POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Development of a hardware module for online learning on spiking neural networks with partial reconfiguration on FPGA.

Supervisors

Prof. Stefano DI CARLO

Prof. Alessandro SAVINO

Ph.D. Alessio CARPEGNA

Candidate

Liqi ZENG

March 2024

Summary

This thesis presents the design and implementation of a new hardware module designed to support online learning in Spiking Neural Networks (SNN) and its partial hardware reconfiguration on the Xilinx Artix-7 FPGA platform. The Spiking Neural Network can simulate the unique way of exchanging information in the form of electrical pulses between neurons in the biological brain. It is widely considered to be an ideal choice for embedded hardware implementation due to its low energy consumption and small size. SNNs are particularly well-suited to resource-efficient local learning algorithms such as Spike Timing-Dependent Plasticity (STDP). The goal of this study is to develop a small hardare module to enable online learning in SNNs. It starts by modeling the algorithm in python, applying it to a simple image classification task. It then transitions to developing a dedicated hardware component while ensuring consistency between the two approaches. Subsequently, this thesis deeply explores the impact of quantization level on learning accuracy and compares different approximate implementations of STDP, aiming to evaluate the specific impact of various calculation methods on the final performance. Such evaluation is crucial for optimizing the area and power consumption of SNN hardware modules. It aims to improve the overall performance and applicability of the SNN models through fine adjustments and provide a resource-efficient solution to the application of Artificial Intelligence in real-world problems. As part of this research, this hardware module combines key components such as the temporary buffer queue and calculation conversion unit. This integration enables dynamic interaction and updating of the SNN weight data, which is stored in SRAM memory cells distributed across the FPGA (Block Ram). This not only significantly improves the flexibility and efficiency of the online learning process, but also ensures the scalability and adaptability of the system by optimizing resource allocation.

The ultimate goal of this project is to deploy a comprehensive, SNN hardware accelerator on the Xilinx Artix-7 FPGA platform that can adapt to changing application requirements and support uninterrupted online learning. Through the results of this research, we expect to promote the development of intelligent systems to a higher level in terms of adaptability and energy efficiency ratio, especially in application scenarios that have strict requirements on performance and energy consumption.

Acknowledgements

I have been helped and guided by a number of key people in the process of completing this thesis. First of all, I would like to thank Prof. DI CARLO, who not only gave me the opportunity to do this research, but also provided valuable advice and support throughout the process. His help was very important to me.

I am also very grateful to Dr. Alessio for his guidance on the content of my thesis. His advice helped me to overcome many challenges.

In addition, I would like to thank my family, who have always supported me and given me strength.

Thank you to all those who have helped me.

Table of Contents

List of Tables

List of Figures

Acronyms

AI	artificial intelligence
SNN	Spiking Neural Networks
LIF	Leaky Integrate-and-Fire Model
STDP	Spike-Timing-Dependent Plasticity
FPGA	Field-Programmable Gate Array
MNIST	Modified National Institute of Standards and Technology dataset
LUT	Look-Up Table
BRAM	Block Random-Access Memory

Chapter 1

Introduction

1.1 Spiking Neural Networks

As society progresses and technology evolves, the need for a deeper understanding and simulation of the human brain's mechanisms has increasingly become a focal point. Neural networks play an increasingly important role in this endeavor, with Spiking Neural Networks (SNNs) serving as a more brain-like neural model. The application domains of SNNs span from basic scientific research to social life applications and industrial fields[1]. In the realms of machine learning and artificial intelligence, where efficiency and low power consumption are crucial, the information processing capability of SNNs stands out[2]. Compared to traditional deep learning models, SNNs have advantages in information representation, temporal dynamics, and energy efficiency, making them more suitable for devices with limited power resources, such as mobile and embedded systems, and more effective in processing dynamic information. In the development of neuroscience research and learning algorithms, SNNs offer a closer approximation to biological neurons. However, training this kind of networks can be challeging due to the non-differentiability of their activation function. In general training algorithms for SNNs result more complex and less efficient. Additionally, the technological advancement of SNNs lags behind that of traditional neural networks, with a lack of mature tools and frameworks.

Spiking neural networks (SNNs) are a distinct branch of Artificial Neural Networks (ANNs), designed to process and transmit information by simulating the bioelectrical activities and communication modes of neurons in biological neural systems. The goal of SNNs is to capture and utilize the efficient computational mechanisms of biological neural systems, particularly their ability to encode complex information through spike signals. The fundamental unit of information in SNNs is the "spike," a discrete event that mimics the electrical activity of neurons. Spike

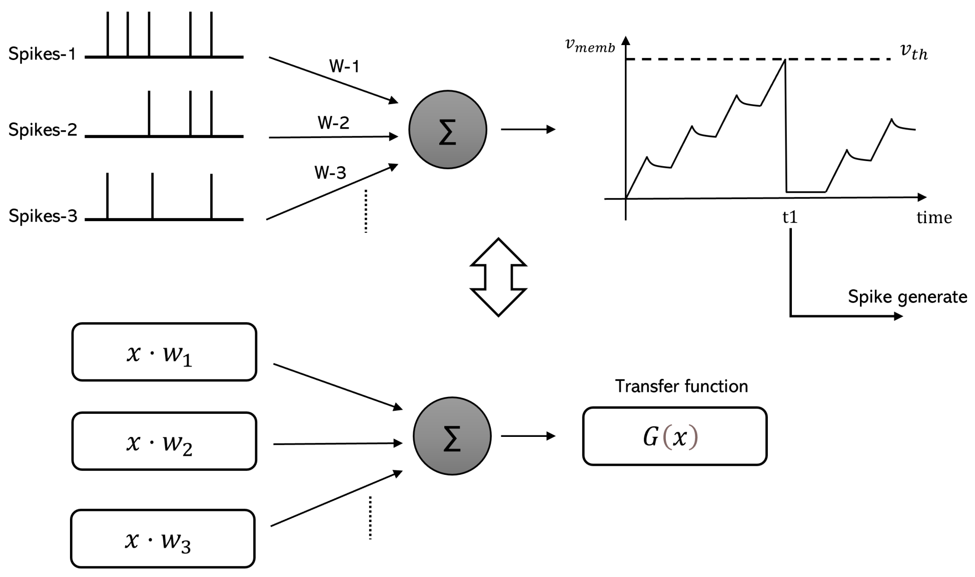


Figure 1.1: SNN network architecture

communication in SNNs emulates the way biological neurons exchange information through rapid voltage changes (spikes).

Neural models in SNNs are designed to simulate the electrophysiological properties of biological neurons. These models range from simple ones that include basic integrate-and-fire mechanisms to complex ones that encompass multiple ion channels, neuronal dynamics, and other biologically realistic details[3]. SNNs represent a highly advanced attempt to simulate the computational mechanisms of the human brain, processing information through the activity of spiking neurons. These networks employ unique and biologically realistic mechanisms such as the integrate-and-fire mechanism, spike transmission and synaptic weight adjustment, and spike encoding to accomplish complex information processing tasks. The integrate-and-fire mechanism simulates the process by which biological neurons accumulate input signals until the membrane potential reaches a threshold, triggering a spike, and then resetting to prepare for the next signal reception. This not only reflects the temporary accumulation of information but also mimics the action potential firing characteristic of biological neurons.

In SNNs, information transmission relies on the transfer of spikes between neurons through synapses, where the weight of each synapse determines the impact of a neuron's spike on another neuron's membrane potential. The adjustment of these synaptic weights is achieved through learning rules, with spike-timing-dependent plasticity (STDP)[4] being a key learning mechanism. STDP adjusts synaptic strengths based on the timing differences between pre- and postsynaptic spikes, enabling experience-based learning and memory through time-dependent synaptic weight adjustments. This mechanism provides SNNs with the ability to dynamically adjust network structures to adapt to environmental changes, making them particularly suited for tasks involving time-series data, such as speech recognition and video processing.

Spike encoding is another crucial concept, involving how information is represented and processed in SNNs. Information can be encoded in the network through differnt mechanisms, like the pattern, frequency, and sequence of spikes, with the choice of encoding method directly affecting network performance. Spike encoding not only allows the network to process information with high efficiency but also enables SNNs to simulate complex temporal dynamics and behavioral patterns[5].

In implementing SNNs, the Leaky Integrate-and-Fire (LIF) model is one of the most commonly used neuron models, providing a relatively simple yet effective way to simulate neuronal electrical activity. The LIF model accounts for the leakage effect of membrane potential, enabling neurons to mimic the capacitive properties and resistive nature of biological neurons. In addition to the LIF model, more complex neuron models such as the Izhikevich model[6] and the Hodgkin-Huxley[7] model are also employed in SNNs, producing a richer variety of neuronal firing patterns and offering deeper insights into the behavior of biological neurons. However these models are generally too complex to be implemented on dedicated hardware circuits, and are out of the scope of this work.

Despite the significant potential of SNNs to simulate the computational capabilities of the brain, their implementation and application still face numerous challenges, including the efficient implementation of learning rules like STDP and the optimization of spike encoding strategies to enhance the efficiency and accuracy of information processing. As understanding of these networks deepens and technology advances, SNNs are expected to play an increasingly important role in the field of artificial intelligence, especially in applications requiring efficient processing of time-series data and the simulation of complex neural dynamics[2].

1.1.1 Leaky Integrate-and-Fire (LIF) Model

The Leaky Integrate-and-Fire (LIF) model, as a fundamental neuronal model in spiking neural networks, offers a refined and effective abstraction of biological neuron behavior. This model simulates the accumulation of neuronal potential, the phenomenon of potential leakage, and the discharge of an output spike when the potential reaches a specific threshold, reflecting how biological neurons respond to external input signals. This simplified mathematical framework not only captures the basic functions of biological neurons, such as receiving, processing, and transmitting information, but also provides an efficient computational tool for studying and simulating complex neural networks.

In the LIF model, changes in the neuron's membrane potential are represented

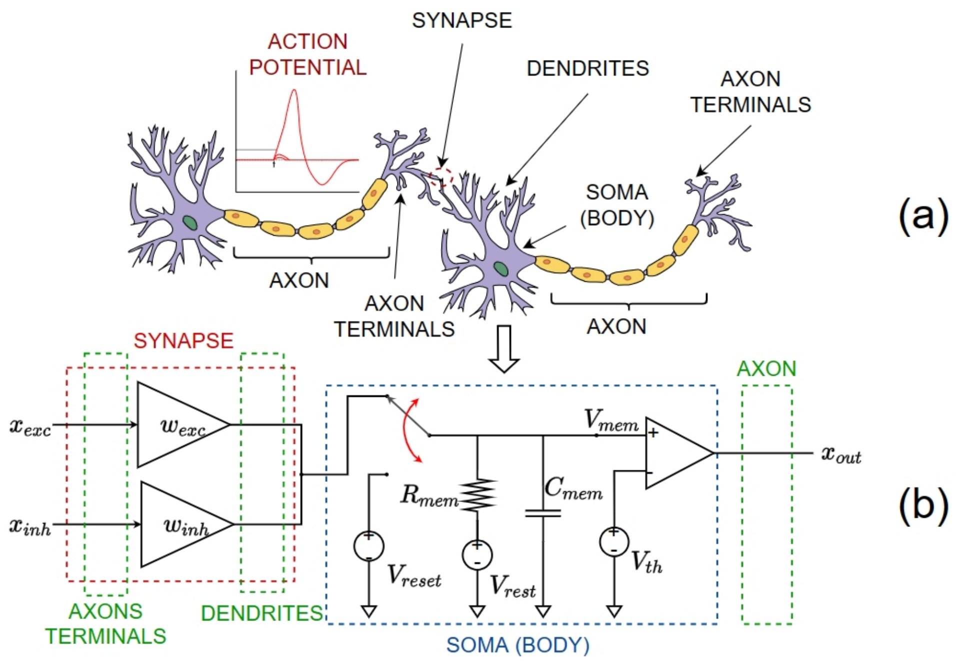


Figure 1.2: LIF Structure Schematic[8]

by a simple differential equation that accounts for the cumulative effect of input signals and the natural decay of the membrane potential. When the membrane potential accumulates enough spikes to exceed a pre-defined threshold, the model stipulates that the neuron fires a spike and immediately resets the membrane potential to a baseline level, followed by a brief refractory period during which it no longer responds to new inputs[9]. This behavior not only mimics the action potential firing mechanism of biological neurons but also reflects how neurons maintain their functionality under continuous stimulation.

Membrane potential equation: In the LIF model, the change of the neuron's membrane potential $V(t)$ with time is described by the following differential equation:

$$\tau_m \frac{dV(t)}{dt} = -[V(t) - V_{\text{rest}}] + R_m I_{\text{in}}(t) \tag{1.1}$$

where τ_m is the membrane time constant, representing the time it takes for the potential to decay to its resting value. It is the product of the membrane resistance R_m and the membrane capacitance C_m, i.e., $\tau_m = R_m \cdot C_m$. $V(t)$ is the membrane potential at time t, V_{rest} is the resting potential, the stable potential of the neuron when there is no input, R_m is the membrane resistance, indicating the resistance of the neuron to the flow of current, and $I_{\text{in}}(t)$ is the input current to the neuron at time t.

The implementation of the LIF model typically involves the solution of this differential equation, which can be accomplished through various numerical methods, such as the Euler method or the Runge-Kutta[10] method. By adjusting model parameters such as the membrane time constant τ_m, resting potential V_{rest}, threshold value $V_{\text{threshold}}$, and reset potential V_{reset}, the LIF model can simulate the behaviors of different types of neurons, providing a flexible and powerful tool for exploring neural network dynamics.

Although the LIF model is relatively simple in form, it plays a significant role in understanding neural dynamics, the information processing mechanisms of neural networks, and the field of neuromorphic computing. Through parameter adjustments, the LIF model can simulate various neuronal behaviors, including excitatory and inhibitory neurons, as well as their reactions under different physiological conditions. This flexibility makes the LIF model a powerful tool for studying how neural networks process, encode, and transmit information, especially in handling time-series data and simulating complex temporal relationships.

Moreover, the computational efficiency of the LIF model makes it particularly favored in large-scale neural network simulations and neuromorphic hardware design. By implementing the LIF model at the hardware level, researchers and engineers can develop efficient neural network processors capable of performing complex computational tasks with extremely low energy consumption, paving the way for the next generation of low-power artificial intelligence systems.

While the LIF model provides an important tool for computational neuroscience, it also has its limitations. The model's simplification overlooks some of the complex mechanisms within neurons, such as ion channel dynamics, neurotransmitter effects, and complex post-synaptic effects, which may be crucial for accurately simulating biological neural systems. Therefore, future research might explore how to integrate the LIF model with more complex neuronal and synaptic models to enhance the model's biological realism and computational capabilities.

Furthermore, with the rapid development of artificial intelligence and machine learning, how to better utilize the LIF model and its variants to design novel learning algorithms and neural network architectures is also an important research direction[9]. By delving into the potential of the LIF model in simulating biological learning processes, researchers can develop more efficient and adaptive artificial intelligence systems, pushing the boundaries of computational neuroscience and neuromorphic engineering further.

1.2 Online Learning

Online learning within SNNs exploits the unique way these networks handle data. This learning strategy allows SNNs to dynamically adjust their internal synaptic weights based on each new input signal (spike) they receive in real-time, enabling the network to adapt to new information patterns and changes in data.

