of each frame with the number of spike indices in that frame.

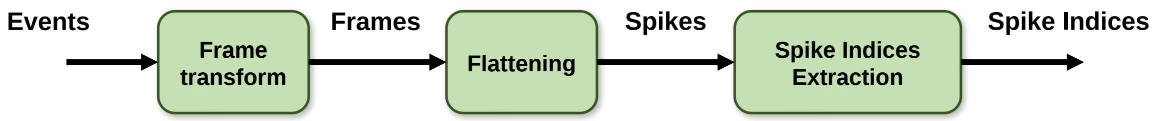

Figure 3.9: N-MNIST pre-processing pipeline.

Depending on the time window, the Tonic frame transform can result in multi-valued (non-binary) spikes due to event overlap, we remedied this by repeating the spike index in proportion to its magnitude.

- **SNN Software Model**

  To test the basic architecture and weights, we constructed a software model for our SNN in Python. The pseudocode for our model impelementation in python is represented in Algorithm 1, where it consists of two parts, first the input data is passed through hidden layer computation before proceeding to the output layer neurons processing.

- **Weights conversion**

  The process of converting model weights into fixed-point equivalents starts with a floating-point tensor input. Each entry in the tensor input is multiplied by 2^S (where S is the number of bits assigned to the fractional part) and then rounded to the nearest integer. The total desired number of bits are masked out using bit manipulation techniques, obtaining a 16-bit value for example would require bitwise-AND'ing the integer with the hexadecimal value `0xFFFF`. The shifted integer is then formatted as a binary string and written to a memory initialization file that can be used with VHDL testbenches; the process is repeated for all the weights in a given layer. For testing purposes, we also converted the resulting fixed-point weights back to floating-point in PyTorch tensor format in order to examine the effects of the conversion on accuracy, this was accomplished by casting the whole part of the fixed-point format into floating-point, and then dividing the fractional part by 2^S before adding it back into the casted floating-point number.

  Table 3.1: Accuracy summary for weight conversion

Weight format	IF	LIF
FP32	97.50%	97.65%
Q2.6	19.90%	21.35%
Q2.14	**97.24%**	**97.31%**

  Fixed-point formats excel in systems where the range of numerical data being processed is well-defined in advance, this is the case in many pre-trained neural network implementations including our own work. This led us to allocate more bits for the fractional part as the weights of our model fall in the range of $[-1.094027, 1.6090962]$, meaning that only two bits need to be dedicated for the whole part of the fixed-point format. Earlier tests explored the use of the Q2.6 format, which decimated the inference accuracy down to 19.9% for the IF network and 21.35% for the LIF network. Hence, 8-bit weights are ostensibly not viable for our implementation. We believe that Q2.14 provides sufficient resolution with negligible loss in inference accuracy, our testing revealed that Q2.14 weights can still achieve a maximum accuracy of about 97.24% for the IF network and 97.31% for the LIF network.

Algorithm 1 run_snn

```
FUNCTION run_snn(inputs):
    # Initialization
    nb_hidden = 200
    nb_outputs = 10
    nb_steps = sizeOf(inputs(2))
    hidden_synapse[nb_hidden] = 0
    Vmem[nb_hidden] = 0
    spikeout[nb_hidden] = 0
    Vmemout[nb_outputs] = 0
    output_syn[nb_outputs] = 0
    # Compute hidden layer activity
    h1_from_input = inputs * w1
for t from 0 TO nb_steps - 1 do
        h1 = h1_from_input(t)
   if Vmem > Threshold then
              spikeout = 1
              Vmem = 0
   else
              spikeout = 0
   end if
        new_Vmem = (Vmem + hidden_synapse) * (1.0 - spikeout)
        APPEND Vmem TO Vmem_rec
        APPEND spikeout TO spiking_rec
        Vmem = new_Vmem
        hidden_synapse = h1
end for
    SET mem_rec TO stack of Vmem_rec
    SET spk_rec TO stack of spiking_rec
    # Readout layer
    h2 = spiking_rec * w2
    INITIALIZE Vmemout_rec WITH Vmemout
for t FROM 0 TO nb_steps - 1 do
        output_syn = h2(t)
        SET Vmemout TO Vmemout + output_syn
        APPEND Vmemout TO Vmemout_rec
end for
    SET Vmemout_rec TO stack of Vmemout_rec
    RETURN Vmemout_rec
END FUNCTION
```

The final format we decided on is the Q2.14 format implemented in 16-bit binary strings, as it assigns as many bits as possible to the fractional part while also being a multiple of 8 for better alignment in operations over byte-wide busses. As a consequence, we truncated the decay factor to the value $\beta = 0.966796875$, which is the closest number representable in the Q2.14 format.

3.3.4 Hardware Implementation and Simulation

Hardware implementation is an iterative process, requirements become clearer as the design becomes more concrete and grounded. But performing full HDL synthesis for a physical FPGA test at every step during this phase is time-consuming and redundant when the implementation is still in its infancy.

After the software proof of concept, we began the hardware implementation of the modules outlined in Chapter 3. These modules were realized using VHDL-2008 Register-Transfer-Level (RTL) modelling, and were each tested by writing an appropriate VHDL testbench that would then be simulated in GHDL. It was clear that we first needed to simulate each building block of our network to validate its functionality, before proceeding to the overall network simulation. The development started from core neuron logic and spike transmission infrastructure, and we progressively worked our way up the hierarchy of modules until we developed and successfully simulated the neural network entity.

- **Spike Index Table**

  The spike index table is a relatively trivial design, it was implemented as a block RAM with read and write pointers similar to a typical FIFO. It includes various generic parameters to tweak the maximum capacity and width of the input spike indices. While FPGAs typically provide dual-ported block RAMs capable of reading and writing at the same time, we did not rely on this property and instead opted to use the single-ported mode, where write operations take precedence over read operations.

  The results of a simple testbench simulation are shown in Figure 3.10 where the table is filled until it reaches its capacity (a depth of 4 in the simulation), the contents of the table are read and tested to verify their integrity. The `inedx_rd_last` signal is asserted at the end of the read operation to indicate the end of the current set of spike indices. It should be noted that after the final index was read, the reading process loops back to the first spike index.

- **IF Neuron**

  The IF neuron model makes use of fixed-point arithmetic to perform weight accumulation and threshold comparison. We made use of standard VHDL packages

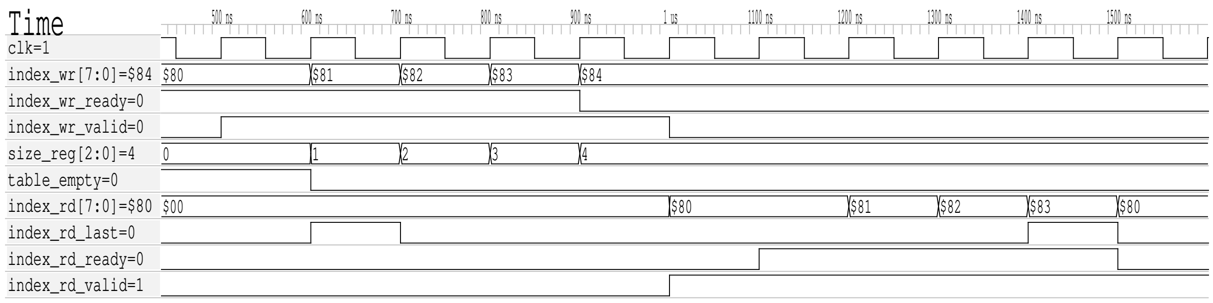

Figure 3.10: Spike index table simulation.

intended for integer arithmetic rather than declare our own types and operators, which simplified the implementation and helped ensure compatibility across different hardware vendors. The port listing generated by TerosHDL for the neuron model is shown in Figure 3.11.

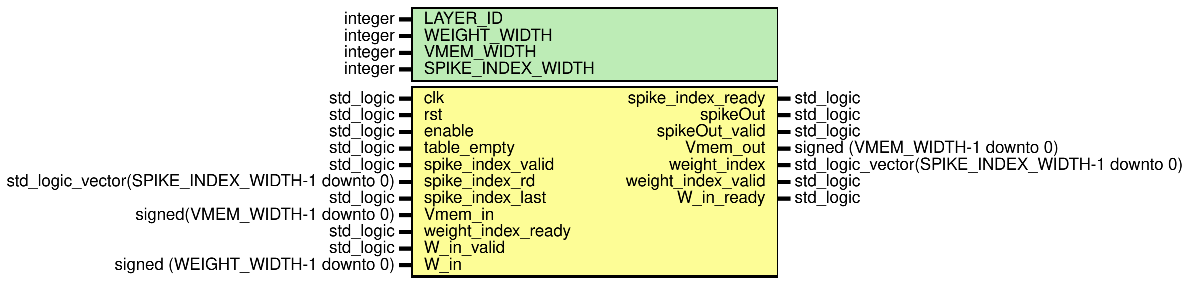

Figure 3.11: Neuron model ports and parameters.

The model features a 24-bit weight accumulation register (`w_acc`) declared as a `signed` bit vector from the `numeric_std` package, the package also defines the addition and comparison operators used in the process of adding incoming weights and comparing the result to a set spiking threshold, the latter was configured to 1.0 as in the software inference model. The only consequence of using Q2.14 fixed-point format for weight encoding is that we had to adjust the binary value of the threshold, which simply involves a left shift of 14 bit positions. It should be noted that the `w_acc` register itself as well as the membrane potential signals use the Q10.14 format, providing headroom for overflows during weight accumulation. Neurons in the readout layer are designed to be non-spiking, in contrast to the spiking neurons in the hidden layer. Our design accommodates both neuron types, configured by specifying a "layer ID" during network instantiation to ensure proper functionality across different layers.

The model was tested using a VHDL testbench featuring an instance of the neuron model, an instance of a spike index table to store input indices and a simple array to represent weight memory. Initially, the weight memory array is filled using a memory initialization file, and the spike index table is pre-filled with four arbitrary spike indices, the neuron is also provided with an arbitrary initial membrane potential `vmem_in`. The neuron model instance is then enabled to start the weight accumulation process, with the resulting waveforms shown in Figure 3.12 (All membrane potential and weight signals are depicted in decimal by assigning a 14-bit fractional part).

