Heterogeneous computing with spiking-neural-network acceleration in a RISC-V-based system-on-chip

by

Xinhu Liu

Student Name	Xinhu
First Surname	Liu

Supervisor: Kofi. Makinwa
Daily Supervisor: Charlotte. Frenkel
Project Duration: Dec, 2022 - Sept, 2023
Faculty: Faculty Electrical Engineering, Mathematics and Computer Science, Delft

Style: TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Abstract

Spiking neural networks (SNNs), which are regarded as the third generation of neural networks, have attracted significant attention due to their promising applications in various scenarios. Based on SNNs, neuromorphic coprocessors, designed to emulate the structure and functionality of biological brains, hold the potential to revolutionize computing. However, these coprocessors encounter challenges related to adaptability and flexibility in various application environments once they are manufactured. To tackle this challenge, our project introduces a neuromorphic System-on-Chip (SoC), which seamlessly integrates a RISC-V CPU with an SNN coprocessor, utilizing sparse time-to-first-spike encoding (TTFS). The primary goal of this SoC is to facilitate the complete reconfigurability of the SNN coprocessor with the RISC-V CPU. By leveraging this neuromorphic SoC and successfully simulating the novel loop learning work model to achieve an accuracy of 92.2% on the MNIST dataset, we demonstrate its capability to adapt the SNN coprocessor for various application scenarios, such as text recognition and face detection.

Contents

Summary i

Nomenclature iv

1 Introduction 1
- 1.1 Contributions . . . 2
- 1.2 Outline . . . 3

2 Background 4
- 2.1 Neural Networks . . . 4
  - 2.1.1 Artificial Neural Networks (ANNs) . . . 4
  - 2.1.2 Learning in Neural Networks . . . 5
  - 2.1.3 Spiking Neural Networks (SNNs) . . . 7
  - 2.1.4 Spike Encoding Schemes . . . 9
  - 2.1.5 Quantization . . . 11
- 2.2 Modern Systems-on-a-Chip (SoC) . . . 12
  - 2.2.1 RISC-V . . . 13
  - 2.2.2 X-Heep . . . 15

3 Heterogeneous Neuromorphic SoC 19
- 3.1 Preliminaries . . . 19
  - 3.1.1 Time-to-First-Spike Encoding on MNIST Dataset . . . 19
  - 3.1.2 Spiking Neuron Model . . . 20
  - 3.1.3 TTFS-based Training . . . 22
  - 3.1.4 Simplification of the Training Algorithm . . . 24
- 3.2 Software Design . . . 24
  - 3.2.1 Baseline . . . 25
  - 3.2.2 Simplified Backpropagation Algorithm . . . 26
  - 3.2.3 Simplified Cuba-LIF Model . . . 30
  - 3.2.4 Quantization of the SNN . . . 34
  - 3.2.5 Loop Learning in Software . . . 37
  - 3.2.6 Results of Software Design . . . 38
- 3.3 Hardware Design . . . 38
  - 3.3.1 Neuromorphic SoC Architecture . . . 40
  - 3.3.2 SNN Processor: TTFS-tinyODIN . . . 40
  - 3.3.3 Cooperation between X-Heep and TTFS-tinyODIN . . . 50
  - 3.3.4 Results and Fallback Plan . . . 54

4 Conclusion and Future Work 55
- 4.1 Conclusion . . . 55
- 4.2 Future Work . . . 56

References 57

A Result of Loop Learning in the SoC 61

Nomenclature

Abbreviations

Abbreviation	Definition
ANN	Artificial Neural Network
SNN	Spiking Neural Network
SoC	System-on-Chip
MCU	Microcontroller Unit
TTFS	Time-To-First-Spike
LIF	Leaky, Integrate and Fire
PTQ	Post-Training Quantization
QAT	Quantization Aware Training
RISA	Reduced Instruction Set Architecture
CISA	Complex Instruction Set Architecture
CPU	Central Processing Unit
MMIO	Memory-Mapped Input Output
Cuba	Current-based

1 Introduction

Since the beginning of the 20th century, Artificial Neural Networks (ANNs) have been developing rapidly [16], evolving from the simple multi-layer perceptrons (MLPs) [19] to the current state-of-the-art large-scale language models (LLMs). Traditional ANNs have achieved remarkable success and have become an inseparable part of most people's lives [54] [36] [35].

Nevertheless, power consumption stands out as a paramount concern. This contravenes the primary objective of ANNs since the original purpose of ANNs was to emulate the cognitive processes of the human brain, which serves as an exceptionally energy-efficient computing system. The human brain, capable of performing 10^{18} mathematical operations per second, achieves this computational prowess while consuming only 20 watts, as documented in [38]. This level of energy efficiency remains far from matched [41] and the gap seems unreachable without a possible paradigm shift.

To fill the gap in energy efficiency, spiking neural networks (SNNs) [37] were put forward and have gained a lot of popularity in the last few years, showing great potential to surpass the energy efficiency of conventional ANNs. Figure 1.1 shows the major differences between SNNs and the conventional ANNs. The design philosophy of SNNs is more closely aligned with the human brain [33]. For example, data transmission in SNNs is spike-based, which is in binary format, compared to the conventional real-world outputs in ANNs. It needs fewer computing resources, theoretically enabling it to achieve better energy efficiency compared to traditional neural networks and thus moving closer to simulating the human brain. Therefore, SNNs introduce new possibilities for applying neural networks to low-power computing scenarios like edge computing.