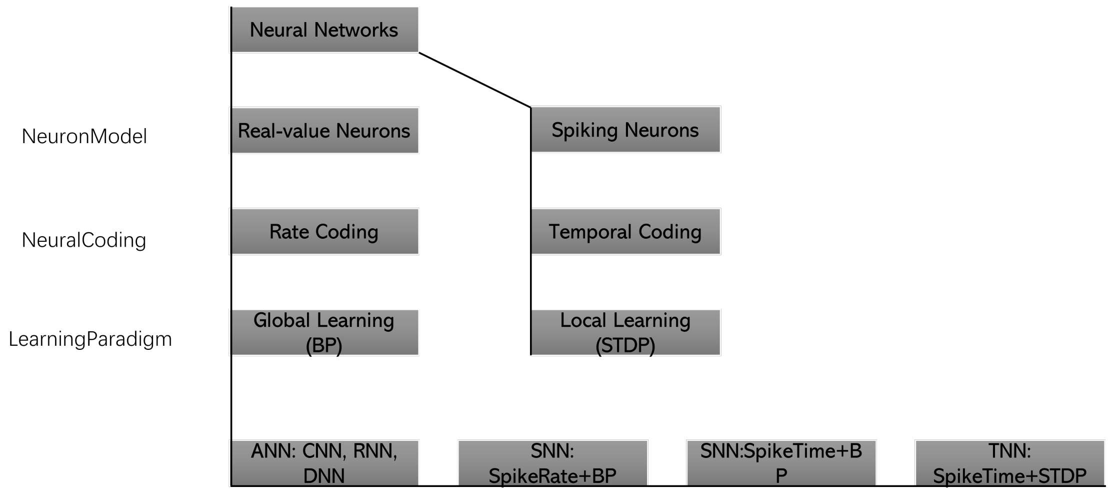


Figure 1.3: Online Learning Hierarchy Chart

In SNNs, neurons respond to input signals by firing spikes, which are transmitted to other neurons via synaptic connections. Each synapse has a weight that determines the impact of a neuron's spike on the membrane potential of the receiving neuron. Online learning involves dynamically adjusting these synaptic weights while the network is executing its task, based on the pattern of spikes received by each neuron. This weight adjustment relies on specific learning rules, such as spike-timing-dependent plasticity (STDP), which strengthen or weakens synaptic connections based on the timing differences between the spikes fired by pre- and post-synaptic neurons.

Through mechanisms like STDP, SNNs can update their internal structure to reflect the information learned from continuously received inputs. This means the network can adapt and respond to environmental changes in real-time without the need to retrain the entire model[11]. For instance, when SNNs are used for pattern recognition tasks, the network can continuously optimize its ability to recognize patterns, maintaining efficient performance even as the data stream gradually changes.

The key to online learning lies in its ability to handle and adapt to dynamically changing data streams, invaluable in many real-time applications. Particularly in SNNs, this learning method is well-suited for processing time-dependent data, such as audio signals or video frame sequences, because the network can utilize the temporal information of spikes for effective learning and decision-making.

Moreover, online learning in SNNs also reflects a simulation of the learning process in biological neural systems, where neurons learn and remember information by adjusting synaptic strengths. This mechanism not only provides insights into how the brain processes and stores information but also inspires the design of efficient, adaptive computational models.

The possible fields of application are many. Some examples are:

1. Real-time Data Processing: Online learning is ideal for applications requiring real-time data processing, such as monitoring systems, real-time transaction analysis, or decision support systems in dynamic environments. In these scenarios, the model needs to immediately react to new data inputs, adjusting its behavior to provide real-time feedback.

2. Adaptive Control Systems: In fields like robotics, autonomous vehicles, and intelligent manufacturing, systems must be able to self-adjust based on environmental changes and new sensor inputs. Online learning enables neural networks to continually learn and adapt to changes in their operating environment, enhancing system adaptability.

3. Personalized Recommendation Systems: Applied in personalized recommendation systems, online learning can update recommendations in real-time based on the latest behaviors and preferences of users, thereby improving user satisfaction and engagement.

4. Natural Language Processing: In tasks such as dialogue systems and machine translation within natural language processing, online learning allows models to adapt to new vocabulary, grammatical structures, and user-specific expressions, increasing accuracy and flexibility.

5. Financial Market Analysis: The dynamic nature of financial markets requires analysis models to quickly adapt to market changes. Online learning enables models to update forecasts and risk assessments in real-time based on the latest market data, helping investors make more timely and informed decisions.

Online learning enhances adaptability and reduces latency in systems by:

- Enabling dynamic updates with new data for real-time trend tracking and anomaly detection.

- Continuously processing data for immediate responses, unlike batch learning's delay.

It's key for adaptive systems, supporting:

- Adjustment to environmental, user behavior, or system state changes, crucial for autonomous vehicles and smart systems.
- Continuous performance optimization during operations, improving efficiency and satisfaction.

Spike-timing-dependent plasticity (STDP) is a widely studied synaptic plasticity mechanism in neuroscience, which adjusts the strength of synaptic connections based on the relative timing of spikes fired by pre- and post-synaptic neurons. This mechanism plays a key role in the learning and memory processes of the nervous system, and its biological basis and computational models have become an important research topic in the field of computational neuroscience.

1.2.1 STDP

The discovery of the STDP mechanism stems from observations of how information is processed in biological neural systems. Biological experiments have shown that the adjustment of synaptic strength depends on the relative timing of spikes from the pre- and post-synaptic neurons. This timing-dependent adjustment is supposed to be one of the basic mechanisms through which the nervous system implements learning and memory functions.

The STDP rule can be simplified as follows: if a spike from a pre-synaptic neuron closely leads to a spike from a post-synaptic neuron (i.e., leading to following), then the synaptic connection is strengthened (long-term potentiation, LTP); conversely, if a spike from the post-synaptic neuron occurs before the spike from the pre-synaptic neuron (i.e., following leads to leading), the synaptic connection is weakened (long-term depression, LTD). The method basically consists in checking the temporal correlation between input and output, and changing the synaptic connection consequently. Through this spike-timing-based adjustment mechanism, STDP enables neural networks to encode stimuli that are repeated or have a specific temporal structure, thereby learning and memorizing these stimuli.

Mathematically, the adjustment rule of STDP is often described by a function that depends on the spike timing difference. If we define $\Delta t = t_{\text{post}} - t_{\text{pre}}$, where t_{post} and t_{pre} are the spike times of the post-synaptic and pre-synaptic neurons respectively, then the change in synaptic weight Δw can be expressed as:

$$\begin{cases} \Delta w = A_+ \exp(-\Delta t/\tau_+), \ if \ \Delta t \geq 0 \\ \Delta w = -A_- \exp(\Delta t/\tau_-), \ if \ \Delta t < 0 \end{cases} \tag{1.2}$$

Here, A_+ and A_- are positive learning rate parameters, τ_+ and τ_- are time constants, which determine the speed and magnitude of synaptic weight adjustments[12].

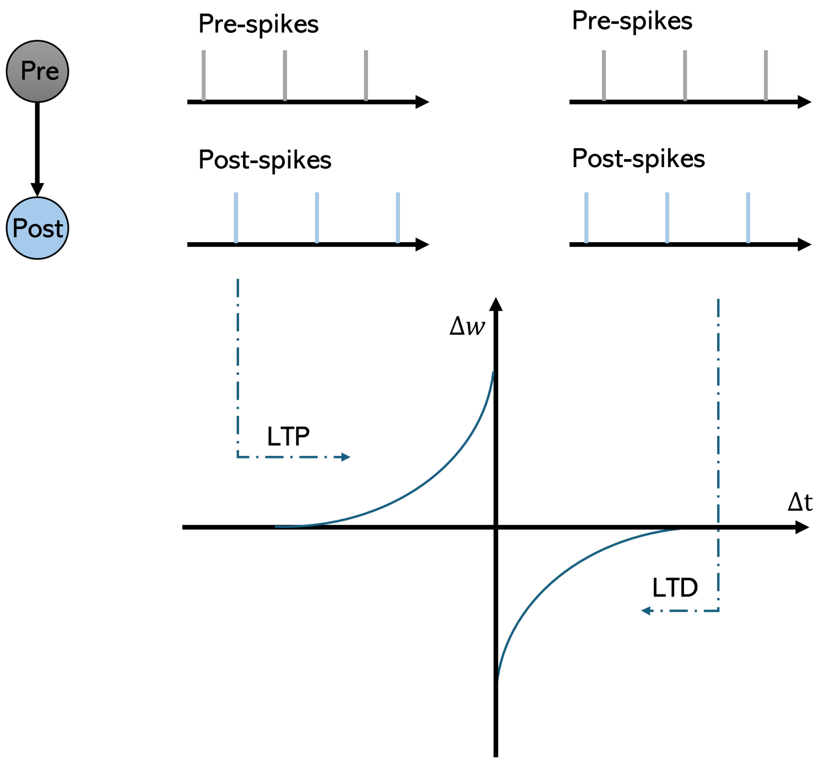


Figure 1.4: STDP Principle Schematic

This mathematical model reveals the time-dependency of STDP adjustments to synaptic weights and how this mechanism can be utilized to encode and learn temporal information.

In computational models, STDP offers an experience-based learning rule that allows neural networks to self-optimize through direct interaction with the environment. This mechanism is particularly well-suited for processing temporal data, such as audio and video, and for performing tasks that require sensitivity to temporal dynamics. By leveraging STDP, spiking neural networks (SNNs) can achieve complex pattern recognition, decision-making, and forecasting tasks without relying on external supervisory signals.

In summary, STDP as a spike-timing-based synaptic plasticity adjustment mechanism, holds profound significance in its biological foundation, working principles, and computational models. It provides an essential theoretical basis for understanding the brain's learning and memory mechanisms and for designing efficient computational neural network models[12].

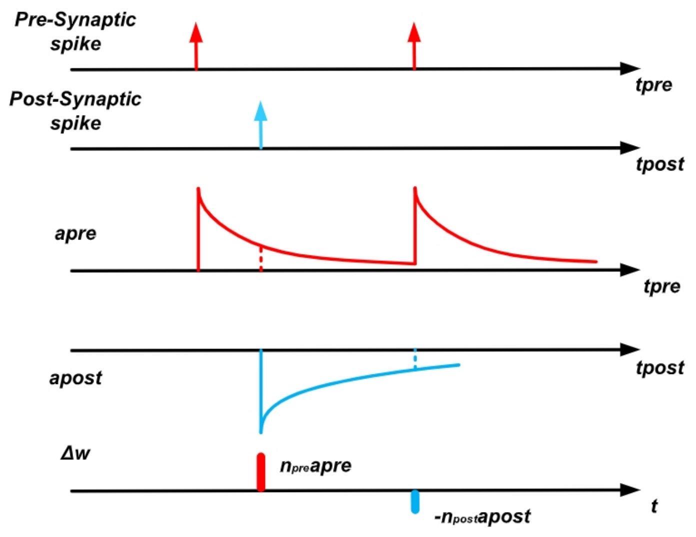


Figure 1.5: LTP/LTD Schematic

1.3 FPGA

Field-Programmable Gate Arrays (FPGAs) are highly flexible digital integrated circuits that allow developers to configure logic blocks and interconnect networks at the hardware level to perform specific logical functions. Unlike traditional integrated circuits with fixed hardware functions before leaving the factory, the programmability of FPGAs makes them an ideal choice for rapidly implementing and prototyping designs, such as custom hardware accelerators, and reconfigurable computing systems. FPGAs consist of core components such as Configurable Logic Blocks (CLBs), programmable Input/Output Blocks (IOBs), interconnect networks, built-in storage elements, and Digital Signal Processing blocks (DSPs)[13]. These components can be configured to realize various logic functions and data transmissions, supporting efficient mathematical operations and complex logic processing.

The programming process of FPGAs usually involves designing logic functions using hardware description languages (such as VHDL or Verilog), converting the design into a logic netlist through synthesis tools, followed by layout and routing (Place and Route), and finally generating a configuration file to be downloaded onto the FPGA, called the bitstream. This process endows FPGAs with programmability and flexibility, suitable for rapid development cycles and applications with high performance requirements.

FPGAs are widely applied in numerous fields such as communications, image and

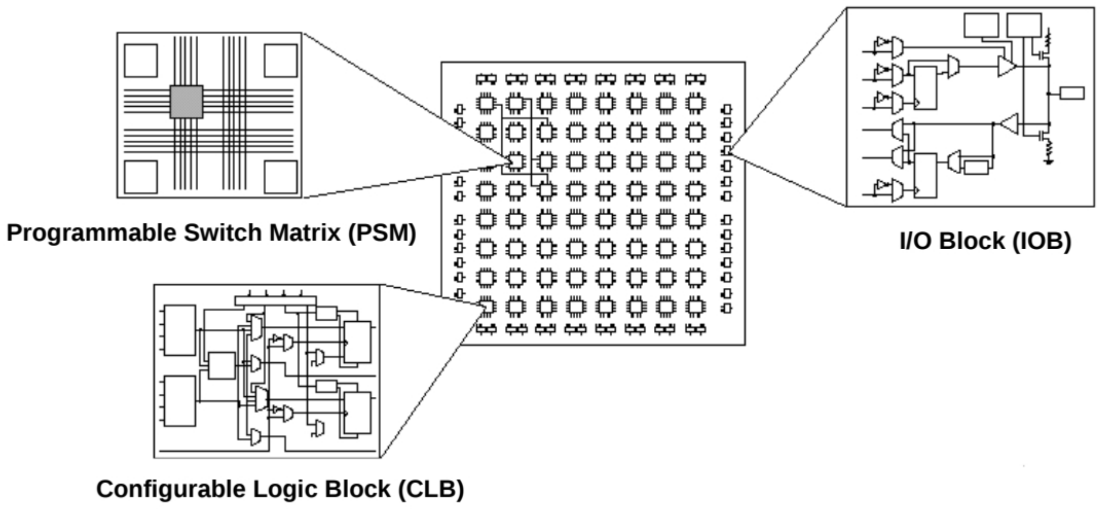

Programmable Switch Matrix (PSM)
I/O Block (IOB)
Configurable Logic Block (CLB)

Figure 1.6: FPGA internal structure[14]

video processing, data centers, embedded systems, and scientific research. Their unique flexibility and high performance provide powerful solutions to meet the rapidly changing technological needs and market requirements. With technological progress and the improvement of development tools, the application fields and importance of FPGAs are expected to continue expanding.

Field-Programmable Gate Array (FPGA) technology plays a crucial role in the acceleration and implementation of neural networks, especially Spiking Neural Networks (SNNs). FPGAs are highly flexible digital integrated circuits that allow developers to configure their logic blocks and connections at the hardware level to perform specific computational tasks. This flexibility not only makes FPGAs an ideal platform for experimentation and prototype development but also allows them to be highly optimized for specific applications, thereby providing outstanding performance and efficiency.

In neural network applications, this means that FPGAs can be customized to efficiently execute specific neural computing operations, such as weight multiplication and addition, while allowing for rapid iterations and updates to accommodate changes or optimizations in algorithms.

For Spiking Neural Networks, FPGAs provide a unique implementation platform that can utilize their parallel processing capabilities to simulate the dynamic behaviors of a large number of neurons and synapses. The operational characteristics of SNNs, such as event-based processing and sparse data communication, highly match the architecture of FPGAs. The parallel resources on FPGAs can be used to simulate the parallel activation of neurons and spike transmission, while their reconfigurability allows for the implementation of complex neural dynamics and learning rules, such as Spike-Timing-Dependent Plasticity (STDP).

Moreover, FPGAs also demonstrate significant energy efficiency advantages in implementing SNNs. Compared to running SNN simulations on CPUs or GPUs, FPGAs can reduce unnecessary computations and energy consumption through hardware optimization, making the deployment of SNNs more suitable for energy-constrained environments, such as mobile devices and edge computing nodes. This is particularly important for applications requiring real-time processing and decision-making, such as sensor data processing in autonomous vehicles or real-time video analysis[13].

However, despite the great potential of FPGAs in accelerating neural networks, especially SNNs, their application in practice also faces a series of challenges. These include the complexity of developing and debugging FPGA applications and the high demands on developers' expertise in hardware description languages and digital logic design. Nonetheless, with the emergence of development tools and higher levels of abstraction, FPGAs are becoming increasingly accessible, opening the door to their broad application in neural network acceleration and implementation.

In summary, FPGA technology provides a unique platform for acceleration and implementation in neural network research and applications, especially SNNs. Through its flexibility, efficiency, and energy advantages, FPGAs not only can drive the research progress of SNNs but also are expected to promote their widespread deployment in areas such as real-time processing and intelligent decision-making. As technology evolves and applications deepen, the role of FPGAs in future intelligent systems will become increasingly significant.