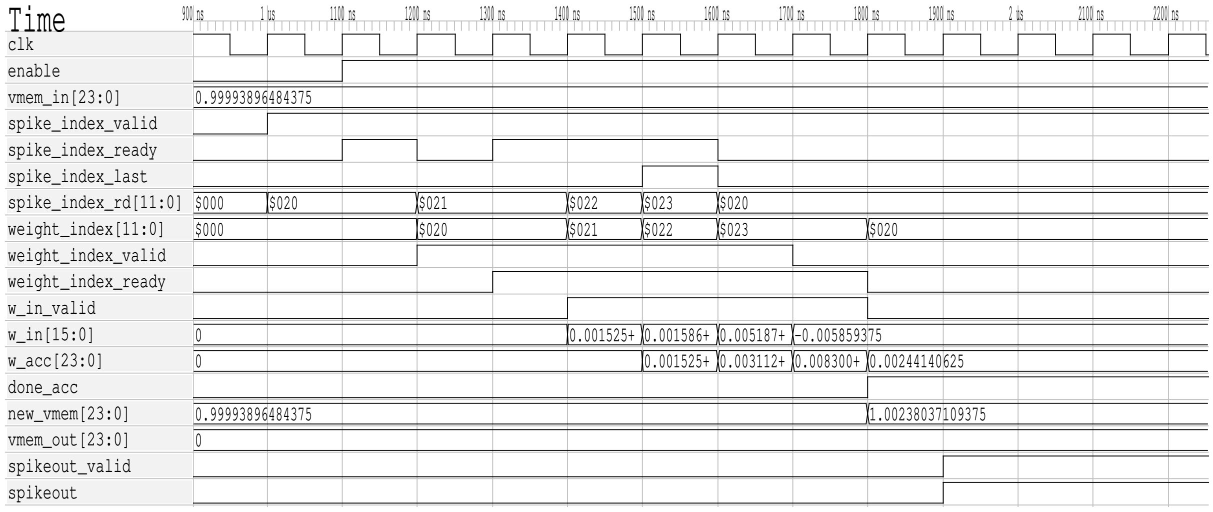

Figure 3.12: IF neuron simulation.

The neuron reads in the first spike index then stalls for one clock cycle as it waits for the weight memory to be ready for the next spike index, as the indices are used to address said memory. Simultaneously, whenever a weight data value (`w_in`) is presented to the neuron, it is added to the weight accumulator. When the last spike index is read (7th clock cycle of Figure 3.12), the neuron stops reading further indices, and when the weight value corresponding to the last index is read, the neuron asserts the `done_acc` signal, indicating the completion of weight accumulation.

Following the accumulation, the neuron's spike output (`spikeout`) is calculated based on the sum of the old membrane potential `vmem_in` and the weight accumulator `w_acc`, the spike output is asserted if the summation (`new_vmem`) is greater than the spiking threshold, otherwise it is left at zero (no spike). The spike output is coupled with a validity strobe `spikeout_valid`, as well as the resulting output membrane voltage `vmem_out`, the latter resets to zero in the event of a spike hence the apparent lack of changes.

- **LIF Neuron**

  As an extension of the IF neuron, most of the implementation details discussed in the earlier IF neuron part apply to the LIF neuron model as well. The most notable addition is the β multiplier for the membrane potential output, which was implemented using a combinational DSP multiplier provided within the FPGA fabric.

For performing the fixed-point multiplication of the membrane potential by the decay factor, a regular integer multiplier was used. The only caveat is that, similar to how an n-bit integer multiplier produces a 2n-bit result, the fixed-point multiplication introduces additional bits. In the case of calculating exponential decay, it is clear that the result will shrink in magnitude rather than expand, rendering the additional bits redundant. But unlike integer multiplication, it is not sufficient to simply ignore the added bits on the left, as the result of a fixed-point multiplication grows in both directions, meaning that both the fractional part and whole part's widths are doubled, shifting the decimal point leftward in the process. We solved this by right-shifting the multiplication result by the number of fractional bits to realign the position of the decimal point.

The VHDL testbench for the LIF neuron was almost identical to the IF neuron. For brevity and completeness, we directly present the resulting waveforms in Figure 3.13.

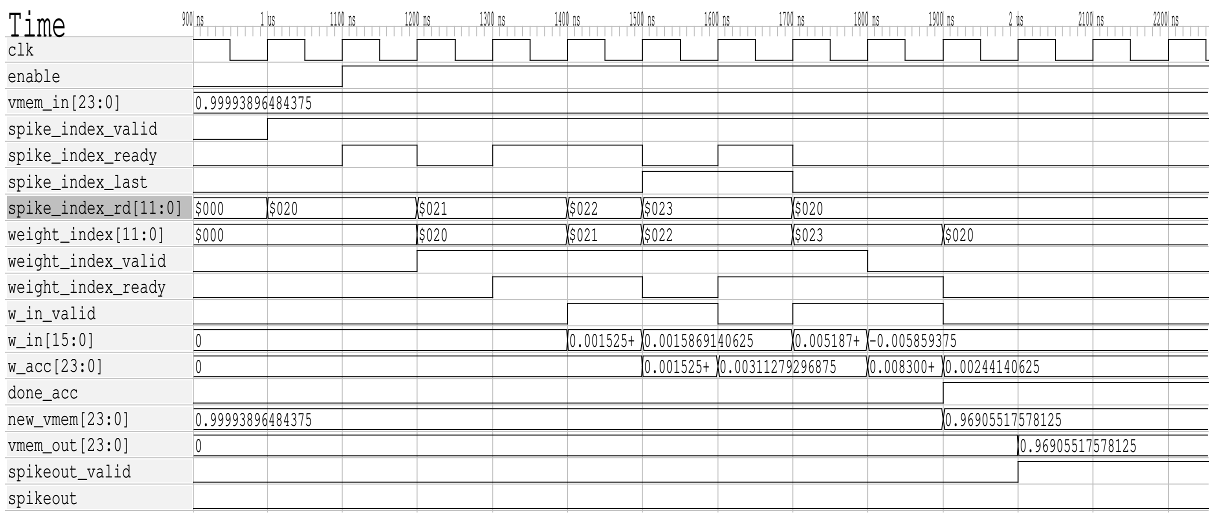

Figure 3.13: LIF neuron simulation.

- **Neuron Wrapper**

The neuron wrapper manages the state data for all the neurons in the layer. It keeps track of the current neuron through the `neuron_index`, and uses that index to address a membrane potential table implemented in block RAM so that the neuron model can read and write the correct neuron state. The `neuron_index` is incremented on the assertion of the `neuron_spikeout` handshake, and the next state data is loaded. As shown in Figure 3.14, the neuron wrapper is parameterized with a generic that chooses the type of neuron being used, the following simulations were performed using the IF neuron.

The weight RAM was organized in a layout that allows ease of addressing. To obtain the weight value W_{ij} associated with a spike input j coming into a neuron

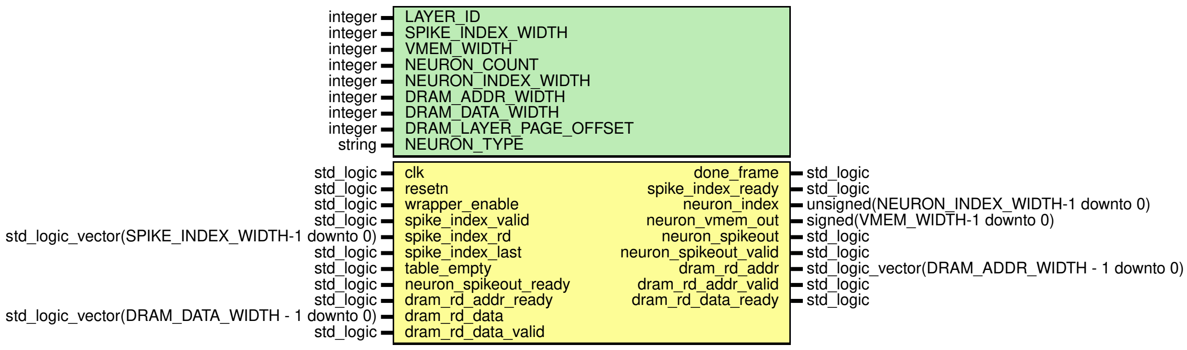
integer LAYER_ID
integer SPIKE_INDEX_WIDTH
integer VMEM_WIDTH
integer NEURON_COUNT
integer NEURON_INDEX_WIDTH
integer DRAM_ADDR_WIDTH
integer DRAM_DATA_WIDTH
integer DRAM_LAYER_PAGE_OFFSET
string NEURON_TYPE
std_logic clk
std_logic resetn
std_logic wrapper_enable
std_logic spike_index_valid
std_logic_vector(SPIKE_INDEX_WIDTH-1 downto 0) spike_index_rd
std_logic spike_index_last
std_logic table_empty
std_logic neuron_spikeout_ready
std_logic dram_rd_addr_ready
std_logic_vector(DRAM_DATA_WIDTH - 1 downto 0) dram_rd_data
std_logic dram_rd_data_valid
done_frame std_logic
spike_index_ready std_logic
neuron_index unsigned(NEURON_INDEX_WIDTH-1 downto 0)
neuron_vmem_out signed(VMEM_WIDTH-1 downto 0)
neuron_spikeout std_logic
neuron_spikeout_valid std_logic
dram_rd_addr std_logic_vector(DRAM_ADDR_WIDTH - 1 downto 0)
dram_rd_addr_valid std_logic
dram_rd_data_ready std_logic

Figure 3.14: Neuron wrapper ports and parameters.

i, the incoming `spike_index_rd` (equivalent to j) is placed on the low-order bits of the weight RAM address bus `dram_rd_addr`, and the `neuron_index` (equivalent to i) is placed on the high-order bits of the address bus. Finally, the top most bit in the address bus `dram_rd_addr` was used to differentiate between the hidden layer and the output layer.