However, despite the highly promising academic research on SNN processors, the design of embedded neuromorphic systems remains challenging. Typical SNN processors [32] [34] do not have on-chip learning capabilities, and thus their internal parameters cannot be modified by other deployments. However, SNN processors that do have learning capabilities are often far less likely to perform as well as off-chip-trained once due to limited on-chip resources [4] [12]. Therefore, flexibility limitations prevent neuromorphic devices from being widely used in edge computing application

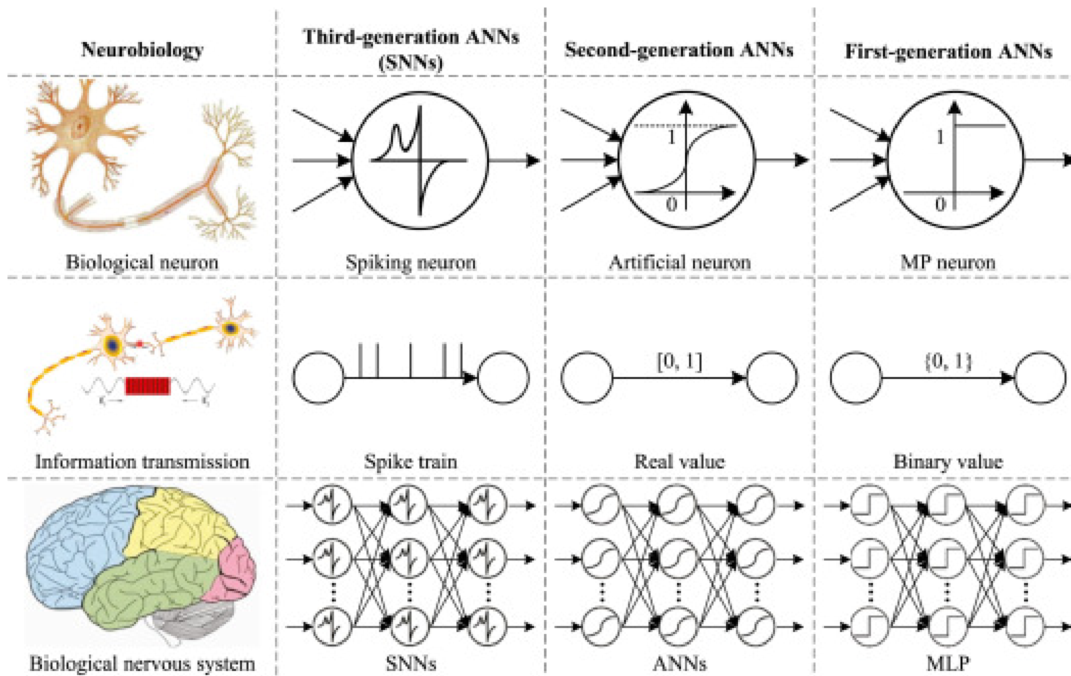


Figure 1.1: Three generations of artificial neural networks: Multilayer perceptron (MLP), ANNs and SNNs. Adapted from[52]

scenarios. To solve this problem, a host central processor unit (CPU) can be used to configure the SNN processor flexibly, so that the SNN processor can be applied to a variety of application scenarios.

In this project, we propose a neuromorphic system-on-chip (SoC) platform that combines a RISC-V-based microcontroller unit (MCU) and an SNN processor. This SoC realizes the balance between the flexibility and efficiency of neuromorphic devices, and it's able to perform "Loop Learning", which is shown in Figure 1.2. In Loop Learning, the RISC-V CPU and SNN processor are in charge of backpropagation and inference respectively: the RISC-V CPU will implement the learning algorithm to update the weights based on the inference result from the SNN processor, and the SNN processor will do the inference based on the updated weights given by RISC-V CPU, the whole data flow generates a loop. Therefore, this SoC gives new possibilities for neuromorphic devices to be applied in low-power computing, where we will investigate sparse spike-based encoding schemes such as time-to-first-spike (TTFS) encoding.

1.1. Contributions

The main contributions of this work can be summarized as follows:

- We simplify the learning algorithm and the neuron model of TTFS-based SNNs from [15]. Compared to the original work, these simplifications reduce the total computation significantly, while losing only 2% accuracy on the MNIST dataset.
- Based on the SNN processor tinyODIN [12], we have developed a new SNN coprocessor, TTFS-tinyODIN, which is capable of supporting inference based on

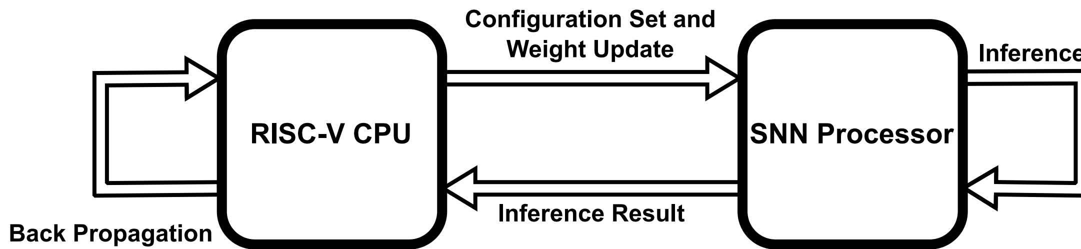


Figure 1.2: Dataflow in the Loop Learning model, where the SNN processor runs the inference and the RISC-V CPU runs the backpropagation to update the weights based on the inference result and then writes back the updated weights to the SNN processor. Inspired from [15]

TTFS encoding. The crossbar structure of TTFS-tinyODIN includes 256 neurons, allowing it to map arbitrary SNN architectures.

- TTFS-tinyODIN performs inference for TTFS-encoded MNIST inputs, which is consistent with the software in the post-implementation simulation results.
- TTFS-tinyODIN is integrated into X-Heep, an open-source MCU platform [45], to create a comprehensive open-source neuromorphic SoC. This SoC enables the configuration of TTFS-tinyODIN via the RISC-V CPU.
- The RISC-V CPU implemented on X-Heep employs a loop learning model in conjunction with TTFS-tinyODIN, resulting in training outcomes identical to those of relevant software. This type of learning loop allows the system to learn across various application scenarios.

1.2. Outline

The rest of the thesis is divided into three chapters.