1.4 Xilinx Vitis Unified Software Platform

With the increasing demand for computing, traditional central processing units (CPUs) are struggling to meet the needs of high-performance computing (HPC) and artificial intelligence (AI) applications. As one of the solutions, hardware gas pedals such as field-programmable gate arrays (FPGAs) and adaptive computational acceleration platforms (ACAPs) have demonstrated their significant advantages in handling parallel computing tasks. However, the complexity of utilizing these advanced hardware, especially the need for a deep understanding of the hardware description language and underlying architecture, has long been a major barrier limiting their widespread adoption[15].The introduction of the Xilinx Vitis Unified Software Platform aims to address this challenge by providing a comprehensive software development environment that greatly simplifies the process of developing applications based on Xilinx hardware.

1. **Unity and Comprehensiveness of the Vitis Platform**

   The Vitis Unified Software Platform provides developers with a unified and comprehensive suite of tools, which not only reduces the difficulty of development, but also shortens the development cycle from concept to product. Compared to traditional FPGA development flows, the Vitis platform enables developers to use high-level programming languages, such as C/C++ and Python, to develop applications, thus avoiding the need to use hardware description languages (HDL) directly. In addition, the platform provides a range of libraries optimized for specific application domains such as data centers, video processing and machine learning, further improving development efficiency and application performance.

2. **The Role of Vitis High-Level Synthesis (HLS)**

   The Vitis platform is designed with an emphasis on openness and scalability, supporting a wide range of Xilinx hardware products including the Versal ACAP family, Zynq UltraScale+ MPSoCs and Alveo accelerator cards. The platform's openness is reflected in its support for third-party libraries and frameworks, enabling developers to easily integrate custom software components. In addition, the Vitis platform offers integration with modern software development tools and environments such as Docker and Jupyter notebooks, further enhancing development flexibility and efficiency.

In summary, the Xilinx Vitis Unified Software Platform greatly simplifies programming and application development for FPGAs and other Xilinx hardware by providing a comprehensive, unified set of development environments. It provides unprecedented opportunities for software developers and algorithm engineers to easily transform their ideas and algorithms into highly efficient hardware-accelerated solutions, driving widespread adoption and innovative use of FPGA technology across a wide range of industries and applications.

1.5 Xilinx Artix-7

The Xilinx Artix-7 series is a family of high-performance, low-power field-programmable gate arrays (FPGAs) from Xilinx, designed to meet the needs of cost-sensitive applications such as portable devices, low-cost wireless communications equipment, and industrial, medical and consumer electronics. The Artix-7 family offers excellent performance-to-power ratio through advanced process technology and Xilinx's proprietary architectural optimizations, which support multiple high-speed serial interface technologies, and flexible configuration and integration capabilities, making it a competitive product in the low to medium density FPGA market[14]. Key features and benefits are:

1. Performance and Power:

   The Artix-7 family utilizes a 28nm Low Power process technology, which enables a significant reduction in power consumption while maintaining the same performance as the previous generation. This makes the Artix-7 ideal for battery-powered and energy-efficient applications.

2. Rich Logic Resources:

   The Artix-7 family offers a rich set of logic units, storage resources, and digital signal processing (DSP) units, enabling designers to implement complex digital logic functions, data processing algorithms, and high-performance signal processing applications.

3. High-Speed Serial Connectivity:

   The Artix-7 family supports a wide range of high-speed serial technologies, including PCI Express® Gen2, Gigabit Ethernet, SATA, and other protocols, which can be used to enable high-speed data transfer and efficient system-level integration.

4. Flexible Configuration and Integration:

   The Artix-7 series supports configuration via SPI, JTAG, and other methods, and supports a partial reconfiguration function that allows specific logic functions in the FPGA to be dynamically modified without downtime, increasing system flexibility and scalability.

5. Comprehensive development tool support:

   Xilinx provides comprehensive development and design tool support, including the Vivado Design Suite. These tools provide full-flow support from design and simulation to debugging and deployment, greatly simplifying the development process and shortening time-to-market.

The Artix-7 FPGA family targets a variety of application areas, including but not limited to: (i) wireless communications: for digital front-end processing in wireless base stations, signal processing in mobile communications, etc.; (ii) industrial automation: for industrial networks, machine vision systems, intelligent sensors and control systems, etc.; (iii) consumer electronics: for video processing, multimedia interfaces, home networking devices, etc.; (iv) Medical devices: for medical imaging, portable diagnostic equipment and patient monitoring systems, etc.

The Xilinx Artix-7 FPGA family offers cost-effective and flexible solutions in a wide range of areas with its high performance, low power consumption, rich logic resources and flexible integration capabilities. By supporting high-speed serial

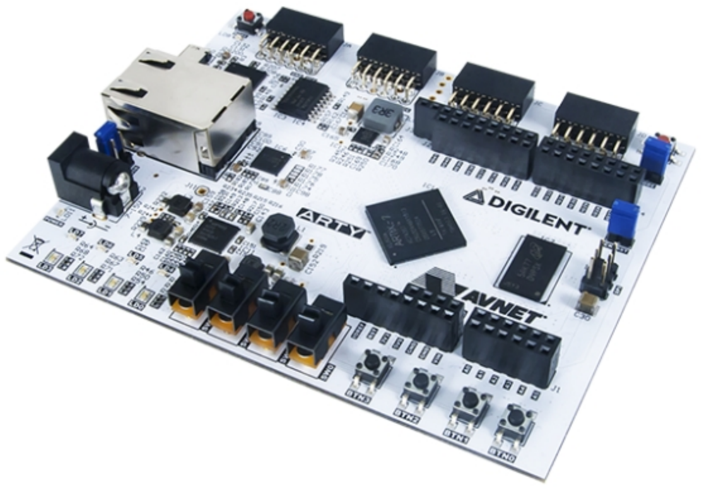


Figure 1.7: Artix-7 Development Board[14]

technology and providing comprehensive development tools, the Artix-7 offers designers the ability to realize innovative designs and shorten product development cycles, making it ideal for low to medium density FPGA applications.

1.6 MNIST

The MNIST dataset, known as Modified National Institute of Standards and Technology database, is a widely recognized and used benchmark dataset, especially in the fields of machine learning and computer vision. It contains gray-scale images of handwritten digit and is intended to provide a standard test-bed for automatic handwritten digit recognition.The MNIST dataset consists of the original NIST dataset simplified and formatted to fit the needs of modern machine learning algorithms[16].

- **The MNIST dataset** consists of 60,000 samples for training and 10,000 samples for testing. Each sample is a 28x28 pixel gray-scale image representing handwritten numbers from 0 to 9. These images are written by different people through different handwriting styles and are intended to cover a wide range of challenges that handwritten digit recognition may face. Each image is matched with a corresponding label, which is a number between 0 and 9 that represents the true value of the handwritten digit in the image.
- **Data Pre-processing**: althoughthe MNIST dataset has already undergone

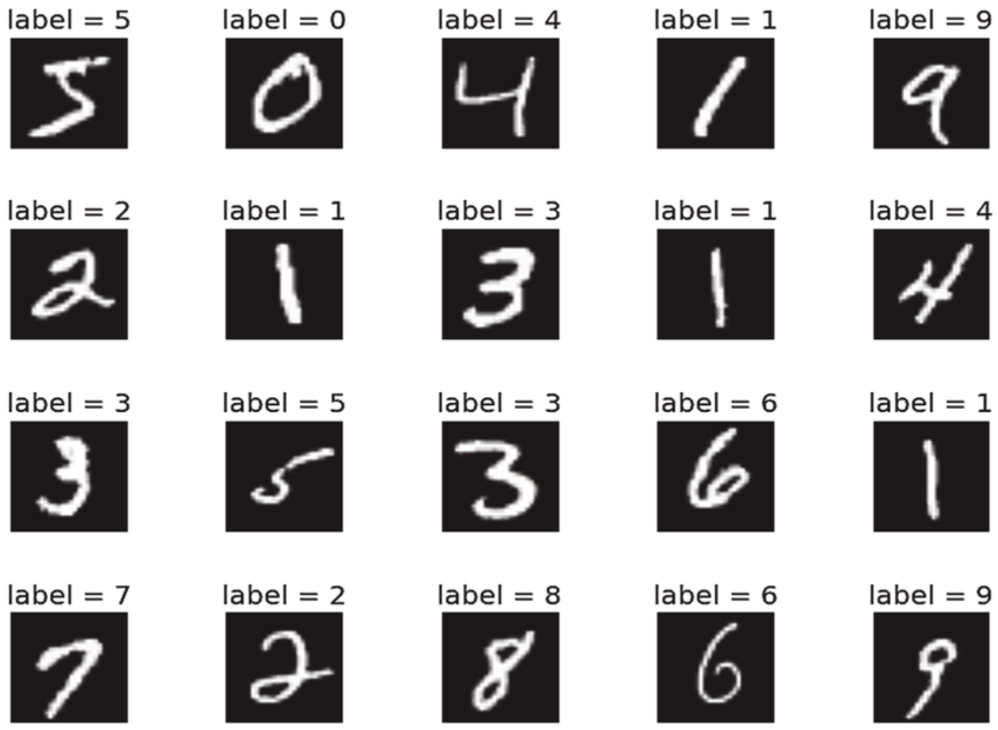


Figure 1.8: MNIST Sample Plot[16]

some degree of preprocessing, such as centrality and size normalization, in real-world machine learning projects it is common to further pre-process and enhance the data according to specific needs. This may include normalization to allow all pixel values to fall within a fixed range, or applying techniques such as rotating, panning, and scaling the image to increase the diversity of the data and improve the generalization of the model.

- **TRAINING PROCESS AND MODEL SELECTION** The goal of the training process is to develop a model that is capable to accurately recognize and predict eamples of handwritten digits that it never saw. The process begins with selecting a model architecture that is appropriate for the task. For a basic image classification task such as MNIST, everything from simple logistic regression to more complex deep learning models such as convolutional neural networks (CNNs) are viable options. With the rapid development of deep learning, convolutional neural networks are preferred for their excellent performance in image recognition tasks.

  Model training involves tuning the parameters of the model using a training dataset to minimize the prediction error. This is usually achieved through optimization algorithms such as backpropagation and gradient descent. The training process gradually improves the performance of the model on the training set, but care must also be taken to prevent overfitting, where the model performs well on the training data but poorly on unseen data.

- **Performance Evaluation** The performance of the model is evaluated by running the model on an independent test dataset. This test set contains unseen images from the training process, providing an opportunity to evaluate the model's ability to generalize. Accuracy, one of the main metrics used to evaluate the performance of the MNIST handwritten digit recognition model, calculates the percentage of images in the test set that are correctly recognized by the model. The MNIST dataset, due to its moderate complexity and ease of processing, has become not only an entry-level challenge in the field of machine learning and computer vision, but also an important benchmark for evaluating and comparing the performance of different algorithms. Despite the availability of more complex and challenging datasets in modern times, MNIST still maintains a wide range of applications and research value in academia and industry, and plays an active role in advancing the development of machine learning algorithms.

Chapter 2

Stdp Python Model

Before designing the VHDL structure for STDP, a Python model of the SNN is built in order to create a baseline with wich the VHDL block will be compared. This also allows for the validation and experimentation of the structures involved, including the implementation of the basic formulas as well as subsequent operations such as weight quantization, in a more flexible and fast environment. This approach allows to rapidly iterate and optimize the algorithms before committing to the detailed design of the hardware description language.

2.1 Python Structure

The design is based on an existing SNN acelerator designed at Politecnico, called Spiker[8]. The core design concept of Spiker is to reduce hardware complexity and improve overall performance through strategic optimization. The Spiker architecture consists of an input layer that acts as a bridge between the pulse-processing core of the network and the outside world, internal spiking layers, and an output layer to translate spikes back in real-world interpretable data. The network implements a clock-driven neuron model in which the membrane potential of a neuron is updated every clock cycle, even if no impulse occurs. At every clock cycle the accelerator checks the presence of spikes in input. If at least one spike is received it loops over all the inputs to understand which ones were active and to integrate the spikes coming from them. This means inputs are processed sequentially. To speed-up the computation, skipping useless operation a logical OR checks the input spikes in parallel: if no spike is present the scan is skipped.

As shown in Figure 2.1, three hierarchical levels can be identified, namely network, layer, and neuron, respectively. In the design of SNNs, numerical data vectors need to be converted into a sequence of spikes in order to mimic the way a biological neural system works. This conversion is crucial because it enables

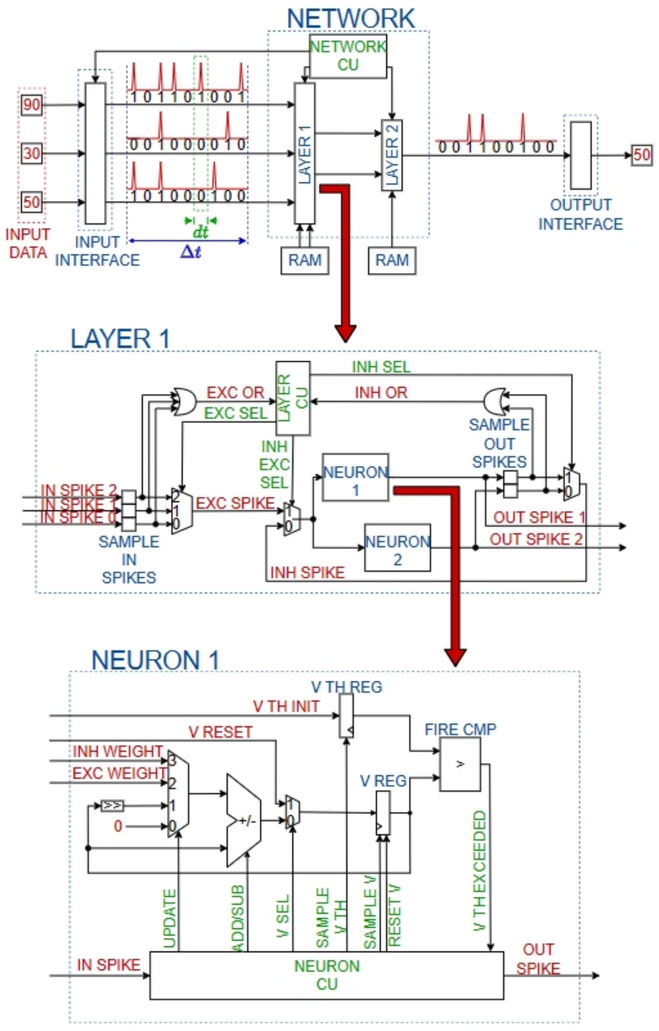


Figure 2.1: SPIKER Hierarchical Structure[8]

the network to interpret and process real-world data by translating them into spikes, which are internally utilized by spiking neurons. To minimize the use of hardware resources, these spikes are represented as single bits in the digital domain. Depending on the type of input data, there are three main types of conversion methods: (i) firing rate coding, (ii) population ranking coding, and (iii) temporal coding. Firing rate coding encodes information by averaging firing rates and is suitable for static data processing, such as images; population rank coding encodes information by the relative firing times of a group of neurons; and temporal coding relies on the precise firing time of each spike and is suitable for dynamic data. Spiker system uses firing rate coding due to its ability to efficiently process static data such as images.

- Within the Spiker system, the input data (e.g., the intensity of a pixel of an image) is processed as an instantaneous magnitude, which is then translated into a correspondent average firing rate. Each sequence of spikes is generated as a Poisson distribution of events, driven by the intensity of the corresponding pixel, which is interpreted as the average probability of a spike occurring within a given time interval. The system starts the transformation process by selecting the duration of the spike sequence and splitting the sequence into multiple computational steps capable of accommodating individual spikes. The length of each step defines the temporal resolution of the network.Interpreting the input value as the average probability of spike occurrence per unit time, the average number of spikes expected in each step is determined by multiplying the input value by the step length. To determine whether a spike should be generated in a particular time step, a random number is generated and compared to the previously obtained average spike count. A spike is generated only if the generated value exceeds the reference, thus producing a statistically independent sequence of random spikes with a temporal distribution that follows a Poisson distribution.

- The output interface converts the spike sequences generated by the network back into numerical information that can be further processed. This is achieved by equipping each output neuron with a simple counter and normalizing the counter value by the duration of the spike sequence to obtain the firing rate of the output neuron. However, since all neurons have the same duration, this operation can be omitted, thus saving resources.