A VHDL testbench was devised to test the neuron wrapper entity, the testbench emulates a RAM block that is initialized with the neuron weights of the output layer. The wrapper was configured with 10 neurons, and connected to a spike index table that is first pre-filled with arbitrary indices. The wrapper is enabled once the spike pre-fill process is done, upon startup it immediately begins loading the previous value of the neuron potential from the state table, starting from a neuron index of zero, and subsequently enables the internal neuron model (Figure 3.15).

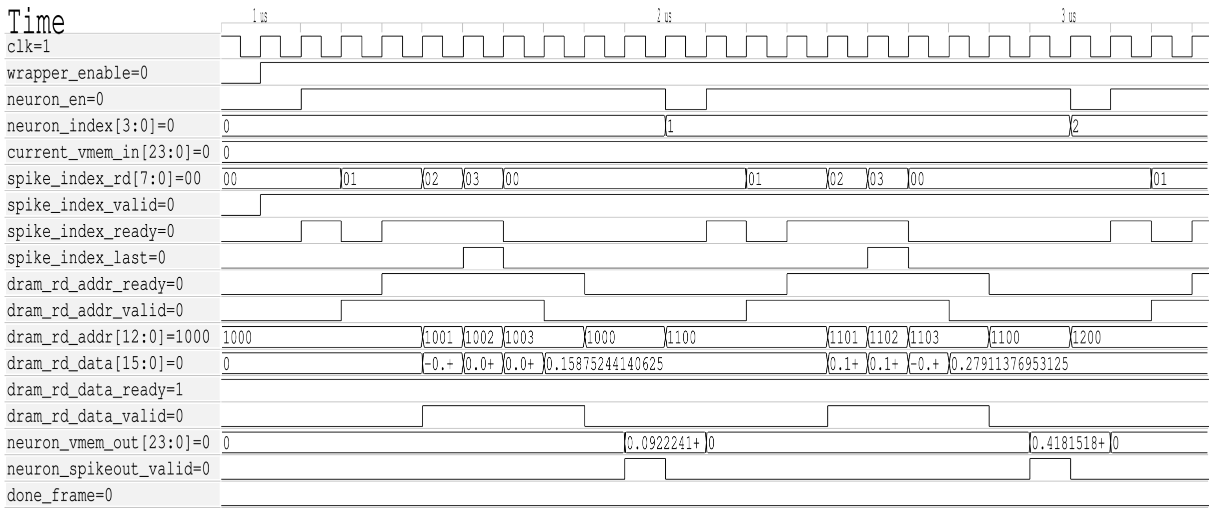

Figure 3.15: Neuron wrapper startup.

When a neuron completes its output activation, the wrapper stores the resulting output membrane potential in the state table. It then increments the neuron index and re-initializes the neuron model by loading the state corresponding to the next neuron, the cyclical action of the spike index table allows all neurons to process the same set of spike indices. Note that for testing purposes, all neurons were configured to have the same initial membrane potential, hence the apparent lack of change of `current_vmem_in`. The process repeats until all neurons have fired their outputs.

At each neuron activation, the wrapper checks if the neuron index reached its final value. When the last neuron achieves an output, the wrapper asserts the `done_frame` signal as shown in Figure 3.16, and freezes the neuron model by de-asserting its enable signal. The wrapper remains in this state until reset, or until its enable signal is toggled, signaling the beginning of a new frame to process.

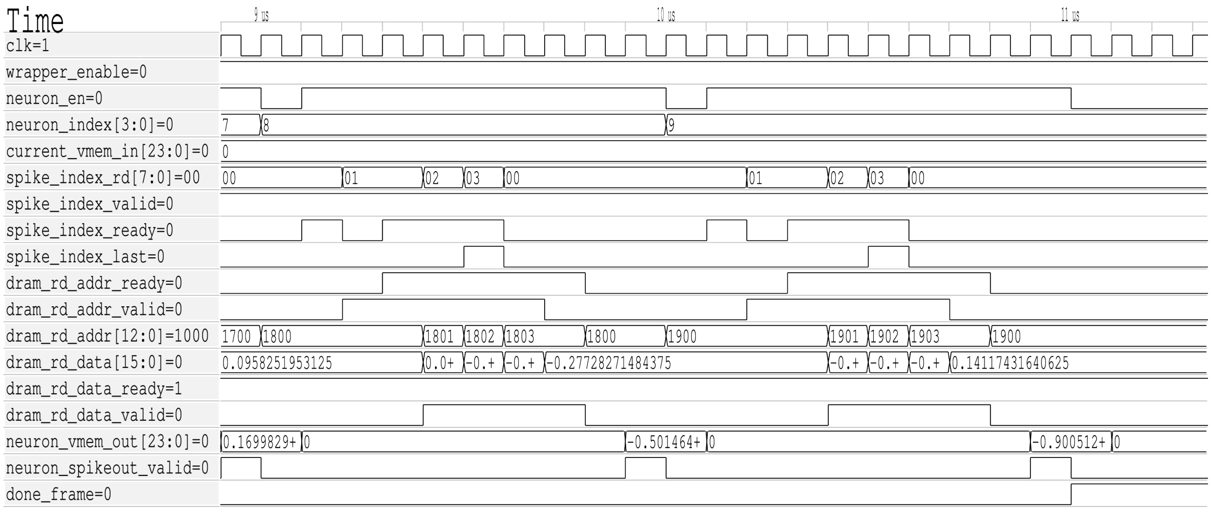

Figure 3.16: Neuron wrapper wind-down.

- **Output Encoder**

  The output encoder implements the maximum membrane potential classification paradigm. As with the neuron models, the output encoder module makes use of fixed-point comparison which was again implemented using the `signed` type provided by `numeric_std`. The history of membrane potential peaks is stored in a block RAM that is addressed by the incoming `neuron_index`. The `neuron_index` associated with the peak recorded potential value `vmax` is stored in the `digit` register which represents the classification output. The full signal listing is shown in Figure 3.17.

  The output encoder testbench features a neuron wrapper containing 4 neurons, a spike index table for supplying arbitrary spikes and the output encoder module. The simulation results are shown in Figure 3.18.

- **Neural Network**

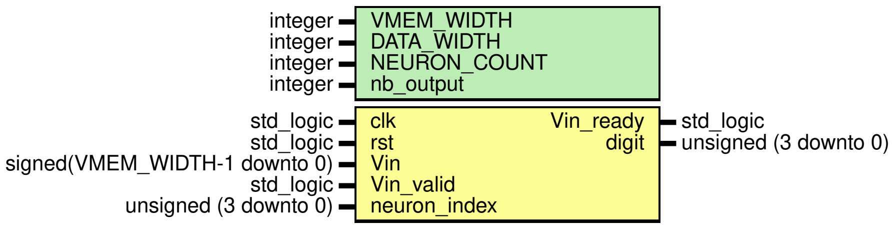

Figure 3.17: Output encoder ports and parameters.

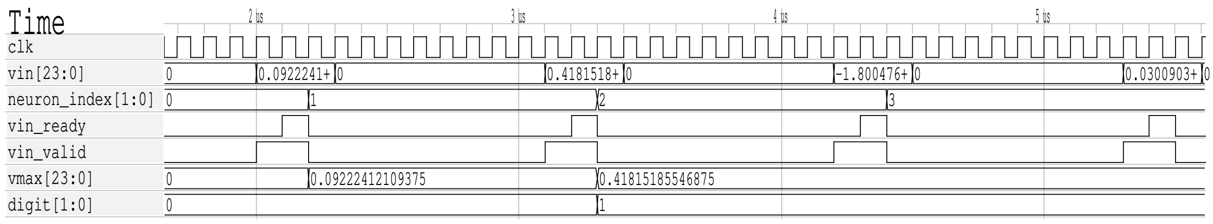

Figure 3.18: Output encoder simulation.

The neural network primarily functions as a container that instantiates the aforementioned modules and connects them together. The most notable block is the control FSM which was primarily implemented as VHDL case-when statements for the next state logic and combinatorial output logic.

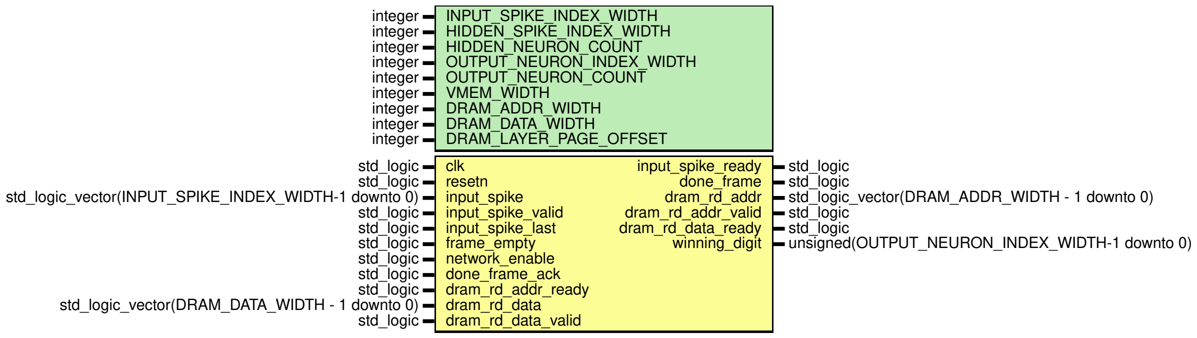

Figure 3.19: Network ports and parameters.

The network's control FSM features one state for each layer in the network, as well as two additional states: an initialization state (the default state upon reset) where the spike index tables are cleared in order to prepare for the next frame, and a "done" state where the predicted output is held stable until it is acknowledged by a receiving module. The FSM advances to the following state when the outputs of the current stage are valid, in the order described by the state transition diagram in Figure 3.20.