- Chapter 2 gives the background of this project, which discusses the development of ANNs and SNNs, and the different encoding schemes of SNNs. This chapter also introduces the MCU platform used in this project, X-Heep, in detail. In addition, this chapter also discusses the trade-off between efficiency and flexibility.
- Chapter 3 details the core parts of this project: the software model training and SoC architecture design. The details of the design in each part are presented in this chapter and the results of the design in each part are also shown.
- Chapter 4 describes the summary of the project and the overall design results. It also identifies areas for future work and room for optimization.

2
Background

This chapter aims to provide an overall understanding of the project's details and design approach by introducing relevant background knowledge. It includes the following topics: artificial neural networks (ANNs), spiking neural networks (SNNs), the four main encoding schemes of SNNs, quantization, the RISC-V instruction set architecture (ISA) and an open-source MCU platform: X-Heep [45].

2.1. Neural Networks

In this section, we provide an objective overview of both the basic network architecture and neuron model of ANNs. Additionally, we detail the step-by-step learning process of ANNs with reference to backpropagation. To compare ANNs with SNNs, we then explain the basic spiking neuron model in SNNs and highlight the differences between the spiking neuron model and the conventional neuron model.

2.1.1. Artificial Neural Networks (ANNs)

In the field of computer science and artificial intelligence, artificial neural networks (ANNs) are computational models based on the neural system of the human brain. ANNs have since evolved into a prominent class of mathematical models. As early as the 1940s, Warren McCulloch and Walter Pitts proposed a mathematical model to simulate the behaviour of biological neurons [39], which is regarded as one of the first neural network theories.

Due to insufficient training methods and computational power, ANNs were not widely used during their early development [42]. However, the introduction of the backpropagation algorithm in the 1980s [28] renewed interest in ANNs, but hardware and dataset limitations still impeded their progress.

However, as computer technology continues to advance, the limitations of hardware resources are gradually being tackled, and neural networks once again garnered widespread interest [1]. Particularly around 2010, with the rise of deep learning, neural networks were extended to deeper structures and formed deep neural networks (DNNs) [31] [20] [51] [48]. Till now, DNNs have achieved significant success in multiple

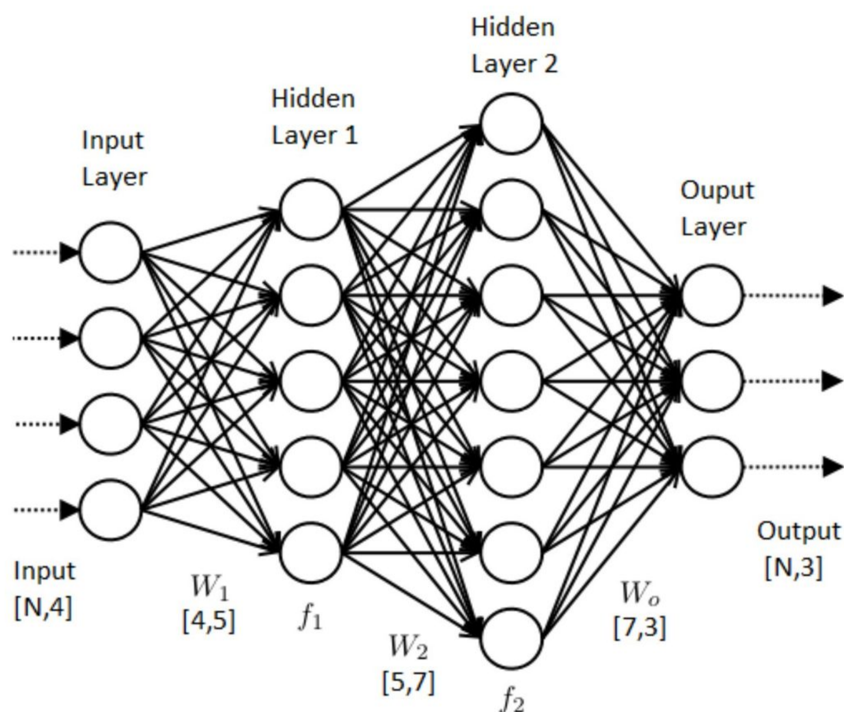


Figure 2.1: One fully connected network with one 4-neuron input layer, two hidden layers, 5-neuron and 7-neurons respectively, and one 3-neuron output layer. The W[x,y] indicates the shape of the weight matrix between two layers, x means the number of neurons in the former layer and y means the number of neurons in the latter layer. Adapted from[26]

application scenarios such as image recognition, natural language processing, and speech recognition [40] [9].

Currently, the fundamental architecture of artificial neural networks is depicted in Figure 2.1 [26]. The structure is composed of a single input layer, various hidden layers, and one output layer. Each layer contains multiple neurons, which are critical components of neural networks. As depicted in Panel A of Figure 2.2, these artificial neurons receive input signals, perform a weighted summation of these inputs, and employ an activation function to produce an output or activation value. This activation value is then transmitted to other neurons in subsequent layers, creating a network of interconnected neurons. For example, in image classification, a neuron in the input layer usually represents one pixel of the input image.