- The network architecture consists of any number of layers connected in a feed-forward structure and managed by a central control unit that organizes the computation in time steps. The central control unit generates a new set of spikes when all layers are ready and initiates parallel computation for all layers. After each layer completes its computation, the central control unit is notified and waits until all layers are complete, then the next computation cycle begins.

- Each layer can contain any number of parallel update neurons, which are managed by the layer control unit. Spiker allows the implementation of inter-layer inhibitory connections, which reduce the neuron membrane potential and prevent it from firing spikes through negative weighted connections. Exploiting the sparsity of the spiking sequence, Spiker skips computations in steps without active spikes to improve performance[8].

2.2 MNIST Spikes Transformation

This section deals with the process of extracting the training images and their corresponding labels from the MNIST dataset, the specific files include:

- train-images-idx3-ubyte: this file contains the pixel point intensity data of the images.
- train-labels-idx1-ubyte: this file contains the corresponding label information of the images.

There are several specific steps performed:

1. data file extraction:
2. Image pixel intensity processing:
3. Labeling information processing:
4. Rate coding conversion:
5. Generation of input spikes file

2.2.1 Random Rate Coding

As mentioned before, Random Rate Coding is used for converting MNIST images to spikes. Unlike regular firing rate coding, random rate coding introduces randomness into the generation process of spikes, so that the specific moments of the spike sequence not only reflect the magnitude of the input data, but also contain certain random variations. This coding method is particularly suitable for those applications that require modeling the random spike firing properties in biological neural systems. The implementation of random rate coding involves the following key steps:

1. Mapping of data to firing probabilities: The input data is first converted into a value representing the firing probability. This probability value determines the likelihood that a neuron will fire spikes in a given time interval. For example, higher data values map to higher firing probabilities, while lower data values correspond to lower probabilities.
2. random number generation: At each computation step, the decision to generate spikes is made by generating a random number (usually between 0 and 1) and comparing it to the emission probability. If the random number is less than or equal to the firing probability, a spike is fired at that step.

3. Spike Sequence Generation: The above process is repeated until the encoding of the entire input data is completed or a set length of time window is reached, resulting in the generation of a sequence of random spikes representing the original input data.

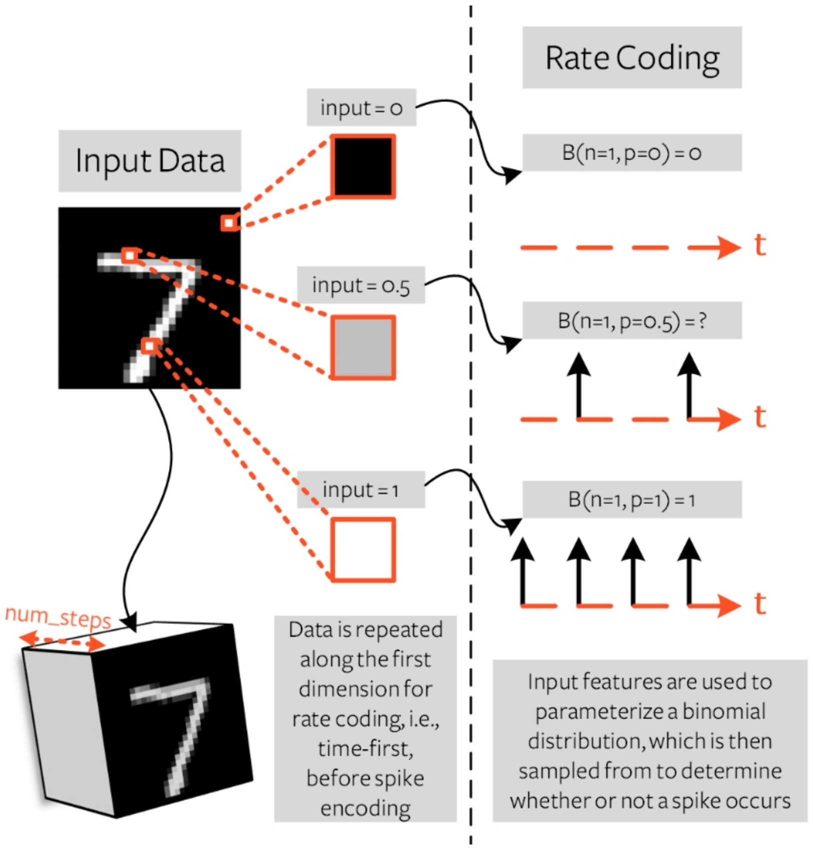

Rate Coding
Input Data
input = 0
B(n=1, p=0) = 0
t
input = 0.5
B(n=1, p=0.5) = ?
t
input = 1
B(n=1, p=1) = 1
t
num_steps
Data is repeated along the first dimension for rate coding, i.e., time-first, before spike encoding
Input features are used to parameterize a binomial distribution, which is then sampled from to determine whether or not a spike occurs

Figure 2.2: Rate-coding Correspondence

2.2.2 Sparsity

- By properly tuning the conversion parameters, the encoding can be driven to generate spike sequences with high sparsity. A sparse spike sequence means that only a few neurons are active during any given time window, while most neurons remain silent.

- Improvement in network processing efficiency: sparse spiking activity reduces the computational burden on the network in processing information. In sparse coding, a large number of background neurons do not generate spikes and therefore do not trigger subsequent post-synaptic processing and weight updating, thus saving computational resources and energy consumption.

Impact of sparsity on SNN performance

- Sparse coding helps to highlight important information features, making it easier for the network to recognize and extract key patterns in the input data. This is because in sparse coding, only the neurons that are most relevant to the task at hand are activated, which reduces redundancy of information. This approach is particularly effective when dealing with specific types of data, as it allows the network to focus on those information features that are most critical to solving the problem, although sparse coding may not be optimal for some data types.
- Enhancement of network robustness: sparsity improves robustness by limiting the network's sensitivity to input noise. In sparse coding, chance or irrelevant input changes are less likely to lead to widespread neuron activation, thus protecting the network from noise.

The combination of random rate coding and sparsity is particularly important in designing efficient and biologically sound SNNs. By adapting random rate coding strategies to optimize sparsity, efficient coding and processing of information can be achieved while improving the accuracy and energy efficiency of the network. In addition, this combination helps to achieve more compact network structures as it reduces the number of neurons and synapses that need to be actively involved in the computation, thereby reducing the complexity and resource requirements of the hardware implementation. In summary, the association between random rate coding and sparsity provides a strategy to improve the performance of SNNs while maintaining biological plausibility, making this network structure more suitable for hardware implementations, especially in resource-constrained environments.

2.3 Stdp Implementation

In realizing the learning mechanism based on spike-time dependent plasticity (STDP), the following steps were taken in this study to ensure the accuracy and biological soundness of the synaptic weight update process:

Listing 2.1: stdp function

```python
def stdp(network, layer, stdpDict, inputSpikes):

    synapseName = "exc2exc" + str(layer)
    layerName = "excLayer" + str(layer)

    # Time step increase
    network[synapseName]["time"] += 0.1

    # Increase the weights for ltp
    ltp(network, synapseName, layerName, stdpDict[synapseName]["
    eta_post"],
        stdpDict[synapseName]["ltp_tau"])

    # Increase the weights for ltd
    ltd(network, synapseName, layerName, stdpDict[synapseName]["
    eta_pre"],
        stdpDict[synapseName]["ltd_tau"], inputSpikes)

    network[synapseName]["weights"][network[synapseName]["weights"] <
     0] = 0
```

1. Layer naming and definition. First, each network layer was assigned a unique name, such as "excLayer" plus the number of the layer, in order to accurately refer to the synapses and neurons of a specific layer during the STDP update process. This step is critical for subsequent synaptic updates and other operations on specific layers.

2. For time step tracking, a variable named "time" is introduced in the dictionary definition of synapses as a counter to track the global time progress. Here a time step of 100ms (0.1s) is considered. This is increased at each time the STDP weight update is performed, thus facilitating the calculation of the time difference required for the weight update.

3. Weight update calculation, next, performs Long Term Potentiation (LTP) and Long Term Depression (LTD) calculations, which are responsible for increasing and decreasing synaptic weights, respectively, following the mechanism explained in 1.2.1. The dynamic adjustment of the weights is achieved by passing the learning rates (eta-post and eta-pre) and time constants (ltp-tau and

ltd-tau) to the corresponding functions. In particular, the LTD computation requires the identification of active synapses based on input spikes.

4. weight regularization, finally, performs a regularization operation of the weights, i.e., all negative weight values are set to 0. This operation avoids the case where the weight values become negative, preventing potential inversion of the network behavior.

Listing 2.2: LTP function

```
FUNCTION LTP(network, synapseName, layerName, eta_post, dt_tau)

    FOR EACH output spike in layerName
        SET post spike time at the output spike index = current time

    INITIALIZE condition_matrix based on the following conditions:
        - The difference between current time and last pre-spike time
    < Time-Window
        - The last pre-spike time /= 0

    FOR EACH output spike in layerName
        INCREMENT weights in synapseName at the output spike index by
    :
            IF condition_matrix = TRUE at the output spike index
              THEN
                eta_post * EXPONENTIAL of negative (current time -
    last pre-spike time) divided by dt_tau
              ELSE
                0
END FUNCTION
```

In the framework of Spike Timing Dependent Plasticity (STDP), LTP (Long Time Program Enhancement) stands for the increase in the strength of a synaptic connection when the pre-synaptic neuronissues spikes before the subsequent neuron in turn issues spikes. This code implements the synaptic weight updating process based on the LTP mechanism in the following steps:

1. Post-synaptic trace update: First, the function updates the post-synaptic traces under a specific synapse name (synapseName) in the network. For all neurons that produce output spikes in the specified layer (layerName), the function assigns the current time (network[synapseName]["time"]) to the corresponding postsynaptic trace to indicate that these neurons are activated at the current moment, and labels them with a timestamp.

2. Conditional matrix computation: Subsequently, the function constructs a conditional matrix by comparing the time difference between the pre- and

post-synaptic events for each synapse. This step aims at determining which pre- and post-synaptic events occur within a predefined time window. The additional condition "not equal to 0" is used to exclude those synapses with zero time difference in case an invalid computation is triggered.

3. Weight update: For those synapses that satisfy the condition (i.e., condition-matrix is true) and produce output spikes in the specified layer, their weights are updated according to the LTP rule. The weights are incremented by the learning rate eta-post multiplied by an exponential decay factor computed based on the time difference between the pre and postsynaptic events and the time constant dt-tau. If the condition is not satisfied, the weights are increased by 0.

Listing 2.3: LTD function

```
FUNCTION LTD(network, synapseName, layerName, eta_pre, dt_tau,
    inputSpikes)

    SET pre-spike times in synapseName for inputSpikes = current time

    CREATE a condition_matrix where:
        - The difference between current time and last post-spike
    time < Time-Window
        - The last post-spike time /= 0

    FOR EACH input spike index in inputSpikes
        DECREASE the weights in synapseName by:
            IF the condition_matrix = TRUE for a post-spike index
              THEN
                eta_pre * EXP(- (current time - last post-spike time)
     / dt_tau)
              ELSE
                0

END FUNCTION
```

This process exemplifies the basic principle of LTD in the STDP rule: if a neuron receives spikes after the subsequent one gives out spikes (i.e., the pre-synaptic activity occurs after the post-synaptic activity), the strength of the connection between these two neurons decreases. This time-based synaptic weight adjustment mechanism is crucial for modeling the forgetting process and refining the learning pattern in neural networks. The process is the same as the LTP function, but in the opposite direction, and the input spikes are transformed into spikes from the input images of MNIST for the first layer.

After the training of each image is completed, the time reference is reset to zero, to start the processing of a new image. The time values in "time" and "pre" and "post" of all synaptic Dicts are zeroed out for the next image. This is because synaptic triggers between different images do not directly affect each other's STDP weight adjustment calculation. Each input (e.g., image) is processed independently, and STDP adjusts the synaptic weights based on the timing of pre- and post-synaptic neuron spikes in this input. Doing so allows the neural network to learn and adapt based on the temporal dynamics of each independent event, which in turn enables the accumulation of knowledge and experience in long-term memory.

MNIST training using learning rates and time constants as shown in Table 2.1.

Parameter	Description	Value
eta_pre	Pre-synaptic learning rate	1e-4 mV
eta_post	Post-synaptic learning rate	1e-3 mV
ltp_tau	Long-Term Potentiation time constant	20 ms
ltd_tau	Long-Term Depression time constant	20 ms

Table 2.1: STDP Parameters

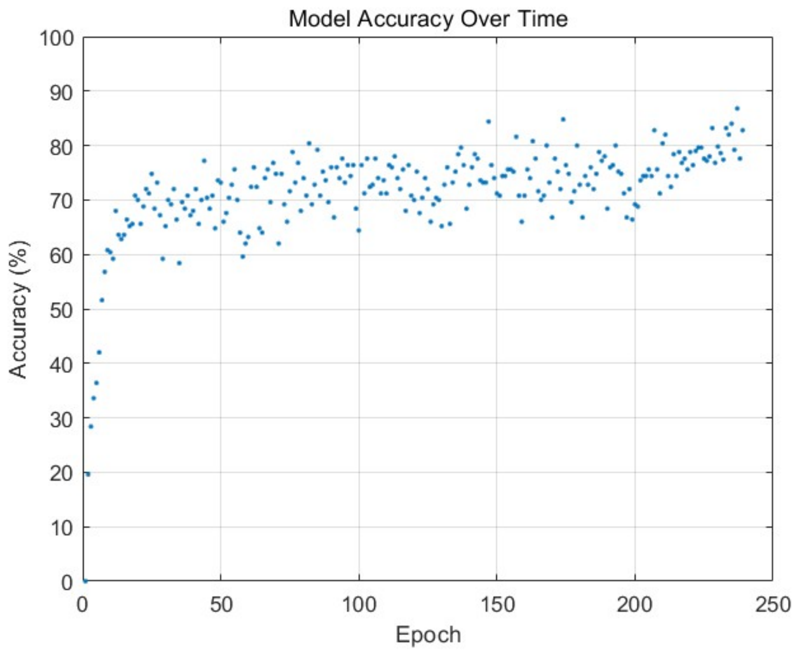


Figure 2.3: Full-precition accuracy scatterplot

The evolution of the network throughout the STDP-based training process is shown in figure 2.3.

- the accuracy is about 70% in the first 50 epochs at the beginning of the training period.

- fluctuating growth in the middle period
- fluctuates between 75%-85% in later stages

Being an unsupervised method it can be seen that the output accuracy is quite noisy. However the network is able to learn quite well

2.4 Quantization

Weight quantization becomes a critical step when the STDP (Spike Timing Dependent Plasticity) model is transferred from a Python simulation environment to a VHDL (Hardware Description Language) implementation. This is because, in contrast to high-level programming languages such as Python, hardware implementations need to take into account storage and processing power limitations.

Purpose of weight quantization

- Reduce storage requirements: quantization reduces the demand on storage resources by reducing the number of bits required for weight values. This is especially important for resource-constrained hardware platforms.
- Simplify computation: Quantization of weights simplifies multiplication operations because using fewer bits allows for simpler hardware logic, which reduces computational complexity and energy consumption.
- Adapting to hardware limitations: In some hardware (e.g., analog cross-arrays or specific storage computing devices), there are limited levels of conductivity available. Quantization ensures that weights can be mapped to these available conductivity levels.

Challenges of weight quantization

- Loss of accuracy: Rounding or truncation during quantization may result in a loss of model accuracy. Designing a quantization strategy requires a careful weighing of the relationship between quantization levels and model performance.
- Quantization noise: The error introduced by quantization can be regarded as a kind of noise, which may affect the learning and generalization ability of the network. How to minimize the impact of quantization noise is a key consideration in the design.

Bit Width	Accuracy (%)
10	73.96
7	73.76
5	73.68
4	71.90
3	70.01
2	27.75
1	12.17

Table 2.2: Quantized weight accuracy

If quantify only the weights, which are then used for inference, as shown in the table 2.2.

As can be seen from the data in the table, when the weight quantization reaches 5 bits or more, the change in the accuracy rate obtained by inference is small, with the change limited to the two-digit decimal range. In contrast, when the weight quantization is less than 5 bits, the decrease in the accuracy is significant. In particular, when the weight quantization is reduced to less than 3 bits, the accuracy rate drops drastically, falling within the 30% to 10% range.