For the network testbench, the weights of the entire network are initialized in an array, and the spike indices of a chosen sample are inserted frame by frame to the network. Based on the size of each frame, the `input_spike_last` signal is asserted to indicate the end of a frame, which prompts the network to stop accepting indices and start hidden layer processing, as shown in Figure 3.21.

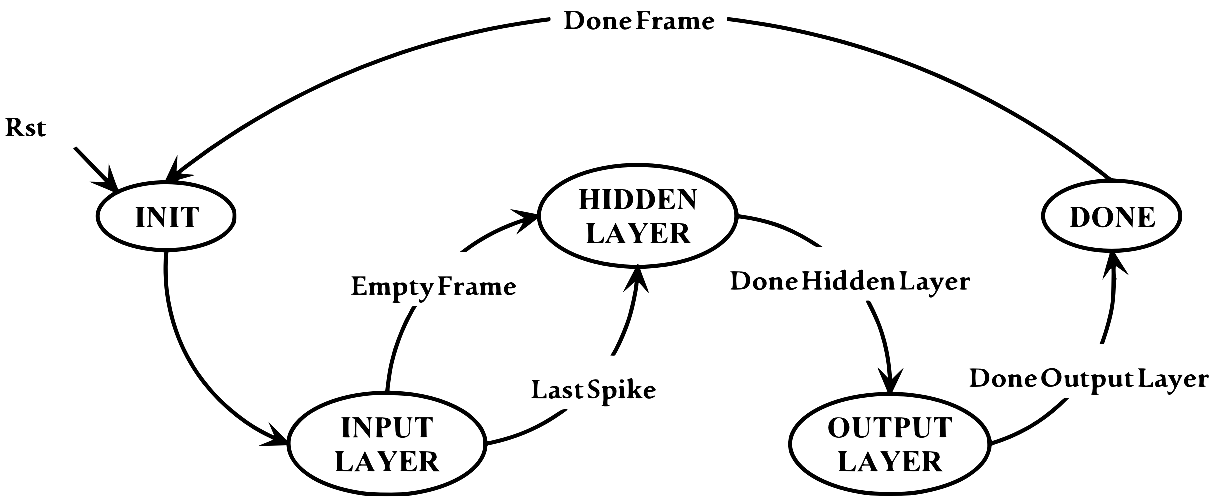

Figure 3.20: State diagram of the network's FSM.

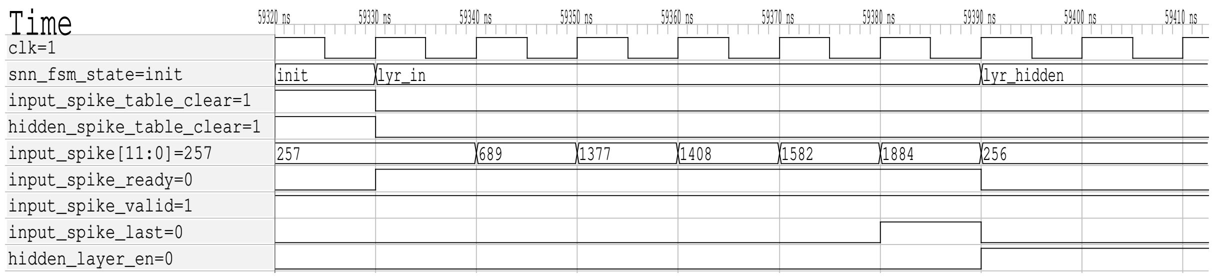

Figure 3.21: Network simulation transitioning into the hidden layer state.

During the hidden layer state, the hidden layer wrapper is given control of the weight RAM buses in order to fetch and accumulate weights. When hidden layer processing is finished, weight RAM control is transferred to the output layer wrapper, which starts processing the indices of the previous layer as demonstrated in Figure 3.22.

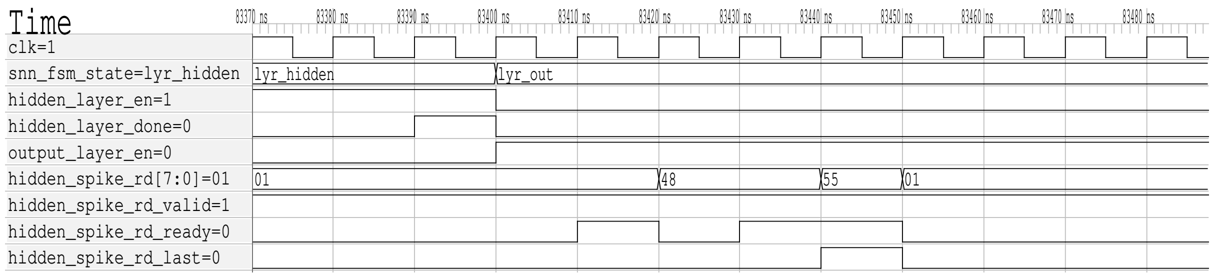

Figure 3.22: Network simulation transitioning into the output layer state.

Finally, when the output layer finishes processing, the digit output is sampled from the output encoder. The network asserts the `done_frame` strobe and awaits acknowledgement from a consumer module, before clearing the index tables in preparation for a new spike frame. These transitions are summarized in Figure 3.23, where the network has correctly classified the digit '8'.

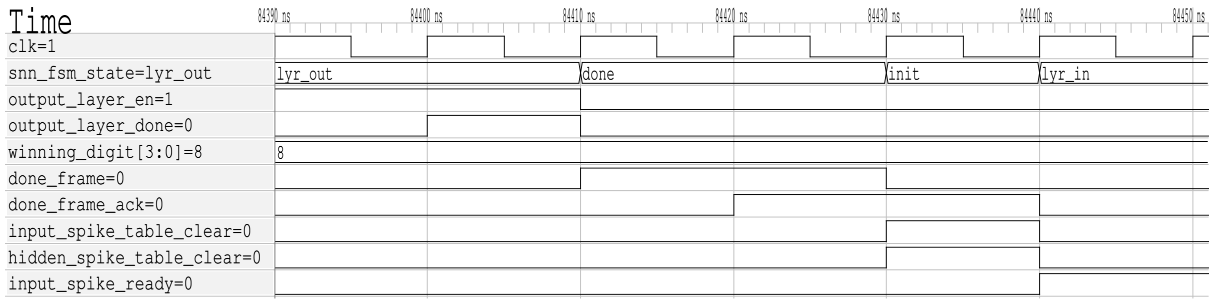

Figure 3.23: Network simulation latching the output and preparing for a new frame.

We have shown the example of processing a single frame in the simulation waveforms, but it should be noted that the actual testbench goes through all the frames in the sample. The rest of the frames were omitted for brevity and clarity.

3.4 Summary

In this chapter, we finalized the design and implementation of our spiking neural network. We performed preliminary testing that demonstrates the correctness of the basic operations needed. In the following chapter, we will put our implementation under rigorous testing and experimentation in order to quantify various metrics and identify any special characteristics.

Chapter 4

Evaluation of the Hardware SNN Implementation

4.1 Introduction

To test and validate our proposed design from Chapter 3, we will apply it to the basic task of handwritten digit classification, using the Neuromorphic-MNIST benchmark dataset discussed in Chapter 2. Following this, we will conduct various experiments to analyze several metrics, focusing on the impact of membrane leakages on different types of spiking neurons.

4.2 Latency and Spiking Activity Analysis

We used the Neural Network simulation testbench to calculate the latency and spiking activity of the hidden layer, as both parameters (especially the latter) are difficult to measure in a live hardware test. This method also eliminates any bottlenecks related to physical weight RAM access and input spike transmissions, and allowed us to focus on the inherent latency of our design.

The testbench recorded the number of clock cycles it took for the system to provide a stable output based on the target sample. The same testbench also recorded the spiking activity of the hidden layer neurons. Using the VHDL `STD.textio` package, a helper module was implemented and used to count and store spiking events on every assertion of `neuron_spikeout` of the hidden layer wrapper.

The experiment was conducted using 300 N-MNIST samples, 30 samples from each digit, and repeated twice: Once for the IF neuron, and once for the LIF neuron.

Table 4.1: Latency and spiking activity summary.

System metric	IF	LIF
Latency (clock cycles)	201800	218700
Hidden layer spiking activity	1521	2194

Both IF and LIF neuron implementations delivered accurate results, with the LIF implementation exhibiting slightly larger latency. As discussed in [14], the larger latency is attributed to the properties of the N-MNIST dataset in relation with the training method being used. Our design also corroborates the difference in spiking activity, with LIF neurons significantly spiking more often per sample. The spike sparsity of IF neurons is a common theme in existing research.

4.3 Power and Resource utilization Analysis

With the testbench simulations confirmed to be functional, we were able to confidently advance to the next step of testing a physical hardware implementation. We settled on the Sipeed Tang Nano 4K FPGA development board to host and test our design.

The transition to real hardware introduces some additional board-specific components, the design of which will be briefly discussed in the following sections.

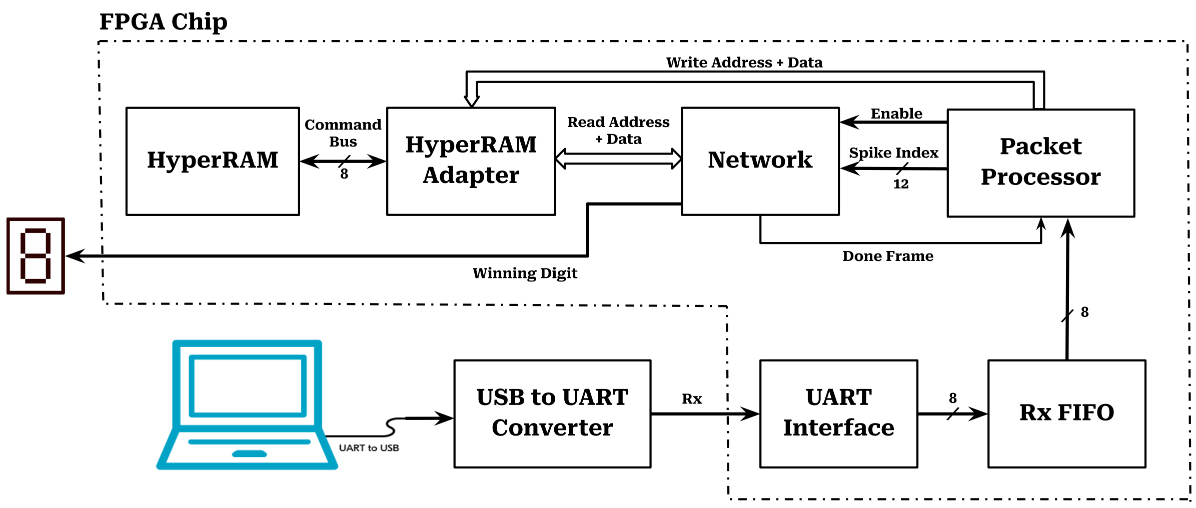

Figure 4.1: Overall diagram of the test setup.