2.1.2. Learning in Neural Networks

To allow the neural network to learn and thus accomplish the target task, backpropagation coupled with gradient descent [43] is typically used. In detail, the whole learning process in artificial neural networks consists of the following steps:

1. Forward propagation: Initially, input data is propagated through the layers of the network, progressively computing the output of each layer, until the model's output is available at the output layer. This procedure is usually called inference. As per equation 2.1, $Input^l, W^l, b^l, Y^l$ indicates the input vector, weight matrix, bias and output of layer l. Y^l is also the input of layer l+1, $Input^{l+1}$. ϕ is the

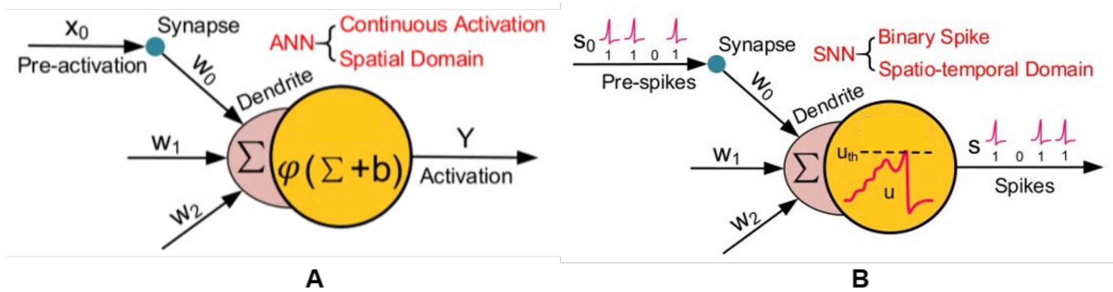


Figure 2.2: Panel A: A typical neuron model in conventional ANNs: the $\sum$ means this neuron accepts the cumulative sum of inputs (x_i) and weights (w_i), the sum together with a bias b is sent to the activation function ϕ and the neuron release the output Y. Panel B: A typical neuron model in SNNs: the $\sum$ means this neuron accepts the input spikes (S_i) and increase the membrane potential by the corresponding weights (w_i), u means the membrane potential of this neuron and U_{th} is the threshold voltage, this neuron will release a spike when u is above the U_{th} then reset. Adapted from u [21]

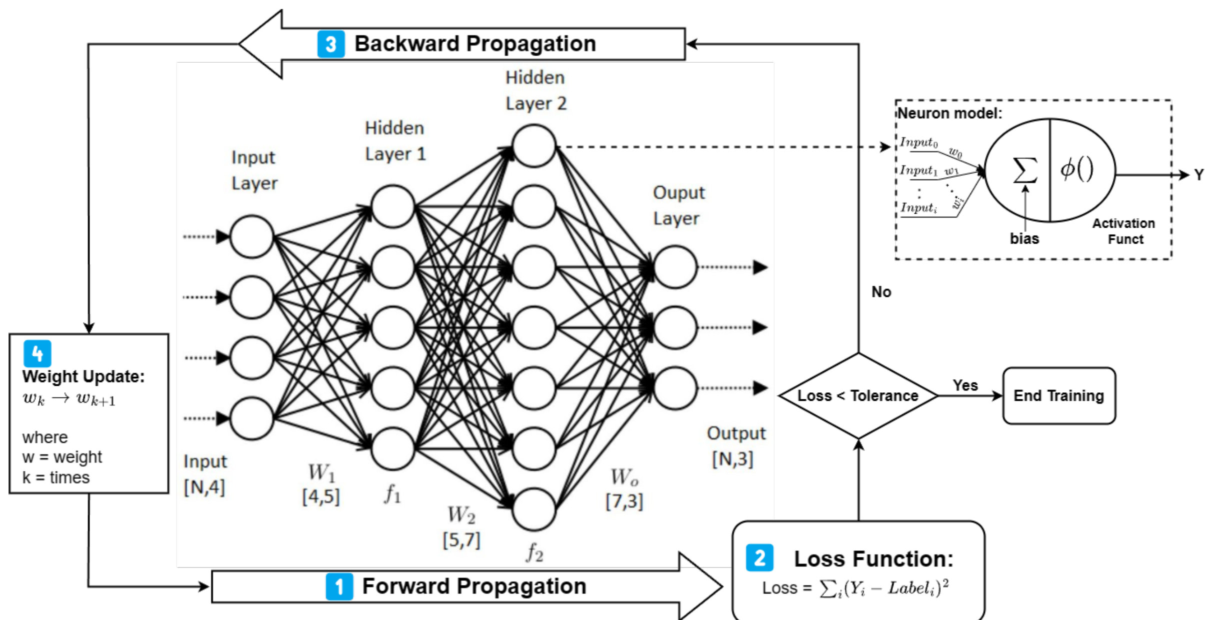


Figure 2.3: The typical learning process in ANNs. Blue labels show the sequence. Step 1: the network carries out forward propagation or inference as per the calculation in the neuron model in Figure 2.2. Step 2: computing the Loss function, the example in the figure is mean squared error (MSE) loss. Step 3: compute the gradients of each weight with the chain rule (Figure 2.4) in backpropagation. Step 4: update the weights according to the gradients from the last step. This figure is modified from [30].

activation function.

$$Y^l = \phi(Input^l \times W^l + b^l) \quad \text{where l is the layer index} \tag{2.1}$$

2. Loss Computation: The network's output is compared with the ground truth (e.g. classification labels), and a pre-defined loss function is employed to quantify the discrepancy between the predicted results and the ground truth. Commonly used loss functions include the mean squared error (MSE) loss or the cross-entropy loss. Equation 2.2 shows the loss function of MSE loss, the Y_i is the output of neuron i in the output layer, and $Label_i$ is the ground truth of this inference.

$$Loss = \sum_i (Y_i - Label_i)^2 \quad \text{where i is the output neuron index} \tag{2.2}$$

3. Backward Propagation: Compute the gradients of the loss function concerning each network weight based on the chain rule, which is shown in Figure 2.4. The chain rule enables the efficient calculation of gradients by breaking down the computation into products of gradients of individual components. The computing of backward propagation is shown in equation 2.3 and Figure 2.4.

$$\frac{\partial Loss}{\partial w^l} = \frac{\partial Loss}{\partial Y^l} * \frac{\partial Y^l}{\partial w^l} \qquad \frac{\partial Loss}{\partial x^l} = \frac{\partial Loss}{\partial Y^l} * \frac{\partial Y^l}{\partial x^l} \quad \text{where l is the layer index} \tag{2.3}$$

4. Weight Update: After obtaining the gradients of network parameters, gradient descent is used to update the weights. The gradient descent algorithm is applied to update parameters in the opposite direction of the gradients (i.e. direction of the steepest descent), iteratively reducing the loss function, where the step size is defined by a hyperparameter called learning rate. The basic equation of this step is called stochastic gradient descent (SGD) [42] which is shown in equation 2.4.

$$w^l_{k+1} = w^l_k + \alpha * \frac{\partial Loss}{\partial w^l_k} \tag{2.4}$$

where

- k is the iteration number,
- l is the layer index,
- α is the learning rate.

