POLITECNINCO DI TORINO

Master's Degree in Embedded System

Master's Degree Thesis

Low-power event driven accelerator for Spiking Neural Networks on FPGA

Supervisors

Prof. Stefano DI CARLO

Prof. Alessandro SAVINO

Prof. Alessio CARPEGNA

Candidate

Filippo MAROSTICA

October 2024

Summary

The exponential growth in both research and industry in recent years has made Machine Learning (*ML*) increasingly complex, rendering applications in the embedded field progressively more challenging. This has intensified the need to reduce power consumption and computational load without sacrificing performance. Hence, the focus of this thesis project is to develop a low-power, event-driven accelerator for Spiking Neural Networks (*SNNs*) on FPGA.
SNNs have gained attention due to their potential to address various challenges in artificial intelligence, including event-based processing, low-power computation, and efficient representation of temporal information.

A Spiking Neural Network is a type of artificial neural network that closely mimics the behavior of biological neurons in the brain. Unlike traditional artificial neural networks, SNNs operate based on discrete events called *spikes*. Moreover, since in biological systems information is encoded not only in the strength of connections between neurons but also in the timing and frequency of spikes, SNNs capture this temporal information by considering the precise timing of spikes as part of computation.
SNNs offer various neuron's models with a trade-off between biological plausibility and performance. The basic neuron is the Leaky Integrate and Fire (*LIF*) neuron, it models neural activity by accumulating input current until it reaches a threshold, causing it to "fire" a spike and then reset; it also incorporates a leaky mechanism where the accumulated charge dissipates over time if not triggered, similar to the leakage of ions in a biological neuron.
In the digital domain, spikes in SNNs can be represented as single-bit events, where a spike is active when it occurs and quiescent otherwise. This representation significantly reduces the memory footprint and interconnection resources required, as activations are compressed to just a single bit.
The Event-Driven approach processes information (*spikes*) only when events occur, in contrast to the Clock-Driven approach where computations happen at regular intervals determined by the system's clock. The Event-Driven approach is chosen to optimize performance in terms of power consumption and occupied area, crucial

for embedded systems with limited computational resources. This technique is particularly advantageous for event-driven sensors, which have become increasingly popular over the years for their flexibility and performances.

The project, EDAMAME (Event-Driven Accelerator to Model And Mimick Encephalon behavior), encapsulates its main characteristics and objectives within its name. The workflow unfolds in two distinct phases.
The first part consists in the creation and training of the SNN using Python. Python, coupled with snnTorch (a framework built on top of PyTorch), facilitates the development and training of SNNs extending the capabilities of PyTorch, taking advantage of its GPU accelerated tensor computation and applying it to networks of spiking neurons; it also offers pre-designed spiking neuron models. For the purposes of the project, apart from all the standard functionalities of snnTorch, a custom event-driven neuron was developed within snnTorch to serve as the central component of the designed networks.

In the event-driven neuron, the neuron only updates when a spike arrives and, between two consecutive spikes, the neuron uses a counter to track how much time has elapsed. This neuron uses the counter value to access a precomputed table or memory that stores the results of the decay function for different time intervals ($e^{-t/\tau}$, where the value of t depends on the input sparsity); this means that when a spike arrives after a certain delay, the neuron simply looks up how much the potential has decayed in the elapsed time instead of recalculating the exponential decay.
After implementing the event-driven neuron model in snnTorch, the next steps involved training and testing the network to compare its performance with the default clock-driven model and also for finding the best approach to implement the event-driven quantization. During testing, inference on the trained model showed that the event-driven and clock-driven networks performed very similarly. Sometimes the event-driven model was slightly better, and sometimes the clock-driven one performed better, but overall, both were highly comparable in accuracy.

A key aspect of the training process was the development of a quantized model. Since the trained network's parameters needed to fit within the limited memory of an FPGA, quantisation reduced memory usage by converting floating-point weights into lower-precision formats, such as 8-bit fixed point. This significantly reduced the model size, making deployment on resource-constrained devices like FPGAs feasible.
To achieve this Post Training Quantization (PTQ) was used. In this technique weights are quantized only after training and, even if has reduced performance compared to Quantization Aware Training (QAT) where weights are quantized during training, has an higher degree of customization to meet the requirements of

the target FPGA at the cost of some accuracy reduction.

Following the training phase, during which various datasets were used to validate the model and network parameters were collected, the project transitions into its second, hardware-centric, phase: implementing the spiking neural network (SNN) on an FPGA using System Verilog.
Key aspects of this implementation involve creating a custom architecture able to efficiently represent the network, use different testbenches to test and simulate the hardware accelerator and compare the results with the results obtained in software, adapting the model to FPGA constraints and go through all the steps for the creation of an hardware project such as simulation, synthesis and finally implementation.

Additionally, a comparative analysis was carried out between other hardware accelerator project built using spiking neural networks. This analysis is critical to understanding the hardware's performance across essential metrics such as latency, throughput, and power efficiency.
The comparison showed that EDAMAME presents a well-rounded design focused on achieving efficiency within FPGA limitations, balancing both performance and power requirements. Operating at 100 MHz, EDAMAME achieves comparable throughput to other accelerators while maintaining low power usage. The choice of the LIF neuron model and an event-driven update mechanism aligns EDAMAME with energy-conscious architectures, contrasting with clock-driven designs. Additionally, EDAMAME's utilization of only 17,274 logic cells and 50 DSPs underscores its resource efficiency, especially in comparison to designs with much higher DSP counts. The relatively compact architecture (784-40-10) optimizes for speed and energy efficiency, though it may limit application in highly complex tasks suited to deeper networks.
With power consumption at just 0.182W, EDAMAME is among the most efficient in its class. The reduction in synapse count to 31,760 further enhances its energy profile. While accuracy, at 88.5%, is slightly lower than some alternatives, EDAMAME's trade-offs make it ideal for embedded applications where efficiency is paramount.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Biological neuron models, also known as spiking neuron models, are mathematical descriptions of the conduction of electrical signals in neurons. Neurons (or nerve cells) are electrically excitable cells within the nervous system, able to fire electric signals, called action potentials, across a neural network.
Central to these models is the description of how the membrane potential (that is, the difference in electric potential between the interior and the exterior of a biological cell) across the cell membrane changes over time. In an experimental setting, stimulating neurons with an electrical current generates an action potential (or spike), that propagates down the neuron's axon. This axon can branch out and connect to a large number of downstream neurons at sites called synapses. At these synapses, the spike can cause the release of neurotransmitters, which in turn can change the voltage potential of downstream neurons: this change can potentially lead to even more spikes in those downstream neurons, thus passing down the signal.

1.1 The Biological neuron

The neuron, as the functional unit of the nervous system, consists of various components:

- **Soma** (also known as Neurosoma or cell body): this is the metabolic center of the neuron, housing the nucleus and other organelles responsible for primary cellular functions. From the soma, certain extensions, known as dendrites, arise.

- **Dendrites**: these are extensions that originate from the soma and are responsible for carrying nerve signals centripetally (from the outside towards the soma). Structurally, the dendritic cytoplasm differs from the axonal cytoplasm,

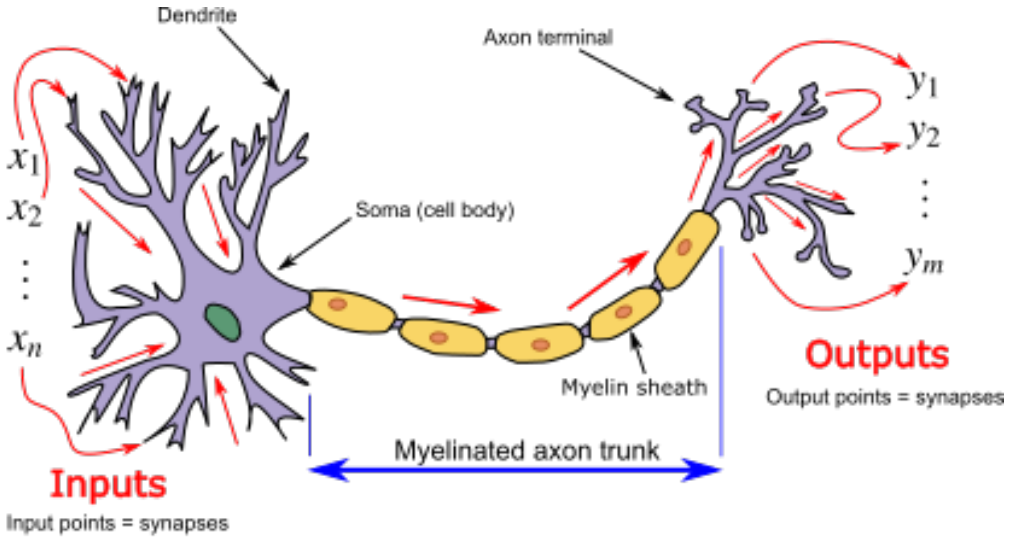

Figure 1.1: Biological Neuron, image taken from Wikipedia [1]

notably due to the presence of mitochondria. In many cases, dendrites may have small protrusions called dendritic spines, which serve as post-synaptic sites for connections with other neurons.

- **Axon**: this is a long, cylindrical structure, generally unbranched, specialized for conducting nerve signals away from the soma (centrifugally). The axon arises from the soma through a region known as the *axon hillock*. The length of an axon can vary widely, reaching up to a meter. At its terminal end, the axon branches into structures known as synapses.

- **Synapses**: these are branching structures originating from the end of the axon, serving as anatomical and functional connections that enable the transmission of nerve impulses to other neurons. Synapses may connect with another neuron, a muscle cell, or an epithelial cell in specialized tissue.

Neuronal synapses can be classified anatomically and physiologically. The anatomical classification recognise three type of synapse: **interneuronal**, if it connects two neurons, **neuromuscular**, if it connects a neuron to a muscle cell, **cytoneuronal**, if it involves a neuron and a specialized epithelial cell.

On the other hand, physiological classification, separate two types of synapses:

1. **Electrical Synapses**: in this type, the action potential passes directly from the presynaptic cell to the postsynaptic cell through transmembrane channels known as gap junctions. This allows for rapid, bidirectional flow of information.

2. **Chemical Synapses**: in this type, information transfer is mediated by chemical substances called neurotransmitters. The presynaptic and postsynaptic cells are separated by a gap known as the *synaptic cleft*. The presynaptic cell contains voltage-gated channels, while the postsynaptic cell has receptors specific to the neurotransmitter.
   Neurotransmitters are released from the presynaptic cell in response to an incoming action potential, binding to receptors on the postsynaptic cell. These receptors are ligand-gated ion channels that open upon neurotransmitter binding, allowing ion entry and altering the membrane potential of the postsynaptic cell. Chemical synapses are generally slower than electrical synapses.

1.1.1 The action potential

Neurons function across all species by transmitting an electrochemical impulse.
The plasma membrane of a neuron is polarized, meaning it exhibits an electrical charge difference between the inside and outside of the cell. This difference is due to a higher concentration of positive ions (sodium ions, Na^+) outside the cell than inside with the consequence of generating an electrical potential difference called the **resting potential**.

The resting potential is maintained by the action of a membrane protein called the sodium-potassium pump, which keeps it stable at approximately –70 mV. The pump achieves this by moving Na^+ ions from the inside to the outside of the cell and potassium ions (K^+) in the opposite direction, from the outside to the inside. The behavior of the two types of ions differs: while potassium ions (K^+) can freely pass through potassium channels, sodium ions (Na^+) cannot move, as the sodium channels remain closed, preventing an even distribution.

The *rest* state is disrupted if the neuron is stimulated, causing the sodium channels to open and the membrane potential to rise from –70 mV to approximately –50 mV, reaching the **threshold value**.
Upon reaching this threshold, many sodium channels open, allowing a rapid influx of Na^+ ions, initiating a sequence of events known as **depolarization** of the membrane; this chain effect continues and, as positive charge concentration inside increases, the potential reverses sharply reaching a value of +30/35 mV (action potential). The time evolution of the neuron potential is shown in Figure 1.2, taken by [2].
Moments later, the sodium channels close, and potassium channels reopen, with the sodium-potassium pump assisting in restoring resting conditions; this process is called **repolarization** of the membrane. If an impulse has been generated, it propagates along the axon membrane of nerve cells.
The picture in Figure 1.3, taken by [2], shows how the neuron manage this ion flux

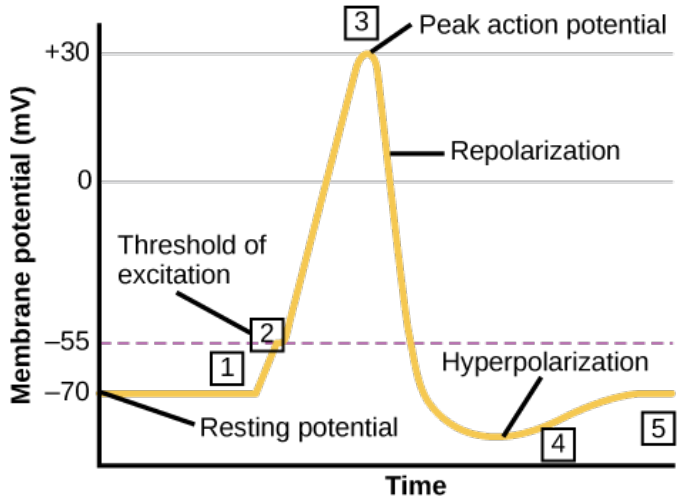

Figure 1.2: Biological neuron membrane potential evolution

when it is stimulated.

The action potential is a localized electrochemical phenomenon and, for impulse transmission to occur, depolarization must propagate from the origin point to the immediately adjacent region. The impulse propagates exclusively in one direction, as the sodium-potassium pump actively restores resting conditions in areas already affected by the action potential.

For a brief moment, the concentration of potassium ions outside the cell becomes higher than under normal resting conditions generating what is called **hyperpolarization** of the membrane, which persists until the –70 mV potential is restored. During this period, which lasts about 2 milliseconds, the membrane cannot respond to any stimulus (**refractory period**), preventing back-propagation of depolarization and effectively ensuring unidirectional transmission of the impulse.

1.2 From the biological neuron to the artificial neuron

It's essential to highlight the differences between biological neurons, traditional artificial neurons, and neurons used in spiking neural networks (SNNs). Biological neurons operate through complex nonlinear processes involving the flow of ions across the membrane, which is modeled, with high accuracy, by the Hodgkin-Huxley equations. These neurons generate action potentials, or spikes, based on dynamic interactions between ion channels, which influence their behavior and communication with other neurons.

In contrast, artificial neurons used in traditional artificial neural networks (ANNs),

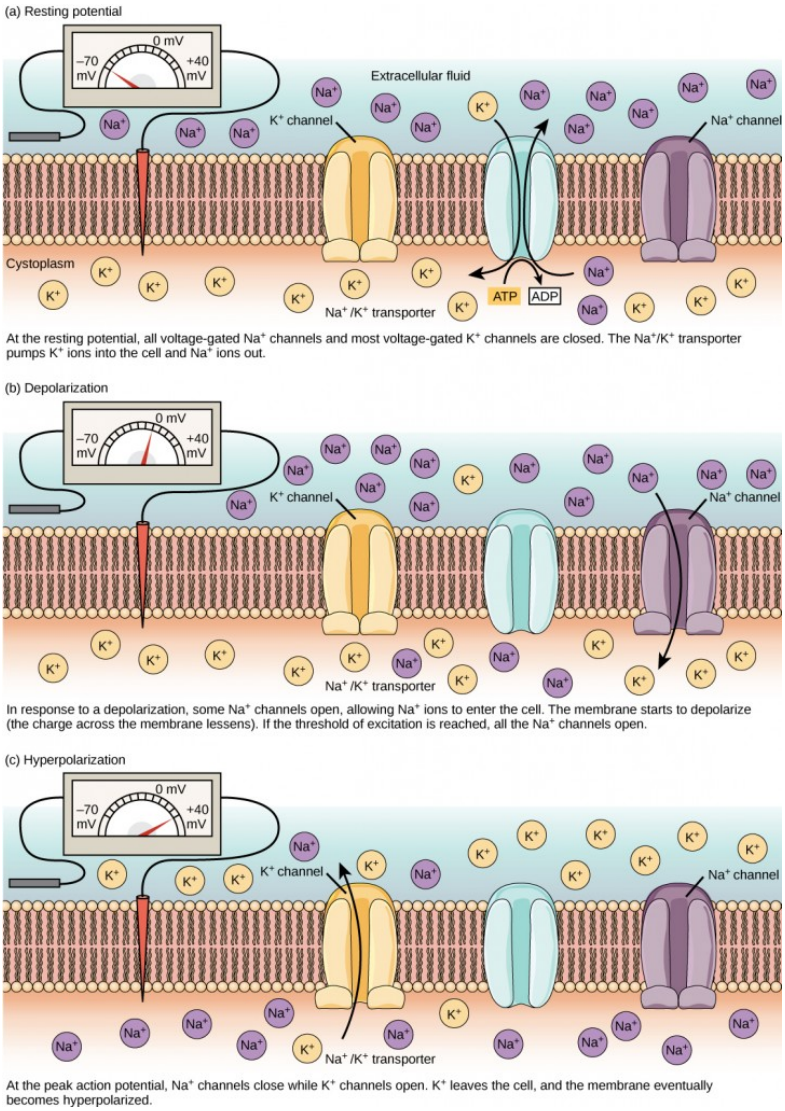

Figure 1.3: How Biological neuron communicate

like multilayer perceptrons or convolutional neural networks (CNNs), operate in a highly simplified manner; they compute weighted sums of their inputs and apply an activation function to determine the output. Unlike biological neurons, traditional artificial neurons do not explicitly represent time or spike-based activity but operate on continuous values, limiting their ability to capture the temporal dynamics seen in real neural processes.

Spiking neural networks (SNNs) introduce a more biologically realistic approach by incorporating neurons like the leaky integrate-and-fire model, which generates spikes based on a neuron's membrane potential crossing a threshold. SNNs simulate

the timing of spikes, allowing them to encode and process temporal information in a manner closer to biological systems. This temporal precision enables SNNs to bridge the gap between the simplicity of artificial neurons and the biological accuracy of models like Hodgkin-Huxley, making them particularly useful for neuromorphic computing and efficient real-time data processing. Figure 1.4, take from [3], shows a comparison of these three models.

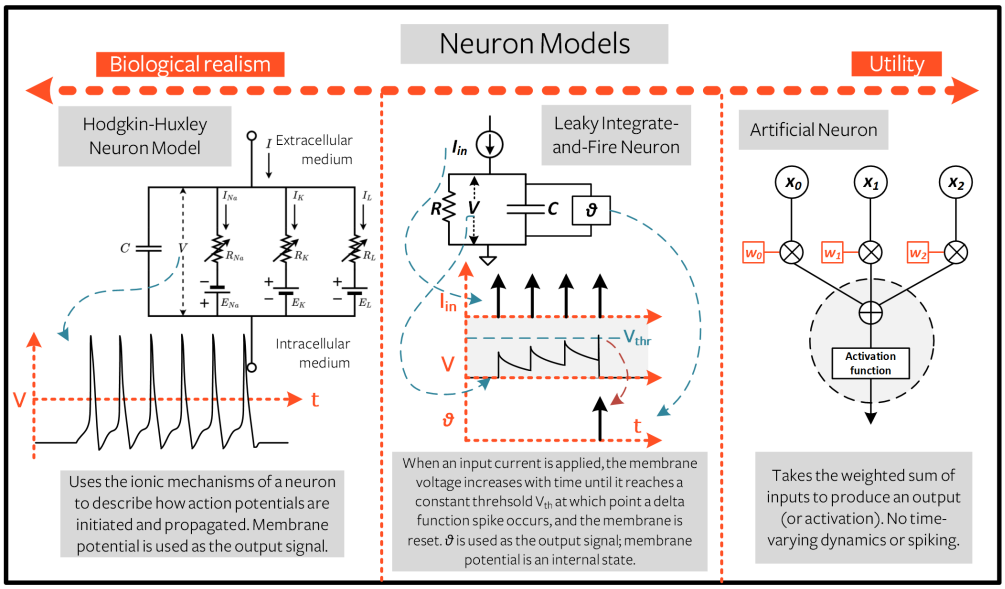

Figure 1.4: Neuron model comparison

1.3 The Neurons classification

Neuron models can be divided into two categories according to the physical units of the interface of the model:

- **Electrical input–output membrane voltage models**: these models produce a prediction for membrane output voltage as a function of electrical stimulation given as current or voltage input. Some models in this category predict only the moment of occurrence of the output spike, other models are more detailed and account for sub-cellular processes.

- **Natural stimulus or pharmacological input neuron models**: the models in this category connect the input stimulus, which can be either pharmacological or natural, to the probability of a spike event. The input stage of these models is not electrical but rather has either pharmacological (chemical)

concentration units, or physical units that characterize an external stimulus such as light, sound, or other forms of physical pressure.

Although neurons can be classified into these two general categories, the variety of biological neuron models is remarkably diverse, often reflecting specific experimental conditions and the complexity of capturing intrinsic neuronal properties. Among these models, the first category (*electrical input–output models*) is particularly significant for artificial neural networks, especially in spiking neural networks (SNNs): in fact these networks rely on simplified models where the spiking behavior mimics biological neurons, laying the groundwork for computational neuroscience.

1.3.1 Electrical input–output membrane voltage models

As stated above, these neuron models are the most relevant for what concern the use of techniques that belong to the human neuron for implementing artificial neurons; in fact, while biophysical models can reproduce electrophysiological results with a high degree of accuracy, their complexity makes them difficult to use at present.

1.3.2 The Hodgkin–Huxley model

The Hodgkin–Huxley (*HH*) model is a model of the relationship between the flow of ionic currents across the neuronal cell membrane and the membrane voltage of the cell. It consists of a set of nonlinear differential equations describing the behavior of ion channels that permeate the cell membrane of the squid giant axon. It is important to note the voltage-current relationship, with multiple voltage-dependent currents ($I_i(t, V)$) charging the cell membrane of capacity C_m, is:

$$C_m \cdot \frac{dV(t)}{dt} = -\sum I_i(t, V) \tag{1.1}$$

The above equation is the time derivative of the law of capacitance, $Q = CV$ where the change of the total charge must be explained as the sum over the currents. The Hodgkin–Huxley model may be extended to include additional ionic currents. The inonic contribution typically include inward Ca^{2+} and Na^{+} input currents as well as several varieties of K^{+} outward currents, including a "leak" current; moreover it is also possible to extend it to take into account the evolution of the concentrations that are otherwise considered constant in the standard version of the neuron.
The high degree of freedom and configurability of this model makes the HH neuron behaviour very close to the "real" neuron however in a model of a complex system of neurons, numerical integration of the equations are computationally expensive. This

problem made simplifications of the model necessary especially in an environment with limited resources as that of an embedded system.

1.3.3 Perfect Integrate-and-fire

One of the earliest models of a neuron is the perfect integrate-and-fire model (non-leaky integrate-and-fire), first investigated in 1907 by Louis Lapicque. A neuron is represented by its membrane voltage V which evolves in time during stimulation with an input current I(t) according to the following relationship:

$$I(t) = C \cdot \frac{dV(t)}{dt} \tag{1.2}$$

which simply represent time derivative of the law of capacitance, $Q = CV$. When an input current is applied, the membrane voltage increases with time until it reaches a constant threshold V_{th}, at which point a delta function spike occurs and the voltage is reset to its resting potential, after that the model continues to run.
In this neuron model the firing frequency is the inverse of the total inter-spike interval so the model can be made more accurate by introducing a refractory period t_{ref} that limits the firing frequency of a neuron by preventing it from firing during that period.
The principal issue with this model is that it describes neither adaptation nor leakage and this characteristic doesn't align with the observed neuronal behavior making it not enough precise and also inefficient when used in Neural Networks.

1.3.4 Leaky integrate-and-fire

The leaky integrate-and-fire (*LIF*) model is an evolution of the integrate-and-fire model described above. It introduces a "leak" term in the membrane potential equation, representing the passive diffusion of ions through the membrane, which better represents the gradual decay of voltage in a biological neuron. This behavior makes the *LIF* neuron more accurate and biologically realistic as it reflects the natural tendency of the neuron to return to its resting state over time.
The model equation is given by:

$$C_m \cdot \frac{dV_m(t)}{dt} = I(t) - \frac{V_m(t)}{R_m} \tag{1.3}$$

From an electrical perspective, the *LIF* neuron can be modeled as an *RC* circuit with a threshold; its behaviour, when input stimuli reaches the neuron, is shown in Figure 1.5, that shows the results obtained by [4].
Each input pulse produces a brief current, and the voltage across the membrane

decays exponentially over time due to the leak term; if the membrane potential reaches a threshold, an output spike (or action potential) is generated, and the membrane voltage is then reset to a resting value. This feature allows the *LIF* model to capture the dynamics of spiking neurons more effectively, and its simplicity makes it widely used in spiking neural networks for computational modeling.
The *LIF* model is central in spiking neural networks because it strikes a balance between computational efficiency and biological realism, providing a tractable yet insightful representation of neuronal dynamics.

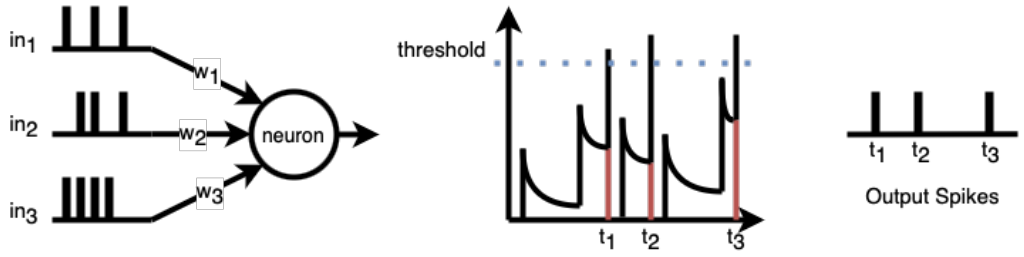

Figure 1.5: LIF neuron model mechanism

1.4 Biological accuracy vs efficiency in SNN

The choice between using a neuron model that closely simulates real biological behavior, such as the Hodgkin-Huxley model, and using a simplified version, like the Leaky Integrate-and-Fire (LIF) model, presents a trade-off between biological accuracy and computational efficiency.
Models like Hodgkin-Huxley provide a highly detailed representation of ion channel dynamics and membrane potentials, making them ideal for studying the precise mechanisms of neuronal behavior. However, this level of detail comes at a cost of high computation intensity, requiring significant processing power, which makes them impractical for large-scale simulations or real-time applications.
On the other hand, simplified models like the LIF neuron offer a much more efficient and scalable solution, obtained with the reduction of the computational overhead by abstracting away many biological complexities; this choice make them particularly advantageous when dealing with artificial systems, such as neuromorphic hardware, where speed and resource constraints are critical factors.
Moreover, when dealing with neuron models, the difference between the human brain and electronic circuits further complicates the direct application of biologically accurate models. The brain operates in a highly parallel and energy-efficient

manner, whereas electronic circuits rely on sequential processing and often consume more power. For this reason using highly detailed models in an embedded system is not only computationally expensive but also unnecessary, as the intricacies of biological ion channels don't translate meaningfully to an electronic system.
Simplified models thus strike a balance, providing enough realism to model key neuronal dynamics while being computationally feasible and adaptable to the differences between biological and electronic systems.