4.3.1 Data Transfer operation (PC-to-FPGA)

Our test setup relies on receiving weight and spike frame data from a computer. Multiple approaches exist to carry out data transmissions, with increasing complexity as the demand rises for higher speeds and lower latency. We settled on the UART protocol for serial data transmissions, as it is low in complexity and UART adapters are ubiquitous thanks to their popularity in embedded systems.

- **UART Protocol**

  The UART protocol is one of the simplest protocols for data transmissions between a PC and an embedded device, such as micro-controllers or in our case, FPGAs. UART is implemented over a variety of physical media and specifications, from legacy RS-232 terminals, to modern USB-based implementations, to even wireless implementations using infrared emitters and receivers. UART speeds lag behind the processing capabilities of even the most low-powered of modern devices, but the lower speed and lower pin count compared to common interfaces allows live-circuit debugging even with less-than-ideal test equipment, and the low complexity leads to minimal failure points.

  The most common full-duplex UART configuration consists of just two signal wires, one for each data direction between any two UART devices. A basic UART transmission consists of an 8-bit data word surrounded by one start bit and at least one stop bit, forming a frame. UART frames are transmitted serially one bit at a time at an implicit baud-rate (transmission speed in bits-per-second) that is agreed upon in advance by the two devices. It is possible to increase the reliability of UART transmissions by appending additional parity bits for error checking, frames that fail the parity check can then be dropped by the receiver.

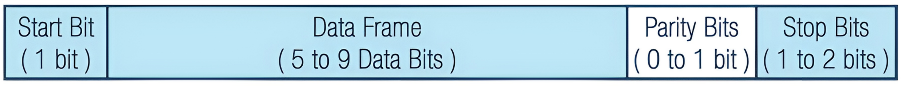

Figure 4.2: General format of a UART packet.

We settled on a 921600 baud configuration with one stop bit and no parity bits for our UART interfaces. This mode was chosen to reduce complexity, as modern UART implementations are reliable enough to not require parity checking, common implementations also do not specify any re-transmission logic by default, and the custom implementation of which is out of scope for our test. The aforementioned configuration also ensures compatibility, as the chosen baud-rate can easily be reached on typical UART adapters and converters.

- **USB-to-UART Converter**

  In the absence of native serial ports on most modern computers, a USB-to-UART converter is required. Such converters are commonly used to re-program microcontrollers without the need for expensive, vendor specific flashing tools. We chose one such converter, the ESP32-CAM-MB.

Figure 4.3: ESP32-CAM-MB USB-to-UART converter, and the ESP32-CAM board.

The ESP32-CAM-MB is a USB-to-UART adapter board designed for the ESP32-CAM micrcontroller development board. Despite being built specifically for programming the latter in an add-on shield form-factor, the ESP32-CAM-MB is an otherwise typical UART adapter based on the WCH CH340 USB-to-UART conversion IC. The adapter supports a variety of UART configurations including varying data widths and parity checking, and features a USB2.0 Micro-B port that connects to a PC for firmware programming and serial terminal debugging.

The UART pins of the adapter board are exposed on a female 2.54mm pin-header. We connected the RX pin to one of the GPIOs on the FPGA board, and configured the UART controller in our design to match the parameters we declared earlier in this section.

- **UART Controller and RX FIFO**

  The final and most vital component of the UART communication is the UART controller on the FPGA itself. Due to time constraints, we opted not to implement a UART controller from scratch, and instead relied on the pre-verified and open-source *Simple UART for FPGA* controller developed by Cabal [35].

  The UART controller was implemented and simulated in VHDL, as well as tested on physical hardware. It does not make any assumptions about the target FPGA and hence can be configured to meet a variety of testing setups. The controller features a write port with a read-valid handshake for transmitting data from the FPGA to the PC, and a read port with only a valid strobe.

  Unfortunately, the controller does not provide any additional flow control or buffering logic, which led us to adding a block-RAM based FIFO between the UART controller and the packet processor. The FIFO buffers incoming UART transmissions when the packet processor is already busy processing a spike frame, rendering it unable to react to further transmissions.

- **Packet processor** The packet processor decodes incoming data transmissions, and performs weight RAM or spike frame initialization depending on the received packet. It writes incoming weight data to the weight memory, and writes received spike indices to the network's input layer, where it also provides additional control signals to mark the beginning and end of a frame.

  Incoming data is brought into the packet processor as a stream of bytes, each individual byte is scanned until the processor detects a packet marker belonging to either defined packet formats: Weight packets and spike frame packets. The processor then reads in the rest of the packet header in preparation for the full payload. The ad-hoc packet formats of weight packets and spike frame packets are defined in Figure 4.4 and Figure 4.5 respectively.

Packet identifier	Weight memory start address	Payload size (in bytes)	Payload (Neuron weights)
1-byte	4-bytes	2-bytes	n-bytes

Figure 4.4: Weight packet format.

Both packets contain a marker (packet ID), payload size field and a payload. The weight packet is unique in that it also features a ”RAM start address” field, which

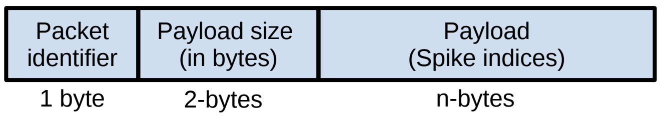

Figure 4.5: Spike frame packet format.

evidently defines the starting location of where the weight payload should be written to. The packet processor keeps track of the current address to write to by incrementing it on each successful reception of a weight value (each value being two bytes long). We also avoided hard-wiring a starting RAM address to improve flexibility of the design by dividing the weight transmission into multiple packets, especially considering the fragmented memory mapping of the neuron weights.

In both packet types, the payload size field defines the length of the payload in bytes. The packet processor assumes that the following given amount of bytes must belong to the same packet, which eliminates any confusion in case the payload data happens to coincide with the designated packet markers. Knowing the payload size is also crucial for spike frame functionality, as it allows the processor to assert the ″last spike″ signal required for the network to begin processing the incoming frame.

In order to decode the various fields of packets from incoming the byte stream, an FSM is used to keep track of processor's position within the packet, with states corresponding to each possible field in the packet. A simplified state diagram of the control logic FSM is shown in Figure 4.6, where the outputs are shown under the state transition events similar to Mealy-style modelling, signals that are not shown in a certain state are not relevant to that phase and are assumed to be in their inactive state.

Per Figure 4.6, the packet processor starts out in the ″packet scan″ state, where it continuously reads each incoming byte through the 8-bit ″RX byte″ port as it looks for special predefined bytes that are used as packet identifiers, the hexadecimal values `0xCA` and `0xFE` are used to identify the weight packets and the spike frame packets respectively. The processor controls the flow of the byte stream using the ″RX ready″

At the end of a weight packet, the packet processor simply returns to the packet scan state to listen for further instructions. Conversely, the neural network begins processing the received frame right after the end of a spike index packet. The processor remains in the ″network busy″ state and will not accept further packets until the network finishes processing, at which point the processor goes back to listening for packets.