2.1.3. Spiking Neural Networks (SNNs)

Undeniably, ANNs have achieved remarkable success in various application scenarios. However, the operational logic of ANNs indeed deviates from their original intention of mimicking the workings of the human brain. One prominent difference lies in the mode of information transmission among neurons. In traditional ANNs, information is transmitted with real-valued amplitudes. In contrast, in the human brain, information

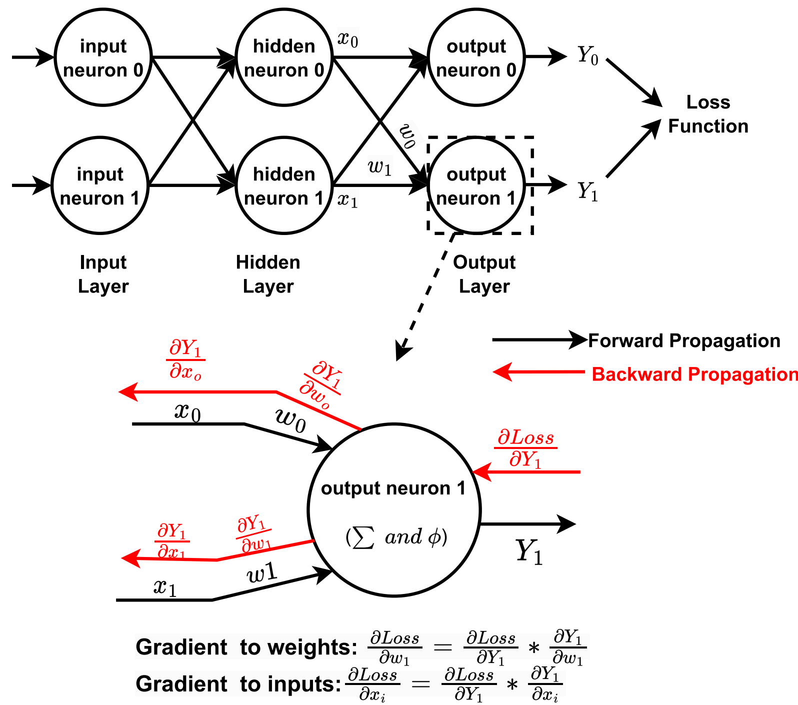


Figure 2.4: The chain rule in backpropagation. The cost of output neuron Y_i is illustrated. where x_i is the input and Y_i is the output. $\frac{\partial Loss}{Y_i}$ is the partial derivative of the loss function with respect to the output. $\frac{\partial Loss}{w_1}$ is the gradient of the loss function used to update w_1, while $\frac{\partial Loss}{x_i}$ is the gradient of the loss function with regard to the input x_i, which will be passed to the former layer.

transmission between neurons occurs through spikes, with relatively fixed amplitudes [13]. Building on this premise, numerous papers have proposed spiking neuron models that aim to reproduce this behaviour.

As of now, researchers have come up with several mathematical models to simulate the behaviour of neurons, but all proposed mathematical models of neurons have faced a trade-off between neuroscientific realism and computational complexity, given the inherent complexity of biophysical mechanisms in actual neural cells. The Hodgkin-Huxley (HH) model [22], while capable of faithfully emulating various behaviours of real neurons, presents significant challenges in hardware implementation due to its complexity. [47] provides an FPGA realization of a simplified HH model, as implementing the original HH model is intractable. Consequently, a more widely adopted alternative in contemporary research is the leaky integrate-and-fire (LIF) model [33]. A classic neural unit of the LIF model is depicted in Panel B of Figure 2.2 [21].

Compared to the neurons in conventional neural networks, the computing inside the LIF neurons is simpler, and basically uses only additions and comparisons. Each neuron has two intrinsic properties called membrane potential and threshold voltage. Once the neuron receives a binary spike from a neuron of the former layer, the membrane potential will increase by the weight between these two neurons, and this is called integration. If there is no spike coming in, the membrane potential will decrease gradually, and this is called leakage. This neuron will release a binary spike once the membrane is above the threshold, and then the membrane potential will be reset. One

basic equation of membrane potential is shown in equation 2.5.

$$\begin{cases} V_{membrane} = -V_{leak}(t) + \sum_i w_i \sum_{t_i} \theta(t - t_i) & \text{if } V_{membrane} \leq V_{threshold} \\ V_{membrane} = V_{reset} & \text{if } V_{membrane} > V_{threshold} \end{cases} \tag{2.5}$$

In the Equation 2.5, all symbols are explained as follows:

- θ is the Heaviside step function,
- $V_{leak}(t)$ is the leakage function,
- $V_{membrane}$ is the membrane potential,
- $V_{threshold}$ is the threshold voltage,
- V_{reset} is the reset voltage after spiking,
- i is the index of the input neurons,
- t_i is the spike times of the input neurons,
- w_i synaptic weight with the $neuron_i$.

The LIF model strikes a balance between accuracy and computational efficiency, making it a popular choice in simulating large-scale neural networks and implementing them in hardware.

The primary distinction between LIF-based SNNs and conventional ANNs lies in the fact that, in traditional ANNs, a neuron's input comprises the sum of the products of its inputs and weights before passing through an activation function. In the LIF neuron model, the neuron accepts all input spikes, increases the membrane potential according to the corresponding weights, and releases a spike when the membrane potential exceeds a certain threshold voltage (Figure 2.2).