But in this design, the delta weights are used as the smallest weight unit because the transmission of the delta weights needs to be considered. Since the learning rates eta_pre and eta_post used previously are 1e-4 and 1e-3 respectively, here we only perform strong quantization for the delta weights so that they act in the training stdp, The quantization of the weights is then determined as the strength of the delta weight quantization is relative to the learning rate.The quantized value of the corresponding weight is the quantized value of the delta weight plus 10. And the accuracy obtained is recorded and the final graph is obtained in figure 2.4.

- Quantization strategy: we implemented different degrees of quantization for delta weights, specifically including 5-bit, 3-bit and 1-bit quantization levels. The model was trained using the MNIST training set, and the model underwent a training process of 35,000 images at each quantization level.The reason for not using full training is that by the time 40,000 training volumes are reached, the inputIntensity decreases so drastically that the amount of spikes on the output is not reached. In order to monitor the performance changes during the training process, we recorded the accuracy of the model every 250 images.
- Observations in the early stages of training: In the early stages of training, we observed a certain degree of deviation in the accuracy of the model when the quantization intensity was high. This suggests that strong quantization may adversely affect the learning effectiveness of the model in the early stages of training.

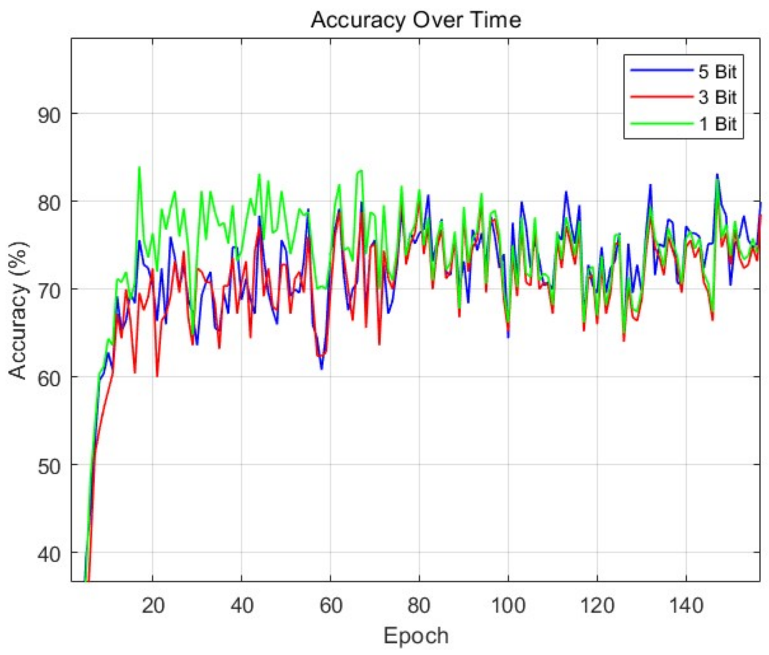


Figure 2.4: Line graph of quantitative results

- Long-term training effect: Despite the bias in the early stages of training, the final accuracy obtained by the models at all quantization levels tends to be close as training continues. This finding suggests that although quantization intensity has an effect on performance in the early stages of training, long-term training can mitigate this effect and the models are able to achieve similar levels of accuracy

Chapter 3

Stdp Vhdl Establish

In the previous section, we verified that the STDP model written in the Python language is feasible for application in SNN networks and achieves high accuracy for a specific learning rate and time constant. The model is equally effective under certain quantization constraints.

In this section, a module implementing the STDP functionality will be built using the VHDL language, aiming to replicate the functionality of the Python version of the STDP module. We separate out the STDP portion of Python and compare the results of the VHDL implementation with it to ensure that both have the same effect.

This section details a module designed for embedding spiking neural networks (SNNs). The main function of this module is to receive the spiking signals generated at both ends of the neuron layer of the SNN, and after a series of internal processing calculations, it finally realizes the exchange of information with the synaptic layer of the SNN and thus adjusts the synaptic weights. This process aims to achieve the learning effect based on the spatio-temporal dynamic remodeling (STDP) mechanism.

This document will introduce the overall structure of the module, including the whole process from receiving input signals to generating output signals. Subsequently, the functions and operation mechanisms of each sub-module will be described in detail one by one. Finally, the comprehensive results of the module are shown to demonstrate its effectiveness and practicality.

3.1 Stdp Structure

In this design, the STDP (Spike-Timing-Dependent Plasticity) architecture is subdivided into four key components to ensure efficient and accurate function realization:

- Encoder
- Queue
- Weight-Trans
- Weight-Bram

The overall structure adopts a left-right symmetric design concept, with the exception of the Weight-Bram, where other parts such as Encoder, Queue and Weight-trans are connected with double devices.

1. The entire STDP architecture is constructed between two layers of the neural network, through which impulses (spikes) from the Input and Output layers are used as entrances to the data stream. When the data stream enters a specific port, it is first processed by the encoder, which outputs the corresponding time value and the address value of the neuron that triggered the spike. Given that this design employs a two-layer neural network structure, where the first layer size is 784 (corresponding to the spike input to MNIST), and the second layer is 400, the inputs and outputs are processed separately for the address-indicating locations corresponding to their respective numbers of neurons.

2. Subsequently, the addresses and timestamps of the neurons are fed in parallel to the Queue at the home end and the opposite end, as shown in Figure Figure 3.1 At the home end they are input as Event-addr and time-attach and at the other end as Event-addr-oppo and time-attach-oppo.At the same time, the event validity indication signal (valid signal) from each encoder is also connected to the Queue at both ends, which serves as an indication signal for entering the queue (Inqueue) at this end and as a start signal for reading the queue information at the other end.

3. Next, the address and time values corresponding to each spike are extracted from the Queue, and these values are fed into the Weight-trans module for processing, where they are computed as addr1 (row-selected address), addr2 (column-selected address), the updated difference in the weights (Delta-Weight), and the resulting valid indication signal (indicate). In addition, since the signal inputs to the front-end and the back-end are in opposite directions, i.e., the corresponding processing methods in the STDP are LTP and LTD, respectively, and the Weight-Trans module is used as a general-purpose type, as shown in Figure 3.2, the row-address signals and column-address signals that are output from the Queue module to the Weight-Trans module are reversed when the ports are connected, ensuring that they both correspond to the same row-address signals and column-address signals. connections to ensure that they both correspond to address values in the same direction layer.

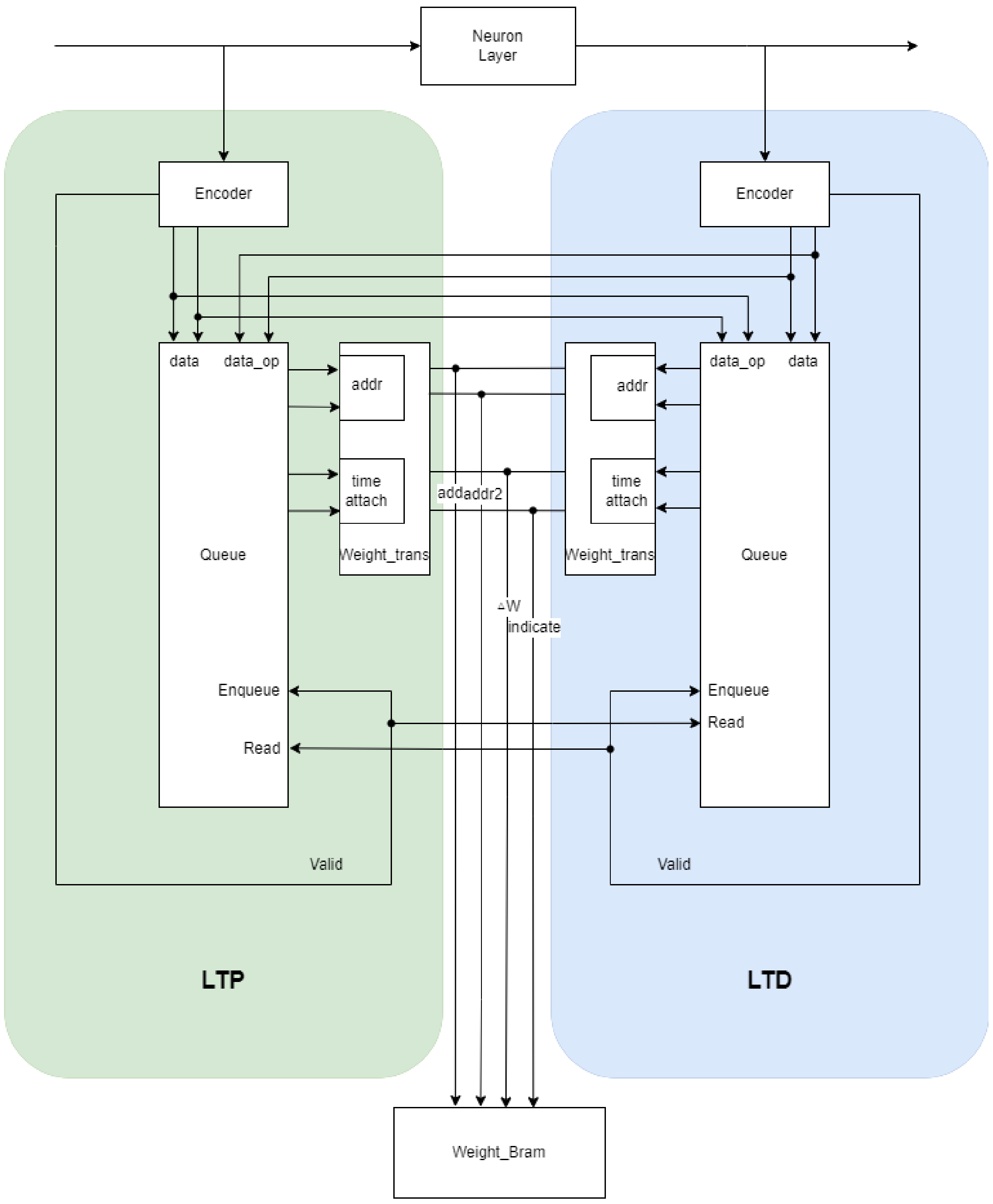


Figure 3.1: STDP overall module structure

4. Finally, the computed address values and weight update values are input into the Weight-Bram module for processing, thus updating the weights stored in

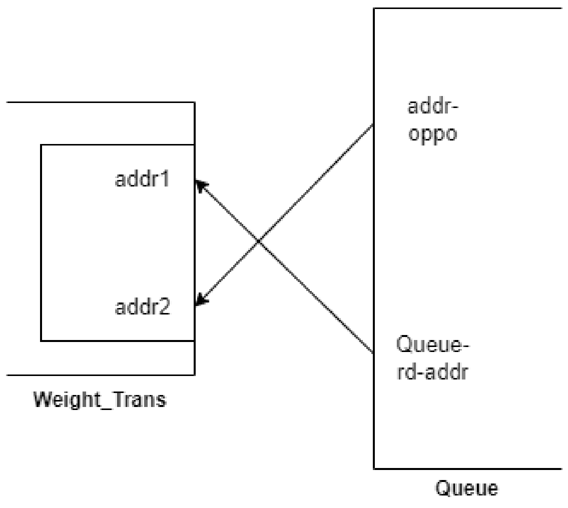


Figure 3.2: LTD address reverse connection

the original synapsis. Through this series of operations, the STDP architecture realizes the weight adjustment based on temporal dynamics, which enables the neural network to adaptively learn and update according to the activity patterns of the input and output layers, thus enhancing the flexibility and efficiency of the neural network's processing and learning capabilities.

3.2 Encoder

In the encoder section, we mainly deal with the spike signals from both input and output ends. The core of the task is to extract the information critical for STDP conversion from these spikes and encode this information to the next level of the system. Since we use spiking neural networks (SNN), a distinctive feature of this type of network is the sparsity of spikes.

In the address processing part of this design, a Time Division Multiplexing (TDM) strategy is adopted to efficiently process and transmit sparse spike signals from the SNN network. Time multiplexing enables efficient encoding and transmission of spikes by dividing time into discrete time slots, assigning each input signal a dedicated time slot. During each given time slot, a specific input channel is activated and its data has exclusive access to the communication line for transmission. For example, in the first time slot, the data of 'input1' is transmitted; then in the second time slot, it switches to 'input2', and so on. This approach ensures that no signal collisions or interference can occur on the communication lines, since each time slot is occupied by only a single input signal.

The implementation of time multiplexing relies on a clock signal (clk) to control the switching of time slots, ensuring that each input signal is accurately transmitted within its designated time slot. This strategy reduces the number of physical connections required. This is particularly important when simulating large-scale connections between units in SNN networks, since a distinctive feature of SNNs is the sparsity of spikes.

However, although the time multiplexing strategy is efficient in terms of physical connections, it also brings additional requirements for time resources. Since each input signal needs to be processed sequentially in a different time slot, this can lead to increased overall latency when processing a large number of input signals. Therefore, the design needs to carefully balance time efficiency and connection resource utilization to optimize system performance. In practical applications, the relationship between transmission efficiency and system response time can be balanced by adjusting the length and allocation strategy of the time slot, as well as optimizing the clock frequency, to ensure an SNN network implementation that is both efficient and can meet real-time processing requirements.

3.3 Queue

The implementation of the queue module consists of the following:

- The building of the module
- Use of memory such as BRAM
- Implementation of in-queue and out-queue functions
- Synchronization of timing
- Storage format of the data

In terms of composing the queue module, the design targets the subsequent symmetrical usage

3.3.1 Queue Module

In the Queue module of this design, we employ signal management and control strategy to ensure that the module can flexibly cope with the dynamic spiking activities in the SNN. The main workflow of the module is shown in Figure 3.3:

1. Spike reception and storage: when a valid spike is detected (Event-Valid signal activation), the module utilizes the address (Event-Address) and timestamp (time-attach) information of the spike to add the spike data into the queue to prepare for subsequent processing.

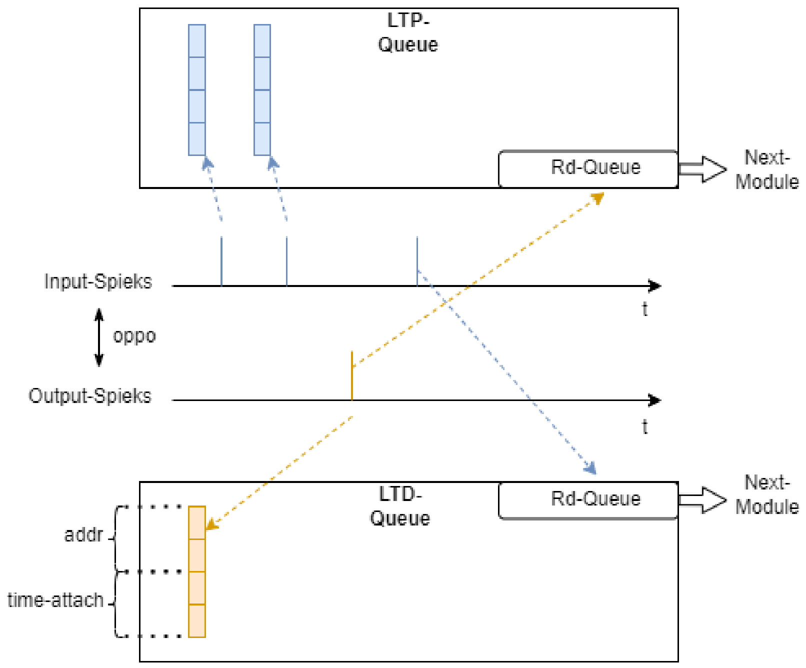


Figure 3.3: Time-triggered on both sides

2. Spike data reading: In the face of a spike data reading request from the opposite end (marked by the Event-Valid-Oppo signal), the module retrieves and provides the required spike data precisely from the queue by using the spike address (Event-Address-Oppo) and timestamp (time-attach-oppo) information provided by the opposite end. This operation corresponds to the Rd-Queue in figure.

The design not only optimizes the processing efficiency of spike information, but also enhances the system's responsiveness to spike activities and guarantees the accuracy and real-time nature of information exchange.