1.5 The Spiking Neural Network

Spiking Neural Networks (SNNs), [5], represent the third generation of artificial neural networks, designed to more closely replicate the behavior of biological neurons compared to traditional models. Unlike standard neural networks, which rely on continuous activation values and compute in fixed time steps, SNNs operate based on discrete events (spikes). This allows them to capture both the strength and precise timing of input stimuli, which is a crucial feature of biological computation. The use of spikes enables SNNs to encode information through temporal patterns, a key distinction from traditional neural networks. This temporal encoding not only mimics the behavior of biological neurons but also makes SNNs highly effective for tasks that require time-based processing, such as sensory data analysis (e.g., vision and auditory signals) and event-driven computing. Their ability to process information over time makes them particularly well-suited for dynamic, real-world applications.
One of the main advantages of SNNs is their energy efficiency. Biological neurons fire only when necessary, and SNNs replicate this sparse, event-driven computation. This means neurons in SNNs only perform calculations when spikes are transmitted, leading to significant power savings compared to traditional artificial neural networks (ANNs), which update all neurons at every time step. As a result, SNNs reduce both computational load and memory usage, making them an attractive choice for low-power hardware.
In terms of architecture, SNNs typically use neuron models like the leaky integrate-and-fire (LIF) neuron and synaptic connections with time-dependent properties. The variety of spiking neuron models available offers a high degree of customizability, allowing networks to be tailored for specific tasks and enabling the creation of biologically inspired, dynamic networks capable of learning and processing complex temporal data. Furthermore, learning in SNNs often involves spike-timing-dependent plasticity (STDP), a mechanism that adjusts synaptic strengths based on the timing of spikes between neurons, further enhancing their ability to mimic biological learning processes.

In conclusion, SNNs [6] represent a significant advancement in neural networks, providing a more biologically realistic and energy-efficient approach to artificial intelligence. As research progresses and dedicated hardware for SNNs continues to evolve, these networks are expected to play a key role in energy-efficient AI, particularly in applications like embedded systems and neuromorphic computing.

Chapter 2

The artificial neural network

There exists different ways of designing SNNs and subsequently to train and run inference using Python and one of them is **snnTorch** [3]. snnTorch is a Python package for performing gradient-based learning with spiking neural networks. It is designed to be used with **PyTorch** [7], as though each spiking neuron were simply another activation in a sequence of layers, extending its capabilities.
snnTorch offers a variety of spiking neuron classes which can simply be treated as activation units with PyTorch; even if its lean requirements enable networks to be trained on CPU, making training possible also on limited performance platforms, snnTorch is also deeply integrated with *torch.autograd* so to take advantage of GPU acceleration in the same way as PyTorch, to further enhance its capabilities. Lastly, the level of integration among PyTorch and snnTorch allows to construct Spiking Neural Networks using a combination of the snnTorch and *torch.nn* packages.
The neurons available in snnTorch are variants of the Leaky Integrate-and-Fire (*LIF*) neuron model offering the developer the possibility to design networks taking advantage of simpler models, as a first order *LIF* or a *Recurrent LIF*, or more complex and advanced ones, as second order Synaptic or its recursive version the RSynaptic, for crafting and configuring the network on the personal needs.

The first steps in the design of the desired network were done using Python by exploiting snnTorch and PyTorch frameworks' capabilities. The wanted result was to create a network using a first order **event-driven** neuron based on the LIF model and subsequently to train this network and verify is effectiveness on the MNIST dataset.

2.1 Clock driven and Event driven

In traditional **clock-driven** approaches neurons are updated at regular intervals based on a fixed time step, whose length is influenced by many parameters such

as the performance capabilities of the chosen platform and also the constraints, like power consumption or heat dissipation especially for devices that work under critical conditions.
Most of everyday electronic devices, including hardware accelerator, are **clock-driven**. In **clock-driven** systems operations are synchronized using a global clock signal that ensures all components update their states at regular intervals. This approach is usually preferred because it simplifies the design and coordination between different components.

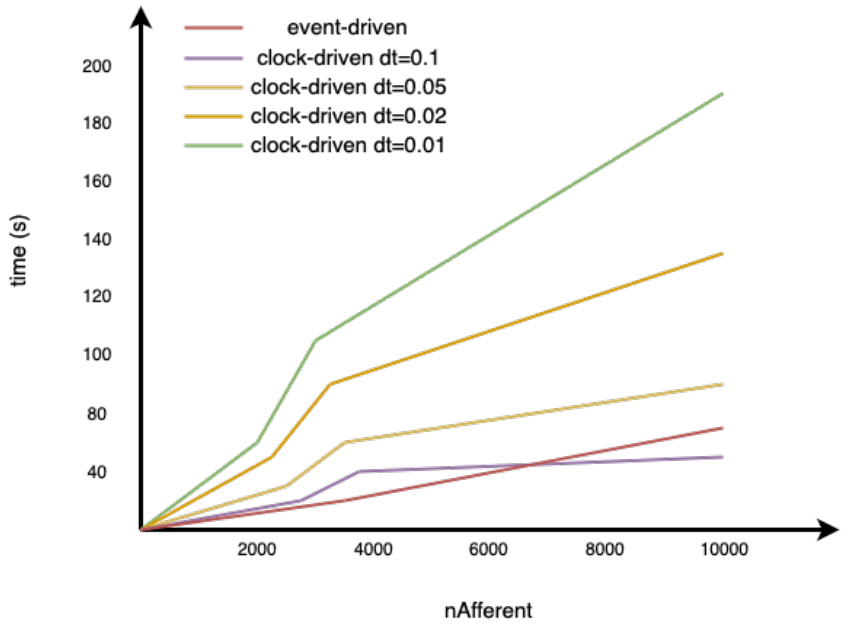

Figure 2.1: Event and Clock driven approach comparison

However the **clock-driven** method has two main disadvantages. First, it limits the temporal precision of neuron updates to the predefined time step; this means that any spike event occurring between time steps will lose some temporal accuracy and it is not possible to recover the accuracy lost in any way since, when the spike arrived, the neuron was involved in executing other operations, this issue become extremely relevant for real-time systems. This drawback, when dealing with SNNs, has important consequences in the performances of the neuron since part of the temporal dynamic is lost going against the nature of biologically inspired neurons used in these systems.
The second disadvantage is that neurons are updated continuously at each time step, even when no spikes are received, resulting in unnecessary computations and

higher energy consumption. This has significant impact on the power consumed by the network especially considering that SNN are usually optimized for working in condition where inputs are sparse, same as for biological neurons.
In contrast, the **event-driven** approach, whose features and characteristics have been deeply discussed in [8], addresses these limitations by only updating neurons when they receive input spikes. This method improves temporal accuracy, as neurons react to spikes as soon as they arrive, providing a finer resolution for spike timing. Furthermore, because neurons are only updated when necessary, event-driven models are more efficient in terms of power consumption.
While the event-driven approach offers clear advantages in timing precision and energy efficiency, it comes with the trade-off of increased computational complexity. Event-driven models require more sophisticated mathematical frameworks to handle asynchronous updates and ensure that neurons react properly to incoming spikes. However, this additional complexity can be justified in systems where power efficiency and timing accuracy are critical, particularly in large-scale SNN implementations or **neuromorphic** hardware where event-driven processing better aligns with the sparse, event-based nature of biological neurons.

Figure 2.1 illustrates a comparison [9], between clock-driven and event-driven simulation approaches in terms of computational time as a function of the number of input connections (*nAfferent*), executed by [10]. The y-axis represents the total simulation time in seconds, while the x-axis represents the number of input connections.
The plot shows how, in the time-driven (*clock driven*) simulations, the computation time increases linearly with both the number of input connections and the resolution of the time step. The smaller the time step (the finer the resolution), the greater the total simulation time.
On the other hand the event-driven simulation shows a much more efficient behavior, with a significantly lower computational time compared to all time-driven simulations, especially as the number of input connections increases. Unlike the time-driven approach, the event-driven simulation time does not vary with the time-step but remains consistently low.

In conclusion, event-driven simulations offer clear computational advantages over time-driven simulations, particularly in cases where input activity is sparse or where minimizing power and processing time is critical. This characteristic makes event-driven models ideal for applications in **neuromorphic** computing where energy-efficient and real-time processing are necessary.

2.2 The LIF model in artificial neural network

As stated above the simpler model that compose the LIF family [11] facilitates the behaviour of biological neuron models assuming that when an input voltage spike occurs, there is an immediate increase in the synaptic current contributing to the membrane potential. The synaptic conductance-based LIF model takes into account the gradual temporal dynamics of the input current due to the fact that in reality the release of neurotransmitters following a spike is a gradual process, not an immediate one.

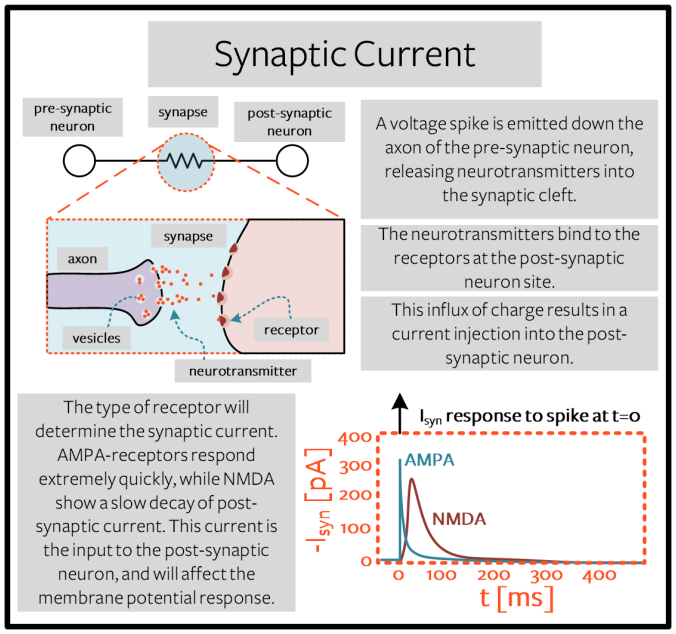

Figure 2.2: Synaptic current

When a pre-synaptic neuron fires, a voltage spike travels down its axon, triggering the release of neurotransmitters from vesicles into the synaptic cleft. These neurotransmitters then activate post-synaptic receptors, influencing the current flow into the post-synaptic neuron. There are two main types of excitatory receptors: **AMPA** and **NMDA**.

1. **AMPA Receptors:** in the biological neuron the conductance of AMPA receptors is voltage-independent, meaning that once the receptor is activated, it rapidly allows ions to pass through regardless of the **membrane potential**. This behavior is what makes AMPA receptors responsible for the initial fast component of excitatory synaptic transmission.

2. **NMDA Receptors:** on the other hand NMDA receptors are involved in a

slower component of excitatory synaptic transmission compared to AMPA receptors due to their voltage-dependent properties. They are particularly important for synaptic plasticity mechanisms such as long-term potentiation (LTP) and long-term depression (LTD).

Figure 2.2, taken from [3], shows how the simplest model of synaptic current assumes a rapid increase followed by a slow exponential decay, as observed in the **AMPA** receptor response.

2.2.1 Mathematical derivation LIF neuron

The **AMPA** model mirrors the membrane potential dynamics of Lapicque's model. From the biological point of view neurons are cells and, like all cells, they are surrounded by a thin membrane (*lipid bilayer*), that has the scope of insulating the conductive saline solution within the neuron from the extracellular medium; however, another function of this membrane is to control what goes in and out of this cell. The membrane is usually impermeable to ions and, thanks to specific channels in the membrane whose activity is electrically regulated, the neuron can exchange ions with the external world controlling the flux of ions entering and exiting the neuron body.

These two behaviors of the biological neuron find a direct translation into an electrical circuit: the RC circuit. In this circuit, the capacitor represents the two conductive solutions separated by an insulator while the resistor models the charge movement toward the neuron.

According to the described model the input current $I_{in}(t)$ is equal to the sum of the contribution on the resistor branch added to that on the capacitor's one, as shown in the Equation 2.1.

$$I_{in}(t) = \frac{U_{mem}(t)}{R} + C \cdot \frac{dU_{mem}(t)}{dt} \tag{2.1}$$

From this expression, the **membrane potential** can be recovered showing that the passive membrane is described by a linear differential equation as shown by Equation 2.2.

$$\tau \frac{dU_{mem}(t)}{dt} = -U_{mem}(t) + RI_{in}(t) \tag{2.2}$$

with $\tau = RC$, representing the *decay rate*.

Representing with U_0 the initial value of the neuron **membrane potential**, with no further inputs, the solution of the previous differential equation leads to the following expression:

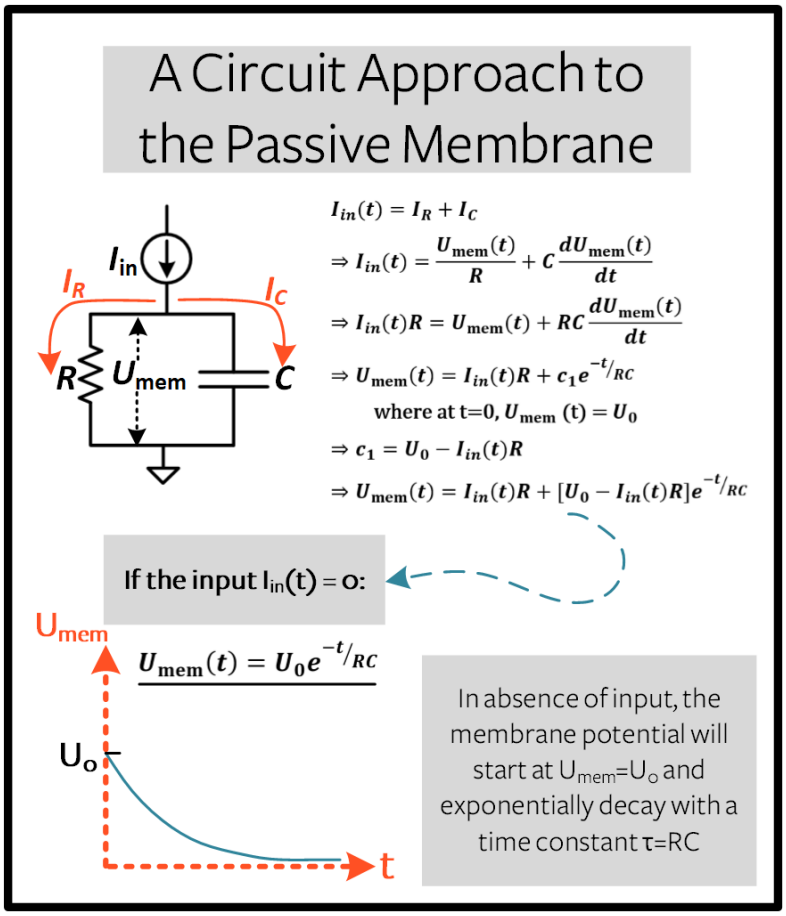

Figure 2.3: Membrane potential equation

$$U_{\text{mem}}(t) = I_{\text{in}}(t)R + [U_0 - I_{\text{in}}(t)R]e^{-\frac{t}{\tau}} \tag{2.3}$$

A further simplification consists in considering the input current $I_{in} = 0$, in this way the previous solution can be further reduced:

$$U_{\text{mem}}(t) = U_0 e^{-\frac{t}{\tau}} \tag{2.4}$$

The Equation 2.4 shows how the circuit, and consequently the neuron, reacts when it starts from a membrane potential U_0 and decays exponentially when no inputs stimuli reaches the neuron. This model well represents the decay behaviour that characterize biological neurons.
Figure 2.3, taken from [3], shows a synthetic representation of the mathematical model described. In the top left the electrical representation of the neuron behaviour

through the RC circuit, on the right the full derivation and in the bottom left how the membrane potential behaves according to the model.

2.3 Clock driven implementation of LIF neuron in snnTorch

Having defined the basic differences between the **event-driven** and **clock-driven** approaches and how neurons behave under each, we can now explore how an event-driven neuron can be implemented using the **snnTorch** framework, particularly focusing on the Leaky neuron model.
In snnTorch, the exponential decay of the neuron's membrane potential over time is approximated using a simplified, quantized model. Calculating the exact exponential decay is a very computationally intensive task so an approximation is employed to make the model more efficient, especially for real-time simulations and hardware implementations. This is where the β factor comes into play.

2.3.1 The β coefficient for the exponential quantization

The beta coefficient (β) is used to approximate the decay between two consecutive time steps of the **membrane potential**.

$$\beta = \frac{U_{mem}(t + \Delta t)}{U_{mem}(t)} \tag{2.5}$$

The value of β describes the ratio between the membrane potential at two successive time steps, allowing for an efficient and simplified calculation of the decay process without needing to compute the full exponential function at every step.
By further simplifying the equation, as shown below, we can obtain a clearer representation of how the quantization of the exponential decay is obtained:

$$\beta = \frac{U_0 e^{-\frac{t+\Delta t}{\tau}}}{U_0 e^{-\frac{t}{\tau}}} = e^{-\frac{\Delta t}{\tau_{mem}}} \tag{2.6}$$

In the expression above τ represents the membrane time constant, which determines how quickly the potential decays, and Δt is the time step. In this formulation the value U_0 cancels out because we are evaluating the ratio between two membrane potentials related to the same neuron under the same conditions.

While the use of the quantization significantly improves computational efficiency, it introduces certain limitations.

- **Loss of precision**: the approximation, compared to the full exponential calculation, lead to a slight loss of precision especially in high-frequency spiking scenarios. This uncertainty can be mitigated introducing smaller time steps ($\Delta t << \tau$, for reasonable accuracy) but it is never canceled completely.
- **Inflexibility**: the β-based model assumes a fixed time constant and constant conditions between time steps. This may not capture more nuanced variations in decay that could occur in more complex neuron models.

Despite these drawbacks, the use of β provides several key advantages that make it a highly effective trade-off in many applications:

- **Efficiency**: the most significant advantage of using β is its computational simplicity. By reducing the complexity of the exponential decay calculation, the model becomes more efficient, enabling real-time simulations and allowing the use of more complex networks without excessive computational overhead.
- **Scalability**: the β approximation allows SNNs to scale better when deployed in hardware or on *neuromorphic* systems, where memory and processing power are limited; the slight loss of precision is an acceptable trade-off for the significant gains in performance.

2.3.2 Complete mathematical model

To model the behavior of a neuron in a Spiking Neural Network (SNN), we start with the general mathematical formula that describes how the neuron's membrane potential evolves over time. The membrane potential $U_{mem}[t]$ is influenced by both the inputs the neuron receives and its internal state.
The equation governing this process is:

$$U_{\text{mem}}[t] = U_{\text{mem}}[t_0] \cdot e^{-\frac{t}{\tau}} + WX[t] - R[t] \tag{2.7}$$

This mathematical expression allows to evaluate the **membrane potential** between two consecutive inputs.

The factor $U_{mem}[t_0]$ is the membrane potential at time t_0 when the last input arrives and it is multiplied for the exponential factor since it decays over time, the $X[t]$ member corresponds to the incoming input at time t and they are weighted according to the W coefficient, lastly, the reset term $R[t]$ is what ensures that, after a spike is fired, the membrane potential drops back down, preventing continuous firing and allowing for subsequent spikes to be generated only when appropriate stimuli are received.

This model is finally adapted using the β-based model for the exponential quantization and the final expression is as follows:

$$U_{\text{mem}}[t + \Delta t] = \beta U_{\text{mem}}[t_0] + WX[t] - R[t] \tag{2.8}$$

2.3.3 "Firing" and Reset mechanism

The concept of a spike, or action potential, is central to SNNs. When the membrane potential U_{mem} exceeds a certain threshold $U_{\text{threshold}}$ the neuron "fires" a spike, sending an output signal to the connected neurons. This binary event defines how the neurons communicate in the network, and it is described by the following condition:

$$S_{out} = \begin{cases} 1 & \text{if } U(t) > U_{\text{threshold}}, \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

The factor S_{out} is the output spike and has a binary format; it takes the value of 1 when the **membrane potential** exceeds the threshold, indicating a spike, and 0 otherwise.

When a Spike occurs the neuron is reset. From a biological point of view this phenomenon is called to as *hyperpolarization*, it consists in a temporarily condition where it is more challenging for the neuron to fire again, thereby conserving energy. In the artificial neuron model, the reset mechanism is used to mimic this process.

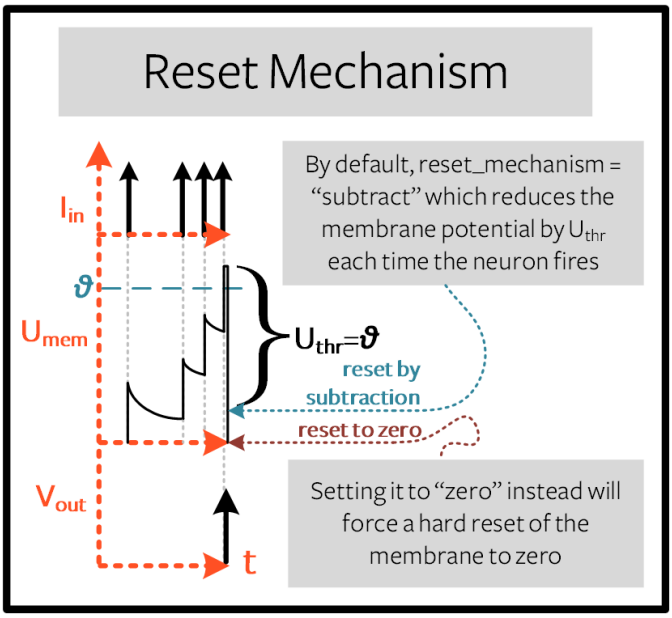

Figure 2.4: LIF neuron reset mechanism

There are three methods to implement the reset mechanism:

- **Reset by subtraction**: this is the default method used in snnTorch neurons, in this case the membrane potential is reduced by subtracting the threshold voltage each time a spike is generated.
- **Reset to zero**: the membrane potential is reset to zero after a spike is triggered, essentially forcing the neuron to start from a neutral state.
- **No reset**: no action is taken after a spike, which can lead to uncontrolled firing and continuous spiking if no other inhibitory mechanisms are applied.

The graph in Figure 2.4, taken from [3], shows how the reset by subtraction and the reset to zero behaves. The main difference between the two approaches is that applying *subtraction* the neuron does not ignore how much the membrane exceeds the threshold and, consequently, the model get close to how the biological neuron behaves. On the other hand, applying a hard reset with *zero* promotes sparsity and potentially less power consumption at the cost of biological accuracy reduction and some performance degradation especially when dealing with correlated information.

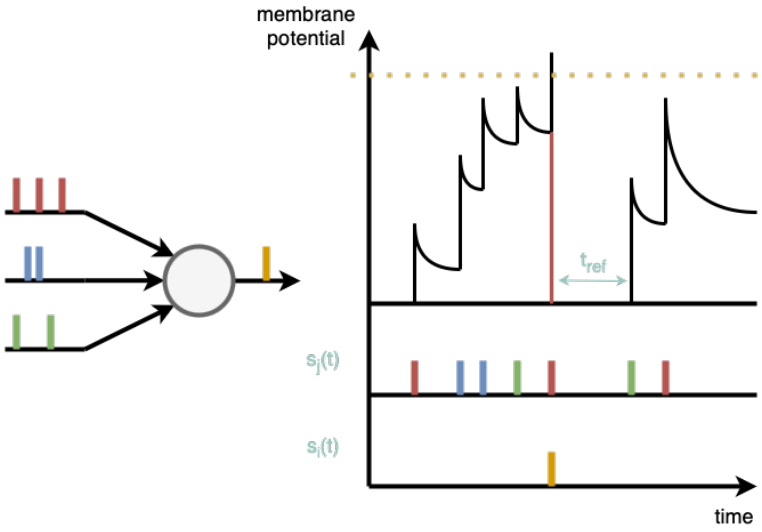

Figure 2.5: LIF neuron behaviour

The plot in Figure 2.5, [12], is a schematic representation of a neurons that is excited by different inputs in different time step and whose **membrane potential**, $v_i(t)$, evolve in time following the previously described relation. The plot $s_j(t)$ shows a flattened representation of the inputs to better visualize their distribution

in time and the plot $s_i(t)$ shows the output spike.
The reset mechanism employed is the *reset to zero* in fact, when the potential reaches the threshold v_{th} (dashed line), the potential goes immediately to zero; this graph shows very well that, with this reset method, the information about how much the threshold has been overcome is "lost". Lastly this model includes a further contribution that is the *refractory period* (Δt_{ref}); it is the brief time after a neuron fires a spike during which it is unable to fire again ensuring that the neuron doesn't immediately reactivate, allowing recovery and preventing continuous spiking, emulating the *hyperpolarization* behavior of a biological neuron.

2.3.4 From clock-driven to event-driven neuron in snnTorch

The previously described method using the β coefficients is highly efficient in terms of hardware utilization but is not well-suited for an event-driven approach. In the standard clock-driven model, the membrane potential is updated at every time step Δt, regardless of whether the input X is *0* or *1*. This continuous update process leads to unnecessary computations when no spike is present, which negates the efficiency advantages typically desired in event-driven systems.

As discussed earlier, the coefficient β represents the ratio between consecutive values of the membrane potential in the absence of input spikes, essentially quantifying the decay over a fixed time interval (Δt). In an event-driven model, to conserve resources, we employ two key components: a counter and a Look-Up Table (LUT).

- **Counter**: this counter is reset every time a new input spike arrives. It keeps track of the elapsed time between consecutive inputs and it is the only active component when the input is *0*.

- **Look-Up Table (*LUT*)**: the LUT is a small memory that stores precomputed decay coefficients (β^i values) corresponding to different time intervals.

The system operates as follows: when the input is *0*, the counter runs continuously, helping conserve resources. By increasing the counter's precision (i.e., increasing the frequency), the system's accuracy can be enhanced. The upper bound is the input source speed, exceeding its frequency results in an unnecessary waste of resources.
When an input spike arrives, the counter stops, and its value is used as an index to access the LUT. The value retrieved from the LUT is used to compute the exponential decay of the membrane potential over the time interval when the input was *0* thus reducing the computational complexity compared to the clock-driven model.

To clarify the process of obtaining the Look-Up Table (LUT) index, we break down the steps for the membrane potential over multiple time steps in an event-driven model. The membrane potential $U_{\text{mem}}[t]$, at any time t, can be expressed using the decay factor β as follows:

$$U_{\text{mem}}[t] = \beta(\Delta t) \cdot U_{\text{mem}}[t_0] \tag{2.10}$$

In the expression $\beta(\Delta t)$ is the LUT and $\Delta t = t - t_0$.