Thanks to the specification of the LIF model, computing becomes much easier compared to that in conventional neuron models. Therefore, the sparse spike-based data transmission and computational workload make SNN processors a good candidate for power-efficient processing compared to traditional ANN processors.

2.1.4. Spike Encoding Schemes

From the previous text, it is clear that the encoding of SNNs varies from that of traditional ANNs. SNNs have spike-based, discrete inputs and outputs lacking amplitude, whereas traditional ANNs have continuous, frame-based inputs with amplitude. To convert continuous frame data into discrete spike data and enable simpler comparison between the performance of SNN processors and ANN processors, it is necessary to encode traditional machine learning datasets, and activations into spikes. Several encoding methods have been proposed in academic literature, such as rate encoding [2], TTFS encoding [14], phase encoding [29], burst encoding [6], and rank-order coding (ROC), as shown in Figure 2.5.

Rate encoding is a prevalent and frequently applied method of encoding. It transforms each input pixel into a frequency of spikes across a timeline based on its amplitude. The timeline is comprised of multiple basic time units, in each of which the pixel may

generate a spike based on its amplitude. The spiking probability is higher for pixels with larger amplitudes. The likelihood of generating spikes typically conforms to a Poisson distribution, similar to that in the brain [2]. During the timeline, higher-amplitude pixels generate a greater number of spikes, as depicted in Figure 2.4, panel A [18].

TTFS encoding [14] indicates that pixels with higher amplitudes generate spikes earlier, indicating their greater importance. Among all input pixels, the one with the highest amplitude spikes earliest while the one with the lowest amplitude spikes latest. Various functions, such as exponential or linear functions, are utilized for mapping the amplitude to spike time. The application of an exponential function for amplitude-to-time mapping is illustrated in Figure 2.4, panel B [18].

Phase encoding is a straightforward encoding technique. During a specific time period, spikes indicate binary 1s, while the lack of spikes represents 0s. Once this period is over, all spikes and absences are combined to form a binary number. A binary decoding process is then performed on all spikes within another specific period to achieve the final outcome. Figure 2.4, panel C [18], demonstrates a case with eight phases within a period, representing a range from 0 to 255.

Burst encoding is a method of representing amplitude values through the release of a burst of spikes within a small time window. The number of spikes in the time window increases with higher amplitude values. Figure 2.4, panel D illustrates this concept [6, 18].

According to [18], TTFS encoding achieves the best computational performance and has the lowest hardware overhead among various encoding methods. This can be attributed to only one spike operation and the utilization of precise timing, which aligns well with the objectives of this project. As a result, this project adopts TTFS encoding.