In addition, the queue design also takes into account data management and access strategies to ensure that data processing is both fast and accurate in an environment where spikes information is exchanged frequently. By optimizing the data structure and access method, this queue module can support high-concurrency spike processing, providing powerful underlying support for the implementation of STDP learning rules in SNN. In addition, this design also provides flexibility for possible future expansion, such as adding more ports to support more complex network topologies, or optimizing timestamp processing to improve the accuracy and efficiency of timing analysis.

Depending on the function to be performed, the core objective of the queue module is to efficiently manage the dynamic flow of spiking information in a

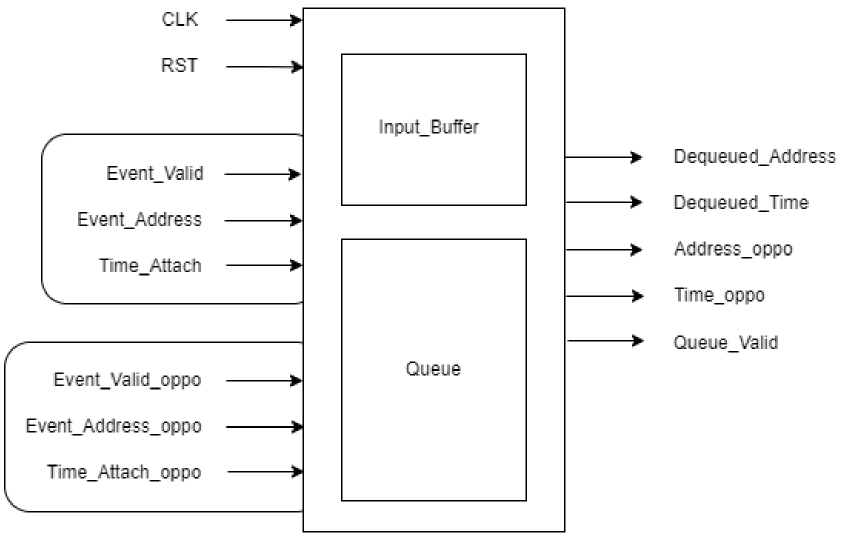


Figure 3.4: Queue Module Port Diagram

pulsed neural network (SNN) environment. The module is specifically designed to handle and coordinate spiking data from both ends of the network, ensuring that information is exchanged and processed accurately and without error in the dynamic SNN environment. This module not only responds to the underlying clock (clk) and reset (rst) signals, but also introduces a series of specialized ports designed to finely tune the reception and management of spike data, as well as interaction with other components in the network.

To introduce how this module handles spike data, it can be divided into two main categories:

For the spike information received by the local end, we have the following ports:

- **Event-Valid:** As a control signal, indicating whether there are currently valid spike events that need to be processed. When Event-Valid is high, it indicates that the queue needs to enqueue the current spike.

- **Event-Address:** carries the neuron address information corresponding to the currently processed spike event. This is key data for locating the source of the spike.

- **Time-Attach:** records the timestamp of the peak event, providing necessary timing information for STDP learning rules.

For a spike message received from the opposite port, the corresponding port includes:

- **Event-Valid-Oppo:**Indicates the valid status of a spike event received from the opposite port. When this signal is high, it indicates that the opposite end has spiked data requesting to read the information stored in the queue on this end.
- **Event-Address-Oppo:** carries neuron address information of the peer spike event.
- **Time-Attach-oppo:** records the timestamp of the peer peak event, providing data support for cross-module timing analysis.

3.3.2 Queue Structure

The queue management system of this design employs a streamlined state control logic to handle and synchronize spiking events in the SNN. , this process is shown in Figure 3.5 and is realized through the following key operations:

1. Data reception and queue entry operations:
    - When the system is in IDLE state, it waits for new spiking events.
    - Once the Event-Valid signal is active, indicating that a spike event has arrived, the system adds it to the queue.
    - If the queue is full, the system takes steps (e.g., looping the queue pointer) to continue storing new events to ensure data continuity.
2. Data reading and evaluation:
    - When the system enters the OUTPUT-DATA state, it is responsible for retrieving spike events from the queue in response to the Event-oppo signal.
    - Each spike event is evaluated based on its timestamp to determine if it falls within the time window defined by the STDP learning rules.
    - Spike events are considered valid inputs to the learning process only if the difference between their timestamps and the current Event-oppo timestamp (time-attach-oppo) is within a predefined threshold.
3. Dynamic management of time window:
    - The time window mechanism is crucial in the OUTPUT-DATA state, which determines which events will be further processed by the system.
    - If the timestamp of a spiking event exceeds the time window threshold, this indicates that the event is no longer influential for the current learning cycle, and the system therefore queues the event out.

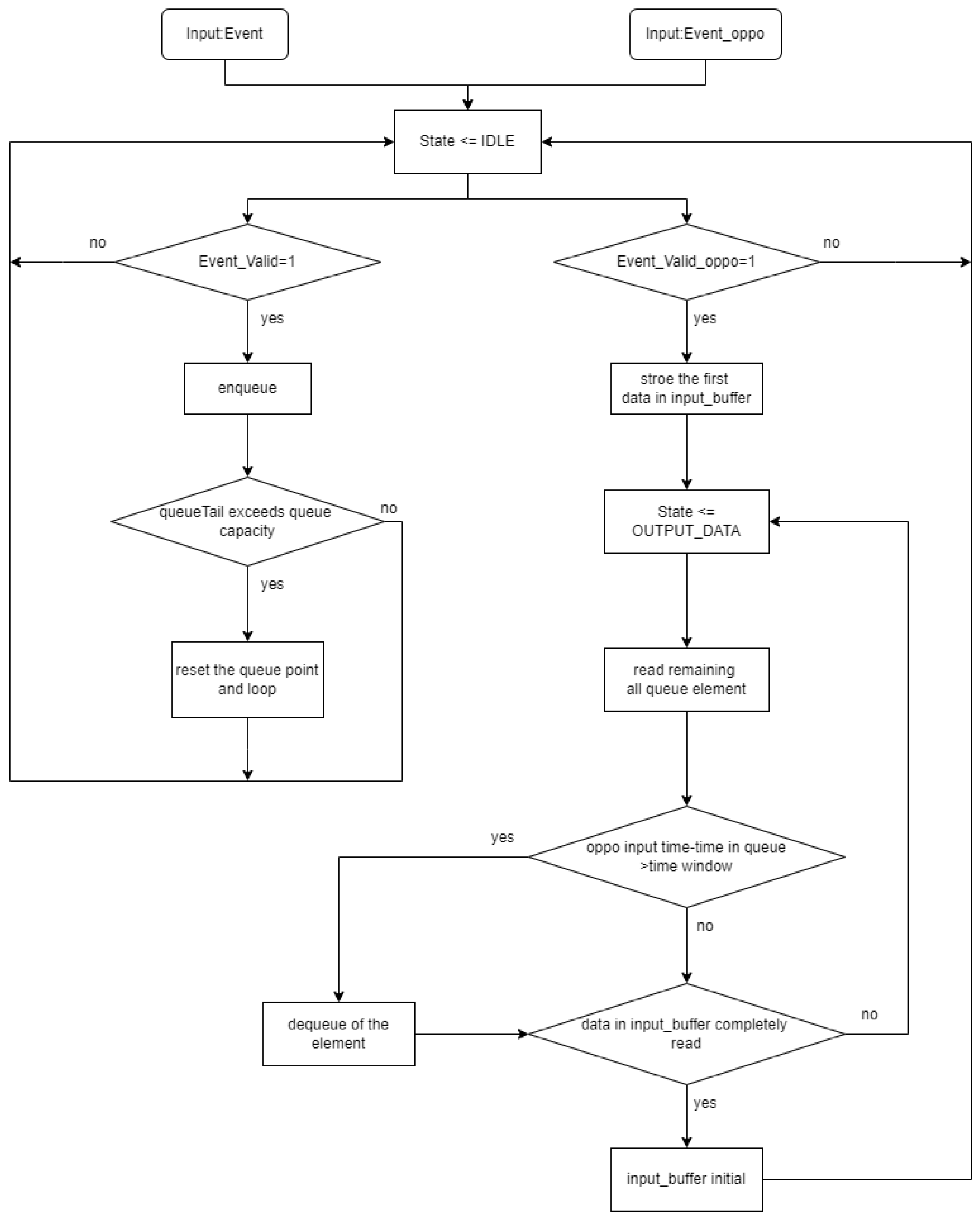

Input:Event
Input:Event_oppo
State <= IDLE
Event_Valid=1
no
yes
Event_Valid_oppo=1
no
yes
enqueue
stroe the first data in input_buffer
queueTail exceeds queue capacity
no
yes
State <= OUTPUT_DATA
reset the queue point and loop
read remaining all queue element
oppo input time-time in queue >time window
yes
no
dequeue of the element
data in input_buffer completely read
no
yes
input_buffer initial

Figure 3.5: Queue module flowchart

4. system reset and preparation:

- After processing all spike events in the queue, the system empties the input-buffer and resets the state machine for receiving the next batch of spike events.
- This ensures that the queue management system is always in an optimal state, ready to respond quickly to new spike events.

Through these steps, the queue management system of this design precisely controls every aspect of the data flow, ensuring that the time window requirements of the STDP mechanism are met. Such treatment not only ensures that only relevant neural events affect the learning process, but also improves the overall network model's responsiveness and learning efficiency.

3.3.3 Queue Storage

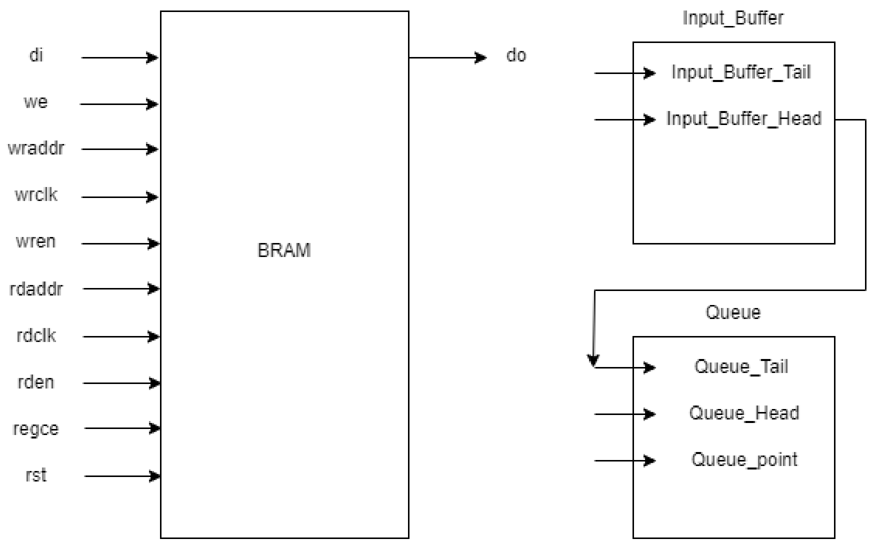


Figure 3.6: BRAM utilization

In modern FPGA (Field Programmable Gate Array) designs, BRAM (Block Random Access Memory) is used as a core component to provide a high-speed and flexible internal storage solution. It features large capacity and low access latency, and is ideally suited for implementing intermediate data storage, buffers, lookup tables, state machines, etc. BRAM's design flexibility and efficiency is reflected in the fact that it can be configured according to the application's needs in terms of its size, bit-width, and mode, and supports multiple configurations, such as single-port RAM, dual-port RAM, or FIFO queuing, and can be configured in cascade or in parallel with multiple BRAM blocks to expand storage capacity or data path width. Utilizing BRAM in FPGA design can significantly improve data processing speed and overall system performance, while reducing dependence on external storage, lowering system power consumption and cost, and enhancing design flexibility and scalability.

In this design, we use the Block RAM (BRAM) of Xilinx FPGA as the key data storage and buffering mechanism to support the implementation of Spike-Timing-Dependent Plasticity (STDP) in the SNN. Through carefully designed configuration, the BRAM unit is integrated into two specialized modules: BRAM-Input-buffer and BRAM-QUEUE. They respectively play different roles in the STDP processing flow, optimizing the data access efficiency and the overall system performance.

- **BRAM-Input-buffer module** Added before the Queue's processing. Responsible for storing the spike signals received by the SNN, in order to solve the problem of mismatched data transmission and processing times and data loss.

- **BRAM-QUEUE module** Manages queues of spiking information to adapt to the randomness of spiking events in SNN. Relying on the capacity and flexibility of BRAM.

Configuration of these two BRAM modules includes selecting the appropriate BRAM size ("36Kb"), optimizing for a specific FPGA device series (such as "7SERIES"), and accurately setting the read and write width to match the structure of spike data. In addition, by adjusting the output register configuration, collision detection mechanism and initial value settings, the accuracy of data processing and the stable operation of the system are further ensured.

When designing these two modules, special attention was paid to the optimization of BRAM configuration to ensure that it can not only meet the SNN processing needs, but also maximize the use of FPGA resources. This BRAM-based data management strategy not only significantly improves the computing and storage efficiency of the SNN model, but also provides powerful hardware support for implementing complex neural network algorithms. Through this design, we successfully combined the theoretical principles of the STDP algorithm with the hardware characteristics of FPGA, demonstrating the possibility of efficiently implementing the SNN model.

As shown in the figure the method involves storing a combination of address and corresponding timestamp as a data element. In our design, the timestamp occupies the lower 5 bits of the combined value, while the upper 10 bits are assigned to the address value. This design decision is based on the consideration of the maximum number of neurons in the neural network layer: the given two-layer SNN is 784 and 400 size respectively. because $2^{10} = 1024$, the upper 10 bits of the address space are enough to cover all necessary address values, ensuring the adequacy and efficiency of address encoding.

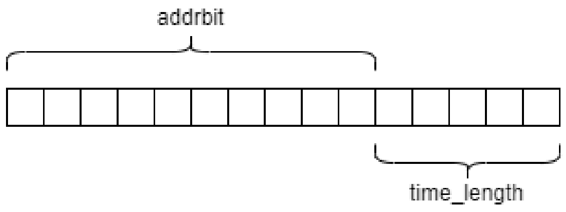


Figure 3.7: Data storage arrangements

When data is written to BRAM, this address-timestamp combination value is input through the data input port (DI). When managing the write address (WRADDR), we adopt a simple and effective method: use the BRAM pointer as the address. Specifically, when writing address data to the BRAM of the input-buffer, we use Input-Buffer-Tail and Input-Buffer-Head as control pointers. These two pointers support input event data (event-oppo value) simple queue operations - that is, enqueuing and dequeuing. Whenever the system returns to the IDLE state and completes a single operation, the input-buffer is initialized to prepare for the next round of data processing. Due to the efficiency of this method, the capacity of the input-buffer far exceeds actual usage requirements, so we do not need to deal with queue overflow.

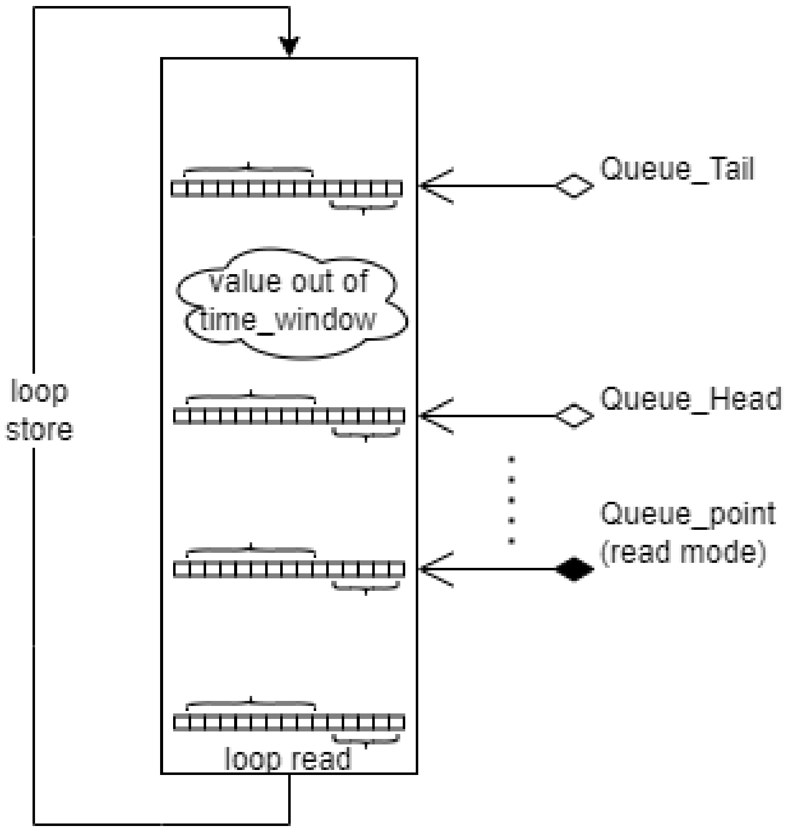


Figure 3.8: Schematic diagram of queue use

The diagram represents the queue management strategy in the SNN model, focusing on three main pointers: Queue-Tail, Queue-Head, and Queue-Point. These pointers facilitate dynamic data storage and retrieval to ensure stable data flow and system functionality. Here's an analysis along with the functionality of the time-window:

- Queue-Tail marks the end of the queue and moves backward as new data elements are enqueued. It ensures that incoming data are continuously added, maintaining the flow's real-time nature. If Queue-Tail reaches the end, the loop store mechanism kicks in, and Queue-Tail wraps around to the queue's start, enabling circular data storage.