Example: Calculate for $t = 3$ with $\Delta t = 1$

We apply this to a specific case where $t = 3$ and $\Delta t = 1$. t represents the time instant where an input spike reaches the network so, in the time interval $[t_0, t] = [0,3]$, the **membrane potential** has an exponential decay behavior without any source of excitation.

$$U_{\text{mem}}[1] = \beta U_{\text{mem}}[0] \tag{2.11}$$
$$U_{\text{mem}}[2] = \beta U_{\text{mem}}[1] \tag{2.12}$$
$$U_{\text{mem}}[3] = \beta U_{\text{mem}}[2] \tag{2.13}$$

By substituting the values of $U_{\text{mem}}[1]$ and $U_{\text{mem}}[2]$ into the expression for $U_{\text{mem}}[3]$, we get:

$$U_{\text{mem}}[3] = \beta \cdot \beta \cdot \beta \cdot U_{\text{mem}}[0] = \beta^3 U_{\text{mem}}[0] \tag{2.14}$$

For a generic time T (time interval between two consecutive *'1'* inputs), the **membrane potential** can be expressed as:

$$U_{\text{mem}}[T] = \beta^T U_{\text{mem}}[0] \tag{2.15}$$

This final equation shows that the membrane potential at a generic time $t = T$ is equal to the initial membrane potential $U_{\text{mem}}[0]$ scaled by the decay factor β raised to the power of T.
Since β^T values repeatably decay over time, we can precompute these values for different time steps T and store them in a Look-Up Table (LUT). When an input arrives we can retrieve the precomputed β^T from the LUT where the value T corresponds to the value stored in the counter register.
The generation of the Look-Up Table is straightforward and can be done with just a few lines of Python code, making it also easier for a successive deployment of the Table in hardware.

Listing 2.1: LUT generation in Python using predefined beta

```
# Define the beta value
beta = 0.8
# Number of elements in the list
num_elements = 60
# Calculate the corresponding U values
LUT = [pow(beta, x) for x in range(num_elements)]
```

This method efficiently adapts the standard clock-driven model to an event-driven approach, significantly reducing computational overhead while maintaining accuracy by leveraging precomputed decay values.

Example: Membrane potential in clock-driven and event-driven

The plot in Figure 2.6 illustrates the behavior of both the clock-driven and event-driven neurons. In this simulation, the initial membrane potential (U_{mem}) is set to 2.5, and the neuron receives two inputs, one at time $t = 10$ and another at time $t = 15$.
In the clock-driven plot, the membrane potential decays exponentially over time. In contrast, during this transient period, the event-driven neuron remains fixed, as it only updates when an input is received. However, upon the arrival of each input, both neurons reach the same value, demonstrating that their fundamental behavior is identical. Finally, when the input at $t = 15$ causes the **membrane potential** to exceed the threshold ($U_{\text{th}} = 3$), both neurons reset to zero.

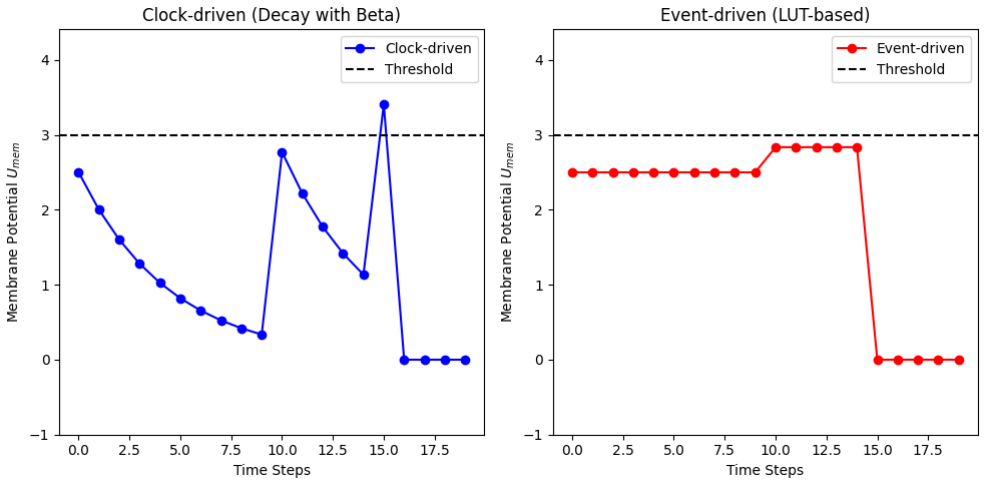

Figure 2.6: LIF neuron clock and event driven comparison

2.4 The network architecture

The following step, after defining and testing the single neuron, was to go through the real implementation of a network using the designed **event-driven** neuron and testing its performance. When using Python, since it is an interpreted language, resources usage is very difficult to monitor so the performed test were done mainly to explore the potentiality of the developed neuron, to verify its functionality and to obtain the network parameters after training as a starting point for the real hardware implementation of the accelerator on FPGA.

2.4.1 The Network Definition

The combined use of snnTorch and PyTorch provides a powerful framework to define, train, and run Spiking Neural Networks (SNNs) efficiently.
In this implementation, we employ the well-known **MNIST dataset**, a benchmark dataset in machine learning, consisting of 60,000 training images and 10,000 test images of handwritten digits (0-9). Each image is grayscale with a resolution of 28x28 pixels.
The reason why MNIST is widely used, especially as a benchmark dataset, it is because it is a relatively simple dataset that enables researchers to test and validate the performance of neural networks, especially for classification tasks. This simplicity is ideal for demonstrating the capability of SNNs in solving classification problems while also allowing for comparison with traditional Artificial Neural Networks (*ANNs*). An example of the items contained in the described dataset is shown in Figure 2.7.

Figure 2.7: MNIST dataset

The network itself is a **three-layer spiking neural network**, as depicted in the scheme in Figure 2.8, taken from [3]. It consists of an input layer, a hidden

layer of spiking neurons, and an output layer. The spiking neurons in the hidden and output layers are modeled using custom event-driven Leaky Integrate-and-Fire (*LIF*) neurons implemented on top snnTorch while the connections between layers are handled by PyTorch modules, such as `nn.Linear()`, which define the synaptic weights between neurons.

The network is composed of two distinct parts:

1. **Network Definition:** the network is implemented as a subclass of *nn.Module*, which is the base class for all neural network modules in **PyTorch**. The architecture comprises two fully connected layers: the first hidden layer (`fc1`) consists of 1000 neurons, and the output layer (`fc2`) consists of 10 neurons, corresponding to the 10 digits (0-9) in the MNIST dataset, to match the dimension of the target classification group.

2. **Forward Method:** the forward method defines how the input image data (`x`) propagates through the network. The image data first passes through the hidden layer (`fc1` and `lif1`), where it is processed by the spiking neurons, next, the output of the hidden layer is passed to the second fully connected layer (`fc2` and `lif2`), then, the network records both the spikes and **membrane potentials** at every time step, lastly, the recorded values, are stacked and returned at the end of the forward pass, providing the output for classification.

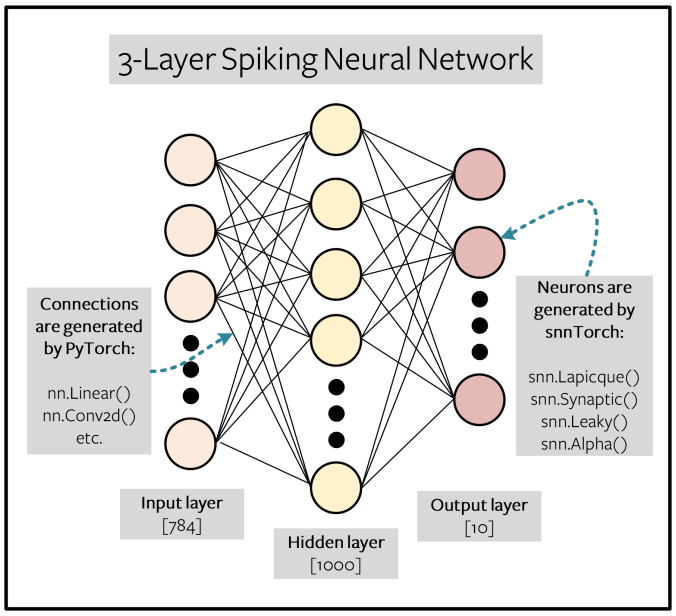

Figure 2.8: MNIST network scheme

The network shown in Figure 2.8 illustrates the three-layer structure: the input layer consists of 784 neurons (28x28 pixels flattened), the hidden layer has 1000

spiking neurons, and the output layer contains 10 neurons, each corresponding to a digit from 0 to 9.

2.4.2 The Network parameters

The choice of a fully connected (FC) network architecture, instead of a more complex structure, is based on the effectiveness of this architecture for handling small-scale, low-resolution image classification problems. Here are the main reasons why the FC network was preferred:

- **Small Dataset**: when dealing with small dataset (in terms of complexity and size) it is feasible to use a fully connected architecture without overcomplicating the network.
- **Feature Learning**: in fully connected layers, each neuron receives input from all neurons in the previous layer, allowing the network to learn global patterns and relationships across the entire input image. This global connectivity is often sufficient to capture the necessary features for accurate classification.
- **Ease of implementation**: fully connected layers, thanks to their regular structure, are easier to implement in a second moment when the network will be implemented in hardware.
- **Future customization**: in this specific application the use of **snnTorch** for developing the network is a preparation step since the target platform is the FPGA; for this reason a FC network give more space for customisation when the network will be deployed in hardware.

Another critical step is the choice of the layers' dimension since this strongly influence performance but also dimension and scalability of the network.

The first layer is the Input Layer; the made choice is to match the number of pixels in the MNIST images, which is $28 \times 28 = 784$ neurons. This is not an unbreakable rule but it is preferable since this ensures that each pixel in the image corresponds to a unique input neuron.
On the other hand the last layer is the output layer; its size must corresponds to the number of classes. Since MNIST contains 10 digits (0-9), the output layer must contain 10 neurons. Each output neuron represents the likelihood of the input image belonging to a specific class.
The last choice consists in how to handle the hidden layer. Usually the hidden layer size is an empirical choice; a larger hidden layer allows the network to learn more complex features but comes at the risk of **overfitting** on the other hand a smaller hidden layer reduces the risk of **overfitting** but might limit the network's ability to learn intricate patterns, encountering underfitting.

Moreover, this choice strongly influences the final dimension of the network; since the hidden layer is the biggest one trying to minimize its dimension helps a lot for reducing the hardware accelerator complexity. The hidden layer's size is a fundamental parameter of the network and the best approach is to test the model under different conditions to verify the effectiveness of such choices.
As a final consideration, in this example, only one hidden layer is used. Adding more layers would increase the model's depth, potentially allowing it to learn more hierarchical features, but this would drastically increases computational complexity.

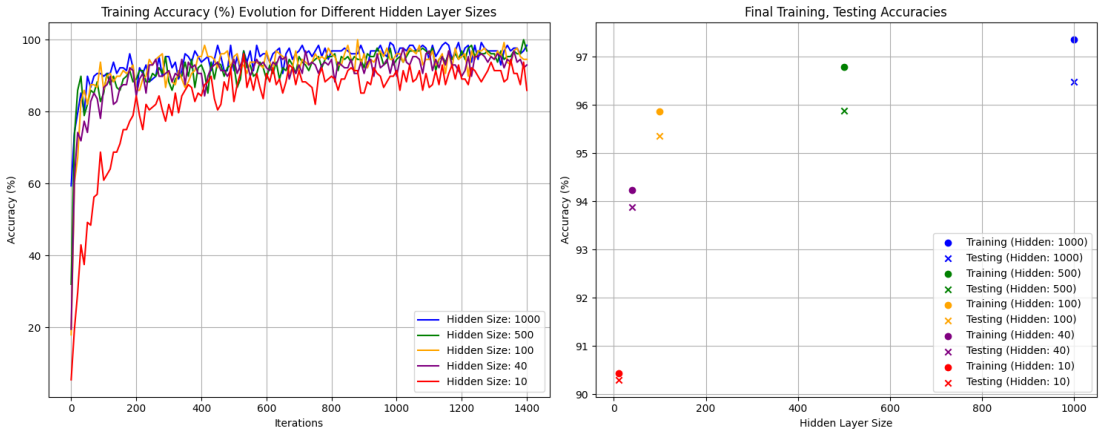

Figure 2.9: MNIST Training with different hidden layer size

The plots in Figure 2.9 demonstrate the evolution in time of the network's training accuracy across different hidden layer sizes. The left plot shows how the training accuracy changes over iterations.
The following key observations can be drawn:

- Larger hidden layers (e.g., 1000 and 500 neurons) tend to reach higher accuracy levels quickly and stabilize around the 95-100% mark; they show minimal fluctuations over time, indicating more stable learning behavior.

- Smaller hidden layers (e.g., 100 neurons and below) also reach reasonable accuracy but at a slower rate, with more oscillation in their learning curves. Particularly, the network with only 10 neurons struggles the most to reach higher accuracy, converging around 90%.

- The relationship between hidden layer size and accuracy is not linear: while reducing the number of neurons decreases the accuracy, the difference is not drastic for sizes like 100 or 40 neurons. This shows that even significantly smaller networks can achieve relatively high performance.

On the other hand the right plot provides a summary of the final training and testing accuracies for each hidden layer size. It reveals:

- Larger hidden layers (1000 and 500 neurons) achieve both high training and testing accuracies, with a minimal gap between the two, suggesting good generalization.

- The hidden layer of 100 neurons and the one with 40 neurons also perform well, achieving around 95% training accuracy and slightly lower testing accuracy, showing that reducing the size still leads to strong performance.

- For smaller networks (10 neurons), the training accuracy drops significantly, and this reduction is mirrored in the testing accuracy.

This analysis shows that reducing the hidden layer size does have an impact on accuracy, but the dependency is not strictly linear. Networks with smaller hidden layers can still perform reasonably well, particularly when considering the trade-off in terms of network complexity.
For example, observing the results obtained for the network with 1000 neurons and for that with 100 neurons while reducing the hidden size by a factor of 10, the testing accuracy drops only slightly (97% to 96%). This represents a significant reduction in network size and computational cost with only a minor impact on performance.
From a practical standpoint, reducing the hidden layer size is an effective strategy to lower the computational complexity and the physical dimension of the network while still maintaining relatively high accuracy.

It's important to note that the choice of dataset plays a significant role in this trade-off. For instance, the MNIST dataset, while commonly used as a benchmark, is relatively simple; this simplicity allows for more aggressive reductions in network size without severely impacting performance while, for more complex datasets, the hidden layer size would have a much greater influence on the network's overall performance, making it a more critical factor to consider.

Listing 2.2: Network Event Driven

```
# Network Architecture
num_inputs = 28 * 28
num_hidden = 1000
num_outputs = 10

# Temporal Dynamics
num_steps = 100
beta = 0.8
```

In the code above, the key parameters for the network are defined, showing how the core architecture and temporal dynamics parameters are defined using PyTorch.

2.5 Training and Inference

The training process for a Spiking Neural Network (*SNN*) involves several key stages, including data preparation, network initialization (described above), loss function and optimizer setup, and iterative training over multiple epochs.

The main steps leading up to the training loop are summarized as follows:

1. **Data Preparation:** in the preparation phase, the MNIST dataset is loaded and transformed into tensors. The dataset is split into training and test sets, and DataLoader objects are created for both sets to facilitate batch processing.

2. **Network Initialization:** the network is defined as described in the architecture section, using layers and neurons adapted to the SNN framework.

3. **Device Setup:** the network is loaded onto the appropriate device, either CPU or GPU. Using a GPU, when available, speeds up both the training and inference phases due to its capacity for parallel processing.

4. **Loss Function Definition:** a loss function measures the discrepancy between predicted and true labels. In this case, the cross-entropy loss function is chosen to quantify classification errors.

5. **Optimizer:** an optimizer is needed to update the network's parameters during training. For this implementation, the Adam optimizer is selected due to its efficiency in handling non-convex optimization problems.

Once these preliminary steps are executed, the network is ready for the training loop, which iterates over the dataset for a predefined number of *epochs* and updates the network's parameters based on the computed loss. Here are the key phases of the training process:

- **Epoch Loop**: the outer loop runs for a specified number of epochs (equal to: `num_epochs`). In each epoch, the network is trained on the entire training dataset once.

- **Minibatch Loop**: during each epoch, the dataset is split into smaller sets called mini-batches. This improves training performance by optimizing memory usage and speeding up gradient calculations.

- **Forward Pass**: for each mini-batch, input data is passed through the network to generate spike records (`spk_rec`) and membrane potential records (`mem_rec`) over multiple time steps.
- **Loss Calculation**: for each time step, the cross-entropy loss is computed between the predicted membrane potentials and the target labels. The total loss accumulates over all time steps.
- **Backpropagation and Optimization**: the loss is used to compute gradients via backpropagation. The optimizer then updates the network's weights using these gradients to minimize the loss.
- **Test Set Evaluation**: periodically, the model's performance is evaluated on the test set. During evaluation, the network is set to inference mode using `net.eval()`, and a forward pass is performed on the test mini-batches. The test loss and accuracy are computed and recorded.
- **Printing Performance Metrics**: the `train_printer` function periodically prints performance metrics, such as training and test loss, as well as accuracy on both the training and test sets.

At the end of the training process, the model is evaluated on the complete test set to calculate the overall accuracy. The trained model parameters are then saved to a file for future use.
Training SNNs is a complex, multi-stage procedure with many interdependent phases. Figure 2.10 shows a comparison between the training process of a standard **clock-driven LIF** neuron (as implemented in snnTorch) and the custom **event-driven LIF** neuron; in the image is also highlighted the point where the two different model reach the highest accuracy (both for testing and training) to show how they both behaves in a very similar way showing a very similar final accuracy obtained at approximately at the same rate.

2.5.1 Running Inference on the Saved Model

After completing the training phase and saving the model's parameters, the final step is to run inference on new, unseen data. Inference is a process that verifies whether the trained model generalizes well beyond the data it was trained on and ensures that the model performs as expected in real-world scenarios. This is done by testing the model on data it has not encountered during the training process, and it is often used as a measure of the model's robustness and generalization capability.
In PyTorch, once the model has been successfully trained, the learned parameters, such as weights and biases of each layer, are saved in the form of a state dictionary,

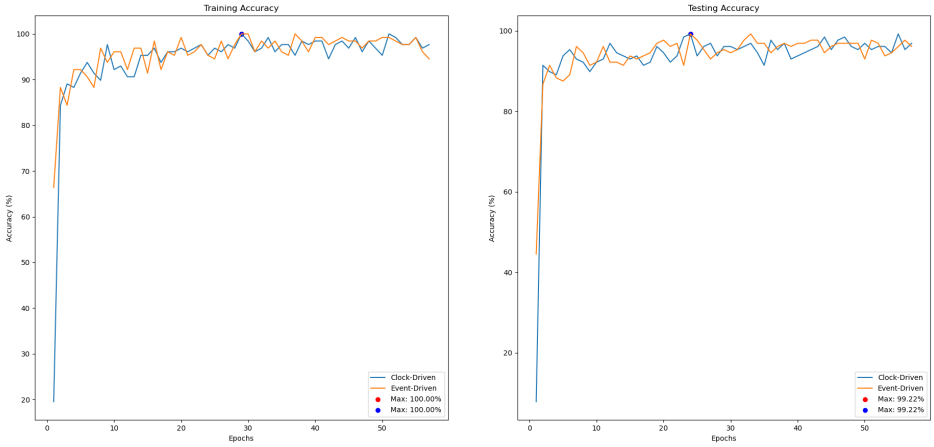

Figure 2.10: MNIST training/testing using standard and custom LIF neuron

or `state_dict`. The state dictionary (*state_dict*) is a Python data structure used in PyTorch to store the parameters of a neural network; it contains the model's learnable parameters organized in a format that maps each parameter to its corresponding tensor. In the dictionary, parameters are organized by layer and neuron, with each entry named according to its layer and the type of parameter (e.g., layer1.weight, layer1.bias), this regular structure facilitates the extraction process of the necessary data.
Saving the `state_dict` ensures that the trained model becomes portable, allowing it to be exported, shared, and deployed across different systems or environments for inference tasks. Furthermore, by decoupling the trained parameters from the model architecture, the `state_dict` allows for flexible reuse of the model without the need for retraining.

The following code snippet demonstrates how easy it is to export the model's learned parameters by saving the `state_dict` at the end of the training phase:

Listing 2.3: Saving the trained model parameters to a state dict

```
# Save the trained model parameters
torch.save(net.state_dict(), "trained_model.pt")
```

To perform inference using this saved model, the first step is to load the saved `state_dict` into a model with the same architecture used during training. It is critical to ensure that the network is built exactly as it was during training since any inconsistencies in the model's architecture will result in failed inference.
Once the parameters are loaded, it is important to set the model to evaluation mode by calling `model.eval()`. This step disables certain layers, such as dropout and batch normalization, which are active during training but not needed during inference. Setting the model to evaluation mode ensures that it behaves correctly

during inference, avoiding any unintended behaviors from training-specific layers. To facilitate and optimize the inference process, the MNIST dataset is divided into subsets, allowing the developer to easily select the desired one. The following code demonstrates how to load the test set subset, load a saved `state_dict`, and prepare the model for inference:

Listing 2.4: Loading the state dict and preparing the model for inference

```
# Select the test-set from the MNIST dataset
mnist_test = datasets.MNIST(data_path, train=False, download=True
, transform=transform)
# Load the saved model parameters
net = Net() # Initialize the model architecture
net.load_state_dict(torch.load("trained_model.pt"))
net.eval()
```

2.6 Quantization for Hardware Deployment

Quantization in general refers to the process of mapping a large set of continuous values or high-precision data to a smaller set of discrete values. The main goal of quantization is to simplify data representation, which can be useful for reducing storage requirements, simplifying computations, and increasing efficiency, particularly in digital systems.

Quantization is used in various fields, such as:

- **Signal Processing**: quantization is applied to analog signals to convert them into digital signals. A generic waveform that contains a range of continuous voltage level (analog values) is converted into a series of discrete voltage levels (digital values) so it can be stored or processed digitally.
- **Image Compression**: in image formats such as JPEG, quantization is used to reduce the precision of pixel values to save storage space while retaining most of the visual information.
- **Neural Networks**: in machine learning, especially for hardware implementation, quantization reduces the precision of weights, biases, and activations from floating-point precision (like 32-bit) to lower precision (like 8-bit integers).

In summary, quantization is the process of reducing precision by converting continuous data into a discrete form; this extremely important when dealing with an Artificial Neural Network because quantization, by reducing the precision of the model's parameters, makes the model more efficient for hardware implementation. However, quantization introduces a trade-off: while it reduces resource usage, it can also negatively affect the model's accuracy if not applied carefully. The challenge

lies in finding the right balance between optimizing the network and maintaining acceptable performance.

2.6.1 Quantization Techniques

There are two main approaches to quantization: Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ). Both methods aim to optimize the model for hardware accelerators, but they differ in their complexity and impact on accuracy.

1. **Quantization-Aware Training** : QAT simulates quantization during the training phase of the neural network. While the model is trained in floating-point precision, it simulates how the parameters will behave when quantized, allowing the model to adapt to lower precision.
   By incorporating quantization during training, the model is more likely to maintain accuracy when deployed in a quantized state. This approach tends to result in higher accuracy but requires more effort during the training process.

2. **Post-Training Quantization** : PTQ is a simpler and faster approach since it is applied after the model has already been trained in floating-point precision in fact the parameters are converted to lower precision after training, typically requiring a calibration step to minimize the accuracy loss. PTQ speeds up the deployment process but can lead to a more significant drop in accuracy compared to QAT.

In summary, PTQ is easier and faster to implement, especially when quick deployment is needed, but it may result in a noticeable drop in accuracy. QAT, on the other hand, requires more effort during training but usually preserves more of the model's accuracy in the quantized form.
In this case, PTQ has been chosen. The reason for this decision is to customize the quantization process as much as possible to adapt it to the specific application. While this method may lead to some performance loss compared to QAT, it allows for greater flexibility and faster deployment. For this reason, as will be explained in Chapter 5, simulating the model and analyzing its behavior will be fundamental to minimize losses and maintain high accuracy during deployment on the hardware accelerator.

Chapter 3

The Spiking Neural Network

The network is made of an input layer, a hidden layer, and an output layer:

- **Input Layer:** the input layer accepts the 28x28 pixel images, flattened into a single dimension of 784 neurons.
- **Hidden Layer:** the hidden layer contains several neurons, this Layer is the more flexible since, according to the requirements of the designer, can be resized.
- **Output Layer:** the output layer has 10 neurons, corresponding to the 10 digit classes (0-9).

This network has been designed specifically to suit the needs of the MNIST dataset, with additional considerations for minimizing the area by keeping the network size as small as possible without compromising performance.
The network consists of two fully connected layers; this choice impacts the overhead in terms of data traveling from one layer to another, but, knowing that the target architecture will have limitations, the designer can approximate low-weight connections to zero to save power.

The key element of this architecture is the neuromorphic neuron [13]. The chosen neuron is an event-driven version of the standard clock-driven leaky integrate-and-fire (*LIF*) neuron available in the **snnTorch** library.

3.1 Hardware implementation

After completing the training and all the steps necessary to create a complete software description of the network, the next step is to move towards the hardware implementation. The first major issue when transitioning from software to hardware is managing the limited resources and correctly scheduling the operations.

The most straightforward way of connecting neurons within a network is to directly route each neuron in the previous layer to neurons in the next layer. This method simplifies the network structure but requires an arbitration mechanism to handle congestion, and it is neither *power-efficient* nor *area-efficient*.
As a starting point for finding a better solution was to draw inspiration from protocols used to manage communications in complex systems, such as Artificial Neural Networks. The studied protocols included those used in the Internet and Network-on-Chip systems. Additionally, interrupt management was analyzed to better understand how to efficiently manage congestion; finally, taking inspiration from all these sources, a custom implementation was created.

3.2 Internet communication characteristics

The term *Internet* refers to a vast collection of standards that govern how communication operates within this global network. Two of the most widely used methods for Internet communication are **Ethernet** and **Wireless** communication, both of which are regulated by the IEEE 802 standards.
Ethernet, a key member of this family, is a widely-used technology for local area networks (LANs) and it regulates how data packets between devices on the same network are transmitted.

3.2.1 How Ethernet works

Ethernet [14] transmits data in units called frames whose name and dimension is summarized in Table 3.1. An Ethernet frame consists of several fields:

1. **Preamble** (PRE): synchronizes communication by waking up the receiver.
2. **Starting Frame Delimiter** (SFD): marks the beginning of the frame.
3. **Destination Address** (DA) **and Source Address** (SA): a six-byte fields indicating the frame's destination and source, respectively.
4. **Length/Type** (L/T): indicates the length of the data or the type of frame, with specific values used for different frame types.
5. **Payload**: contains the actual data to be transmitted, which can be up to 1500 bytes. Padding is added if the data is less than 46 bytes.
6. **PAD**: ensures the minimum frame length of 64 bytes.
7. **Frame Check Sequence** (FCS): a CRC value for error detection, ensuring frame integrity during transmission.