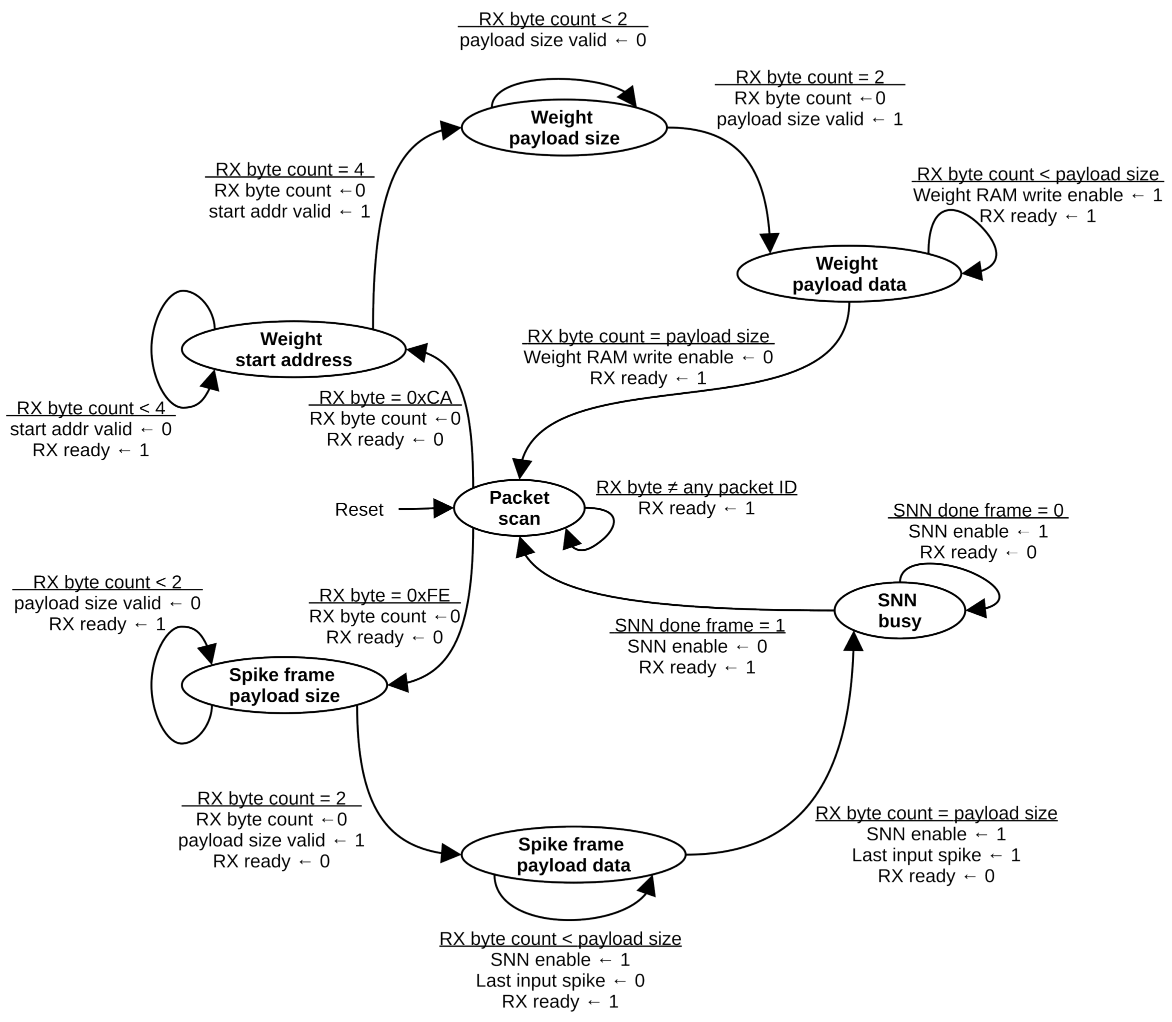

Figure 4.6: State diagram of the packet processor FSM.

4.3.2 Weight Memory

In the matter of storing model weights, it was apparent from the preliminary phases of design that using some form of dynamic RAM would offer the best flexibility, cost and scalability. But such memory would also come with numerous caveats, not the least of which is the issue of DRAM refresh. We resorted to using the Winbond 4Mx16-bit HyperRAM memory, packaged on-chip along with the GW1NSR-4C FPGA, to help us overcome the refresh issue and vastly simplify our design.

- **HyperRAM** Essentially, HyperRAM is a form of pseudo-static RAM or PSRAM, which is merely an array of capacitor-based SDRAM coupled with a controller integrated on the same chip that performs automatic refresh and fulfills memory read/write requests. The controller internally reads and re-writes DRAM contents on a row-by-row basis under a strict deadline, rejuvenating the data which would otherwise be lost to leakages. The controller also helps reduce pin-counts on HyperRAM chips by using a single ”command bus” for both address and data transmissions.

FPGA soft-core solutions are also commonly used to solve the DRAM refresh problem with conventional SDRAM chips, but one major benefit of integrating the refresh controller is that it does not rely on the correct implementation or integration of a user-supplied DRAM controller, which would add an additional failure point within an already complex design.

A downside of the HyperRAM interface is that it requires address and data transfers to be time-multiplexed on the same command bus, in accordance to a packet format defined by the manufacturer. We employed a pre-fabricated IP block provided by the Gowin EDA to fulfill the bulk of the HyperRAM interfacing logic, but we had to provide additional glue logic that adapts the IP block to our design.

- **HyperRAM Adapter**

  The Gowin HyperRAM IP block is designed around burst transfers, that is reading or writing successive memory locations in a single operation for maximum performance, as can be seen in Figures 4.7 and 4.8. Even though the memory is word-addressable with a word size of 16-bit, these operations are performed with a 32-bit double-word data bus. This comes at odds with our single-word (16-bit) access based design, yet we were able to conform to both constraints by only considering the first double-word in a burst, and only considering either the lower or upper word within that transfer depending on the desired address.

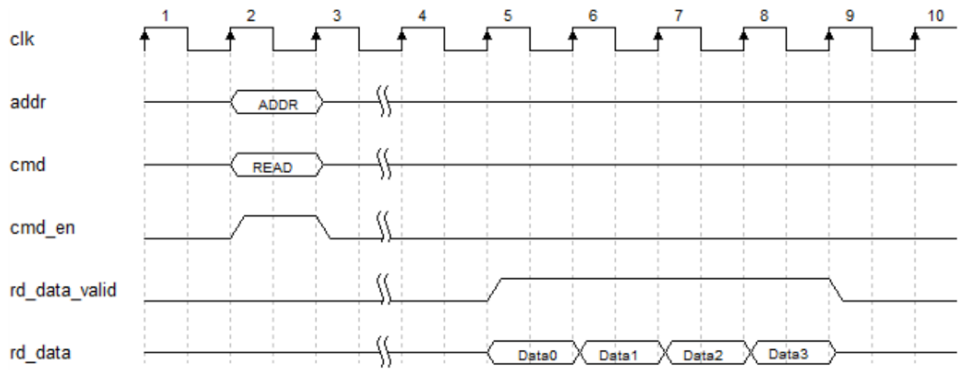

Figure 4.7: Burst read transfer using the Gowin HyperRAM IP.

To read a single 16-bit entry from the HyperRAM, the adapter places the desired address in the IP block’s address bus save for the least significant bit, which is fixed to zero to preserve 32-bit alignment; we concluded in an earlier test that the IP block misbehaves when operations are not 32-bit aligned. When the read request is fulfilled, the IP block reads a burst of data; a total of four 32-bit double-words starting from the requested address. However, our design can only ac-

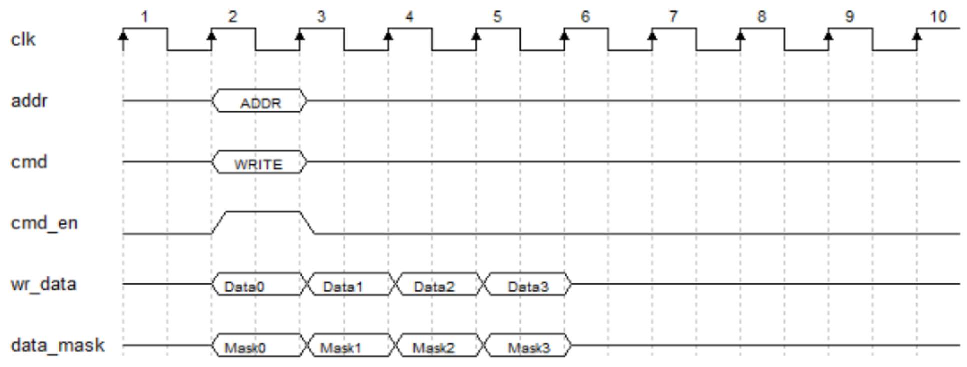

Figure 4.8: Burst write transfer using the Gowin HyperRAM IP.

cept one word at a time. To remedy this, the read valid strobe of the IP block is scanned for changes, and a transfer from the IP block's data bus to the data bus of the adapter is only performed when a low-to-high transition on the read strobe occurs, ensuring that the neural network only receives the desired (first) word in a burst. Additionally, since the IP block operates on 32-bit double-words, a single word has to be selected out of every transfer on the IP block's data bus. The least significant bit of the desired address is used to determine which word would be read: If it is zero, the lower 16-bits are chosen by the adapter's data bus multiplexer. If it is one then the upper 16-bits are chosen instead.

Write operations are handled in a similar vein, but the roles of the adapter and the IP block are somewhat reversed. The adapter still provides the write address (the least significant bit is excluded once again), along with the data to be written into that address. Furthermore, it must also provide a "byte mask", which is a group of data validity strobes defined on a byte-by-byte basis. The mask value is chosen based on the least significant bit of the desired write address: If it is zero then the mask indicates that the lower 16-bits are valid, and the adapter places the data to be written on the lower 16-bits of the IP block's 32-bit data bus. If it is one, then the mask signifies that only the upper 16-bits are valid, and the write data is placed on the upper 16-bits of the double-word write bus. The mask indicates valid data only for one clock cycle, before it is de-asserted such that no bytes on the write bus are valid. This conforms to the IP block's burst design, as it expects multiple double-words to be written unless it is explicitly forbidden from doing so. The mask also remains in the de-asserted state when the adapter is idle as a safeguard against any data corruption from unforeseen edge cases.

When a memory request is acknowledged, regardless of whether it is a read or a write, the adapter blocks further operations until a timeout counter elapses. This obstruction is imposed by the IP block, as it is incapable of queuing commands while already processing a command, thus commands issued during a

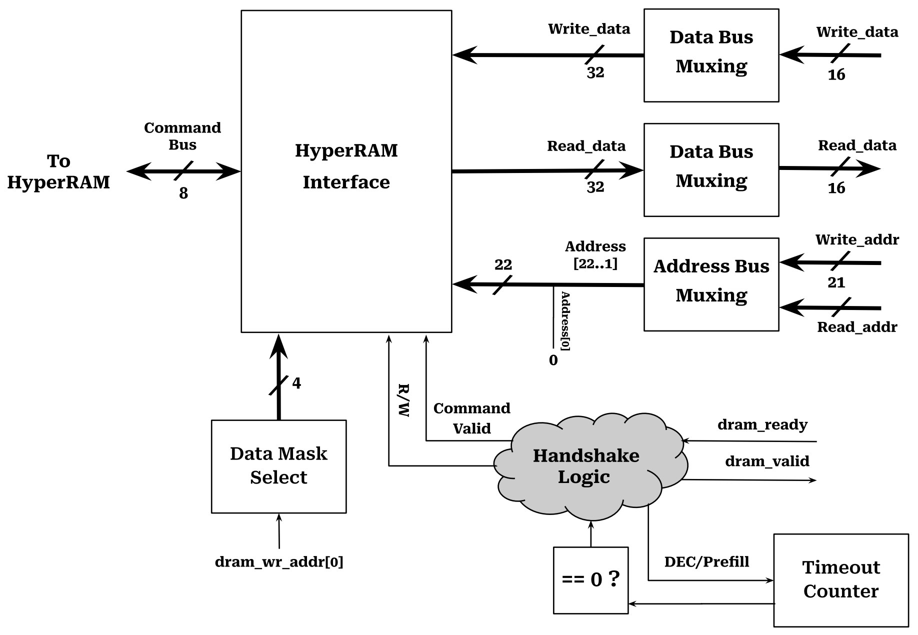

Figure 4.9: HyperRAM adapter block diagram.

busy state are simply dropped without notice. The timeout counts the number of busy cycles that the IP block must go through before being able to process a new command, in our configuration this timeout is given by the manufacturer as 19 clock cycles.