- Queue-Head is the pointer that leads the queue, indicating where data will be dequeued next. It moves sequentially through the queue, ensuring that data are processed in their arrival order, which is critical for maintaining temporal coherence.

- Queue-Point is in read mode and is used during the data output phase. It can traverse the queue independently of Queue-Head, allowing for efficient data access without disrupting the enqueuing process.
- The time-window feature ensures that only data within a certain temporal threshold are considered for processing, which prevents outdated data from affecting current computations. This feature is represented by the shaded areas marked "value out of time-window," indicating that data outside this time frame are ignored during dequeue operations.

The loop read and loop store notations signify that the queue is circular, and both reading from and writing to the queue can continue indefinitely without the risk of overflow or underflow, as long as data management is properly synchronized.

This system provides a robust method for managing data within an SNN, enabling the network to handle spikes efficiently and in an ordered fashion. The time-window mechanism further enhances the model by filtering out irrelevant data, thereby optimizing the processing and increasing the reliability of the network's response to stimuli.

3.3.4 Time Synchronization

In the queue management system, by introducing the Queue-Valid signal as an indicator of output validity, the system can accurately control and manage the data flow. As shown in the figure, for each received Event-oppo signal (values 1, 7, 6 in the example), the system will accordingly extract the current remaining address value (e.g. 3, 7, 6, 2) from the queue and its The corresponding timestamp (eg 1, 2, 3, 3). These extracted values are then passed to the next processing level via the queue's output port for further manipulation.

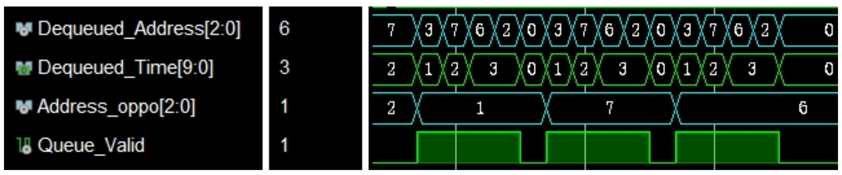


Figure 3.9: Read data timing correspondence

3.4 Weight-Trans

The design converts the input temporal information into weight update values (ΔW) by means of pulse time difference dependent plasticity (STDP) computation, and its core features and functions include:

- STDP-based weight update: the design utilizes the STDP principle to calculate the weight update value (ΔW) based on the time difference between the input pulses
- Use of Event-Valid-oppo port: is the key signal that controls the start of the weight update calculation.
- Synchronized output of address information and weight update values: Ensures that the processed address information and one of the corresponding weight update values can be output synchronously. This is critical to ensure that the data flow is at least carried out in subsequent processing steps.
- Guarantee of real-time and accuracy: This design guarantees the real-time and accuracy of the weight adjustment process by ensuring the synchronization of data and the detection of event effects during the weight adjustment process, which meets the demand for fast and accurate processing.

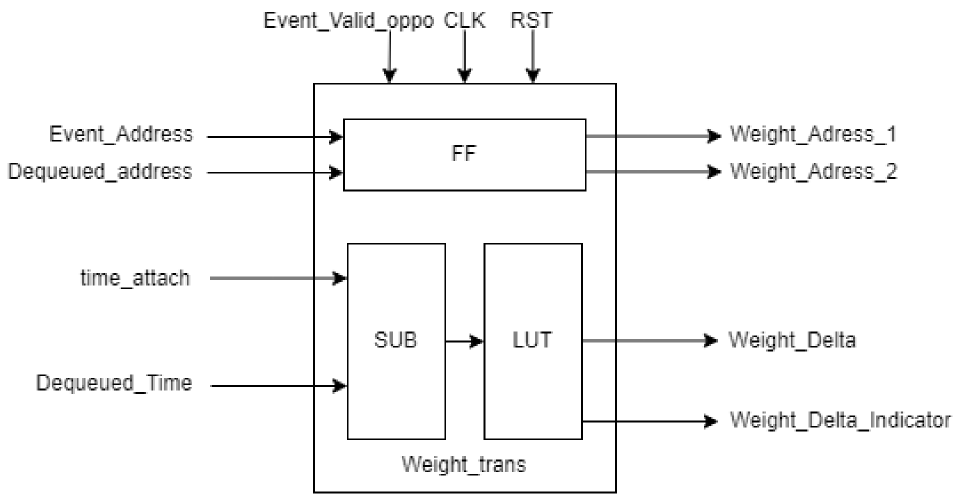


Figure 3.10: Weight-Trans Module Diagram

In the STDP conversion section, the time computation involves the time-attach from the front-end for this port input and the time value read from the queue. First, a subtractor is used to calculate the difference between these two input time values to get the time difference Δt. For this VHDL design, we temporarily use a lookup table (LUT) to map the corresponding $\Delta weights$ values.

We have already discussed the specific effects of different delta weight values in the previous section on python modeling, so we can accordingly use MATLAB to generate the corresponding lookup tables based on the underlying STDP formulae and taking into account the width of the time window. From this we can get the corresponding LUT values directly on MATLAB. For positive and negative windows, we generate two corresponding LUTs and embed them in the weight-trans model, using Δt as an index to select the output $\Delta weights$.

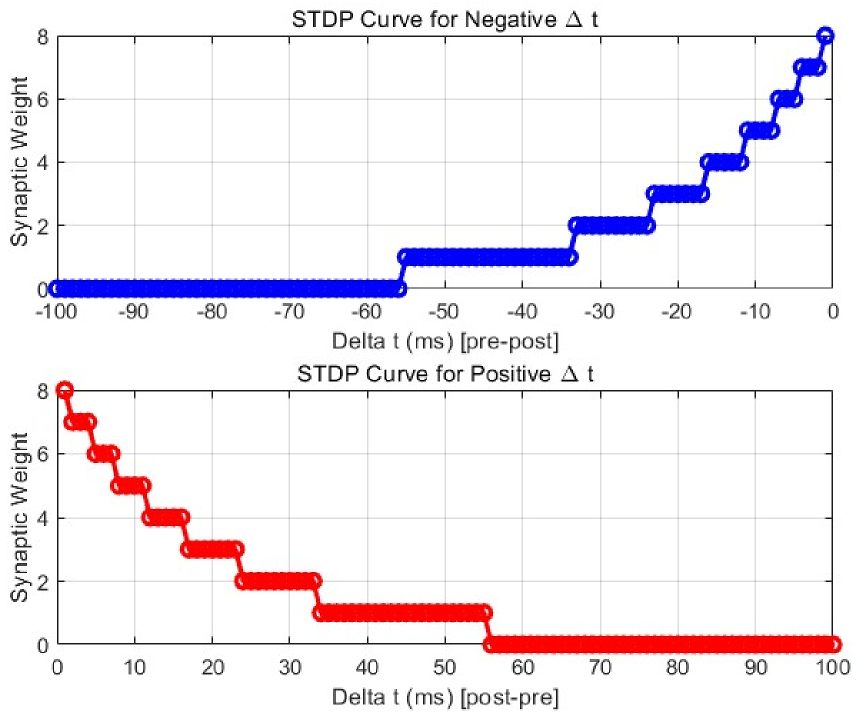


Figure 3.11: Matlab LUT data generation

In order to ensure the synchronous output of address and time information, in the weight conversion module (weight-trans), we simply add a flip-flop (flip-flop) to delay the signal by one clock cycle. In addition, a Weight-Delta-Indicator signal is introduced to represent the weight change value and the validity of its corresponding address. This design aims to ensure the consistency and synchronization of data processing, ensuring that the relevant address and time information can be output accurately together with each weight update.

3.5 Weight-Bram

After completing the STDP conversion and obtaining a series of weight modification values corresponding to each input, the next key step is to integrate these modification values into the corresponding weights of the synapses in the original SNN network. This process requires a mechanism to efficiently retrieve and update synaptic weights from storage. To this end, a specialized weight management system is implemented that accesses and modifies synaptic weights based on weight update values calculated by the STDP algorithm. This system ensures that weight adjustments are accurately reflected on the network's synaptic connections, enabling dynamic learning and adaptation based on temporal spike patterns.

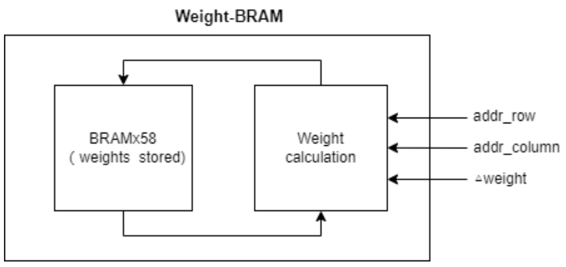


Figure 3.12: Enter Caption

In the VHDL implementation of SNN, the storage strategy of weights is crucial for network performance and resource utilization. In this design, for example, the input size of the first layer is 784 and the second layer is 400, and the total number of weights to be stored reaches 313600. For this reason, we chose to use a Series 7 36KB BRAM for weight storage, and special consideration was given to the BRAM configuration to adapt to the storage requirements of the SNN.

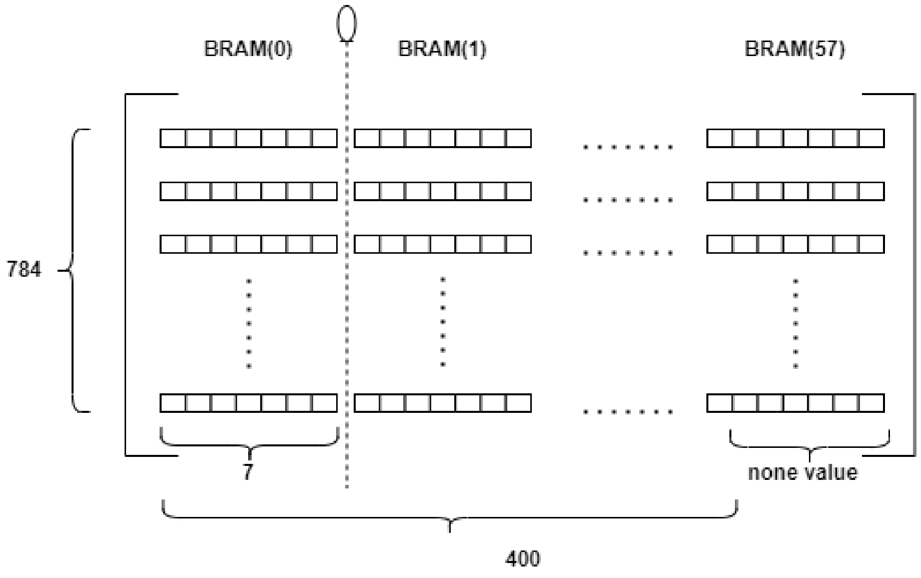


Figure 3.13: Methods for storing weights in BRAM

The BRAM configuration includes both single-line write and read lengths (word-length) set to 36 bits and all initial weight values set to 0. In this design, the quantization width (weight-bit-width) of individual weights is set to 5 bits, which is an adjustable parameter so that it can be adjusted in subsequent experiments to observe the effect of different quantization levels on the experimental results. effect of different quantization levels. Based on the 36-bit word-length, each line can store up to 7 weight values (36/5=7).

This design adopts a unique transpose matrix storage strategy, i.e., the number of rows of the weight matrix to be stored is set to 784, and the number of columns is set to 400. according to this storage strategy, the overall number of BRAMs needed is 400/7 + 1 = 58. As shown in the figure, the BRAM is divided into 58 cells (numbered from 0 to 57), and in each BRAM cell, each row has a width of 36 bits and is capable of storing 7 weight values, resulting in a total of 784 columns.

In addition, the transposed matrix storage method not only optimizes the storage efficiency, but also simplifies the access and update process of the weights, especially when performing the STDP learning rule. This approach allows parallel processing of weight updates for multiple neurons, which significantly improves the computational efficiency. Meanwhile, the tunability of the quantization width facilitates the study of the impact of different accuracies on the learning performance, further enhancing the flexibility of SNN design and the breadth of experiments.

Table 3.1: Encoding Table for N=3

Decimal	**Binary**	**Unary**	**One-hot**
0	000	0	00000001
1	001	1	00000010
2	010	10	00000100
3	011	11	00001000
4	100	100	00010000
5	101	101	00100000
6	110	110	01000000
7	111	111	10000000

This system uses 58 BRAM cells to store weights and manages these BRAM cells through precise read and write control. The write enable port (wren) is configured as BRAM-sel to specify the target BRAM cell for the current operation. The system uses a One-Hot Encoding decoder to optimize the address selection process by converting the N-bit input signals into uniquely corresponding high level outputs, precisely selecting individual BRAM cells for operation, thus simplifying the address resolution process and improving the operational efficiency of the system.

The use of Block Random Access Memory (BRAM) for weight storage is a key strategy for data management in Spiking Neural Networks (SNNs). Initially, the network weights are set, and as the network operates, these weights are modified by Spike-Timing-Dependent Plasticity (STDP). A state machine controls the BRAM during these processes, which includes:

- IDLE (idle state): The system initializes and prepares for operation, ensuring readiness for weight adjustments.
- Weight-fetch (weight extraction state): This state is used in the early stage of network construction, where the initial weight values of synapses are assigned through external data sources (e.g., file reading), which is a key step in network construction and weight initialization.
- Weight-stdp-rd (STDP read state): During STDP operations, it reads specific weight values from BRAM, providing data for time-difference-based weight adjustments.
- Weight-stdp-wr (STDP write state): Writes the updated weights back to BRAM, completing the dynamic weight adjustment process.

Through this state machine control mechanism, not only can the initialization and dynamic learning process of the network be clearly separated, but also the efficiency and accuracy of weight access can be improved. In addition, the introduction of the state machine provides better scalability and flexibility for the system, which makes modification and optimization for the future possible and further enhances the capability of the SNN in handling complex tasks.