Frame field	Description	Byte Count
PRE	Preamble	7
SFD	Starting Frame Delimiter	1
DA	Destination Address	2 - 6
SA	Source Address	2 - 6
L/T	Length/Type	2
Dati	Data	0 - 1500
PAD	Padding	0 - 46
FCS	Frame Check Sequence	4

Table 3.1: Ethernet Frame Fields and Byte Count

3.2.2 Is Ethernet model suitable for SNN?

While **Ethernet** is an effective model for communication in computer networks, it is not well-suited for implementing spiking neural networks (SNNs) in hardware. Here are several reasons why Ethernet's model does not satisfy the requirements of SNNs:

- **Broadcast Communication:** Ethernet is primarily designed for packet-switched networks where data can be sent to a specific address or broadcast to all devices on the network; on the other hand SNNs require precise, point-to-point communication between neurons. Broadcasting data to all neurons, or most of them, would be highly inefficient leading to significant performance degradation.
- **Network Size and Addressing:** the MAC addresses used by Ethernet for identifying devices becomes cumbersome in large-scale networks. Since SNNs typically consist of a very large number of neurons the addressing scheme in Ethernet would not scale well.
- **Scalability:** SNNs need to scale efficiently to simulate large neural networks while Ethernet, for its characteristics, encounters performance degradation when the numbers of devices increases.
- **Communication Overhead:** Ethernet introduces overhead through its various frame fields which are necessary for reliable data transmission in computer networks but in an artificial neural network, where minimal latency and overhead are required, the additional data fields in Ethernet frames would introduce unnecessary latency and processing overhead.
- **Data Granularity:** SNNs need to transmit very small amounts of data (spike events) so the large frame size in Ethernet would be strongly inefficient for

the frequent, small data transmissions characteristic of SNNs.

3.3 Network on Chip (NoC) characteristics

A Network on Chip (*NoC*) [15] is a communication framework designed to connect multiple processing elements (*PEs*) within a single integrated circuit. It provides an efficient way to interconnect various components such as CPUs, GPUs, memory modules, and specialized accelerators (Figure 3.1).

3.3.1 How Network on Chip works

The Network on Chip communication model has the following characteristics:

1. **Topology:** NoC employs various topologies like mesh, torus, ring, and star to interconnect PEs.

2. **Routing:** data packets are routed from source to destination using routing algorithms; the scope of these algorithms is to determine the path taken by data packets.

3. **Communication Protocol:** NoC uses packet-based communication where data is encapsulated in packets and each of them typically contains a header (with source and destination addresses), payload (data), and sometimes a tail.

4. **Switches and Buffers:** at each node in the network, switches or routers manage the incoming and outgoing data packets while the job of buffers is to temporarily store packets during routing to prevent data loss and manage congestion.

5. **Flow control:** these are all the mechanism that ensure an efficient data transfer avoiding congestion.

Finally one of the biggest advantages of NoC is its flexibility, allowing designers to fully customize the network to meet specific needs.

3.3.2 Is NoC model suitable for SNN?

Having defined the main features of NoC is now necessary to analyze if this architecture is suitable for creating a SNN to satisfy our needs. Here are the main reasons why NoC model doesn't satisfy the requirements for our SNNs:

- **Connection Scheme:** SNNs require both a simple and direct communication pathways between neurons to minimize as much as possible power usage and

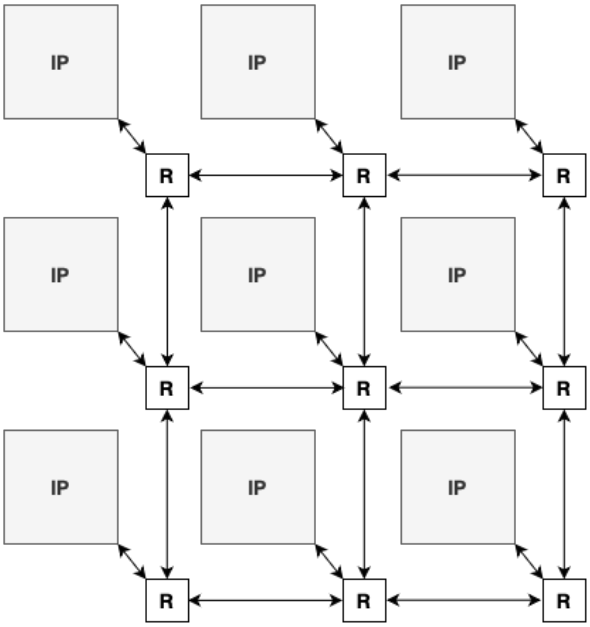

Figure 3.1: NoC scheme with Mesh structure

latency. The complex routing and switching mechanisms in NoC are overkill and inefficient for SNNs, leading to unnecessary power consumption.

- **Broadcast Communication:** NoC are optimized mainly for point-to-point and multicasting communication while SNNs require mostly broadcasting. Implementing such communication may turns out to be inefficient and complex, requiring additional hardware and power.
- **Power consumption:** SNNs require low-power communication mechanisms to simulate the energy efficiency of biological brains and this is in contrast with he complex switching, buffering, and routing mechanisms of NoC which is acceptable for general-purpose processing but not ideal for energy-efficient neural network implementations.
- **Routing Overhead:** SNNs need minimal overhead to ensure real-time performance. The routing overhead in NoC would introduce unnecessary delays and complexity, without satisfying the fast communication required in neural networks.

While Network on Chip (NoC) provides an efficient and scalable communication framework for integrated circuits, it is not well-suited for implementing spiking

neural networks. The described issues in using NoC for implementing the architecture of a SNN are significant drawbacks. However, in some applications where very large Artificial Neural Networks (*ANNs*) are implemented, NoC can be used for regulating the communication between building blocks, each made of thousands of neurons.

3.4 Interrupt management characteristics

Managing **interrupt** congestion is crucial when multiple interrupts require access to shared resources. Effective interrupt handling ensures that high-priority tasks are executed promptly without causing significant delays in the system and ensuring to execute all necessary operations avoiding collision between requests.
There exists many different ways how interrupt congestion is typically managed and every system can implement the strategy or an hybrid one that best satisfy its requirements; however interrupt management techniques can be divided into five categories:

1. **Priority-Based Handling:** each interrupt has an assigned priority, both in a static or dynamic way, based on the criticality; higher priority interrupts are serviced first, ensuring critical tasks are handled promptly.

2. **Interrupt Queuing:** in this case interrupts are queued in a buffer and the adopted strategy consists in processing interrupts in the order they arrive. This approach can be combined with priority assignment to increase the efficiency or, in simpler application, interrupts are just executed in order.

3. **Interrupt Masking:** in this approach interrupts have priorities and lower priority interrupts can be temporarily masked or disabled when a higher priority interrupt is being serviced.

4. **Distributed Interrupt Handling:** this approach is reserved to system that have multiple execution units (such as multi-core systems) for example in a multi-core systems, interrupts can be distributed across different cores to balance the load and avoid congestion.

5. **Hardware Timers and Interrupt Controllers:** hardware timers and dedicated interrupt controllers manage the timing and distribution of interrupts. This approach is by far the most complicated one and needs some specialized hardware resources.

3.4.1 Similarity with SNN Input Management

In a spiking neural network, input spikes from the previous layer must be processed by the neurons in the current layer; this is analogous to handling multiple interrupts requiring access to shared hardware resources.
The design of the artificial neural network and how data need to be processed inside it make the ANN different from different perspective with respect to a CPU that needs to control **interrupt** and manage their corresponding routine.
In SNNs the use of custom hardware to manage simultaneous input stimuli, as done for interrupt, should be reduced as much as possible to limit both power consumption and area occupation; for this reason solutions as Interrupt Masking or Priority-Based Handling are suitable. However Interrupt Queuing can be an interesting starting point for building a custom method to manage neuron operations without the need of extra hardware even though this require some method to synchronize inputs and to manage the priority.

3.5 Custom architecture characteristics

Given the limitations but also the advantages of existing communication frameworks such as Ethernet and Network on Chip (*NoC*) for implementing spiking neural networks (SNNs) and inheriting some ideas from the way interrupts are manged, a custom architecture has been designed.
Here are summarized some of the main characteristics of the custom architecture.

- **Data structure:** one of the key characteristics of this custom architecture is how data are represented. In the context of Ethernet and NoC, a significant disadvantage was the overhead introduced by data packets. To address this issue, the custom architecture adopts a simplified data representation where each neuron's output is either a 0 or a 1, corresponding to a *non-spike* or a *spike*, respectively.
  In this way, by representing the outputs as an ordered sequence of 0's and 1's, the data can be processed and transmitted more efficiently compared to complex packet-based systems but correct synchronization must be ensured.

- **Priority management:** this is an extremely important aspect to take care of since, in SNNs, the order in which operations are executed is crucial; for this reason, to simplify the internal structure of the network, each input is processed in order and only after the end of all the operations generated by the previous one.
  This assumption is coherent with the way sensors, such as *event-driven* camera, works; they work at a certain frequency and once every specified period of time an output is generated and given as an input to the network. This

characteristic requires the accelerator to be fast enough to generate an output before the output from the sensor reaches the network input.

- **Mesh Structure:** the architecture is organized in a mesh structure with N rows and K columns, this is done for avoiding a 1-dimension network with too many rows or with big building blocks that would turns out into an inefficient structure both for power consumption and area optimization, ensuring that the architecture can scale efficiently without excessive resource usage.

- **Input Spike Distribution:** input spikes are divided into N groups (same as the number of rows) of dimension H and each group is sent to a different row of the mesh. This is done for ensuring that elements in the same row work on the same data. To simplify the architecture, there is no data transfer between columns.

- **Building Blocks:** each building block corresponds to a specific set of J neurons, the choice of the number of neurons effects both latency and area utilization.

- **Data Transfer and Processing:** once each block has processed its group of input spikes (from the previous layer or from the input bus), the results are transferred to the block directly below it. If each block requires G cycles to process an input block, then in a total of $G \times N$ cycles, all blocks (and consequently all neurons) will have processed the incoming input.

- **Parallel Input and Serial Processing:** input blocks arrive in parallel through a bus that delivers data packets to all the blocks that build a row and are transmitted in parallel between blocks in the same column. On the other hand, operations inside the neuron are executed serially, with inputs shifted one at a time and then elaborated by the processing element inside the single block.

- **Scalable Processing and efficient cycle management:** the design allows for scalable processing as each block processes its input independently before transferring the results to the block below. Moreover by requiring $G \times N$ cycles for complete processing, the architecture ensures that all neurons in the network process their inputs in a synchronized manner, maintaining the temporal accuracy that is crucial for SNNs.

In summary the processing flow can be divided into the following phases:

1. Input Distribution: input spikes coming from the previous layer are divided into N groups and sent to the respective rows.

2. Processing: each block, after receiving through the bus the inputs, processes them serially, bit after bit.

3. Cycle Management: in G cycles, all blocks complete their processing.

4. Data Transfer: once the elaboration phase has elapsed, each block transfers its input, strategically saved inside a register during the execution of operations, to the next block in the same column and a feedback connection allows to route data from the last to the first block of the column.

5. End of operation: when all the blocks in a column as successfully processed all the data from the previous layer for each block the *spike* or *non-spike* (1 or 0) is calculated and sent in parallel to the next layer.

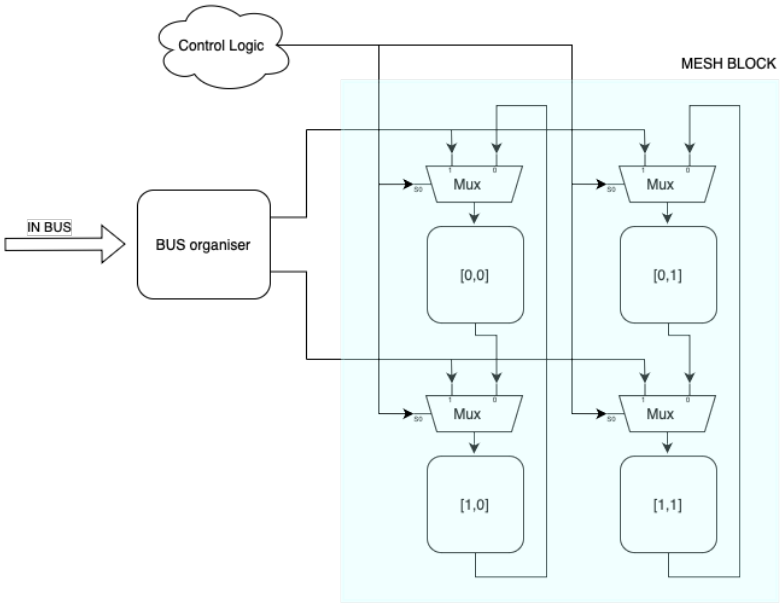

Figure 3.2: 2×2 Mesh scheme

This custom architecture addresses the specific needs of our SNNs, providing a scalable, efficient, and low-power solution compared to a traditional communication models. An example of a 2×2 mesh is shown in Figure 3.2; in the picture a simplified scheme of the connections is shown and the main components of the mesh are schematically represented.

3.5.1 Custom architecture optimizations

Starting from this basic description some optimization can be applied to further customize and to adapt the network to the specific needs of SNNs.
Here some possible personalization are described:

- **Input Buffer Optimization:** inside each building block there is an input buffer that is used to collect the input data and to serially distribute it to the processing elements.
  The buffer becomes critical when its dimension increases both for power and area issue so it is important to choose a buffer size that allows for efficient data processing without excessive memory usage. This involves finding a trade-off where a smaller buffer reduces power and area requirements but increases the frequency of memory access, which can impact performance.

- **Number of Neurons per Block:** this parameter is related to the buffer dimension and, as before, some considerations are needed both in terms of area and power.
  Assuming that the number of neurons per block is J and that each input packet is made of H bits we have two possible scenarios:

  1. Maximizing Performance: with J=H, the performance is maximized because each neuron processes one input, but this consumes more power and area furthermore going above this value is completely useless.
  2. Resource Optimization: reducing J by a factor (i.e. 2 or 3) can save space and power leading to better resource reuse and optimized area. This reduction increases latency but is acceptable when processing events from sensors or cameras operating at low frequencies (a few Hz), where ultra-fast response times are not critical.

- **Memory Access Optimization:** optimizing memory is crucial to reduce the bottleneck introduced by memory elements.
  A possible solution to this problem comes from reducing memory access contention for example aligning data access patterns with memory architecture to reduce access time and improve throughput. Another possibility is to utilize local memory within each block to store frequently accessed data. By combining this approach the latency due to slow memory can be strongly minimized.

- **Idle System Optimization:** design the system to stay *idle* when an input bit is 0, meaning resources are only consumed when the input bit is 1. This approach, even if it doesn't effect latency, leverages the typically sparse activity in spiking neural networks to reduce power consumption significantly without introducing undesired desynchronization due to dropping 0 inputs.

3.6 Address Event representation (AER) standard

Address Event Representation (*AER*) is a communication protocol widely used in Spiking Neural Networks (*SNNs*) for transmitting information between neurons, neuronal populations, or between a sensor and the corresponding network. In AER, events are conveyed as series of address-event pairs. The key characteristics of AER are outlined below:

- **Event-Based Communication:** AER communicates information via discrete events, where each event represents a neuronal spike. This form of communication can either transmit the entire state of the network at fixed intervals (synchronous communication) or only when spikes occur (asynchronous communication). The latter results in a highly efficient, event-driven communication system, reducing unnecessary data transmission.

- **Address and Time:** each AER event consists of two parts: an address and a timestamp. The address corresponds to the identity of the neuron or network node that generates the spike, and the timestamp indicates the exact time the event occurred.
  The addresses can be either unique for every neuron in the network or reused within each layer, depending on the network structure. Proper event mapping mechanisms are required to ensure that spikes are transmitted accurately from their source to the appropriate destination.

- **Temporal Precision:** the AER protocol provides high temporal precision by timestamping events with fine-grained resolution ensuring precise synchronization and accurate representation of spike timing, making it especially useful in real-time neural processing.

- **Scalability and Efficiency:** the event-based nature of AER enables scalability, especially in large-scale neural systems. By only transmitting events when spikes occur, AER minimizes data bandwidth and energy consumption, making it an efficient mechanism for high-density neural processing systems.

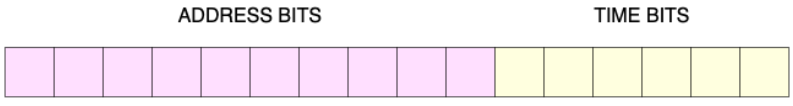

Figure 3.3: AER string format

From the perspective of the network architecture, each AER event is represented as a string of bits, as depicted in Figure 3.3. The first part of the string, starting from the Most Significant Bit (*MSB*), encodes the address, while the second part, ending at the Least Significant Bit (*LSB*), represents the timestamp or temporal information. Inside the architecture, spikes are often represented as bit strings, where each bit corresponds to the presence (*1*) or absence (*0*) of a spike. This representation simplifies the internal transmission of information and is enabled by translators at both the input and output of the accelerator.
The Address Event Representation (*AER*) is critical for interfacing with external systems, allowing for minimal data transfer overhead and enabling efficient communication within the SNN. By maintaining compatibility with the AER standard, this communication protocol ensures the SNN can be easily integrated with other external systems and devices, supporting interoperability across diverse platforms.

Here is an Example of the AER Conversion Process. Consider a random string of binary values representing a simple sequence of neuronal spikes: `1010110010` at a certain time instant t_1.
Each bit in this string represents whether a neuron has spiked (1) or not (0). The conversion of this binary spike representation into AER format involves assigning an address (in a neural network the position of a spike in a sequence is the address of the neuron who generate that spike) to each spike and capturing the time at which the spike occurs.
Observing the first two ones of the string we can say that there is a spike at position 1 and another spike at position 4. The corresponding AER representation for these events would look like:

- `Address: 00001, Time:` t_1
- `Address: 00100, Time:` t_1

After processing all events, the final AER string for the two spikes could look something like: `00001_`t_1`, 00100_`t_1`, 00101_`t_1`, 00111_`t_1`, 01001_`t_1, then, substituting a generic value for $t_1 = 0101$ the resulting string would be `000010101, 001000101, 001010101, 001110101, 010010101`.
This string provides the complete address-event information, where each address is linked to a specific neuron and timestamp, facilitating communication with external devices using the AER protocol.

Chapter 4

Network software testing

To ensure the functionality and correctness of the design, before diving into the hardware development of the model, a preliminary verification was performed using a Python version of the spiking neural network that behaves exactly as the hardware accelerator.
This network consists of 784 inputs, a first layer (*hidden layer*) with 40 neurons organized in a 2×2 mesh structure (10 neurons per mesh element) and with 10 outputs. Following this, a second layer with 10 neurons arranged in a 2×1 mesh structure (5 neurons per mesh element) was used leading to 10 final outputs. An high-level representation of the network is shown in Figure 4.1.
The primary objective of this setup was to test the functionality of the network and those of the developed environment and to finally debug any potential issues in the design.

The simulation steps are as follows:

- **Training the Network**: the network was initially trained using PyTorch and snnTorch with the goal of extracting the network's parameters. The training is performed using the custom event-driven neuron and the parameters from the trained network are saved in a text file extracting the useful information from the `state_dict` generated with PyTorch.

- **Network Preparation**: two automated scripts were employed to prepare the network for simulation. The first script creates the initialization files for the required memories then another script is in charge of generating a file for the simulation extracting from the MNIST dataset a sample and converting it into the wanted AER format.
  For this preliminary test the first script is not strictly needed since the Python simulation could extract the parameters directly from the dictionary but, in this way, the script is tested before using it for the real hardware implementation.

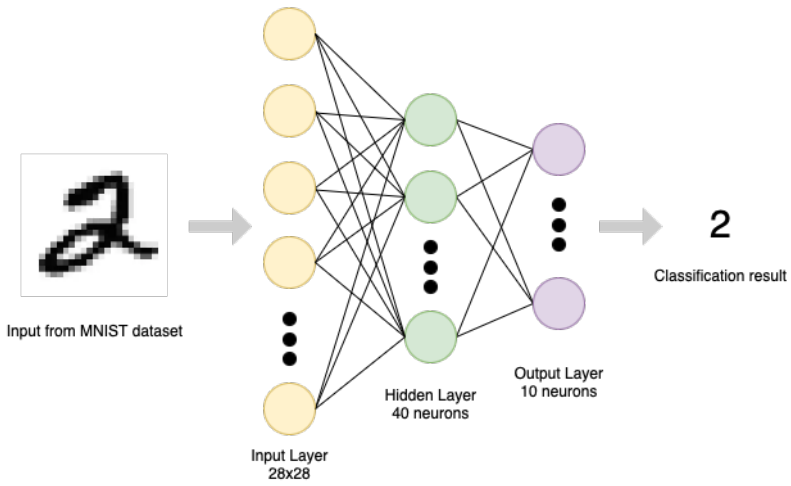

Figure 4.1: Network scheme

- **Network simulation**: the simulation was executed using Python. The intermediate results were analyzed to verify the correctness of the network's functionality and to identify any potential design errors.

4.1 Training the Network

The network's training is conducted using PyTorch, as described in Section 2.5, and the training output file is the `state_dict`.
After extracting the state dictionary, a Python script was used to convert it into a simple text file; the code iterates through all the entries of the `state_dict` and prints only the necessary parameters (weights, biases, etc.).
This conversion step is crucial for making the model's learned parameters more accessible for further processing, serving as the foundation for subsequent processing steps, where automatic scripts will handle parameter conversions for deploying the network in hardware.

4.2 Network Preparation - Automatic Scripts

The automatic scripts employed in this phase are two and they were both developed using Python for its ease of use and for its characteristics of being merely deployed on different platforms.

4.2.1 Parameters extraction and memory initialization

The first script is used to extract from the state dictionary, obtained through the training phase, the necessary parameters as though weights and biases.
With this automated script, the elements are first extracted from the file generated in the previous stage through the *state_dict* conversion and stored inside temporary variables, then, the script is in charge of converting from floating-point representation into the necessary integer format, finally, they go through a quite articulated process where the data are re-organized to meets the requirements of the hardware accelerator.
The last step is the generation of a *".coe"* (*Coefficient*) file filling it with the parameters extracted and then elaborated as described above.
The *".coe"* files, produced by the first script, are used for memory initialization; Vivado offers the possibility of generating a memory (*ROM/RAM*) using its IP block set of instruments and the coefficient file is used to initialize the memory.
This last step, as described in the introduction of the chapter, is not strictly necessary during the simulation phase but it is fundamental for ensuring coherence with how the accelerator will works when it will access the parameters. For this reason, also in the software simulation, the files containing the parameters will be accessed by the program.

4.2.2 Sample extraction and AER conversion

The second script is used for extracting a sample from the MNIST dataset and converting it into Address Event Representation using *rate coding*.
One of the features of Spiking Neural Networks (*SNNs*) is that they are made to exploit time-varying data and, since the MNIST is not a time-varying dataset, there are two options for using MNIST with an SNN:

1. Repeatedly pass the same training sample $X \in \mathbb{R}^{m \cdot n}$ to the network at each time step. This is like converting MNIST into a static, unchanging video. Each element of X can take a high precision value normalized between 0 and 1: $X_{ij} \in [0,1]$.

2. Convert the input into a spike train of sequence length `num_steps`, where each feature/pixel takes on a discrete value $X_{ij} \in 0,1$. In this case, MNIST is converted into a time-varying sequence of spikes that features a relations to the original image.

The first method is straightforward but does not fully leverage the temporal dynamics inherent to spiking neural networks (*SNNs*) therefore data-to-spike conversion (*encoding*) methods based on the second method are preferable.
The *snntorch.spikegen* module of SNNTorch offers several functions to simplify the

process of converting data into spikes. There are three available options for spike encoding:

- **Rate Coding**: implemented as *spikegen.rate*, this method uses the input features to determine the frequency of spikes.
- **Latency Coding**: available through *spikegen.latency*, this method utilizes input features to determine the timing of spikes.
- **Delta Modulation**: accessed via *spikegen.delta*, this approach generates spikes based on the temporal changes in input features.

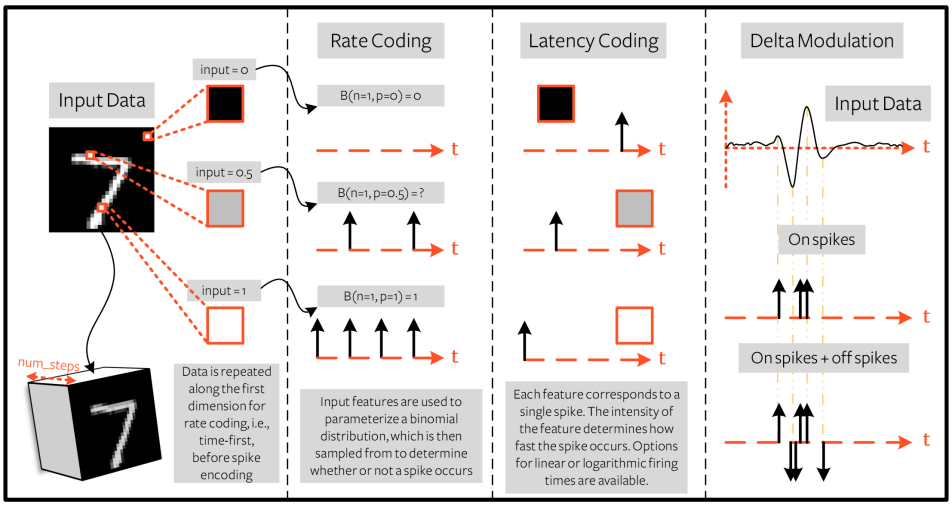

Figure 4.2: MNIST conversion process for SNNs deployment

When converting input data into a rate code each normalized input feature X_{ij} is used as the probability an event (*spike*) occurs at any given time step, returning a rate-coded value Rij. This can be treated as a Bernoulli trial: $R_{ij} \sim B_{ij}$, where the number of trials is $n = 1$, and the probability of success (spiking) is $p = X_{ij}$. Explicitly, the probability a spike occurs is: $P(R_{ij} = 1) = X_{ij} = 1 - P(R_{ij} = 0)$. For an image from the MNIST dataset, the described probability of spiking corresponds to the pixel value; a white pixel corresponds to a 100% probability of spiking, and a black pixel will never generate a spike (0% probability). Figure 4.2 shows how the three different approaches described above behave for converting an image into the corresponding SNN's friendly representation.

In a similar way, *spikegen.rate* can be used to generate a rate-coded sample of data. As each sample of MNIST is a static element (*image*) we can use `num_steps` to repeat it across time; in all the examples this parameter has been fixed at a value of 100. Figure 4.3 shows the output of the conversion using a raster plot.

Raster plots are a highly informative visualization technique for examining neural activity over time, both for individual neurons and across all channels simultaneously but they can also be used to visualize spiking activity in relation to particular stimuli.
In this context, raster plots are used to display the spiking activity at the input of the neural network. As shown, the horizontal axis represents 100-time steps (`num_steps`), while the vertical axis consists of 784 spots (28×28), corresponding to the pixels of an image. This representation not only illustrates the result of the data-to-spike conversion process but also effectively highlights the inherent **sparsity** of spiking neural networks.