4.3.3 Results and Discussion

The hardware was primarily monitored for power consumption under different scenarios, and resource usage. A full accuracy test was not feasible due to the low speed of UART transmission, but to our knowledge the hardware implementation closely mimics the simulations with no significant discrepancies.

Basic tests using the setup shown in figure 4.10 were first performed to make sure the design recognizes digit samples correctly. To view the output, we used a simple 7-segment digit display driven using the network's predicted output.

- **Resource utilization** Table 4.2 summarizes the resources used by our spiking neuron network on FPGA for the two implemented types of spiking neurons. The table was derived from the synthesis summary given by the GOWIN IDE. It can be seen that both network types consume the same number of Look Up Tables, Flip-flops, and Block RAMs, except for the extra DSP blocks used by the LIF neuron due to the beta multiplication. Hence, IF network hardware system uses less resources than the LIF network.

- **Power consumption** The power consumed by the FPGA board was measured us-

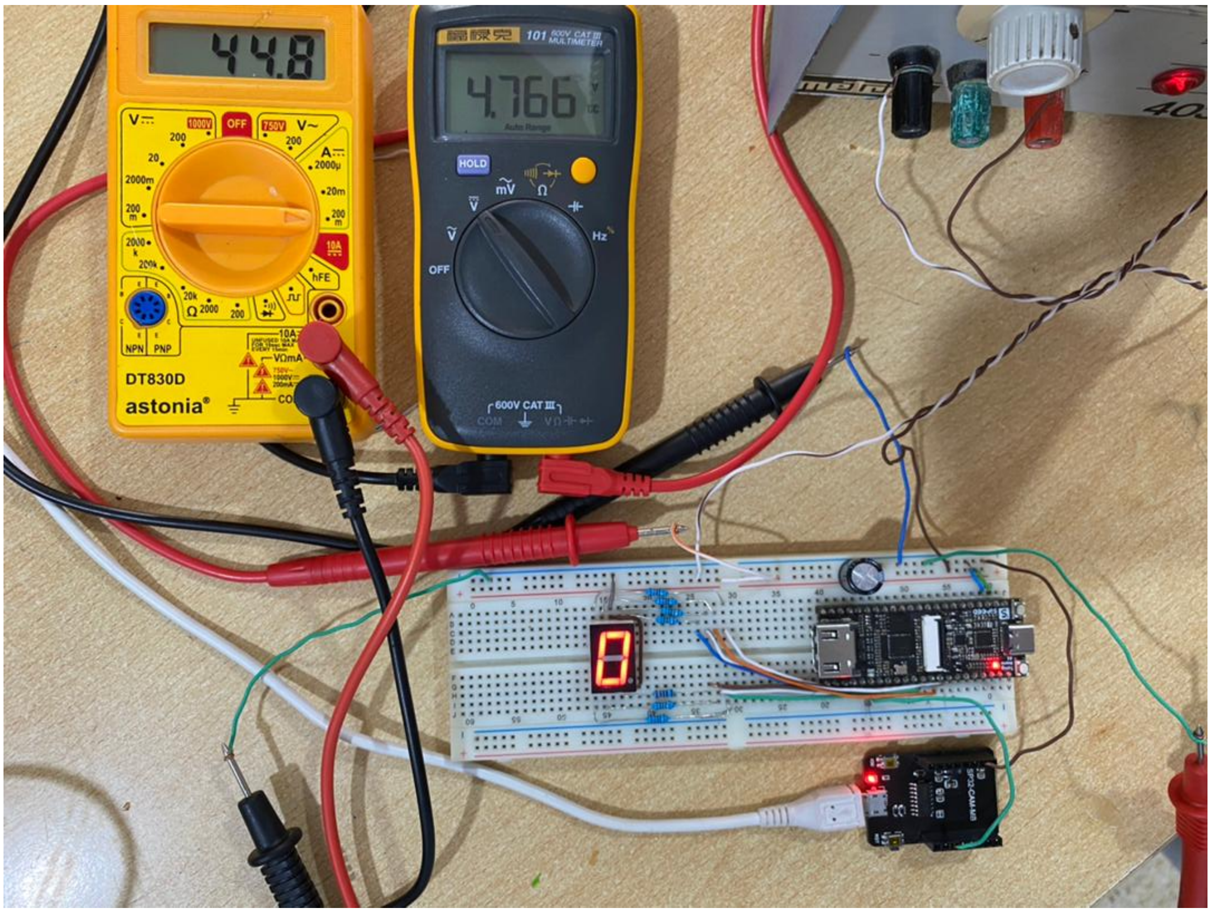

Figure 4.10: Experimental Setup.

ing a voltmeter and an ammeter and then multiplying their results. Three trials were performed using the same digit sample, each trial consisted of three power consumption recordings: idle consumption, idle consumption with the reset button held, and active load consumption during spike processing.

The experiment starts with powering the board using an external 5V power supply, then sending all the network's weights. For the idle consumption test, we waited until the power metrics stabilized to collect the measurements: the voltage across the FPGA board's power inputs, and the current flowing out of the board towards power supply ground. We then held the board in the reset state and measured its power consumption, in order to quantify the static (DC) power leakages (Note that thanks to the HyperRAM's self-refresh behavior, the weight data is completely unaffected by the reset). Finally, the design was put under load by continuously sending spike frames. The spike sample was sent in a loop, subsequently recording the power measurements as soon as they stabilized. The experimental setup is shown in Figure 4.10, with the only remark being that the 7-segment display was entirely removed from the circuit in order to prevent potential voltage sagging or stray current losses.

Table 4.2: FPGA resource usage summary.

Module		LUT4	Flip-flops	BRAM (bits)	DSP
Input layer	**Spike Index Table**	29	20	6144	0
Hidden layer	**IF neuron**	80	66	0	0
	LIF neuron	80	66	0	4
	Neuron Wrapper	14	11	4800	0
	Spike Index Table	19	18	2048	0
Output layer	**IF neuron**	47	61	0	0
	LIF neuron	47	61	0	4
	Neuron Wrapper	11	7	240	0
Output encoder		90	31	240	0
Neural network		43	3	0	0
Total	**IF network**	333	217	13472	0
	LIF network	333	217	13472	8

The same experiment was repeated three times: once with the IF neuron (using IF weights), once with the LIF neuron with a β of 0.966796875 (using LIF weights), and once again with the LIF neuron but using a β of 1, effectively making it behave like an IF neuron (using IF weights). The purpose of the first two experiments was to quantify the difference in consumption depending on neuron model dynamics, and the last experiment aimed to isolate the exact cause behind any differences, as the LIF neuron with β set to 1 should exhibit similar spiking characteristics to the regular IF implementation.

Table 4.3: Power consumption summary.

System state	IF	LIF	
		β = 0.966796875	β = 1
Idle power (mW)	220.43	252.93	250.19
Reset power (mW)	189.09	226.98	223.83
Load power (mW)	**235.6**	273.74	266.51

It was not unreasonable to predict that IF neurons would consume less power than LIF neurons given the difference in spiking activity measured in Experiment II. However, setting β to one provided us with further insight. While the spiking activity contributed a non-negligible percentage of the power consumption (a load power increase of 2.71% when going from $\beta = 1$ to $\beta = 0.966796875$), it is clear that most of the power consumed is dependent on the circuit design itself, where switching from the multiplier-less IF to the IF-like LIF ($\beta = 1$) results in a 13.11% power increase under load. Testing different implementations of β multiplication could provide additional insight.

Overall, the system delivers correct results with evaluated accuracy and latency, all while maintaining low resource utilization and power consumption.

4.4 Summary

In this chapter, we demonstrated the simulation and implementation of the proposed design of Spiking Neural Networks on FPGA. By conducting different experiments, we could analyze different metrics, highlighting the important experimental results and findings, and being able to answer the question asked in the beginning of this work.

General Conclusion

This work demonstrates that neuromorphic computing is a highly interdisciplinary field, encompassing computer science, machine learning, digital systems, and computational and theoretical neurosciences. As a new generation of AI, neuromorphic computing addresses the computational limitations and rigidity of traditional AI systems, which rely on deterministic and context-lacking interpretations of events. This emerging technology promises to enhance AI's capabilities to align more closely with human cognition.

However, to be able to benefit of such paradigm, understanding the effect of membrane leakages between the different neuron types on hardware is of a quite interest. For this reason, this work attempted to propose a generic and efficient digital hardware design of spiking neural network, implement it and test it on FPGA for a handwritten digit classification task, adapting the benchmark dataset "Neuromorphic-MNIST".

The virtual neuron based methodology derived tended to allocate very low resources compared to conventional architectures existing in literature, which makes it powerful in scaling for more complex network topologies. The network was designed in a generic manner, making it easily customizable and flexible for any other data formats, spiking neuron types, and other classification tasks.

The analysis of different metrics was done based on the implementation of the two different types of spiking neuron models, Integrate-and-Fire and Leaky Integrate-and-Fire neurons. It was first necessary to validate the system performance by analysing its accuracy.

The Leaky IF neuron outperformed the simple IF that does not have leaks by reaching an accuracy of 97.31% and closely matched the IF neuron by reaching a 97.20% accuracy. It was hence, clear that LIF neuron is more biologically plausible and accurate than the IF neuron model.

It was then found that the Leaky IF neuron has higher spiking activity than the IF neuron, this is due to the leak in membrane introduced for the LIF neuron, enabling it to spike more than the IF. This was further justified by computing the power consumption of both neuron models based networks, where truly the less spiky IF neuron turned to consume slightly less power than the Leaky one with approximately 46.51 mW on our test FPGA board.