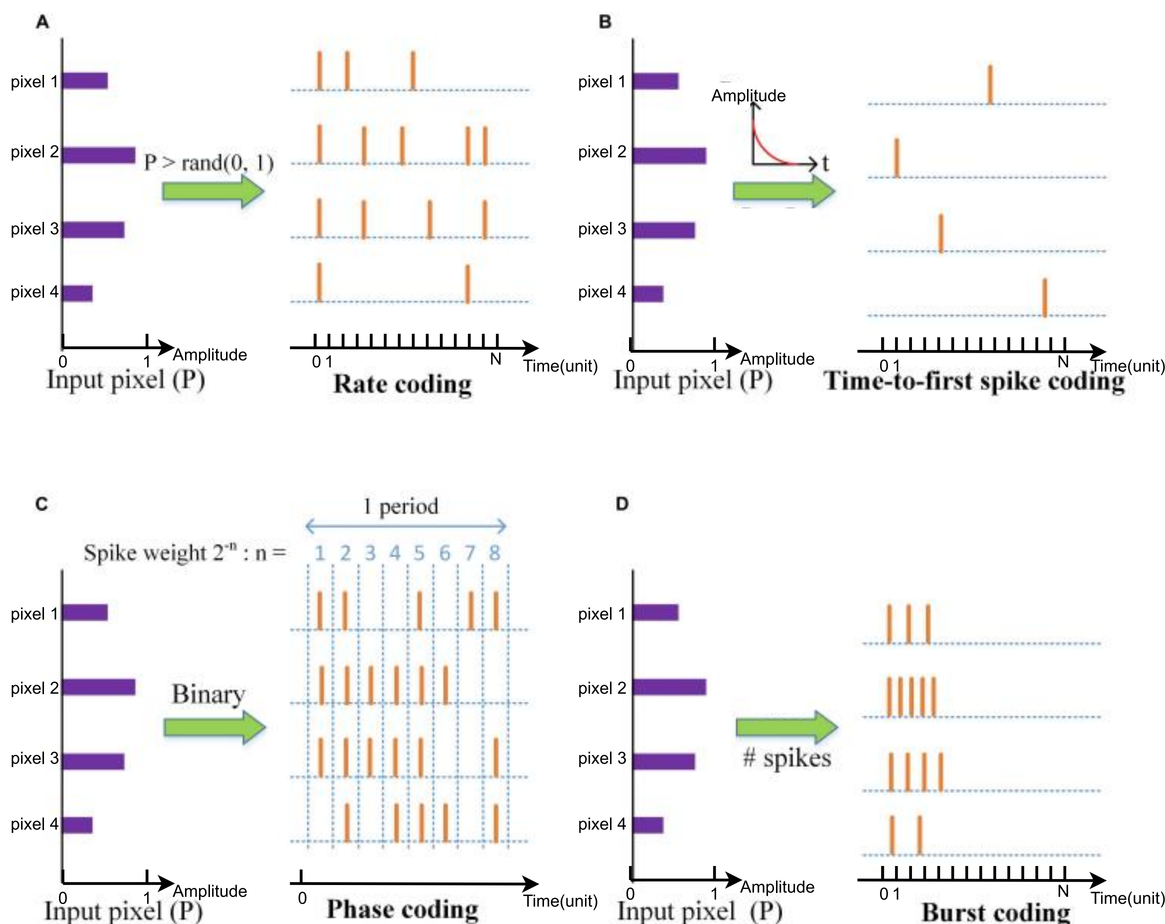


Figure 2.5: An illustration of four spike encoding schemes, the amplitude of pixel 2 is the highest while the amplitude of pixel 4 is the lowest. (A) Rate coding: pixel 2 produces the highest number of spikes over time, while pixel 4 produces the least. (B) Time-to-first spike coding: pixel 2 produces the earliest spikes, while pixel 4 produces the latest spikes. (C) Phase coding: in one period which contains 8 time units, pixels 1 to pixel 4 are encoded as 11001011, 11111100, 11111001, and 01011101 respectively. (D) Burst coding, the spike generated by pixel 2 is the densest while the spikes from pixel 4 are the sparest. Modified from [18]

2.1.5. Quantization

Quantization is a widely used technique in neural networks [24] [8]. Typically, when processing neural networks, we work with floating point numbers that range from 0 to 1 and have a data type of float32. Quantization is defined more broadly as the process of converting a high-precision floating-point number into a lower-precision fixed-point number, such as uint8, commonly known as quantization [25], which is illustrated in Figure 2.6.

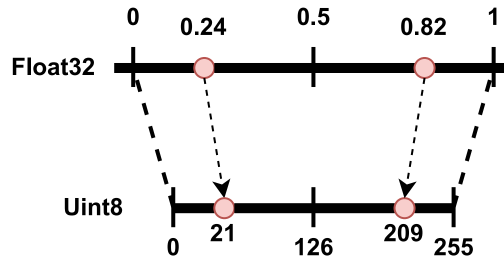


Figure 2.6: The example of quantizing a float32 value which is between 0-1 to a uint8 value which is between 0-255. The float32 0 will be the uint32 0 and the float32 1 will be the uint32 255, the float32 0.24 will be quantized to uint8 21 and the float32 0.82 will be quantized to uint8 209.

Appropriate quantization can significantly decrease the workload of neural network computation by reducing memory and bandwidth demand without sacrificing accuracy. The equations for general quantization are Equation 2.6 and Equation 2.7:

$$f = S(q - Z) \qquad q = round(\frac{f}{S} + Z) \tag{2.6}$$

where

$$S = \frac{f_{max} - f_{min}}{q_{max} - q_{min}} \qquad Z = round(q_{max} - \frac{f_{max}}{S}) \tag{2.7}$$

- f is the floating-point number, f_{max} and $f_m in$ denote the maximum and minimum of f respectively,
- q is the integer number after quantization, q_{max} and q_{min} denote the maximum and minimum of q respectively.
- S is the scale between f and q,
- Z is the zero point which denotes the integer that corresponds to the quantization of a 0 in a floating-point number.

There are two categories of quantization - post-training quantization (PTQ) and quantization-aware training (QAT) - based on the timing of quantization. PTQ involves using full-precision values during neural network training, followed by quantizing computation-related parameters, including weights, inputs, and outputs after training. PTQ is a commonly used quantization method as it is easily implementable. QAT involves quantizing specific parameters during training, which therefore requires retraining the network. This approach is typically employed when post-training quantization results in significant accuracy loss.

2.2. Modern Systems-on-a-Chip (SoC)

Typically, an entire system comprising a central processing unit (CPU) and one or more co-processors, along with several other modules, is referred to as a System-on-Chip (SoC). The CPU is the most important unit in a SoC system. The RISC-V-based CPU has been garnering increasing interest owing to its various advantages such as being open-source and modular in design [17].

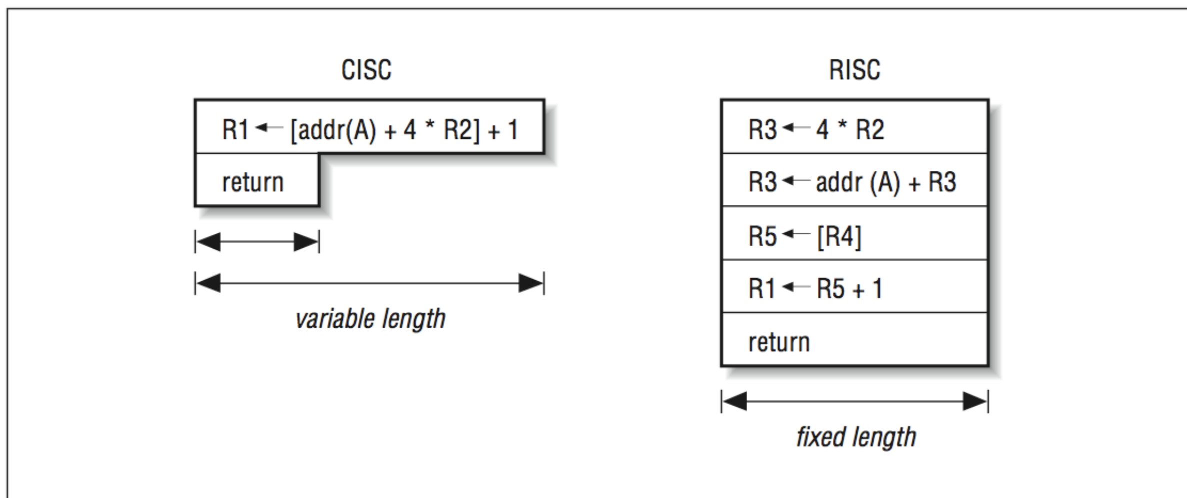


Figure 2.7: The instructions comparison between RISC and CISC in running R1=[addr(A)+4*R2]+1. In RISC, this command is divided into 4 basic instructions while in CISC there is only one complicated instruction. Adapted from [11]

However, a complete neuromorphic SoC entails more than merely a CPU and a neuromorphic coprocessor; other modules are also indispensable. Thus, we select as our platform, the X-Heep microcontroller unit (MCU), which is RISC-V based [45]. This section will outline the specifics of both RISC-V and X-Heep in order to provide a comprehensive understanding of the final neuromorphic SoC.