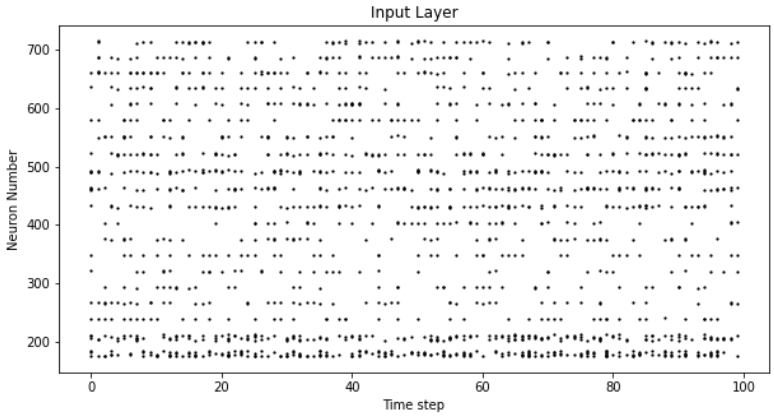

Figure 4.3: MNIST sample raster plot

Thanks to the functions of the snnTorch's framework the process of converting an image into the corresponding SNN format is straightforward. After that, to ensure compatibility with the input interface of the network, the information encoded into spikes must be converted into the Address Event Representation (*AER*) format.
The output from the conversion of the images into spikes is a set of `num_steps` strings of length 28×28 (784 pixels); converting such information into the AER format requires to describe each 1 (*spike*) bit of every string with the corresponding encoding. In this final conversion, the bit length is first defined, a certain portion of each string will be dedicated to the address (*address_portion*) part and the next portion to the time (*time_portion*). Formally, for obtaining such quantities, the following formulas are used:

$$time\_portion = \log_2(\text{num\_step}) = 6.64 \approx 7$$

$$address\_portion = \log_2(\text{num\_inputs}) = 9.61 \approx 10$$

where *num_steps* is equal to 100 and *num_inputs* is 784 (28x28).

Having defined the bit length the script goes through every time step, extract only the elements that represent a spike (bit = *1*) and convert its position into the binary address on the defined number of bits and, for each spike, write a line in a *.txt* file.

4.3 The quantization process

The script in charge of extracting the parameters and initializing the memories implement also diverse steps for normalization and quantisation of the parameters. The first step, before moving to the actual quantization, consists in adapting the input generation process.
In snnTorch the function `spikegen.rate` can be used to generate a rate-coded sample of data. Since the MNIST dataset consists of static images, we need to repeat the image data across multiple time steps; this is achieved by specifying the number of time steps using the `num_steps` parameter, ensuring that the static image is represented over a temporal sequence. Another important argument of the *spikegen.rate* function is the `gain`, which controls the spiking frequency by scaling the intensity of each pixel's grayscale value.
The MNIST dataset contains grayscale images, where a significant portion of the image consists of a white background. This characteristic is particularly relevant because, using the rate-coding mechanism, a white pixel corresponds to a 100% spiking rate at every time step. In contrast, only a small portion of the image represents the relevant information (i.e., the digits). By reducing the `gain` parameter, the spiking rate for less relevant parts of the image can be decreased, effectively reducing unnecessary computations. Figure 4.4 shows how different `gain` values change the spiking behavior of a sample from the MNIST dataset.

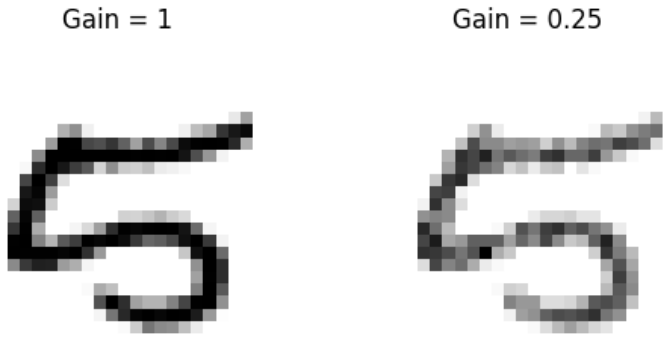

Figure 4.4: MNIST rate coding with different gain values

Choosing the appropriate *gain* is not trivial. While reducing the gain can

optimize resource usage (as fewer spikes reduce computational load), it can also negatively impact accuracy. If the gain is reduced too much, essential information might not be adequately represented, leading to significant degradation in model performance.

4.3.1 Input optimization and parameters normalization

The next step involves normalizing the network's parameters for deployment on hardware accelerators. In this process, a **conversion factor** is used to transform the floating-point parameters, trained in PyTorch, into integer values. This conversion is essential for hardware platforms, which operate more efficiently with fixed-point arithmetic or integer-based computations.
The conversion process involves multiplying each floating-point parameter, extracted from the `state_dict` of the trained network, by the chosen conversion factor and then truncating the result to obtain an integer value. The conversion factor must be consistent across all network parameters to ensure coherence between the floating-point training and the hardware implementation.

The **normalization** of the parameters effectively behaves like a binary shift operation. In fact, the factor is chosen as a power of 2, meaning a conversion factor of 10, for example, corresponds to $2^{10} = 1024$. This is important because in binary arithmetic, multiplying by a power of 2 is equivalent to performing a left shift simplifying the multiplication process in hardware.
For example, if we have a floating-point number 0.035474, and we apply a conversion factor of 10 (which is equivalent to multiplying by $2^{10} = 1024$), the conversion process is as follows:

$$\text{floating-point value} : 0.035474$$

$$\text{multiply by factor} : 0.035474 \times 2^{10} = 36.334$$

$$\text{truncate to integer} : 36$$

In this case, after applying the factor and truncating, we get the integer value 36. However, it's important to note that when using a factor of 2^{10}, any information smaller than 10^{-3} is effectively considered negligible. This is because 2^{10} is approximately 10^3, meaning that when rounding, we are discarding all information smaller than 10^{-3}. This consideration is valid for every chosen factor but, as shown in the example, if the factor is close to a power of 10 it is easier to visualize this loss of precision but also how much information we are actually keeping with the conversion.

In practice, the choice of the factor depends on the dataset and the range of values present in the model's parameters. For a relatively simple dataset like MNIST, it is easier to find a factor that is not too large, as the data don't require high precision to maintain accuracy. However, for more complex datasets or networks, careful tuning of this factor becomes more critical to prevent significant information loss during normalization.

This conversion process has two main advantages:

- **Reduced Computational Complexity**: Integer operations are faster and less resource-intensive compared to floating-point operations. Integer arithmetic can be implemented more efficiently on hardware, consuming fewer resources in terms of power and silicon area.

- **Improved Determinism**: Integer arithmetic typically offers more deterministic behavior, as it avoids the precision issues and rounding errors associated with floating-point arithmetic.

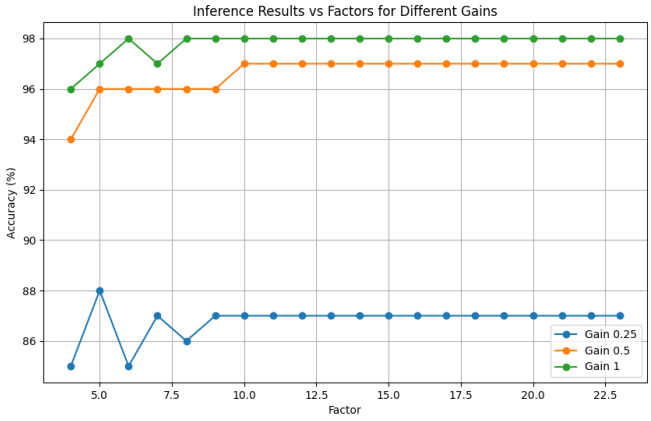

Figure 4.5: Inference results using different gain and conversion factor

On the other hand, as discussed above regarding the selection of the `gain` coefficient, finding a trade-off between reducing the memory footprint and increasing computational efficiency without drastically compromising performance is not trivial. To address this challenge an analysis was conducted to examine the combined effects of adjusting the `gain` during input generation and modifying the conversion factor. The results are shown in Figure 4.5, where the graph illustrates the network's accuracy as a function of different gains and conversion factors.

On the x-axis, the conversion factor is shown, while on the y-axis, the accuracy is displayed in percentage. What can be observed is that in all three cases (`gain`

= 1/0.5/0.25), the network saturates with a conversion factor of 10 or greater; therefore, it would be unnecessary to select a larger conversion factor, as it doesn't yield any further improvements in accuracy. This is an important information since it shows that everything that is bigger than $2^{factor} = 2^{10}$ is negligible.
The plot shows that the network achieves its best result when the `gain` value is set to 1 (corresponding to 100% spike generation). However, it is important to note that with a 50% reduction in input spikes (`gain` = 0.5), the accuracy is practically preserved, showing only a minimal reduction of about 1%.
Nevertheless, when the `gain` is reduced further (example in blue with a gain = 25%), the performance degrades significantly, and the reduction in accuracy becomes non-negligible.
Based on the results, the optimal configuration to maintain high accuracy while limiting computational load is to select a `gain` of 0.5 (50% reduction) coupled with a conversion factor of 10 for the float-to-integer conversion.

4.3.2 Parameters quantization

The next step is to select the number of bits used to actually represent the parameters in hardware; this decision is crucial, as it impacts both power consumption and silicon area. While choosing a higher number of bits keeps the precision high, it also increases the size of the hardware components used by the accelerator, resulting in greater power consumption and silicon area utilization.
It is important to mention that in this simulation phase, the concept of bits is just an artificial mechanism. In fact, since the simulations are performed in Python, all the operations are executed using integer number; however the truncation process is extremely useful to simulate how the real system will behaves when deployed in hardware using binary representation.
An important consideration when choosing the bit-width is how it handles parameter saturation. For instance, suppose we have chosen 4 as the bit-width of the parameters; if a parameter's value can fit within a 4-bit integer it can be represented without issue, however, if the value exceeds the range that can be represented by 4 bits (e.g., [-8;7] in signed 4-bit integers), saturation will occur. Saturation means that the value will be clipped to the maximum or minimum representable number, depending on its sign. For example:

$$\text{Given a 4-bit signed integer range: } [-8, 7]$$

$$\text{For } x = 5, \quad -8 < x < 7, \quad \text{so} \quad x_{\text{converted}} = 5$$

$$\text{For } x = 20, \quad x > 7, \quad \text{so} \quad x_{\text{converted}} = 7$$

Saturation can introduce errors into the network, especially when critical parameters are affected. Therefore, when selecting the bit-width, it is essential to strike a

balance between precision and hardware constraints to avoid excessive saturation that might degrade the model's performance.
To determine the optimal bit-width for our specific application, a simulation was conducted. The `gain` used in the generation of the input was fixed at 0.5 (50%), and the *conversion factor* was chosen as 10. Thus, the only parameter that varied during the simulation was the number of bits used to represent the quantized parameters.
In the plot shown in Figure 4.6, the x-axis represents the bit-width, while the y-axis displays accuracy as a percentage. The blue points correspond to results from multiple simulations using different set of data to ensure the reliability of the outcome. The green line connects the average accuracy for each bit-width, showing the overall trend, while the red dot highlights the best solution in terms of both accuracy and bit minimization.
From the simulation, we can observe that when the bit-width is small (between 4 and 6 bits), the accuracy is significantly compromised and the performances have high unpredictable fluctuations; this happens because lower bit-widths do not capture enough detail in the parameter representation. However, as the bit-width increases beyond 7 or 8 bits, the accuracy plateaus, meaning that further increases in bit-width no longer provide substantial accuracy improvements but merely consume additional resources; this is because most of the relevant information is already captured by 8/9 bits.
In this scenario, 9 bits represent a *Pareto point* namely a point where we achieve the best balance between minimizing resource usage (bit-width) and maximizing performance (accuracy). In the context of optimization, a Pareto point is a solution where no other option can improve one objective (e.g., accuracy) without worsening another (e.g., resource usage).

4.3.3 Accumulator quantization

The final step in optimizing the system involves saturating the membrane potential of the LIF neuron model. In this model, when inputs arrive, the system first accumulates the weights of the incoming inputs then, when all inputs have been processed, it applies the exponential decay and performs the remaining mathematical operations. The accumulator, which handles this process, stores signed integer values but predicting the fluctuations of this accumulator is challenging because they depend on both the number of inputs and the specific parameter values involved.
To optimize the bit-width of the accumulator, a quantitative analysis was conducted, similar to the approach used for determining the parameter bit-width. The network was tested with various input sets, and the bit-width of the accumulator

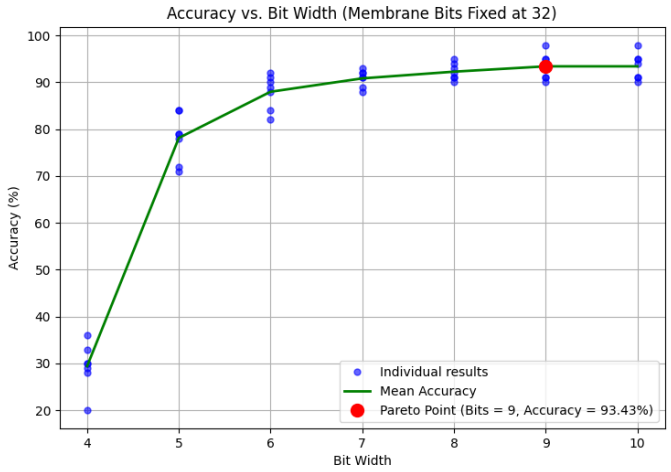

Figure 4.6: Inference with different different bits for parameters' quantization

was adjusted with each simulation.
The results, shown in Figure 4.7, illustrate the effect of varying the bit-width of the membrane potential, which ranged from 7 bits (two bits smaller than parameters) to 15 bits. As with previous optimizations, the objective is to find a *Pareto point* to get a balance between accuracy and resource usage.
The analysis shows that the behavior is very similar to that observed during the quantization of the parameters and it indicates that using 11 bits strikes a good compromise, maintaining high accuracy while reducing resource consumption. While 11 bits is a safe choice, further experiments on the actual hardware accelerator might require to increase the bit-width to 12 or to reduce it to 10.

4.3.4 Final considerations

The normalization process can be summarized in three main phases:

- **Choosing the Gain**: this phase reduces the neuron activity. By selecting an appropriate gain value, we ensure that neural responses remain within a manageable range, minimizing excessive spikes and computational overhead.
- **Selecting the Normalization Factor**: this phase involves scaling the network's inputs using facilitating efficient processing on hardware.
- **Quantizing the Parameters and Accumulator**: finally, the network parameters are quantized to reduce bit-width while maintaining accuracy. At this stage, we also determine the optimal bit-width for the accumulator to

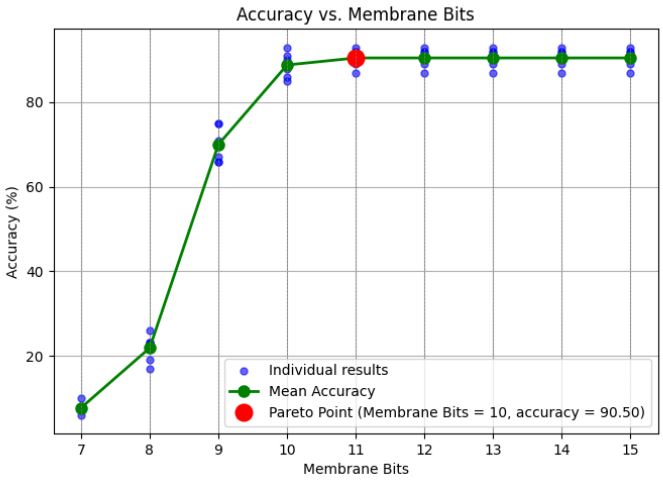

Figure 4.7: Inference with different different bits for accumulator's quantization

handle the summation of inputs efficiently. The goal is to minimize resource usage without compromising model performance.

When performing inference with the trained network, an initial accuracy of approximately 95-96% was observed. The trained model was then passed to the Python program, where it operated at maximum precision without any optimization, yielding an average accuracy of 94-95%, comparable to the behavior observed with PyTorch.
Once optimization began, with a gain set at 50% and a normalization factor of 10, the accuracy increased to approximately 96-97%. The most critical phase was during quantization, where the impact on accuracy was more pronounced. Simulations showed that with 9-bit quantization for both weights and biases, the accuracy dropped to approximately 93%. Finally, after applying 11-bit quantization to the accumulator, the accuracy settled at 90%.
While this behavior might seem unexpected, it is largely due to the characteristics of the MNIST dataset. The simplicity of MNIST allows for model simplifications, which can sometimes lead to performance improvements rather than degradation. The differences between these results and the earlier steps are therefore minor and still fall within a comparable range.

4.4 Network simulation

The last step is the network simulation. The simulation consists of multiple layers of computation where each layer performs specific operations.

The code reads the inputs in the AER format, processes them through weight accumulation, applies a membrane potential decay mechanism, generates spikes and finally computes and evaluates the accuracy of the model based on its predicted results.

1. **Parameters Initialization**: in the code, some useful parameters are defined such as the number of rows (`ROW_num`), columns (`COL_num`), processing elements (`N_PE`) or inputs (`num_inputs`) to define the structure of the layers.
   Moreover, various other constants, like `bit_line_length` or `thr_factor` (threshold factor), are set at the beginning to optimize code portability and customization of the simulation.

2. **Reading AER Data**: as a preliminary step the code reads Address-Event Representation (AER) formatted input data. The function `process_AER_file()` reads and processes the file. The resulting data is stored in a 2D array where each row represents a time step, and the columns correspond to input neurons that are active (set to 1).

3. **Bias Conversion**: the bias values are converted to integer format using the function `convert_bias()`. This function multiplies the bias by a scaling factor (defined as $2^{\texttt{thr\_factor}}$) and saturate them at the maximum allowable range depending on the bit-width dedicated to the parameters.

4. **Processing Weights**: the weights, saved in a dedicated file through the automatic script, are stored inside an array. After this operation, the network has finished the setup and it is ready to start processing the inputs

5. **Simulation Loop**: the main simulation takes place inside the `simulation()` function, which processes input stimuli over multiple time steps. For each time step, the code iterates over all layers, performing the following steps:

   - Weight Accumulation: for each layer, the code initializes an accumulator and spike matrix; then it iterates over all inputs and the weights are accumulated. The resulting values are accumulated in the `accumulator` matrix and saturation limits are applied to prevent overflow.
   - Membrane Potential Update: when the input processing ends the membrane potentials are updated using an event-driven exponential decay approach. The new membrane potential is then computed by adding the accumulated weights and bias to the decayed potential.
   - Spike Generation: spike is generated when the membrane potential exceeds a predefined threshold, which is set as $2^{\texttt{factor}}$ (in line with the normalization of the network's parameters); when a spike occurs, the

membrane potential is reset to zero. The spike vector is stored for future processing by subsequent layers.

6. **Results and Accuracy Calculation**: after processing all inputs through both layers, the final output spikes are analyzed and the neuron with the maximum number of spikes is considered the predicted output. The accuracy is calculated by comparing this prediction with the target label extracted from the input file.
   The simulation allows to process more than one input at a time by repeating this process iteratively, in this case, the accuracy is computed evaluating over the set of inputs the number of correct predictions over the total predictions.

This code simulates an event-driven hardware accelerator that processes inputs across multiple layers using spiking neurons, closely mimicking the behavior of the real hardware implementation.
Certain steps, such as weight extraction and weight accumulation, are intentionally "not optimized" to replicate how the hardware will retrieve parameters from memory and accumulate their values. This approach aims to make the simulation as realistic as possible, accurately reflecting the eventual hardware behavior.
The simulation script has an high degree of customizability to test different configurations of the network. This propriety has been used to analyze the behaviour of the system under different conditions and to test different configurations as a preparation step for the final deployment of the "real" accelerator on hardware.

Chapter 5

Network - hardware structure

The network architecture is designed with several specialized components that manage the flow and processing of data. In total the network is made of the following modules:

1. **Control Unit** (CU): the Control Unit is responsible for managing all operational states of the network using a Mealy state machine. It controls the flow of operations throughout the system, ensuring that each component executes its tasks in the correct sequence and timing.

2. **Processing Element** (PE): the Processing Element, named in this way for the similarities with the Network-on-Chip (NOC) model, serves as the core of the neural network.
   The PE implements the functionalities of a set of N neurons and it performs essential mathematical operations such as membrane potential computations or weight accumulations. The PE is essentially the heart of the network, where all the computations are executed and the actual processing of input data occurs.

3. **Mesh Block**: Mesh Blocks are structural components that facilitate the flow of information to and from the Processing Elements; every mesh block is connected directly to one PE in a way to ensures efficient communication within the network's mesh structure. This block, as for the PE, is inspired from the structure of NoC.

4. **Layer**: a Layer, in this network, is a collection of Mesh Blocks. It defines the overall architecture and control the distribution of weights, inputs and outputs within the blocks.

Every Layer component has numerous parameters that can be set when istantiating it to customize the shape and the internal organization of the Layer. The parameters are:

- `IN_block`: number of bits delivered to each ROW. This defines the input width for each row of the MESH.
- `piso_cnt`: logarithm (base 2) of the `IN_block`. This represents the dimension of the counter needed to track the number of bits in the Parallel-In-Serial-Out (PISO) buffer as they are sent to the neuron.
- `ROW_number`: number of rows in the MESH.
- `COL_number`: number of columns in the MESH.
- `N_PE`: number of Neurons in each Processing Element.
- `ROM_weights_depth`: number of lines in the ROM (Read-Only Memory) storing the weights.
- `ROM_weights_width`: width of each line in the ROM storing the weights.
- `ROM_weights_addr`: logarithm (base 2) of the `ROM_weights_depth`. This is the number of bits in the address for the ROM storing the weights. This is calculated as the
- `ROM_biases_addr`: number of bits in the address for the ROM storing the biases.
- `counter_dim`: dimension of the counter used to track the number of iterations through the mesh. This is calculated as the logarithm (base 2) of the `ROW_number`, representing the counter's size for navigating rows during computation.
- `layer_index`: index of the Layer in a multi-layer architecture. This helps distinguish between different layers when multiple layers are present in the network.

5. **Memory**: each Layer has a single port ROM where the weights are stored. The dimension of the memory depends on the number of inputs and on the number of neurons of the layer. The memories are defined as external component with respect to the Layer to raise the network configurability.

6. **Input Translator**: this component is responsible for converting Address Event Representation (AER) formatted inputs into a string of bits.

7. **Output Translator**: performing the inverse operation of the Input Translator, the Output Translator converts the processed bit strings back into AER format.

8. **Network**: the Network component is the top level entity and it encompasses all other components. It integrates the Control Unit, Layers, Input Translator, Output Translator and finally the memories, ensuring that all parts are interconnected and function cohesively.
   The Network component is in charge of addressing the synchronization issues between different Layers, facilitating smooth data processing across the entire system.

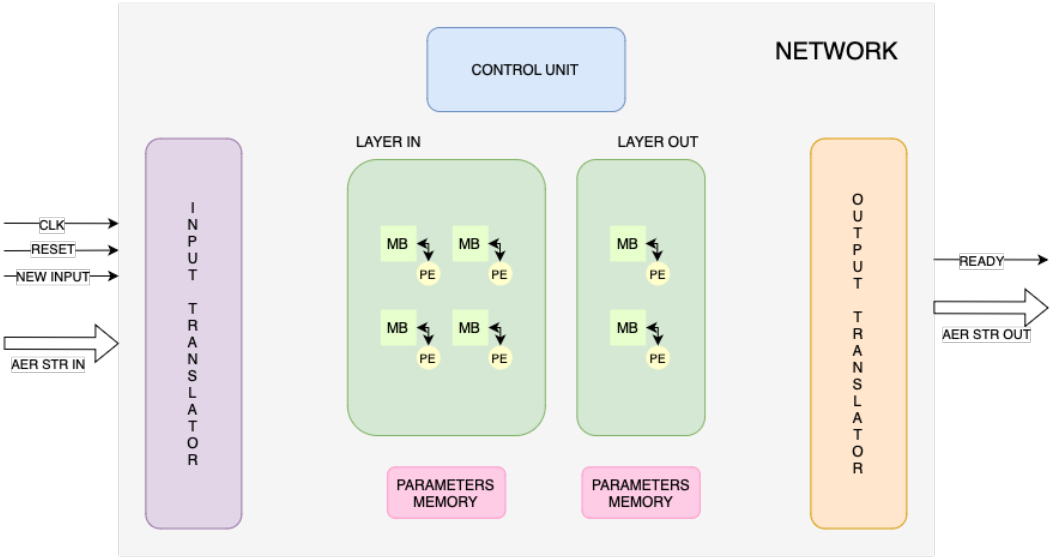

Figure 5.1: Network high level scheme

An external components has been deployed to enhance easy customization and modification of the network's configuration, enabling new versions to be created with minimal changes dedicated.
This package named `constants.sv` is used to store all configurable parameters of the network; an example of such parameters are the number of neurons in each layer, the number of bits dedicated to the parameters (either weights or biases) or exponential decay coefficients (β).

5.0.1 External interface

The external interface of the hardware accelerator is designed to be as simple as possible, minimizing the number of connections to external systems and enhancing compatibility.
The external connections are as follows:

- **Input Signals**:
  1. `clk`: clock signal used for synchronization and timing.
  2. `reset`: reset signal that reset all the registers and components of the architecture.
  3. `new_input`: a boolean flag that can be either *1* or *0*, indicating when a new input is available. This signal must remains high as long as new inputs are present.
  4. `AER_string_in`: the input string in Address Event Representation format, with its length being the sum of the *ADDRESS* bits and *TIME* bits, as explained in Section 3.6.
- **Output Signals**:
  1. `ready`: a boolean flag indicating the readiness of the architecture to receive new input stimuli. A value of *1* means the accelerator is ready, while *0* indicates it is busy processing.
  2. `AER_string_out`: similar to the input string but represents the output of the network after processing the inputs.

An high level representation of the structure of the architecture is shown in Figure 5.1 and the external configuration is also shown. The picture represent a simplified version of the network made of an input layer organised as a 2×2 mesh and of an output layer organized as a 2×1 mesh.

5.1 The Finite State Machine

The network's Finite State Machine (*FSM*) is designed as a Mealy machine. In this type of **FSM** the outputs depend not only on the current state but also on the current inputs. This allows for more responsive behavior compared to a Moore machine, where the outputs depend solely on the current state.

5.1.1 FSM States Description

The **FSM** (Finite State Machine) of the network operates through a series of states to manage the flow of data and control operations; a scheme of all the states of the Finite State Machine is shown in the Figure 5.2. Below is a detailed description of each state:

- **IDLE**: the FSM is in a waiting state, looking for a new input signal. Once a new input is detected, the FSM transitions to the next state to begin processing.

- **EVENT IN**: when a new input arrives, the FSM enters this state. Here, every clock cycle, a new input in Address Event Representation format is received and the Control Unit manages the translation process through the Input Translator component, converting AER format to the corresponding bit string suitable for processing.
- **TIME CALCULATION**: when an input from a different time step is detected this state is activated and the delta time between the previous and the current set of input is evaluated and, once this evaluation is complete, the translation process concludes.
- **LOAD IN BUS**: at this point, the input bus from the Input Translator delivers the bit string to the first Layer of the network and, from the second input set, the first layer sends the output spikes to the following layer. This state prepares the Layer/s to start processing the inputs, marking the transition from input translation to input processing.
- **BUF IN**: in this state, bits are sent serially from the Layer to the Mesh Block and subsequently to the Processing Element (PE). During each clock cycle a bit is processed, contributing to weight accumulation within the PE.
- **BUF WAIT**: the Layer communicates its status to the CU. If the input bus still contains data to be processed, the FSM returns to the `BUF IN` state otherwise, if the input string has been fully processed, the FSM transitions to the next state.
- **OUT SHIFT**: in this state the CU controls data transmission between Mesh Blocks belonging to different rows of the matrix. If all inputs have been processed, the FSM moves to the `MEMBRANE DECAY` state to calculate exponential decay, otherwise, it transitions to the `LOAD INTERNAL` state to continue weight accumulation.
- **LOAD INTERNAL**: this state manages data transmission inside the Mesh and prepares the Layer for the next set of weight accumulation operations; the FSM then moves back to the `BUF IN` state. The behaviour of this state is equal to `LOAD IN BUS` but, instead of loading the data from the input bus, either from the input translator or, in case of the second layer, from the previous layer, the data are delivered from one row to the following one in a circular manner.
- **MEMBRANE DECAY**: this state initiates the membrane decay calculation, when all input has been processed. The accumulator and the bias of each neuron are added together and the membrane potential absolute value is evaluated as a preparation step for the exponential decay evaluation.