Further inspection included the study of system latency and resources utilization for both networks. the two network designs uses same number of resources excepts for the DSP blocks, which are used only by the LIF neuron network, making it complex and expensive when converting to ASIC.

The most important findings and contributions of our work can be briefly summed up as follows:

- We proposed a generic low resource allocation design of SNN, which tends to be power efficient when implementing.
- The membrane leakage has a significant effect on spiking activity, hence on the overall power consumption of the network.
- IF neuron is simple, hardware friendly, and sufficient in basic applications (low complexity datasets.)

One of the limitations of our work is that our design do not follows the Pipe-lining paradigm, which can be investigated in the future to decrease the system latency. Moreover, it is worth investigating the effect of synapses leakages emulated by the CUBA-LIF neuron model, by redoing the same experiments on a modified LIF neuron to adapt the CUBA-LIF mechanism. Furthermore, Our design was only tested and validated on a simple classification task, adopting the benchmark dataset "N-MNIST", which is not very rich in temporal information. Hence, we propose to test our network on more rich spatio-temporal datasets, for instance: SHD (Spiking Heidelberg Digits), CIFAR10-DVS, Tactile Braille Letters, and others. This would allow us to further study the different types of spiking neurons and their best fit for specific tasks.

Bibliography

[1] G.E. Moore. “Cramming more components onto integrated circuits”. In: *Proceedings of the IEEE* 86.1 (Jan. 1998), pp. 82–85. DOI: 10.1109/jproc.1998.658762. URL: https://doi.org/10.1109/jproc.1998.658762.

[2] Audrey Woods. *The Death of Moore's Law: What it means and what might fill the gap going forward*. https://cap.csail.mit.edu/death-moores-law-what-it-means-and-what-might-fill-gap-going-forward. Accessed: 2024-06-06. 2021.

[3] Charles Shipley and Stephen Jodis. “Programming Languages Classification”. In: *Encyclopedia of Information Systems*. Ed. by Hossein Bidgoli. New York: Elsevier, 2003, pp. 545–552. ISBN: 978-0-12-227240-0. DOI: https://doi.org/10.1016/B0-12-227240-4/00138-6. URL: https://www.sciencedirect.com/science/article/pii/B0122272404001386.

[4] The AGI Podcast. *What is Spiking Neural Network?* Accessed: 2024-06-05. 2019. URL: https://medium.com/@theagipodcast/what-is-spiking-neural-network-fe818ecd0d1b.

[5] Lenz Gregor. *Event Cameras*. Accessed: 2024-06-05. 2023. URL: https://lenzgregor.com/posts/event-cameras/.

[6] Gideon Hinz et al. “Online Multi-object Tracking-by-Clustering for Intelligent Transportation System with Neuromorphic Vision Sensor”. In: *KI 2017: Advances in Artificial Intelligence*. Ed. by Gabriele Kern-Isberner, Johannes Fürnkranz, and Matthias Thimm. Vol. 10505. Lecture Notes in Computer Science. Springer, Cham, 2017, pp. 204–216. DOI: 10.1007/978-3-319-67190-1_11. URL: https://doi.org/10.1007/978-3-319-67190-1_11.

[7] Fabrizio Ottati. *Spiking Neurons: a Digital hardware implementation*. Jan. 2023. URL: https://open-neuromorphic.org/blog/spiking-neurons-digital-hardware-implementation/.

[8] Nandakumar S.R. et al. “Building Brain-Inspired Computing Systems: Examining the Role of Nanoscale Devices”. In: *IEEE Nanotechnology Magazine* 12 (Sept. 2018), pp. 19–35. DOI: 10.1109/MNANO.2018.2845078.

[9] Michael Pfeiffer and Thomas Pfeil. "Deep Learning With Spiking Neurons: Opportunities and Challenges". In: *Frontiers in Neuroscience* 12 (2018). ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00774. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2018.00774.

[10] Lyes Khacef, Nassim Abderrahmane, and Benoît Miramond. "Confronting machine-learning with neuroscience for neuromorphic architectures design". In: *2018 International Joint Conference on Neural Networks (IJCNN)*. 2018, pp. 1–8. DOI: 10.1109/IJCNN.2018.8489241.

[11] Murat Isik. *A Survey of Spiking Neural Network Accelerator on FPGA*. 2023. arXiv: 2307.03910 [cs.AR].

[12] A.L. Hodgkin and A.F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *Journal of Physiology* 117 (1952), pp. 500–544.

[13] Jiankun Chen et al. *SAR Image Classification Based on Spiking Neural Network through Spike-Time Dependent Plasticity and Gradient Descent*. June 2021.

[14] Bouanane Mohamed Sadek et al. "Impact of spiking neurons leakages and network recurrences on event-based spatio-temporal pattern recognition". In: *Frontiers in Neuroscience* 17 (2023), p. 1244675.

[15] Qiang Yu et al. "Temporal Encoding and Multispike Learning Framework for Efficient Recognition of Visual Patterns". In: *IEEE Transactions on Neural Networks and Learning Systems* 33.8 (2022), pp. 3387–3399. DOI: 10.1109/TNNLS.2021.3052804.

[16] Hesham Mostafa. "Supervised Learning Based on Temporal Coding in Spiking Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 29.7 (2018), pp. 3227–3235. DOI: 10.1109/TNNLS.2017.2726060.

[17] Ling Zhang et al. "A Cost-Efficient High-Speed VLSI Architecture for Spiking Convolutional Neural Network Inference Using Time-Step Binary Spike Maps". In: *Sensors* 21.18 (2021). ISSN: 1424-8220. DOI: 10.3390/s21186006. URL: https://www.mdpi.com/1424-8220/21/18/6006.

[18] Simon Davidson and Steve B. Furber. "Comparison of Artificial and Spiking Neural Networks on Digital Hardware". In: *Frontiers in Neuroscience* 15 (2021). ISSN: 1662-453X. DOI: 10.3389/fnins.2021.651141. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2021.651141.

[19] N. Reddy. “FPGA Realization of a High-Speed Spiking Neural Network with Modified LIF Neurons for Pattern Recognition”. In: *Tuijin Jishu/Journal of Propulsion Technology* 44 (Oct. 2023), pp. 8526–8541. ISSN: 1001-4055.

[20] Anand Sankaran et al. “An Event-driven Recurrent Spiking Neural Network Architecture for Efficient Inference on FPGA”. In: *Proceedings of the International Conference on Neuromorphic Systems 2022*. ICONS ’22. Knoxville, TN, USA: Association for Computing Machinery, 2022. ISBN: 9781450397896. DOI: 10.1145/3546790.3546802. URL: https://doi.org/10.1145/3546790.3546802.

[21] R. Paz et al. “Test Infrastructure for Address-Event-Representation Communications”. In: *Computational Intelligence and Bioinspired Systems*. Ed. by Joan Cabestany, Alberto Prieto, and Francisco Sandoval. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 518–526. ISBN: 978-3-540-32106-4.

[22] Fopefolu Folowosele et al. *Wireless systems could improve neural prostheses*. https://spie.org/news/0854-wireless-systems-could-improve-neural-prostheses. Accessed: 2024-05-17. 2007.

[23] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[24] Rui Ponte Costa et al. *Cortical microcircuits as gated-recurrent neural networks*. 2018. arXiv: 1711.02448 [q-bio.NC].

[25] Lei Deng et al. “Rethinking the performance comparison between SNNS and ANNS”. In: *Neural Networks* 121 (2020), pp. 294–307. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2019.09.005. URL: https://www.sciencedirect.com/science/article/pii/S0893608019302667.

[26] Hian Hian See et al. *ST-MNIST – The Spiking Tactile MNIST Neuromorphic Dataset*. 2020. arXiv: 2005.04319 [cs.NE].

[27] Amirreza Yousefzadeh, Teresa Serrano-Gotarredona, and Bernabé Linares-Barranco. *MNIST-DVS and FLASH-MNIST-DVS Databases*. http://www2.imse-cnm.csic.es/caviar/MNISTDVS.html. Accessed: 2024-05-17.

[28] Garrick Orchard et al. “Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades”. In: *Frontiers in Neuroscience* 9 (2015). ISSN: 1662-453X. DOI: 10.3389/fnins.2015.00437. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2015.00437.

[29] Ralf Engbert. "Microsaccades: a microcosm for research on oculomotor control, attention, and visual perception". In: *Visual Perception*. Ed. by S. Martinez-Conde et al. Vol. 154. Progress in Brain Research. Elsevier, 2006, pp. 177–192. DOI: https://doi.org/10.1016/S0079-6123(06)54009-9. URL: https://www.sciencedirect.com/science/article/pii/S0079612306540099.

[30] *TMS320C64x DSP Library Programmer's Reference*. Available at https://www.ti.com/lit/ug/spru565b/spru565b.pdf. Texas Instruments Incorporated. 2003. Chap. Appendix A.2.

[31] Dat Ngo and Bongsoon Kang. "Taylor-Series-Based Reconfigurability of Gamma Correction in Hardware Designs". In: *Electronics* 10 (Aug. 2021), p. 1959. DOI: 10.3390/electronics10161959.

[32] Ling Zhang. "A Cost-Efficient High-Speed VLSI Architecture for Spiking Convolutional Neural Network Inference Using Time-Step Binary Spike Maps". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 67.3 (2020), pp. 759–772.

[33] *GW1NSR Series of FPGA Products Datasheet*. Available at https://cdn.gowinsemi.com.cn/DS861E.pdf. Guangdong Gowin Semiconductor Corporation. 2018. Chap. 3.

[34] Gregor Lenz et al. *Tonic: event-based datasets and transformations.* Version 0.4.0. Documentation available under https://tonic.readthedocs.io. July 2021. DOI: 10.5281/zenodo.5079802. URL: https://doi.org/10.5281/zenodo.5079802.

[35] Jakub Cabal. *Simple UART for FPGA*. https://github.com/jakubcabal/uart-for-fpga. 2016.