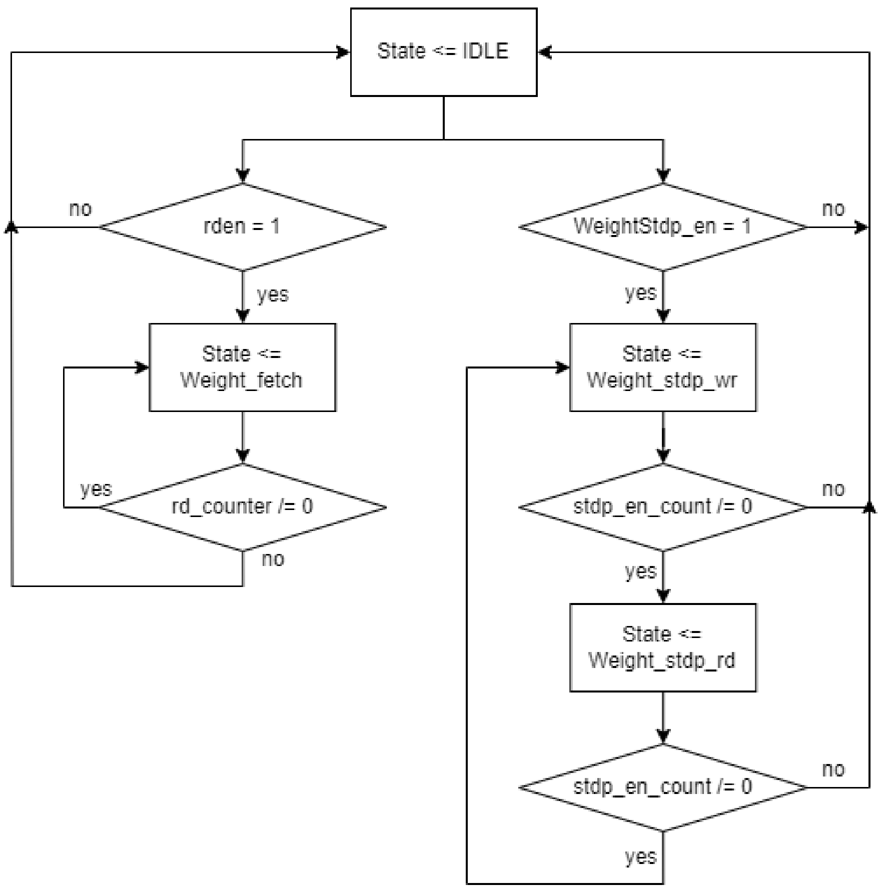


Figure 3.14: Weight-BRAM Flowchart

In the state machine's design, one trigger initializes the SNN, and another starts the learning phase, allowing smooth transitions within the system's operations.

In this flowchart, the wake-up of the system from the IDLE state is triggered by two conditions:

- Weight read: when rden is activated, the system enters the Weight-fetch state, in which the system reads the initial weight values from the outside and stores them in BRAM for weight initialization in the network establishment phase.
- Weight update: When WeightStdp-en is activated, the system enters the Weight-stdp-wr state, which indicates that the STDP mechanism triggers the weight update, at which time the system writes the updated weight values into BRAM.

These two triggering conditions correspond to different phases of neural network initialization and weight adjustment in the learning process, respectively. Since rden is only activated in the initialization phase, it does not conflict with the weight learning process of WeightStdp-en. In the weight update phase, the system first extracts the weight values to be changed and their addresses, and then updates them in real time based on STDP rules, which ensures the dynamic learning capability of the network. Through this mechanism, the integrity of data and the efficiency of access are ensured throughout the life cycle of the neural network.

Listing 3.1: address select

```
bram_num := to_integer(unsigned(WeightStdp_addr2_stor(addr_counter)))
     / 7;
bram_inner_num := to_integer(unsigned(WeightStdp_addr2_stor(
    addr_counter))) mod 7;

weight_line := data_out_buffer(bram_num);

weight := weight_line((bram_inner_num+1)*weights_bit_width-1
                downto bram_inner_num*weights_bit_width);

weight := weight + Weight_Delta_stor(weight_stor_count);
                weight_stor_count <= weight_stor_count + 1;
                weight_line((bram_inner_num+1)*weights_bit_width-1
                downto bram_inner_num*weights_bit_width)
                := weight;
```

1. The storage and updating of weights is one of the key aspects in the implementation of SNN. This design utilizes multiple BRAM cells to store the weight values, where the bram-num represents the BRAM number where the desired weights are located, as determined by addr2 output from the STDP module. addr1 represents the row number of the weights in the BRAM, while addr2 points to the address of the second-layer neuron - in 400 neurons is equivalent to selecting a specific column in the BRAM.

   Since each BRAM is capable of storing weight values for up to 7 neurons, a limit determined by both word-length and weight-bit-width, the specific BRAM number (BRAM-num) of the desired weight can be determined by dividing addr2 by 7. Next, calculating the remainder of addr2 divided by 7 allows determining the exact position of the target weight in its BRAM (denoted as bram-inner-num).

2. After obtaining the exact location of the weights, the next step is to extract the weight values at that location and perform the necessary calculations. This involves updating the weight value by adding the original weight value to the change in weight (Delta-Weight) calculated by the STDP algorithm. Eventually, the updated weight value is written back to its original position in the original extracted row to replace the previously stored weight value.

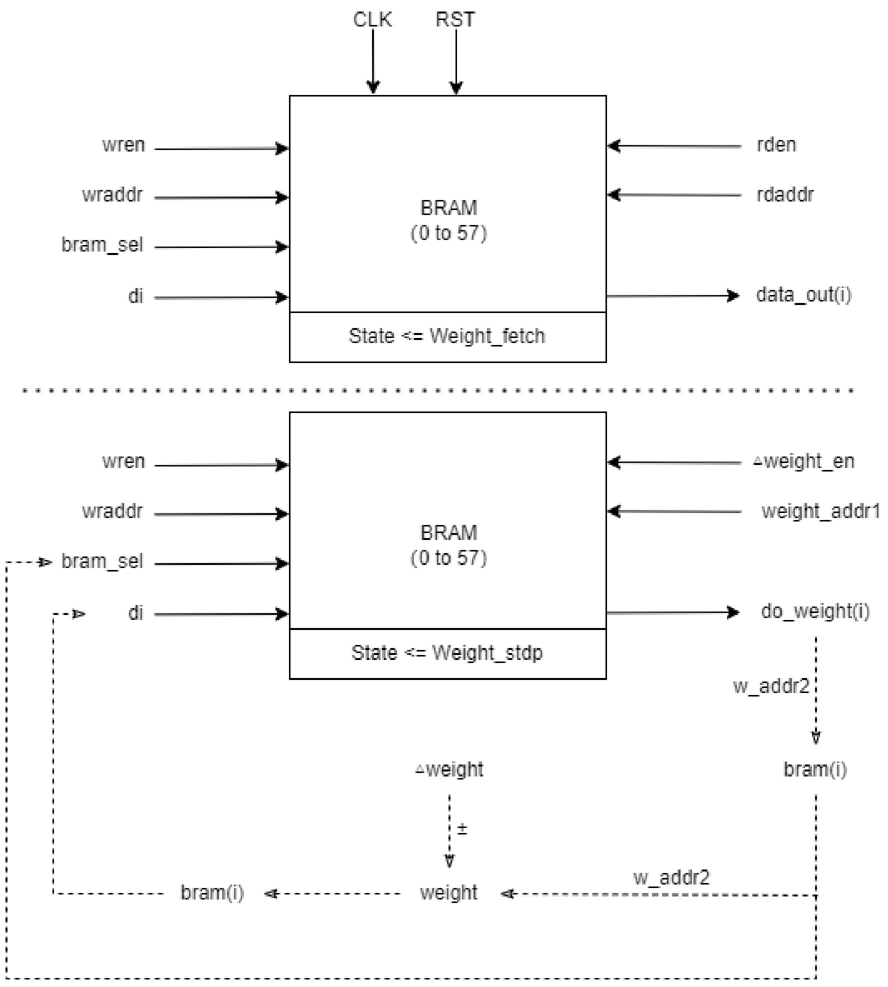


Figure 3.15: Weight-BRAM process structure

This design effectively supports the dynamic learning process of SNN by accurately locating and updating the weight values in BRAM. By using addr2 for locating the BRAM number where the weight is located and its position within the BRAM, and addr1 for determining the row position of the weight in the BRAM, this design not only realizes the efficient management of weight storage, but also ensures the accuracy and fast response of the weight update process.

3. The process then moves to the next critical state, which is to write the computed and modified rows of weight values back to the BRAM. The operations involved in this phase are typical of the BRAM write process. It is important to note that the write address wraddr port receives the value of addr1

determined during the STDP calculation, which represents the row address of the weight data (ranging from 1 to 784), while the write enable signal wren (i.e., the bram-sel port) corresponds to the value of the bram-num previously calculated by dividing addr2 by 7.

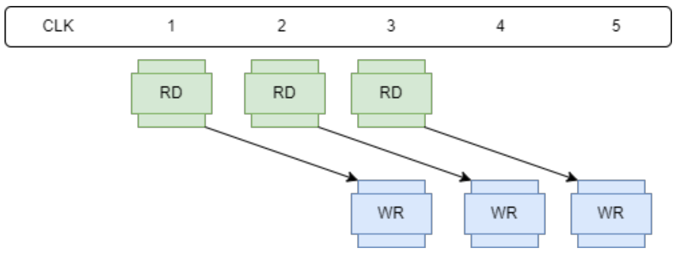


Figure 3.16: Enter Caption

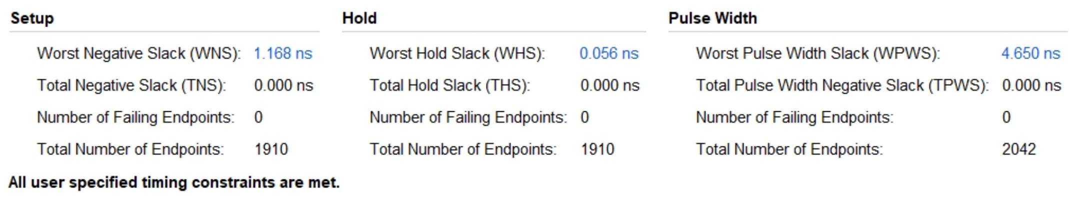
4. As shown in the figure, the timing control strategy for reads and writes to the same memory address in Block Random Access Memory (BRAM) is illustrated. In order to avoid data conflicts due to the read and write cycle delays inherent in BRAM (operations take effect on the next clock cycle), a delay of two clock cycles is introduced between read and write operations.

3.6 Synthesis

The synthesis part of this design is analyzed for resource utilization, timing analysis and power consumption analysis.

Site-Type	**Stdp (Util%)**	**Weight-BRAM &Stdp (Util%)**	**Total-Available**
Slice LUTs*	5.51	10.54	41000
LUT as Logic	5.51	9.52	41000
LUT as Memory	0.00	3.13	13400
Slice Registers	2.48	5.45	82000
Register as Flip Flop	2.48	5.45	82000
Register as Latch	0.00	0.00	82000
F7 Muxes	0.77	2.23	20500
F8 Muxes	0.74	1.50	10250
Block RAM Tile	2.96	46.30	135
RAMB36/FIFO*	2.96	45.93	135
RAMB18	0.00	0.37	270

Table 3.2: Resource Utilization Summary



Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.168 ns	Worst Hold Slack (WHS):	0.056 ns	Worst Pulse Width Slack (WPWS):	4.650 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1910	Total Number of Endpoints:	1910	Total Number of Endpoints:	2042

All user specified timing constraints are met.

Figure 3.17: Time report summary

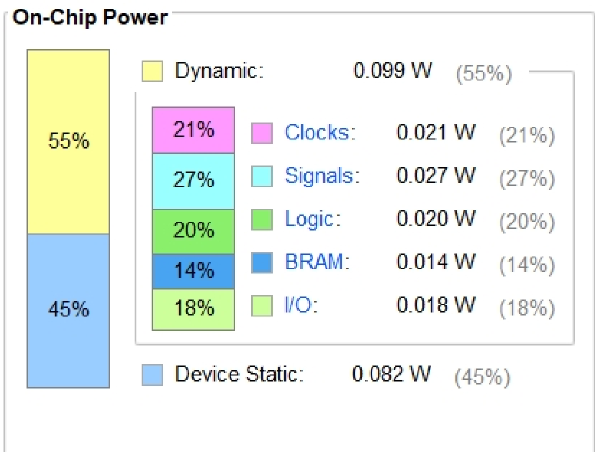


Figure 3.18: Power report summary

In our comprehensive analysis, we evaluated the overall performance only for the STDP mechanism and its use in combination with Weight-BRAM.

- As shown in Table 3.2, we analyzed the resource utilization of the Stdp module versus the Weight-BRAM & Stdp module. In the Weight-BRAM & Stdp module, the Block RAM Tile is utilized at 46.30%, which is significantly higher than the 2.96% of the Stdp module, mainly because it is used as a device for storing weights.

- Timing analysis: Worst Negative Slack (WNS) is 1.168 ns, Worst Hold Slack (WHS) is 0.056 ns, and Worst Pulse Width Slack (WPWS) is 4.650 ns, and all user-defined timing constraints are satisfied.

- Power Consumption Analysis: The dynamic power consumption is 0.099 Watts, which is 55% of the total power consumption, with clock, signal, logic, BRAM and I/O accounting for 21%, 27%, 20%, 14% and 18% of the dynamic power consumption, respectively.

Bibliography

[1] Kashu Yamazaki, Viet-Khoa Vo-Ho, Darshan Bulsara, and Ngan Le. «Spiking Neural Networks and Their Applications: A Review». In: *Brain Sciences* 12 (2022) (cit. on p. 1).

[2] W. Gerstner and W. M. Kistler. *Spiking neuron models: Single neurons, populations, plasticity.* Cambridge University Press, 2002 (cit. on pp. 1, 3).

[3] A. L. Hodgkin and A. F. Huxley. «A quantitative description of membrane current and its application to conduction and excitation in nerve». eng. In: *The Journal of Physiology* 117.4 (Aug. 1952), pp. 500–544. ISSN: 0022-3751 (cit. on p. 2).

[4] Henry Markram, Wulfram Gerstner, and Per Jesper Sjöström. «Spike-Timing-Dependent Plasticity: A Comprehensive Overview». In: *Frontiers in Synaptic Neuroscience* 4 (July 2012). Publisher: Frontiers. ISSN: 1663-3563. (Visited on 03/13/2024) (cit. on p. 2).

[5] Bojian Yin, Federico Corradi, and Sander Bohté. «Effective and Efficient Spiking Recurrent Neural Networks». In: *ERCIM News* (July 2021) (cit. on p. 3).

[6] Authors' Names. «Memristive Izhikevich Spiking Neuron Model and Its Application in Oscillatory Associative Memory». In: *Frontiers in Neuroscience* Volume Number (2022), Page Numbers (cit. on p. 3).

[7] Author(s) Names. «Action Potential Initiation in the Hodgkin-Huxley Model». In: *PLOS Computational Biology* (Publication Year) (cit. on p. 3).

[8] Alessio Carpegna, Alessandro Savino, and Stefano Di Carlo. «Spiker: an FPGA-optimized Hardware accelerator for Spiking Neural Networks». In: *arXiv* (2022) (cit. on pp. 4, 18–20).

[9] M. Mozafari, S. R. Kheradpisheh, T. Masquelier, A. Nowzari-Dalini, and M. Ganjtabesh. «First-spike-based visual categorization using reward-modulated stdp». In: *IEEE Trans. Neural Networks Learn. Syst.* 29 (2018), pp. 6178–6190 (cit. on pp. 4, 5).

[10] Xiangfeng Yang and Yuanyuan Shen. «Runge-Kutta Method for Solving Uncertain Differential Equations». In: *Journal of Uncertainty Analysis and Applications* 3.17 (2015) (cit. on p. 5).

[11] Jesus L Lobo, Javier Del Ser, Albert Bifet, and Nikola Kasabov. «Spiking Neural Networks and online learning: An overview and perspectives». In: *Neural Networks* 121 (2020), pp. 88–100 (cit. on p. 6).

[12] Errui Zhou, Liang Fang, and Binbin Yang. «Memristive Spiking Neural Networks Trained with Unsupervised STDP». In: *Electronics* 7.12 (2018) (cit. on pp. 8, 9).

[13] Runchun Wang, Gregory Cohen, Klaus M. Stiefel, Tara Julia Hamilton, Jonathan Tapson, and André van Schaik. «An FPGA Implementation of a Polychronous Spiking Neural Network with Delay Adaptation». In: *Frontiers in Neuroscience* 7 (2013) (cit. on pp. 10, 12).

[14] Xilinx. *Artix-7 FPGA Product Documentation.* 2024. URL: `https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html#documentation` (cit. on pp. 11, 13, 15).

[15] Xiping Ju, Biao Fang, Rui Yan, Xiaoliang Xu, and Huajin Tang. «An FPGA Implementation of Deep Spiking Neural Networks for Low-Power and Fast Classification». In: *Neural Computation* 32.1 (2020) (cit. on p. 12).

[16] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. «The MNIST database of handwritten digits». In: *IEEE* 10.8 (1998) (cit. on pp. 15, 16).