- **FINISH DECAY**: in this state the actual decay is calculated, a multiplier is employed to calculate the product between the membrane potential and the β coefficient.

- **DECAY NORMALIZATION**: finally, in this state, the sign of the membrane potential is restored and, depending on whether this is the first pass through this state, the FSM transitions to `MEMBRANE COMPUTATION FIRST` for the initial computation phase or `MEMBRANE COMPUTATION` for subsequent evaluations.

- **MEMBRANE COMPUTATION**: this state performs the final operations for evaluating firing conditions, determining whether a neuron spikes or not. The operations executed in this last two states are among the most computationally intensive tasks for the PE and consequently for the entire network.

- **MEMBRANE COMPUTATION FIRST**: similar to the previous state (`MEMBRANE COMPUTATION`) but with specific adjustments to handle the initial stage of operation (Stage 1 described in section 5.2).

- **OUT CONVERSION**: in this state, the FSM manages the conversion of output bit strings from the last layer back into AER format; this processes persists for as many clock cycles as there are neurons in the last layer and it is executed by the Output translator (in our network the process last 10 clock cycles).
  When the conversion completes the FSM can either return to `EVENT IN` if more inputs are queued, transitions to the `LAST ROUND` state to manage the final cycle of operations (Stage 3 described in section 5.2) or moves to the `OUT` state if processing is complete.

- **LAST ROUND**: this state ensures proper synchronization during the final stage of operation (Stage 3) and initiates the last cycle of processing operations.

- **OUT**: the final state of the FSM, where a single clock cycle is used to reset all components, making the network ready to receive new inputs.

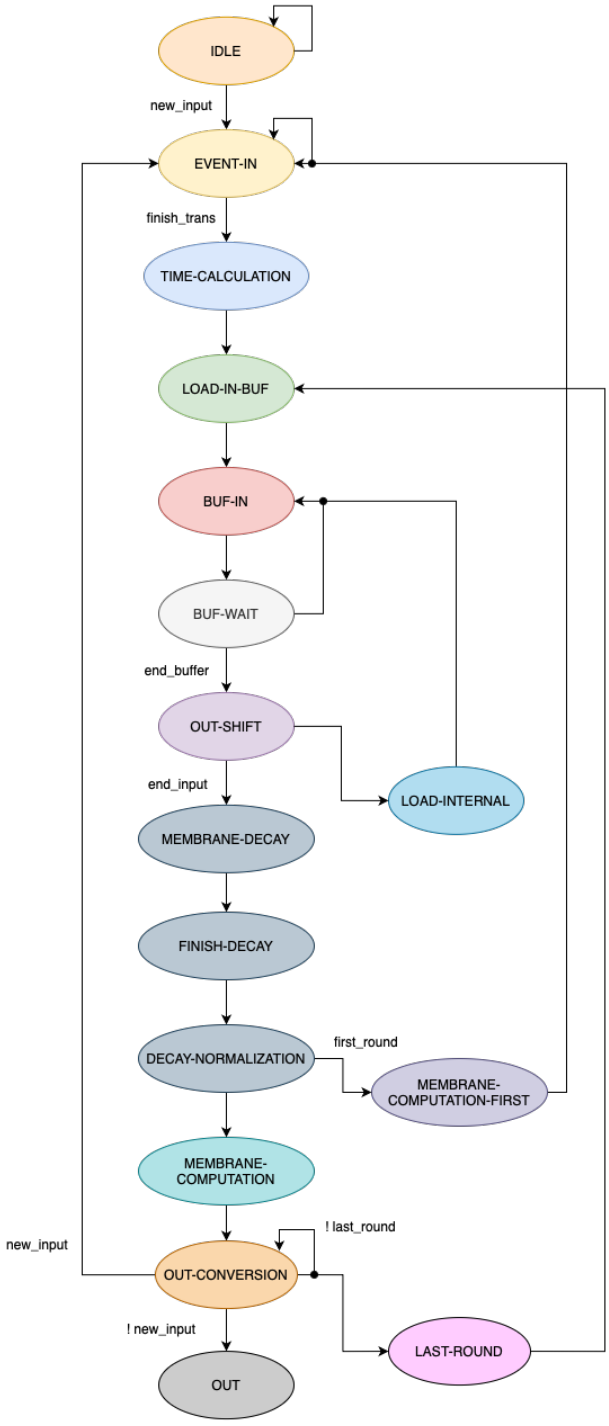

Figure 5.2: Finite State Machine flow

5.2 Optimized Structure for Two-Layer Operation

The network has been specifically optimized to handle two layers, which is suitable for tasks such as processing the MNIST dataset that are simple enough to not require any more complex network structure but can be applied, increasing the number of neurons for each layer, also on more complex dataset.
When the network receive a new sets of inputs the operations are executed in 3 stages:

- **Stage 1**: in the first stage, only the first layer is active, processing the initial set of inputs; this stage last as long as the layer has processed all the inputs, latency is approximately $2 \cdot num\_inputs$.
- **Stage 2**: in the second stage, both layers operate simultaneously. The first layer processes the next set of inputs while the second layer processes the outputs generated by the first layer in the previous time step.
  This stage is the longest and last as long as all input sets have been processed. Assuming that every set is processed in a time $t = K$ this stage lasts $t = K \cdot (num\_sets - 1)$.
- **Stage 3**: in the final stage, the first layer ceases operation, and the second layer processes the last set of inputs from the first layer. This stage last $2 \cdot num\_inputs$ where, this time, the number of inputs depends on the output of the first layer.

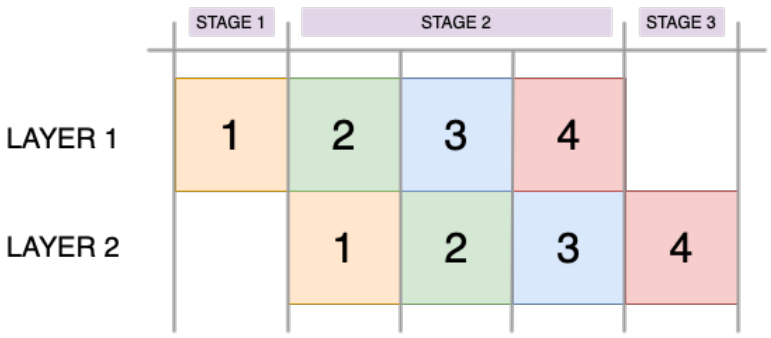

Figure 5.3: Pipeline execution

The layers are designed to operate in a sort of pipelined fashion; the scheme in Figure 5.3 show in a very simplified way how the two layers work. In the example

is assumed that there are 4 sets of inputs and each square block represents all the operations described in the Section 5.1.
This pipelined processing ensures efficient use of resources and minimizes idle time. However, due to the different sizes and processing times of the two layers, there might be situations where one layer completes its tasks before the other. In such cases, the layer that finishes first enters a "wait" state where it follows the FSM in the same way as the other layer is doing but the *PEs* are disabled so no operation is performed until the other layer completes its processing. This wait mechanism ensures synchronization and prevents data loss or errors due to unsynchronized operations.
Overall, this structured and optimized design allows the network to handle neural computations effectively, ensuring high performance and adaptability for different tasks and datasets.

5.3 Memory management

One of the major challenges in this design was the size of the neural network model, which significantly complicates the creation of efficient embedded models due to limitations in the available resources.
FPGAs provide the option of using Block RAM (*BRAM*), which is optimized to save space while maximizing memory access speed. BRAMs in AMD FPGAs are ideal for these applications, as they allow extensive customization and optimization through Vivado's Block Memory Generator that provides various solutions in terms of external interface (Native, AXI or AXI-4), of memory types (single port RAM, single port ROM,...), of speed, of error correction,....

The strategy adopted for this design was to allocate separate memories for each layer of the neural network each containing the weights organised through the script for parameters' extraction. Each **BRAM** was configured as a single-port ROM with a three-port interface: clock, address, and data out; during configuration, the enable signal was set to always active to simplify memory management and the memory was loaded with *.coe* (coefficient) file so that, if the dimension of the layer remains constant, the memory content can be easily updated changing this file content. A scheme showing the external interface of the memory is shown in Figure 5.4.

In the developed system two memories have been employed. The first memory block stores the input layer weights, with 784 lines of 360 bits each while the second memory block is smaller, consisting of 40 lines of 90 bits.
The reason for separating the memories belonging to each layer is to simplify the parameters' storing for future optimization and for fine-tuning and also to reduce

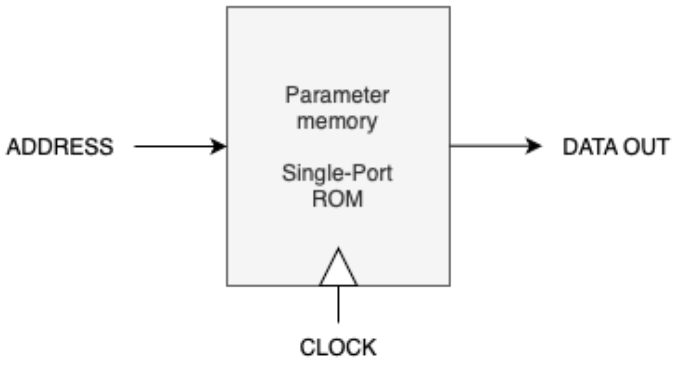

Figure 5.4: Parameter Memory interface

the bus dimension. Each layer, when an input is received by the system, accesses the corresponding element in memory that contains the necessary weights for the computation.

5.4 Accelerator usage

To properly interface with the accelerator, you need to follow a structured procedure to ensure data is properly transmitted to and processed by the accelerator, while also allowing you to retrieve the results accurately from the output bus. This description assumes you are using an AER (Address Event Representation) format to send and receive data.

1. **Wait for the Accelerator to be Ready**: the first step requires to monitor the `ready` signal. It will remain low (*0*) while the accelerator is busy; once the `ready` signal rises to high (*1*), the accelerator is prepared and ready to accept inputs.

2. **Reset the Accelerator**: when the accelerator is ready you must first reset the accelerator to bring it in a known state and avoid unwanted behaviour due to unsynchronized state of one or more components of the system. This can be achieved by setting the `reset` signal high (*1*) for at least one clock cycle, and then set it low (*0*) to release the reset and allow the accelerator to start operating normally.

3. **Send New Input Data**: after monitoring again the `ready` signal and having verified that it is high (*1*), raise the `new_input` signal to indicate that a new set of inputs will be sent to the accelerator.

The `AER_string_in` bus is used to send input data in AER format. Keep sending the input data monitoring the `ready` signal; if the accelerator detects a new timestep (i.e., data belonging to a different timestep from the previous set) the accelerator will stop reading new data, the ready signal will go low (*0*) and the system will start processing the received data.

4. **Pause Input Transmission While the Accelerator is Processing**: as soon as the accelerator starts processing, the `ready` signal will go low (*0*), the input transmission must be interrupted otherwise any further inputs will be lost so.

5. **Monitor the Output and Wait for the Ready Signal**: while the `ready` signal is low (*0*), monitor the `AER_string_out` bus for results.
   When the input processing comes to the end the accelerator will output the results, this require to monitor continuosly the `AER_string_out` bus until the `ready` signal returns to high (*1*), indicating that the accelerator has finished processing the current input set and is ready to receive more input data.

6. **Repeat the Input Transmission Process**: once the `ready` signal is high again, repeat the input transmission process:

   - Keep the `new_input` signal high.
   - Send the next set of inputs using the `AER_string_in` bus.
   - Pause again when the `ready` signal goes low, wait for the accelerator to process, and read the results from the `AER_string_out` bus.

7. **Final Output and Shutdown**: After sending the last set of inputs, while monitoring for the value on the output bus, wait for the `ready` signal to go high one final time. This will be the output from the network corresponding to the last input set you sent.

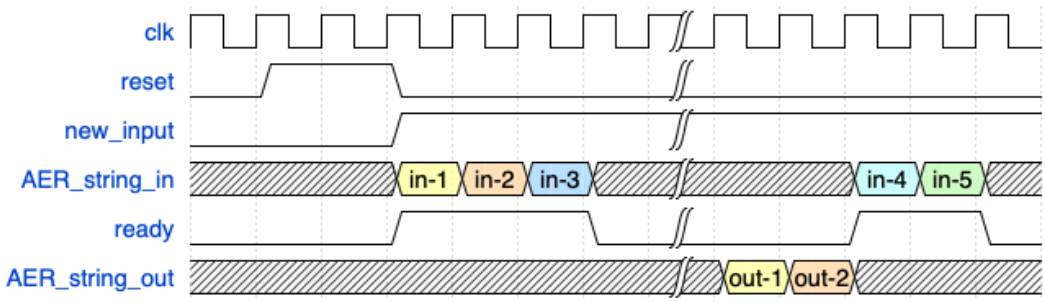

Figure 5.5: Accelerator Waveform example

5.4.1 Example Description

The waveform in Figure 5.5 illustrates how the system handles multiple time steps of input (two in the example), monitors the `ready` signal and ensures that input data from different time steps is processed correctly.
Moreover the timing shows how the `ready` signal plays a crucial role in controlling when data can be sent and when it is being processed to avoid data loss and correct synchronization with the input source.

1. Reset and Initial Setup:
    - Initially, the system is in the reset state, as indicated by the `reset` signal being high (*1*).
    - The reset is deactivated (*0*) after a few clock cycles, and the system is ready to accept new inputs.
2. First input set (*in-1 and in-2*):
    - The `ready` signal is high (*1*), meaning the accelerator is ready to receive inputs.
    - The `new_input` signal goes high, indicating that new input data is being sent and `AER_string_in` shows the values `in-1` and `in-2`, which are inputs for the first time step.
3. Second Input Set:
    - The third input (`in-3`) belongs to a different time step compared to `in-1` and `in-2`. The accelerator detects this change in timestep, as indicated by the `ready` signal dropping to *0*.
    - The system is now processing the previous set of inputs (`in-1` and `in-2`) while the information related to `in-3` is stored in memory ready for the next input transmission. The input transmission must be interrupted while the system processes the data otherwise the data would be lost.
4. Output generation from first input set (*out-1 and out-2*):
    - After processing the first set of inputs, the accelerator produces outputs `out-1` and `out-2`, visible on the `AER_string_out` bus.
    - Once the output is available, the `ready` signal rises again (*1*), indicating that the accelerator is ready for more inputs.
5. Second Input Set cont. (*in-4*):

- The next set of inputs, made of two inputs (`in-3`, received in the first input set transmission, and `in-4`), is transmitted after `ready` is high.
- As with `in-3`, input `in-5` belongs to a different time step, causing `ready` to drop again as the system processes the data; as before `in-5` is saved in a temporary memory.

Chapter 6

HW simulation, synthesis and implementation

6.1 Hardware Simulation

The simulation process supports testing with either a single input or multiple inputs. Single-input testing is used to assess parameters like network latency and power per image, while multi-input testing provides insights into accuracy across a larger dataset, similar to the SW simulation. To ensure consistency and comparability with the SW simulation, the HW simulation process follows specific steps that are shown in the flow chart in Figure 6.1.

Parameters extraction

First, the same script used in the SW simulation is employed to extract 100 random samples from the MNIST dataset. These samples are saved in a designated folder and reused for HW simulation, ensuring that both HW and SW simulations evaluate the same data. Each time the script is relaunched, it generates new random samples, enabling tests on varying data sets.
Next, weight and bias conversion is handled automatically by a script. For weights, the script takes the SW simulation's weight files, stored as integers (`parameter_layer.coe`) and converts them to binary format for hardware deployment (`weight_layer.coe`), maintaining consistency between simulations; this conversion process elaborates both the weights of the input and of the output layer. After the weight conversion, biases are extracted from the *state_dict* in their floating-point format. These are then converted to integers and subsequently to binary values. The converted bias values are printed on-screen and manually copied into the source code, simplifying their deployment in HW due to the low number

of biases involved.

Input processing

Once the network has the correct parameters the accelerator is ready to start the input processing. The simulation begins by launching the testbench, which iteratively processes all elements in the folder containing the input samples.
The testbench (`tb._mnist.sv`), for each sample, reset the accelerator and then starts the input processing; this process increase the latency but this small performance loss ensures that the accelerator always starts from a known state and emulates how the accelerator would be employed in a real context.
The output from the accelerator, which are strings in the AER format, is written into a file in a specific format. At first, for every sample, the corresponding target value is extracted from the input file and reported in the output one then every string is written line by line finally, once the processing of the inputs is complete, a *"STOP"* line is written to signify the end of that sample's output.
Additionally, in the TCL console of Vivado, when an input is processed, is shown the target value and when the simulation is complete, to help the designer monitor the evolution of the process.

Output interpretation

The final step is the output interpretation; the interpretation of the output differs significantly between traditional and spiking neural networks.
In a conventional non-spiking neural network the output layer typically represents the network's classification of the input data where each neuron corresponds to a different class while, for a multi-class classification problem the interpretation is more complex and the network assigns a probability to each class, with the neuron yielding the highest activation value representing the predicted class.
This supervised learning approach is based on the final activations as confidence levels, and the neuron with the maximum activation signifies the model's best guess.
In a Spiking Neural Network (*SNN*) the interpretation of outputs depends on the encoding method use to convert the input information into spikes. With **rate coding** the class prediction is based on the neuron with the highest firing rate or spike count, on the other hand, if the chosen method is **latency coding** the output interpretation is based on which neuron spikes first, assuming that faster spiking corresponds to stronger confidence in a given class.

For the hardware simulation of the accelerator **rate coding** is employed and the number of output neurons match the number of classes so, for interpreting the output of the network, the neuron that spikes most frequently represent the

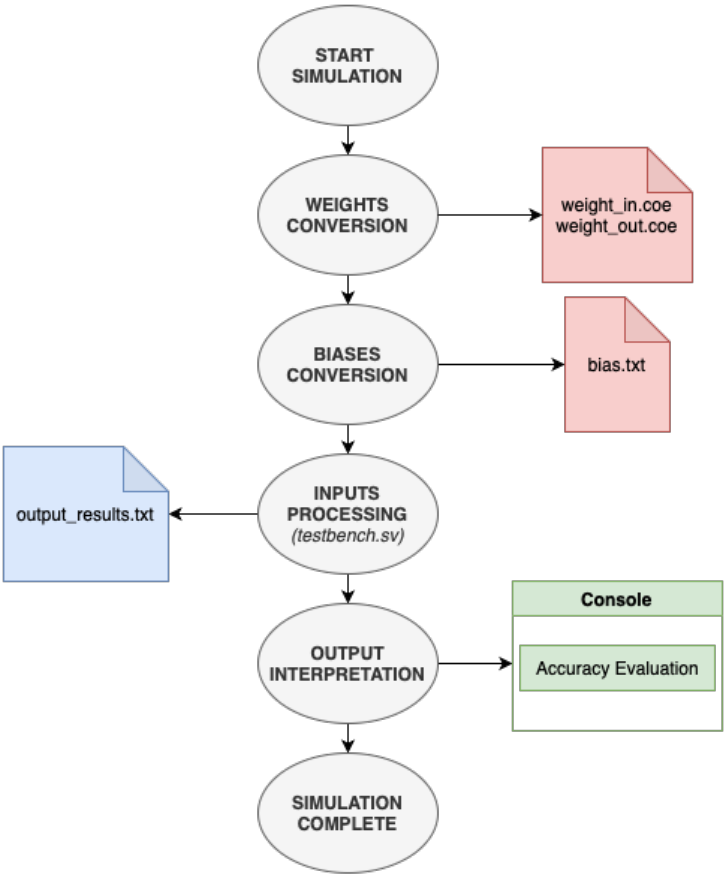

Figure 6.1: HW Simulaiton flow chart

predicted class. For doing so a dedicated script processes the output file, generated by the HW simulation *testbench*, where the classification results are stored.
For each sample the script begins by extracting the target value from the first line of the output string; it then initializes a 10-element vector (corresponding to the 10 classes in the *MNIST* dataset) with zeros and, for each AER-formatted string in the output, the script extracts address and timing information, incrementing the vector element corresponding to that address. The presence of the *"STOP"* line marks the end of the current output's processing, allowing the script to analyze the vector by identifying the maximum value, which represents the predicted class. Lastly a comparison between the predicted and expected classes is then executed. This process is repeated for all outputs, and finally, accuracy is computed over the entire test set.

6.2 Synthesis and Implementation

In FPGA design using Vivado, **synthesis** and **implementation** are two crucial stages that convert a high-level hardware description into a deployable, optimized design on an FPGA.
Each process plays a distinct role in preparing a digital design for hardware execution:

1. **Synthesis**: this step translates the functional design, written in **HDL** (Hardware Description Language), into a gate-level netlist optimizing for factors like area and speed. The output of synthesis serves as a blueprint for how logical operations are mapped onto the FPGA's resources.

2. **Implementation**: it focuses on physically placing and routing the design within the FPGA's fabric by assigning physical locations to each component and connects them according to timing and area constraints.

6.2.1 Synthesis process

The step that follows simulation is synthesis. The synthesis process translates a high-level hardware description, written in **System Verilog**, into a gate-level representation and represents the first major step in preparing a design for deployment on an FPGA.

Synthesis consists of the following three main steps:

- **Parsing and Elaboration**: the HDL code is read and analyzed to identify modules, hierarchies and finally checked for any syntax errors.

- **Optimization and Technology Mapping**: if the design passes parsing, Vivado optimizes it, mainly to reduce area and increase speed. Moreover, in this stage, components are mapped to primitive gates or LUTs (Look-Up Tables) available on the FPGA.

- **Netlist Generation**: this final step of synthesis creates a **netlist**, a structural representation of the design, that outlines the interconnections between logic gates and other elements.

The initial parsing step is independent of the chosen platform and is consistent across different FPGAs. In contrast, the Optimization and Netlist Generation stages are technology-specific, meaning that Vivado must consider the hardware resources available on the target FPGA.

The chosen platform for deployment is the **PYNQ-Z2** board, an FPGA development board designed for the Xilinx Zynq-7000 SoC (System on Chip), which

combines ARM processing with FPGA fabric. The PYNQ-Z2 board includes a dual-core ARM Cortex-A9 processor alongside the programmable logic (PL) resources of the Zynq FPGA, making it ideal for applications requiring a combination of embedded processing and reconfigurable hardware. This board also provides multiple interfaces, such as HDMI, audio, and communication protocols, and offers General Purpose I/O (*GPIO*) headers compatible with Arduino and Raspberry Pi pins, making it highly flexible for prototyping.

In Vivado, the target device is first specified, either by selecting it from the installed device list or importing board files. After that, the **constraints file** is customized using the Vivado Editor to specify I/O ports and clock configuration. A constraints file (*XDC* file) defines physical parameters that guide the synthesis and implementation processes. These constraints are crucial to ensure that the design operates as intended on the physical board, by mapping logical signals to physical pins, defining clock speeds, and setting other essential parameters.

For the hardware accelerator, external interfaces require the `ready`, the `reset` and the `new_input` PIN, as well as PINs for input and output strings in AER format. The PYNQ-Z2 board's GPIO headers provide 24 Arduino and 28 Raspberry Pi-compatible pins, making it possible to distribute the input and output signals of the accelerator across these 52 pins. This choice enables contiguous grouping of pins for both input and output strings, facilitating system usage and debugging.

Using these *GPIO* pins offers several advantages:

- **Power Measurement for Simulation**: utilizing these pins simplifies power measurement, increasing the confidence level in power estimation by providing a reference during simulation.

- **Physical Accessibility**: the accessible GPIO headers simplify the connection of external components or interfaces, enabling easy input and output testing.

- **Flexibility and Interfacing Options**: these GPIO pins can interface with various devices, including custom-designed interfaces, enhancing the versatility of the board for a wide range of applications.

Finally, the *XDC* file also specifies the clock constraint. For this synthesis and implementation, a $100MHz$ clock frequency is chosen to align with other accelerator specifications, ensuring easy comparison of results.

The outcome of the synthesis stage serves as input for the next stage: **implementation**.

6.2.2 Implementation process

The **implementation** phase in Vivado transforms the synthesized design into a configuration that can be deployed on a specific FPGA platform.
Following **synthesis**, the implementation process maps the gate-level netlist to the physical resources of the target FPGA, optimizing for timing, area, and power based on platform constraints. Since the hardware resources and interconnect architecture vary between FPGA models, the implementation phase, as for the synthesis one, is inherently platform-dependent.

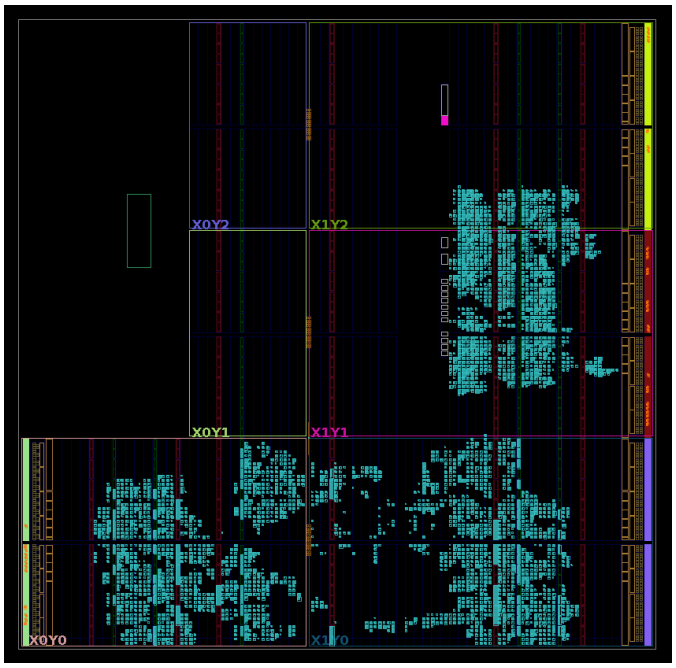

Figure 6.2: Accelerator's device scheme

The major steps in Vivado's **implementation** process are as follows:

- **Opt Design** (*Optimization*): in this initial step, Vivado refines the synthesized netlist by applying optimizations tailored to reduce area usage and improve timing performance.
  Some optimization examples are logic cells' merging, unused logic's trimming or specific FPGA features' optimization such as block RAMs (*BRAM*) or Digital Signal Processing (*DSP*) slices. The results of the optimization can be investigated by looking at the *Utilization Report*.
- **Place Design** (*Placement*): the placement maps logical elements onto the FPGA's physical resources, assigning each LUT, flip-flop, or memory block a

specific location on the FPGA fabric.
The goal of this procedure is to optimize data flow by minimizing interconnect delays and ensuring that the components are placed as close as possible to each other based on signal paths and design constraints.

- **Route Design** (*Routing*): this stage involves connecting the placed elements using the FPGA's interconnect resources. Vivado uses internal algorithms to determine the optimal paths to ensure signal integrity while limiting the delay and power consumption.
  Routing can be tailored with additional constraints to further optimize the design especially if a specific timing performance or signal routing is necessary. For this design, standard settings are used, allowing Vivado's routing algorithms to find an optimal solution.

- **Bitstream Generation**: once placement and routing are complete, Vivado generates the *bitstream*, a binary file that encodes the hardware configuration for the FPGA. This *bitstream* can be directly loaded onto the FPGA for testing and deployment.

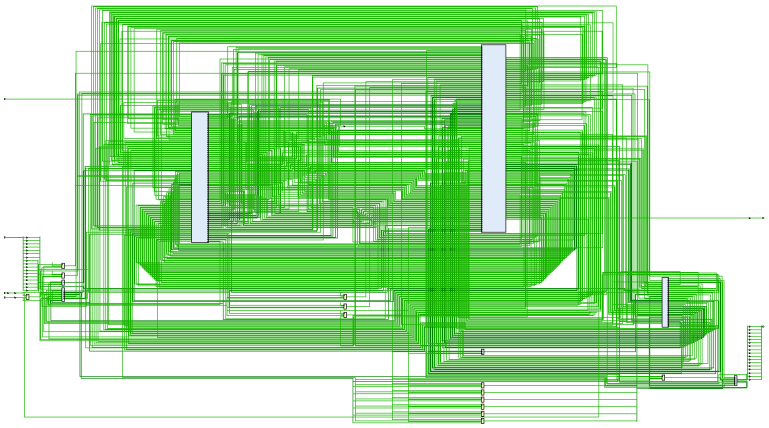

Figure 6.3: Accelerator's schematic

Vivado allows for extensive customization throughout the implementation process, such as: setting timing constraints, enforcing specific setup and hold times, defining area constraints, keeping certain logic blocks within specified regions, or enabling power optimizations for energy-efficient designs.
The design choice was to keep the default configuration; this decision comes from the idea of letting Vivado make decisions automatically to verify the "real" system performances without the intervention of the designer, however, better results could be obtained by setting more stringent constraints.

The visual resource distribution (Figure 6.2) shows organized clustering by functionality across different regions labeled *X*Y** with a more relevant concentration in regions *X0Y0*, *X1Y0* and *X1Y1*. The Device scheme from Vivado helps identify where denser regions, where the logic activity is more intense, are located.
On the other hand the schematic in Figure 6.3 illustrates a dense arrangement of routing paths, as highlighted by the green lines. Major logic blocks appear highly interconnected, showing that the design leverages parallel data paths extensively. This organization supports efficient data processing and signal transfer which is crucial for complex computational tasks within the FPGA.

6.3 Resource Utilization Summary

The utilization report for the FPGA design indicates a moderate use of resources, as shown in Table 6.1. The design utilizes a small portion of Slice LUTs and DSPs, while the memory resources are minimally utilized.
The breakdown of slice logic resources is presented in Table 6.2.

Resource Type	**Used**	**Available**	**Utilization (%)**
Slice LUTs	8011	53200	15.06
Slice Registers	9263	106400	8.71
F7 Muxes	204	26600	0.77
F8 Muxes	100	13300	0.75
Block RAM Tile	11.5	140	8.21
RAMB36/FIFO	11	140	7.86
RAMB18	1	280	0.36
DSP Blocks	50	220	22.73

Table 6.1: Resource Utilization Summary

The most significant contributors to resource allocation in this design are LUTs and DSP blocks. Approximately 15% of available LUTs and 22.73% of DSP resources are utilized, indicating moderate usage with a focus on computational components. These results align with expectations, as the network was specifically designed to minimize resource usage. The primary complexity lies within the computational blocks (neurons), while the remaining resource demands come from neuron interconnections and control logic that synchronize and manage the flow of operations.

As outlined above, Vivado leverages DSP (*Digital Signal Processing*) blocks to optimize performance and resource efficiency in FPGA designs. DSP blocks offer several key advantages: they provide efficient resource usage, as they are specialized

Slice Logic Components	Used	Available	Utilization (%)
Slices	3156	13300	23.73
Slice LUTs	8011	53200	15.06
LUTs as Logic	8011	53200	15.06
LUTs as Memory	0	17400	0.00
Slice Registers	9263	106400	8.71
Registers with internal LUT usage	240	-	-

Table 6.2: Detailed Slice Logic Breakdown

for complex arithmetic operations like multiply-accumulate (MAC), enhancing the accelerator's performance during intensive computational tasks. Additionally, DSP blocks are engineered for high-speed arithmetic which reduces the number of clock cycles required, thereby increasing overall throughput. Another benefit that make DSP convenient is their lower power consumption relative to traditional logic elements, contributing to overall power efficiency and making them ideal for low-power applications.
Vivado's synthesis engine automatically infers and optimizes DSP block usage based on the design's requirements and constraints resulting in an improved placement and utilization, reducing area and power consumption while achieving performance goals.

6.3.1 Power and Timing analysis

Vivado's Power Report provides three main analyses of power consumption, offering detailed insights into resource efficiency and power distribution across the design's components.

The first section, the **Power Summary**, offers a high-level overview of total on-chip power, measured at 0.182 W. This includes both dynamic (0.074 W) and static power (0.108 W), indicating that the majority of power consumption is static. Additionally, the junction temperature is calculated at 27.1 °C, suggesting safe operational thermal levels.

The second report, **On-Chip Components Power Consumption**, breaks down power usage across individual FPGA resources. DSP blocks, utilized at 22.73%, consume 0.018 W, reflecting their central role in handling arithmetic operations while I/O components show a high utilization rate of 30.40%, though they contribute only 0.003 W to the total power, suggesting that a more efficient approach for output connectivity may be advantageous, rather than directly routing pins from the accelerator interface to the FPGA's embedded pins.
Furthermore, Slice Logic, comprising LUTs and registers, forms a significant part of the power profile, while Block RAMs consume minimal power, indicating efficient

Parameter	**Value**
Total On-Chip Power (W)	0.182
Design Power Budget (W)	Unspecified
Power Budget Margin (W)	NA
Dynamic (W)	0.074
Device Static (W)	0.108
Effective TJA (C/W)	11.5
Max Ambient (C)	82.9
Junction Temperature (C)	27.1
Confidence Level	Low
Setting File	—
Simulation Activity File	—
Design Nets Matched	NA

Table 6.3: Power Summary

resource usage for parameter storage.

On-Chip Component	**Power (W)**	**Used**	**Available**	**Utilization (%)**
Clocks	0.021	3	—	—
Slice Logic	0.013	20583	—	—
LUT as Logic	0.011	8011	53200	15.06
CARRY4	0.001	1034	13300	7.77
Register	<0.001	9263	106400	8.71
F7/F8 Muxes	<0.001	304	53200	0.57
Others	0.000	72	—	—
Signals	0.012	15011	—	—
Block RAM	0.008	11.5	140	8.21
DSPs	0.018	50	220	22.73
I/O	0.003	38	125	30.40
Static Power	0.108			
Total	**0.182**			

Table 6.4: On-Chip Components Power Consumption

The final report, **Power Consumption by Hierarchy**, details power distribution across specific design hierarchies. The `Layer_INPUT` component shows the highest power usage, consuming 0.045 W, primarily due to the arithmetic-heavy operations within `MeshBlock` and `ProcessingElement` instances aligning with the expectations as this block is the largest, containing 40 neurons and 31,360 synapses

(784×40).
In contrast, the `Translate_Event`, `input_memory`, and `output_memory` components exhibit relatively lower power usage due to their smaller, energy-efficient design. This distribution highlights that processing-intensive tasks are the primary drivers of power consumption, underscoring the need for optimized arithmetic handling to further enhance power efficiency across the design.

Name	**Power (W)**
network	0.074
ControlUnit	0.001
Layer_INPUT	**0.045**
gen_row[0].gen_col[0].MeshBlock	0.011
ProcessingElement	0.009
gen_row[0].gen_col[1].MeshBlock	0.011
ProcessingElement	0.009
gen_row[1].gen_col[0].MeshBlock	0.011
ProcessingElement	0.009
gen_row[1].gen_col[1].MeshBlock	0.011
ProcessingElement	0.009
Layer_OUTPUT	0.010
gen_row[0].gen_col[0].MeshBlock	0.006
ProcessingElement	0.003
gen_row[1].gen_col[0].MeshBlock	0.004
ProcessingElement	0.003
Translate_Event	0.005
input_memory	0.006
U0	0.006
inst_blk_mem_gen	0.006
output_memory	0.003
U0	0.003
inst_blk_mem_gen	0.003

Table 6.5: Power Consumption by Hierarchy

The **timing analysis** provided by Vivado shows three main timing metrics (shown in Table 6.6):

- **Setup Timing:** the worst negative slack (WNS) is measured at 1.066 ns, which is sufficient for our 100 MHz clock. This indicates that the timing constraints are comfortably met, with no negative slack or failing endpoints. The positive slack allows for some margin in the design, which is crucial for

ensuring reliable operation under varying conditions.

- **Hold Timing:** the worst hold slack (WHS) is 0.037 ns, showing that the design meets hold timing requirements without any failing endpoints. This is particularly important in ensuring that the data is stable at the clock edge, preventing data corruption due to timing violations; moreover, the absence of failing endpoints further confirms the robustness of the design.
- **Pulse Width Timing:** the worst pulse width slack (WPWS) is 4.500 ns, with no negative slack, further confirming that pulse width timing constraints are met. This generous slack ensures that the pulse width is long enough for reliable detection and processing of signals by downstream components.

To achieve the required *100 MHz* clock speed, the critical path within the neuron, specifically from the accumulator result to the final output, was split across different processing states. This approach minimizes the path delay by dividing operations, such as exponential decay of the membrane and bias accumulation, into separate stages; in this way the design meets timing requirements despite the high computational load on the neuron module.

Additionally, the timing analysis indicates that there are zero failing endpoints across all metrics, suggesting the design's robust against timing violations. The comprehensive slack across setup, hold, and pulse width timings provides further assurance that the design will perform reliably in its intended application. Overall, these results demonstrate a successful alignment of the design with the targeted clock frequency and performance criteria.

Timing Type	Parameter	Value
Setup	Worst Negative Slack (WNS)	1.066 ns
	Total Negative Slack (TNS)	0.000 ns
	Worst Hold Slack (WHS)	0.037 ns
Hold	Total Hold Slack (THS)	0.000 ns
	Number of Failing Endpoints	0
Pulse Width	Worst Pulse Width Slack (WPWS)	4.500 ns
	Total Pulse Width Negative Slack (TPWS)	0.000 ns

Table 6.6: Timing Analysis Summary Constraints

6.3.2 Hardware Accelerators comparison

The Table 6.7 shows a comparison of the performance of the EDAMAME accelerator and similar accelerator projects. All results were obtained by testing the hardware

using the MNIST dataset to ensure consistency. Given the dataset's nature, each *static* image was converted into a spike representation over 100 `time_steps`, as the accelerator's latency depends on the number of cycles required to process the input.

- **Clock Frequency (f_{clk})**: EDAMAME operates at 100 MHz, consistent with most other designs, though some operate at higher frequencies, such as Nevarez et al. [16] at 200 MHz or Li et al. [17] at 300 MHz. While higher frequencies can increase throughput, they also impact on power consumption and design complexity.
- **Neuron and Weight Bit Width (bw)**: the average neuron bit width across other designs is approximately 11.5 bits, close to EDAMAME's 12-bit width while the average weight bit width among other designs is about 11 bits, whereas EDAMAME uses 9 bits. This lower precision in both neuron and weight bit widths slightly impacts accuracy but significantly benefits memory savings and reduces power consumption.
- **Neuron Model and Update Mechanism**: EDAMAME uses the LIF neuron model, a popular choice among accelerators due to its balance between performance and efficient resource use. EDAMAME's event-based update approach aligns with designs focused on energy efficiency, while others adopt clock-driven mechanisms.
- **FPGA Logic Cells and DSPs**: EDAMAME uses around 17,274 logic cells, much lower than the average of 31,000 in other designs, highlighting its focus on resource efficiency. With 50 DSPs, EDAMAME achieves a balanced use of DSP resources; some accelerators use no DSPs, while others, prioritizing performance, have more DSP-intensive designs (e.g., Li et al. [17] with 288 DSPs).
- **Architecture**: EDAMAME's architecture (784-40-10) is relatively lightweight compared to deeper networks like Gerlinghoff et al. [18] (32×32×1–6c5 p2–16c5–p2–120c5–120–84–10) or like Panchapakesan et al. [19] (28x28-32c3-p2-32c3-p2-256-10). This structure favors speed and energy efficiency, though it limits the accelerator's applicability for more complex tasks that benefit from deeper architectures.
- **Power, Number of Synapses, and Accuracy**: EDAMAME consumes approximately 0.182W, one of the lowest power consumptions among the designs. Its simpler structure reduces the number of synapses to 31,760, contributing to its efficiency; however, EDAMAME's trade-offs in power and area utilization result in an accuracy of 88.5%, slightly lower than that of other designs.

Overall, EDAMAME demonstrates an effective balance between efficiency and performance. Its low power consumption and resource usage make it highly optimized for FPGA constraints. The moderate architecture and parameter choices yield strong results in accuracy and energy efficiency, particularly suitable for embedded applications.

Design	**Liu et al. [20]**	**Nevarez et al. [16]**	**Li et al. [17]**	**Gerlinghoff et al. [18]**	**Panchapakesan et al. [19]**	**Khodamoradi et al. [21]**
Year	2023	2021	2023	2022	2021	2021
f_{clk} **[MHz]**	100	200	300	200	200	N/R
Neuron bw	8	8	12	N/R	4	N/R
Weights bw	8	8	8	3	4	N/R
Update	Clock	Clock	Clock	Clock	Event	Event
Model	IF	Spike-by-Spike (SbS)	LIF	LIF	IF	LIF
FPGA	XA7Z020	XC7Z020	XCZU3EG	XCVU13P	XCZU9EG	XA7Z020
Avail. BRAM	140	140	216	2688	912	140
Used BRAM	N/R	16	50	N/R	N/R	40.5
Avail. DSP	220	220	360	12288	2520	220
Used DSP	0	46	288	0	N/R	11
Avail. logic cells	159,600	159,600	211,680	3,088,800	822,240	159,600
Used logic cells	27,551	23,704	15,000	51,000	N/R	39,368
Arch	28x28-32c3-p2-32c3-p2-256-10	28x28x2-32c5-p2-64c5-p2-1024-10	28x28-16c3-64c3-p2-182c3-256c3-10	32x32x1-6c5-p2-16c5-p2-120c5-120-84-10	28x28-32c3-p2-32c3-p2-256-10	28x28-16c7-24c7-32c7-10
#syn	8,960	75,776	2,560	25,320	10,752	320
$\mathbf{T}_{lat/img}$ **[ms]**	0.27	1.67	0.49	0.29	0.08	N/R
Power [W]	0.28	0.22	2.55	3.40	N/R	N/R
E/img [mJ]	0.076	0.37	1.250	0.986	N/R	N/R
E/syn [nJ]	8.48	4.88	488	38.9	N/R	N/R
Accuracy	99.00%	98.84%	98.12%	99.10%	99.30%	98.50%
Design	**Han et al. [22]**	**Gupta et al. [23]**	**Li et al. [24]**	**Carpegna (SPIKER) et al. [25]**	**Carpegna (SPIKER+) et al. [26]**	**EDAMAME (this work)**
Year	2020	2020	2021	2022	2024	2024
f_{clk} **[MHz]**	200	100	100	100	100	100
Neuron bw	16	24	16	16	6	12
Weights bw	16	24	16	16	4	9
Update	Event	Event	Hybrid	Clock	Clock	Event
Model	LIF	LIF	LIF	LIF	LIF	LIF
FPGA	XC7Z045	XC6VLX240T	XC7VX485	XC7Z020	XC7Z020	XC7Z020
Avail. BRAM	545	416	2,060	140	140	140
Used BRAM	40.5	162	N/R	45	18	11.5
Avail. DSP	900	768	2,800	220	220	220
Used DSP	0	64	N/R	0	0	50
Avail. logic cells	655,800	452,160	485,760	159,600	159,600	159,600
Used logic cells	12,690	79,468	N/R	55,998	7,612	17,274
Arch	784-1024-1024-10	784-16	784-200-100-10	784-400	784-128-10	784-40-10
#syn	1,861,632	12,544	177,800	313,600	101,632	31,760
$\mathbf{T}_{lat/img}$ **[ms]**	6.21	0.50	3.15	0.22	0.78	1.62
Power [W]	0.477	N/R	1.6	59.09	0.18	0.182
E/img [mJ]	2.96	N/R	5.04	13	0.14	0.29
E/syn [nJ]	1.59	N/R	28	41	1.37	9.13
Accuracy	97.06%	N/R	92.93%	73.96%	93.85%	88.50%

Table 6.7: Combined Data Set with Additional Designs

Chapter 7

Further improvements

In this chapter, we explore potential enhancements to the current accelerator design aimed at increasing efficiency, accuracy, and adaptability in deployment.
The first improvement centers on replacing the current Leaky Integrate-and-Fire (LIF) neuron model with a **Synaptic** neuron model. This transition offers the opportunity to better emulate biological neural networks by incorporating **Synaptic** dynamics. The **Synaptic** model's ability to account for dynamic neural interactions is anticipated to significantly boost the network's expressiveness and robustness.
The second improvement consists in integrating the **Brevitas** framework for quantization, the design can achieve higher computational efficiency and reduced memory footprint thanks to the Quantization Aware Training that this framework owns. Quantization with **Brevitas** is designed to retain essential model characteristics even with lower bit precision, enabling the accelerator to operate effectively under stringent hardware constraints.

7.1 The Synaptic neuron

There exists a large variety of artificial neuron models, they go from biophysically accurate models to extremely simple artificial neuron that compose most of the deep learning network. Biological plausibility has been at the center of the research when building artificial neuron.

The similarity between nerve membranes and RC circuits was observed by Louis Lapicque in 1907. He stimulated the nerve fiber of a frog with a brief electrical pulse, and found that neuron membranes could be approximated as a capacitor with a leakage.
Somewhere in the middle between biological plausibility and practicality there is the leaky integrate-and-fire (*LIF*) neuron model. This specific type of neuron takes

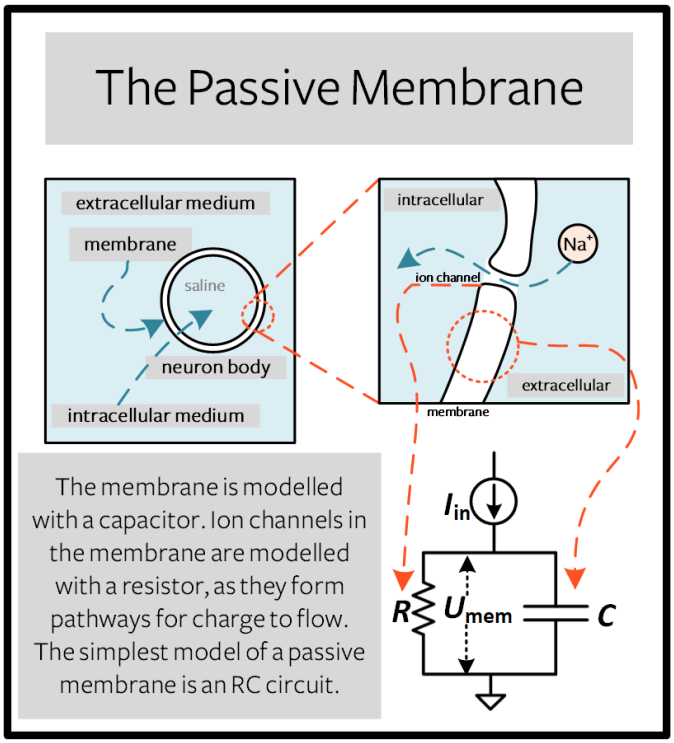

Figure 7.1: Passive Membrane in biological neuron

the sum of weighted inputs and integrates them over time with a leakage; then, if the integrated value exceeds a threshold, the *LIF* neuron will emit a voltage spike. The research has enlarged the number of available models and nowadays there exists different neurons that are part of the *LIF* family and they mainly differ in how the previously described electrical model is implemented.

The *LIF* neurons's family, in Artificial Neural Network, is made of different neurons classified for their similarity with the Biological neuron and for their performance in ANN, with the **Synaptic** neuron being one of them and representing a good trade off between performance and semblance.
The main characteristic of the **Synaptic** neuron is that it embodies two exponentially decaying terms:

- **Synaptic current** (I_{syn}) : it represents the flow of electric charge across the synapse in response to the flow of neurotransmitters inside the neuron.
- **Membrane potential** (U_{mem}) : it is the electric potential difference across the neuronal membrane. The Membrane potential reflects the neuron's response to inputs, leading to the generation of spikes.

The drawing in Figure 7.2 shows the time evolution of the **Synaptic** Current

(left) and the Membrane Potential (right) in response of a series of input stimuli (S_{in}) and the resulting output (S_{out}).

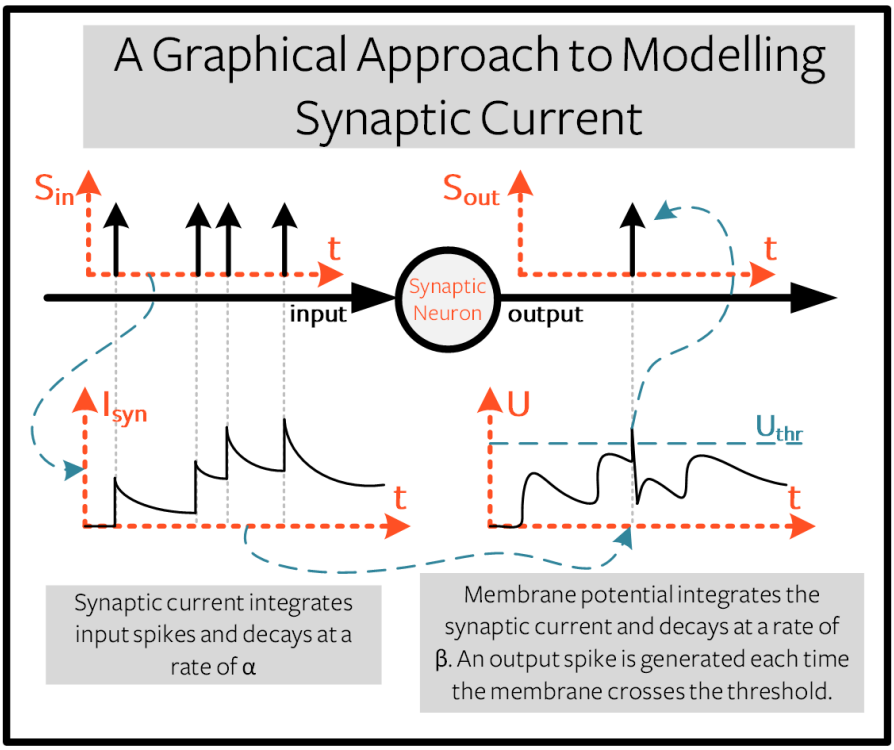

Figure 7.2: Synaptic neuron behaviour

7.1.1 Synaptic neuron mathematical model

As explained for the standard *LIF* model in Chapter 2 the evaluation of an exponential decay is not an easy task due to the complexity of its calculations, leading to the necessity of a quantized version; as a consequence two coefficients, α and β, are used, representing the ratio between subsequent terms of the exponential:

$$\begin{aligned} \alpha &= I_{syn}(t + \Delta t)/I_{syn}(t) = e^{-\frac{\Delta t}{\tau_{syn}}} \\ \beta &= U_{mem}(t + \Delta t)/U_{mem}(t) = e^{-\frac{\Delta t}{\tau_{mem}}} \end{aligned} \tag{7.1}$$

In this expression τ_{syn} and τ_{mem} represent the time constants; they are chosen by the designer and their values affects the network performance as well as the exponential decay behavior.

With the introduction of the α and β coefficients we can evaluate the **Synaptic** current and the Membrane potential; the mathematical expressions that model

their behavior are the following:

$$\begin{aligned} I_{\text{syn}}[t+1] &= \alpha I_{\text{syn}}[t] + WX[t+1] \\ U_{\text{mem}}[t+1] &= \beta U_{\text{mem}}[t] + I_{\text{syn}}[t+1] - R[t] \end{aligned} \tag{7.2}$$

In the expression in Equation 7.2 the term $WX[t+1]$ is the product between the weight W associated to a certain connection and the input $X[t+1]$, while $R[t]$ refers to the reset mechanism that regulates the behaviour of the neuron when the membrane potential exceed the threshold voltage U_{thr}.

In a similar to that of the standard *LIF* neuron, based on the input and the evolution over time of the **Synaptic** current and the membrane potential, it is possible to evaluate the expression for the output spike S_{out}

$$S_{out} = \begin{cases} 1 & \text{if } U(t) > U_{\text{threshold}}, \\ 0 & \text{otherwise.} \end{cases} \tag{7.3}$$

7.1.2 From clock-driven to event-driven synaptic neuron

The mathematical model described in the previous paragraph works well when a **clock-driven** approach is used. As can be observed, the α and β parameters, used to quantized the exponential decay, allows to evaluate at every time instant (defined by the clock signal) the current value for I_{syn} and U_{mem}. If, instead, an **event-driven** approach is adopted some modifications are requested.

As first step we can rewrite the expressions for the **Synaptic** current and the membrane potential using the "real" exponential decay at a specific time instant $t = T$:

$$\begin{aligned} I_{\text{syn}}[T] &= e^{-\frac{T}{\tau_{syn}}} I_{\text{syn}}[0] + WX[t+1] \\ U_{\text{mem}}[T] &= e^{-\frac{T}{\tau_{mem}}} U_{\text{mem}}[0] + I_{\text{syn}}[T] - R[T] \end{aligned} \tag{7.4}$$

The main difference with respect to the definition given by equation 7.2 resides in the meaning of time t. As described before, the **event-driven** approach became more efficient as the sparsity of the datas increases, leading us to define $I_{\text{syn}}[0]$ and $U_{\text{mem}}[0]$ as the values of **Synaptic** current and membrane potential when the neuron receives the previous non zero input while $I_{\text{syn}}[T]$ and $U_{\text{mem}}[T]$ represent the value of **Synaptic** current and of membrane potential at the instant T when an input different from zero arrives.

The Figures 7.3 and 7.4 show the different results of membrane potential ($U_{\text{mem}}[t]$) between the **clock-driven** model (blue curve) and the **event-driven** one (orange curve) for different initial conditions:

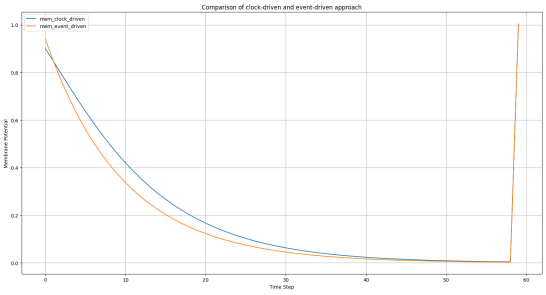

Figure 7.3: Small synaptic current

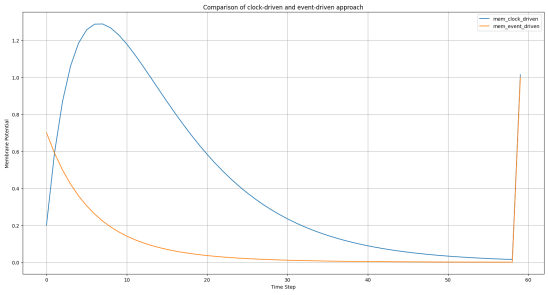

Figure 7.4: Big synaptic current

1. Figure 7.4 : $I_{\text{syn}}[0] = 0.5$ and $U_{\text{mem}}[0] = 0.2$
2. Figure 7.3 : $I_{\text{syn}}[0] = 0.04$ and $U_{\text{mem}}[0] = 0.9$

In the pictures the starting time is $t = 0$ and, after a time $T = 60$, the neuron receives an input whose value is equal to 1; this is the reason for the fast jump on the far right of the graph.

The chosen initial condition are just for simulations purposes but, as shown by the two graphs, the performance of the **event-driven** approach is very poor compared with the **clock-driven** one. It is important to notice that the accuracy of the model is strongly influenced by the initial value $I_{\text{syn}}[0]$. When the value increases the error increases too caused by the fact that if calculations are executed at the end, as in the **event-driven** approach, all intermediate terms (between $t = 0$ and $t = T - 1$) of the **Synaptic** current that sum up to the $i - th$ term of membrane potential, are lost.

For the previously explained reasons this mathematical approach requires some additional complexity to integrate in the expression also the contribution due to the "lost terms".
To better understand what the "lost terms" are and to find a way to generalize the expressions we can calculate the result that we obtain using the **clock-driven** approach, described by equation 7.2, assuming that an input arrives at $t = 4$ (time_step) and with the following parameters: W=1 and R=0.

$$\begin{aligned} I_{\text{syn}}[0] &= I_{\text{syn\_init}} & U_{\text{mem}}[0] &= U_{\text{mem\_init}} \\ I_{\text{syn}}[1] &= \alpha I_{\text{syn}}[0] & U_{\text{mem}}[1] &= \beta U_{\text{mem}}[0] + I_{\text{syn}}[1] \\ I_{\text{syn}}[2] &= \alpha I_{\text{syn}}[1] & U_{\text{mem}}[2] &= \beta U_{\text{mem}}[1] + I_{\text{syn}}[2] \\ I_{\text{syn}}[3] &= \alpha I_{\text{syn}}[2] + X[3] & U_{\text{mem}}[3] &= \beta U_{\text{mem}}[2] + I_{\text{syn}}[3] \end{aligned} \tag{7.5}$$

The equations 7.5 shows how I_{syn} and U_{mem} vary as a function of the time steps. From this preliminary mathematical analysis it can be seen as if each term, both for the **Synaptic** current and membrane potential, depends on the previous one.

Having defined each terms, the membrane potential at time $t = 4$ can be calculated by substituting in the last expression the value of I_{syn} and U_{mem} as a function of their respective initial value:

$$U_{\text{mem\_clk\_drv}}[4] = \beta^4 U_{\text{mem\_init}}[0] + \alpha I_{\text{syn}}[0](\beta^3 + \beta^2\alpha + \beta\alpha^2 + \alpha^2) + X[3] \quad (7.6)$$

At first glance the formula might look very different from the equation 7.4 but if we express its result for $t = 4$ in the same way substituting $e^{-\frac{t}{\tau_{syn}}}$ with α^t and same for $e^{-\frac{t}{\tau_{mem}}}$ with β^t we get:

$$U_{\text{mem\_evn\_drv}}[4] = \beta^4 U_{\text{mem\_init}}[0] + \alpha^3 I_{\text{syn}}[0] + X[3] \quad (7.7)$$

By observing this expression we can notice that the term with β^4 represent the exponential decay after a time $t = 4$, the term with α^3 is the exponential decay of the **Synaptic** current after a time $t = 3$ and finally the term $X[3]$ is the non-zero input received at time $t = 4$.

By subtracting the equation 7.7 from 7.6, we can get the "lost terms"

$$lost\_terms = \alpha\beta^3 I_{\text{syn}}[0] + \alpha^2\beta^2 I_{\text{syn}}[0] + \alpha^3\beta I_{\text{syn}}[0] \quad (7.8)$$

The term $\alpha\beta^3$ is the **Synaptic** current $I_{\text{syn}}[1]$ integrated three times (β^3), the term $\alpha^2\beta^2$ is the **Synaptic** current $I_{\text{syn}}[2]$ integrated two times (β^2) and finally the term $\alpha^3\beta$ is the **Synaptic** current $I_{\text{syn}}[1]$ integrated one time (β).

These are exactly the terms that we don't consider when we directly calculate the membrane potential at a general $t = 4$, in fact, they corresponds to the terms $t \in (1 : 3)$.

The last step is to generalized the expression for the calculation of the membrane potential at a generic time instant T. When observing equation 7.6 we can notice that it can be splited in four parts:

1. The exponential decay of the membrane potential at instant $t = T$(i.e. $\beta^4 U_{mem_init}[0]$)
2. The product between α and the initial value of $I_{syn}[0]$ (i.e. $\alpha I_{syn}[0]$
3. A polynomial made of β and α coefficients (i.e. $\beta^3 + \beta^2\alpha + \beta\alpha^2 + \alpha^2$)
4. Finally the contribution of the input at instant T (i.e. $X[3]$)

The first, the second and the last element of the equation are very simple to be predicted so what is left is the third one. This polynomial, as was observed before, has a very regular form since it includes the "lost terms" and the power of α related to the $T - 2$ term; in Python code the polynomial can be calculated, for a generic time T, as follows:

Listing 7.1: Polynomial calculation

```
    if t == 0:
        poly[t] = 0
    elif t == 1:
        poly[t] = 1
    else:
        for n in range(t) :
            if n == 0:
                poly[t] += beta**(t-n-1)
            else:
                poly[t] += (beta**(t-n-1)) * alpha**(n)
```

In this code the various terms are calculated based on the input value t that represent the time instant at which the neuron received a non zero input.

Having defined all the necessary terms that are needed to calculate the membrane potential using the **event-driven** approach a generalized set of formulas 7.9 and 7.10 is used and Figure 7.5 shows the comparison between the **event-driven** and **clock-driven** simulations.
The graph shows that the two curves perfectly overlap showing how, with this new model, the behaviour of the membrane potential is correctly evaluated.

$$U_{\text{mem}}[t] = \beta^t U_{\text{mem}}[0] + \alpha I_{\text{syn}}[0](poly_t) + X[t-1] \tag{7.9}$$

$$poly_t = \begin{cases} t & \text{if } t = (0,1) \\ \sum_{n=0}^{t-1}(\beta^{t-n-1} * \alpha^n) & otherwise \end{cases} \tag{7.10}$$

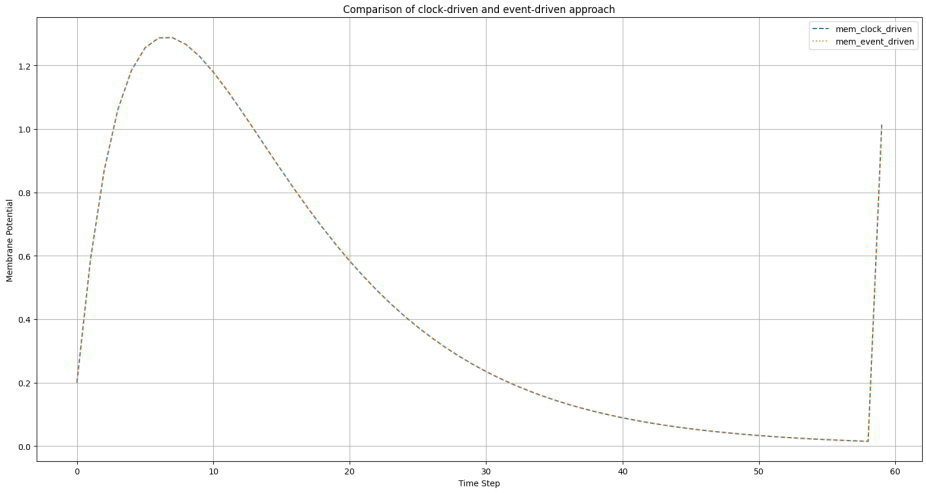

Figure 7.5: Event vs clock driven approach

7.1.3 Synaptic event-driven neuron criticality

To discuss the criticality of using **Synaptic** neurons in spiking neural networks, it's important to analyze the evolution of the membrane potential in response to inputs. As shown in Figure 7.5, when a **Synaptic** neuron receives an input, its membrane potential rises before decaying exponentially back to its resting state. This behavior reveals a challenge with **event-driven** neurons, especially during the initial transient phase where the potential increases.
In a **clock-driven** neuron, the membrane potential is continuously monitored at each time step. If the membrane potential crosses the firing threshold during this transient, the neuron fires, ensuring precise spike timing. However, in an **event-driven** neuron, the neuron remains "asleep" during the decay phase and only "wakes up" when a new input arrives. As a result, the neuron may miss potential threshold crossings during the decay, leading to missed firings and negatively impacting the overall network accuracy.

To address this issue, three possible solutions can be proposed:

- **Accepting the Event-Driven Compromise**: in this approach, the neuron remains 100% **event-driven** and ignores any potential threshold crossings during the transient increasing phase. The neuron only monitors the membrane potential when an input arrives, similar to the behavior of Leaky Integrate-and-Fire (LIF) neurons. This method significantly reduces power consumption and simplifies implementation, but it comes at the cost of reduced accuracy, particularly in terms of both missed spikes and spike timing precision.

- **Hybrid Neuron Model**: a more advanced solution involves adopting a hybrid neuron model. Upon receiving an input, the neuron briefly switches to a **clock-driven** mode, allowing it to monitor the membrane potential during the transient phase and, if the membrane potential crosses the threshold during this period, the neuron fires. Once the potential begins to decay or stabilize, the neuron reverts to its **event-driven** state; this implementation offers improved accuracy and timing precision but increases power consumption.

- **Future Evolution Prediction**: this approach is a hybrid between the previous two. When the neuron receives an input, it estimates the maximum potential value that will be reached during the transient phase; if this predicted value exceeds the firing threshold, the neuron either switches to **clock-driven** mode to ensure precise timing or spikes immediately, simplifying the process. This method strikes a balance between accuracy and resource efficiency, as it limits **clock-driven** operations to situations where they are necessary, while also preserving computational simplicity by spiking immediately when possible.

7.1.4 Network simulation

The last step to verify the effectiveness of this neuron is to use it inside a Spiking Neural Network and, after executing a suitable training, analyze its performance; the MNIST dataset has been chosen as a benchmark and the neuron has been simulated "Accepting the Event-Driven Compromise" so it is actually ignoring what happens during the growing transient.

The neural network chosen for testing the functionalities of the **event-driven Synaptic** neuron is a three-layer architecture with a similar shape to that used for the LIF neuron. It has an input size of 784 neurons, a hidden layer of 1000 neurons, and an output layer of 10 neurons (for classification into 10 categories specifically for the MNIST dataset); all the layers are fully connected ones.
The network was trained and tested using the MNIST dataset to then compare the results with the custom **event-driven** LIF neuron and the results are shown in Figure 7.6. On the left plot the training accuracy evolution are shown, the curves show that both neuron types perform similarly in terms of the final accuracy however the custom neuron (orange) exhibits faster convergence than the standard one and it appears to have more stable performance in terms of oscillations. On the right plot the testing accuracy evolution are shown; in testing, the custom neuron achieves a higher maximum accuracy (98.44%) than the standard neuron (96.88%), indicating better generalization, both networks stabilize similarly after epoch 15, though the custom neuron retains a small but consistent advantage over the standard neuron.

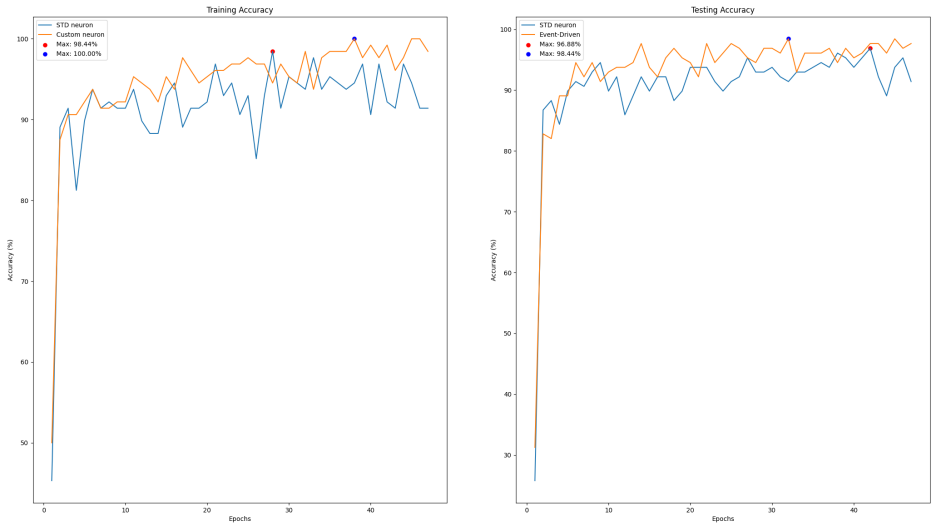

Figure 7.6: MNIST training/testing using standard and custom synaptic neuron

The results obtained with the **Synaptic** neuron show strong performance, even though the chosen strategy for the **event-driven** neuron tends to sacrifice some efficiency. However, the increased complexity of the **Synaptic** neuron, compared to the simpler LIF neuron, makes it more suitable for tackling more challenging tasks. The MNIST dataset, being relatively simple, doesn't fully test the neuron's capabilities however, the **Synaptic event-driven** neuron implement using this approximation, if used in more complex scenarios, could result in a noticeable drop in accuracy.

Testing the **Synaptic** neuron's behavior was useful in validating certain hypotheses formed from the LIF neuron. The primary distinction between the **clock-driven** and **event-driven** neurons lies in the latter's reliance on a simpler model that more deeply approximates the network's exponential dynamics. While this might lead one to expect weaker performance from the simpler **event-driven** neuron, this is not the case for the MNIST dataset; in fact, the simplicity of MNIST allows the less complex neurons to perform more effectively.

7.2 Quantization with Brevitas

Brevitas is an open-source library specifically designed for neural network quantization in PyTorch. Its primary goal is to facilitate the training of quantized neural networks using Quantization-Aware Training (QAT) and enhancing the capabilities of PyTorch with a series of Custom Layers, built on top of the standard ones of PyTorch, that owns an high level of parameters for optimizing the quantization process.
The main capabilities of **Brevitas** can be summarized as follows:

- Brevitas allows both Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ).
- Supports customizable bit-widths for both weights and activations offering flexibility for balancing precision, memory footprint, and computational efficiency. **Brevitas** is apable of handling integer quantization from binary (1-bit) up to 8-bit or higher.
- Provides quantized versions of common operations such as Linear and Convolutional layers other than quantized activation functions like ReLU, batch normalization, and pooling layers.

The key advantage of **Brevitas** is its seamless integration with PyTorch, enabling the developers to easly introduce quantization into their existing workflows without needing to switch frameworks. Moreover **Brevitas** provides greater flexibility, compared to PyTorch's native quantization, which typically focus on standard 8-bit

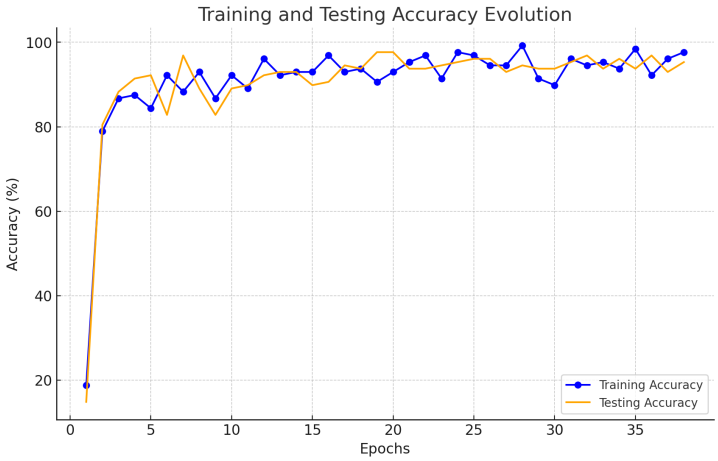

Figure 7.7: MNIST training/testing using Brevitas quantization

quantization; this level of customizability enhance PyTorch's networks flexibility and it is especially important for deploying models on custom hardware architectures that require non-standard bit-widths.
The plot in Figure 7.7 shows the evolution in time both for training and inference using **Brevitas** framework on the usual network built from the MNIST classification (784 inputs - 40 hidden layer - 10 output layer) where weights are quantized of 4 bits while input activations and biases are quantized on 8 bits.

To illustrate how simple it is to incorporate **Brevitas**, let's compare a standard PyTorch Linear layer with a quantized version from **Brevitas**:

Listing 7.2: quantized network

```
# PyTorch standard Linear laeyr
self.fc_std = nn.Linear(num_inputs, num_hidden, bias = True)
# Brevitas Linear layer
self.fc_brv = qnn.QuantLinear(num_inputs, num_hidden, bias = True,
    weight_bit_width=4, input_quant=Int8ActPerTensorFloat, bias_quant=
    Int8Bias)
```

In a standard PyTorch 'nn.Linear' layer the framework uses 32-bit floating-point weights and activations; on the other hand the **Brevitas** layer, 'QuantLinear', allows for quantization of both weights and activations. In this example, weights are quantized to *4-bit* precision, and the activations and biases are quantized to *8-bit* integers. The addition of quantization-related parameters like *weight_bit_width*, *input_quant*, and *bias_quant* is the primary difference, but integrating **Brevitas** is otherwise quite straightforward.

7.2.1 Using Brevitas in Hardware Deployment

While **Brevitas** provides significant advantages for quantization-aware training in software, there are some critical issues when moving from training to hardware deployment:

- **Brevitas quantization process** may not map directly to hardware like FPGAs or custom accelerators. Most hardware platforms support only specific types of quantization, typically 8-bit or binary formats. This means that custom quantization schemes, such as 4-bit or 6-bit used during training, may be difficult to implement in hardware without custom logic.
- **Hardware Acceleration:** when designing custom hardware, converting the Brevitas-trained model to hardware can be challenging.
  Even if **Brevitas** significantly simplifies the quantization process in software and helps developers build custom quantized models with reduced effort in terms of code production compared to other methods, the processes executed by PyTorch and **Brevitas** under the hood to optimize the quantization make direct translation from software to hardware difficult.
  Using tools like HLS (High-Level Synthesis) might simplify the deployment of **Brevitas** models onto custom hardware, but the level of customizability on the hardware side would be significantly compromised.

In conclusion, while **Brevitas** is a powerful tool for training quantized models in PyTorch, integrating these models into custom hardware accelerators requires careful planning and adaptation to the constraints of the target hardware platform.

Bibliography

[1] *Biological neuron model*. Oct. 2024. URL: https://en.wikipedia.org/wiki/Biological_neuron_model (cit. on p. 2).

[2] *How Neurons Communicate*. URL: https://opentextbc.ca/biology/chapter/16-2-how-neurons-communicate/ (cit. on p. 3).

[3] Jason K Eshraghian, Max Ward, Emre Neftci, Xinxin Wang, Gregor Lenz, Girish Dwivedi, Mohammed Bennamoun, Doo Seok Jeong, and Wei D Lu. «Training spiking neural networks using lessons from deep learning». In: *Proceedings of the IEEE* 111.9 (2023), pp. 1016–1054 (cit. on pp. 6, 12, 16, 17, 21, 25).

[4] Adarsh Kosta and Kaushik Roy. *Adaptive-SpikeNet: Event-based Optical Flow Estimation using Spiking Neural Networks with Learnable Neuronal Dynamics*. Sept. 2022. DOI: 10.48550/arXiv.2209.11741 (cit. on p. 8).

[5] *Spiking neural network*. Oct. 2024. URL: https://en.wikipedia.org/wiki/Spiking_neural_network (cit. on p. 10).

[6] *Compressed Latent Replays for Lightweight Continual Learning on Spiking Neural Networks | IEEE Conference Publication | IEEE Xplore*. URL: https://ieeexplore.ieee.org/abstract/document/10682744 (visited on 10/08/2024) (cit. on p. 11).

[7] Adam Paszke et al. «PyTorch: An Imperative Style, High-Performance Deep Learning Library». In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on p. 12).

[8] Mahyar Shahsavari, David Thomas, Marcel van Gerven, Andrew Brown, and Wayne Luk. «Advancements in spiking neural network communication and synchronization techniques for event-driven neuromorphic systems». In: *Array* 20 (2023), p. 100323. ISSN: 2590-0056. DOI: https://doi.org/10.1016/j.array.2023.100323. URL: https://www.sciencedirect.com/science/article/pii/S2590005623000486 (cit. on p. 14).

[9] *Difference between Clock-driven and Event-driven Scheduling*. May 2020. URL: https://www.geeksforgeeks.org/difference-between-clock-driven-and-event-driven-scheduling/ (cit. on p. 14).

[10] Limiao Ning, Junfei Dong, Rong Xiao, Kay Tan, and Huajin Tang. «Event-driven spiking neural networks with spike-based learning». In: *Memetic Computing* 15 (May 2023), pp. 1–13. DOI: 10.1007/s12293-023-00391-2 (cit. on p. 14).

[11] Sijia Lu and Feng Xu. «Linear leaky-integrate-and-fire neuron model based spiking neural networks and its mapping relationship to deep neural networks». In: *Frontiers in Neuroscience* 16 (2022). ISSN: 1662-453X. DOI: 10.3389/fnins.2022.857513. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2022.857513 (cit. on p. 15).

[12] Federico Paredes-Vallés, Kirk Scheper, and Guido Croon. *Unsupervised Learning of a Hierarchical Spiking Neural Network for Optical Flow Estimation: From Events to Global Motion Perception*. Mar. 2019. DOI: 10.48550/arXiv.1807.10936 (cit. on p. 21).

[13] Dario Padovano, Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. «SpikeExplorer: Hardware-Oriented Design Space Exploration for Spiking Neural Networks on FPGA». en. In: *Electronics* 13.9 (Jan. 2024). Number: 9 Publisher: Multidisciplinary Digital Publishing Institute, p. 1744. ISSN: 2079-9292. DOI: 10.3390/electronics13091744. URL: https://www.mdpi.com/2079-9292/13/9/1744 (visited on 09/06/2024) (cit. on p. 35).

[14] *Ethernet*. Sept. 2024. URL: https://en.wikipedia.org/wiki/Ethernet (cit. on p. 36).

[15] *Network on a chip*. Sept. 2024. URL: https://en.wikipedia.org/wiki/Network_on_a_chip (cit. on p. 38).

[16] Yarib Nevarez, David Rotermund, Klaus R. Pawelzik, and Alberto Garcia-Ortiz. «Accelerating Spike-by-Spike Neural Networks on FPGA With Hybrid Custom Floating-Point and Logarithmic Dot-Product Approximation». In: *IEEE Access* 9 (2021). Conference Name: IEEE Access, pp. 80603–80620. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3085216 (cit. on pp. 86, 87).

[17] Jindong Li, Guobin Shen, Dongcheng Zhao, Qian Zhang, and Yi Zeng. «FireFly: A High-Throughput Hardware Accelerator for Spiking Neural Networks With Efficient DSP and Memory Optimization». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 31.8 (Aug. 2023). Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems, pp. 1178–1191. ISSN: 1557-9999. DOI: 10.1109/TVLSI.2023.3279349 (cit. on pp. 86, 87).

[18] Daniel Gerlinghoff, Zhehui Wang, Xiaozhe Gu, Rick Siow Mong Goh, and Tao Luo. «E3NE: An End-to-End Framework for Accelerating Spiking Neural Networks With Emerging Neural Encoding on FPGAs». English. In: *IEEE Transactions on Parallel and Distributed Systems* 33.11 (Nov. 2022). Publisher: IEEE Computer Society, pp. 3207–3219. ISSN: 1045-9219. DOI: 10.1109/TPDS.2021.3128945 (cit. on pp. 86, 87).

[19] Sathish Panchapakesan, Zhenman Fang, and Jian Li. «SyncNN: Evaluating and Accelerating Spiking Neural Networks on FPGAs». In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. ISSN: 1946-1488. Aug. 2021, pp. 286–293. DOI: 10.1109/FPL53798.2021.00058 (cit. on pp. 86, 87).

[20] Hanwen Liu, Yi Chen, Zihang Zeng, Malu Zhang, and Hong Qu. «A Low Power and Low Latency FPGA-Based Spiking Neural Network Accelerator». In: *2023 International Joint Conference on Neural Networks (IJCNN)*. ISSN: 2161-4407. June 2023, pp. 1–8. DOI: 10.1109/IJCNN54540.2023.10191153 (cit. on p. 87).

[21] Alireza Khodamoradi, Kristof Denolf, and Ryan Kastner. «S2N2: A FPGA Accelerator for Streaming Spiking Neural Networks». In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 194–205. ISBN: 978-1-4503-8218-2. DOI: 10.1145/3431920.3439283 (cit. on p. 87).

[22] Jianhui Han, Zhaolin Li, Weimin Zheng, and Youhui Zhang. «Hardware implementation of spiking neural networks on FPGA». In: *Tsinghua Science and Technology* 25.4 (Aug. 2020). Conference Name: Tsinghua Science and Technology, pp. 479–486. ISSN: 1007-0214. DOI: 10.26599/TST.2019.9010019 (cit. on p. 87).

[23] Shikhar Gupta, Arpan Vyas, and Gaurav Trivedi. «FPGA Implementation of Simplified Spiking Neural Network». In: *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. Nov. 2020, pp. 1–4. DOI: 10.1109/ICECS49266.2020.9294790 (cit. on p. 87).

[24] Sixu Li, Zhaomin Zhang, Ruixin Mao, Jianbiao Xiao, Liang Chang, and Jun Zhou. «A Fast and Energy-Efficient SNN Processor With Adaptive Clock/Event-Driven Computation Scheme and Online Learning». In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 68.4 (Apr. 2021). Conference Name: IEEE Transactions on Circuits and Systems I: Regular Papers, pp. 1543–1552. ISSN: 1558-0806. DOI: 10.1109/TCSI.2021.3052885 (cit. on p. 87).

[25] Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. «Spiker: an FPGA-optimized Hardware accelerator for Spiking Neural Networks». In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. ISSN: 2159-3477. July 2022, pp. 14–19. DOI: 10.1109/ISVLSI54635.2022.00016 (cit. on p. 87).

[26] Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. *Spiker+: a framework for the generation of efficient Spiking Neural Networks FPGA accelerators for inference at the edge*. arXiv:2401.01141 [cs]. Jan. 2024. DOI: 10.48550/arXiv.2401.01141. URL: http://arxiv.org/abs/2401.01141 (visited on 01/26/2024) (cit. on p. 87).