2.2.1. RISC-V

Instruction set architectures (ISAs) define a computer architecture by specifying the instruction set and the operations performed by those instructions. ISAs are categorized into two main types: the complex instruction set architecture (CISC) [10], represented by the x86 architecture, and the reduced instruction set architecture (RISC) [49], represented by ARM [23], MIPS [27] and so on.

CISC architectures are characterized by having specialized instructions tailored to different functionalities. As a result, CPUs belonging to this type of instruction set perform well in handling special tasks and have high computational power. However, CISC designs tend to suffer from significant power consumption and heat generation issues due to those complex instructions. Characteristics of the above make them more suitable for high-performance computing environments such as servers.

In contrast, RISC aims to keep it simple. In RISC, the commonly used instructions are designed to be straightforward and highly efficient, while less frequently used instructions are constructed through combinations of simpler instructions. One example is shown in Figure 2.7: for the same result, the RISC needs five fixed-length common instructions while the CISC needs only two variable-length instructions.

As a consequence, RISC performs worse in implementing specialized functionalities compared to CISC, but makes the cases sense efficient. Despite potentially lower performance in comparison to x86-based computers, RISC architectures possess an advantage in terms of lower power consumption, making them well-suited for mobile

Table 2.1: A list of basic instructions and extensions for RISC-V [53]

Base	Description
RV32E	Base 32-bit ISA with 16 registers
RV32I	Base 32-bit ISA
RV64I	Base 64-bit ISA
RV128I	Base 64-bit ISA
Extension	Description
A	Atomic instructions
B	Bit manipulation
C	Compressed instructions
D	Double-precision floating-point
F	Single-precision floating-point
G	Shorthand for IMAFD extensions
H	Hypervisor extension
J	Dynamically translated languages
L	Decimal floating-point
M	Integer multiplication and division
N	User-level interrupts
P	Packed-SIMD instructions
Q	Quad-precision floating-point
S	Supervisor mode
T	Transactional memory
V	Vector operations

devices and other power-constrained applications.

RISC-V is an open-source ISA belonging to RISC. In RISC-V, the instruction set is designed to be concise, standardized, and extensible, aiming to provide an efficient, flexible ISA. In addition to being open-source, another key feature of RISC-V is modularity. All RISC-V processors support a specific set of functionalities that remain fixed. This inherent modularity facilitates development. Furthermore, based on this foundation, designers have the flexibility to incorporate additional functionalities by adding modules that support specific features. This provides designers with a vast design space, allowing them to tailor different designs according to their requirements. By adding only the necessary modules on top of the basic RISC-V processor, designers can fulfill their specific needs while avoiding redundant modules that may lead to power consumption wastage, thus ensuring both functional completeness and efficient design.

The foundational extension is the integer instruction set (RV32I or RV64I), comprising basic integer arithmetic and data transfer instructions. Additionally, a range of optional standard extensions exist, such as floating-point extension (RV32F and RV64F), vector extension (RV32V and RV64V), and encryption extension (RV32E and RV64E), among others. The supported RISC-V extensions are listed in Table 2.1.

2.2.2. X-Heep

Neural network processors often cannot work independently by themselves, so they need a host CPU to control them. In this project, we chose a RISC-V-core-based MCU as the host to control the neural network processor. Nowadays, there are many open-source RISC-V-based MCUs, such as the Rocket chip [3] from Berkeley University and PULPissimo [46] from ETH Zurich. These kinds of mature platforms involve tons of files and a complex environment setup, which makes it hard to develop custom designs on the existing platform. Considering the design difficulty and flexibility of these platforms, we finally chose a flexible and simple platform instead: X-Heep [45], from EPFL. The X-Heep architecture and padframe are provided in Figure 2.8 and Figure 2.9.

X-Heep is a RISC-V-based MCU that is designed to be configurable and also supports the integration of various neural network processors. The most important feature of X-Heep is that it is highly customizable, which is because the memory size, peripheral modules and other files in X-Heep are generated by Python scripts.

Figure 2.8 shows the default infrastructure of X-Heep, which consists of the following main modules:

- CPU: X-Heep supports the selection of multiple CPUs to meet different application scenarios, among which the available CPUs are IBEX, CV32E40X, and CV32E40P, which are all open-source RISC-V CPUs.
- Bus: X-Heep's on-chip bus protocol is the open bus interface (OBI) protocol [44], a simple protocol that supports one-to-many or many-to-many communication.
- Memory: X-Heep divides the memory into several banks, each with a fixed size of 64KB. The number of instantiated banks is decided according to the size of the memory needed.
- Debug module: X-Heep uses the JTAG protocol for debugging.
- Peripheral module: X-Heep supports a variety of communication protocols to communicate with the outside world, including I2C, SPI, UART, and GPIO, and supports the use of direct memory access (DMA) to control the data transfer between the internal and external memories.
- System control module: This part includes modules such as a programmable platform level interrupt controller (PLIC), a power controller, a timer and other control-signal-related modules.

Figure 2.9 shows the pins exposed to the external from X-Heep, where the top two bits of the GPIO and the I2C are multiplexed. The registers in the Pad Control module determine who uses the pin during operation. This padframe multiplexing allows customizable and flexible accommodation for pad-limited designs.

X-Heep is a tiny MCU, which only includes the necessary basic modules. At the same time, X-Heep is highly customizable. The design of the existing modules can be modified easily according to the designers' own needs, thus allowing for efficient SoCs tailored to their application.

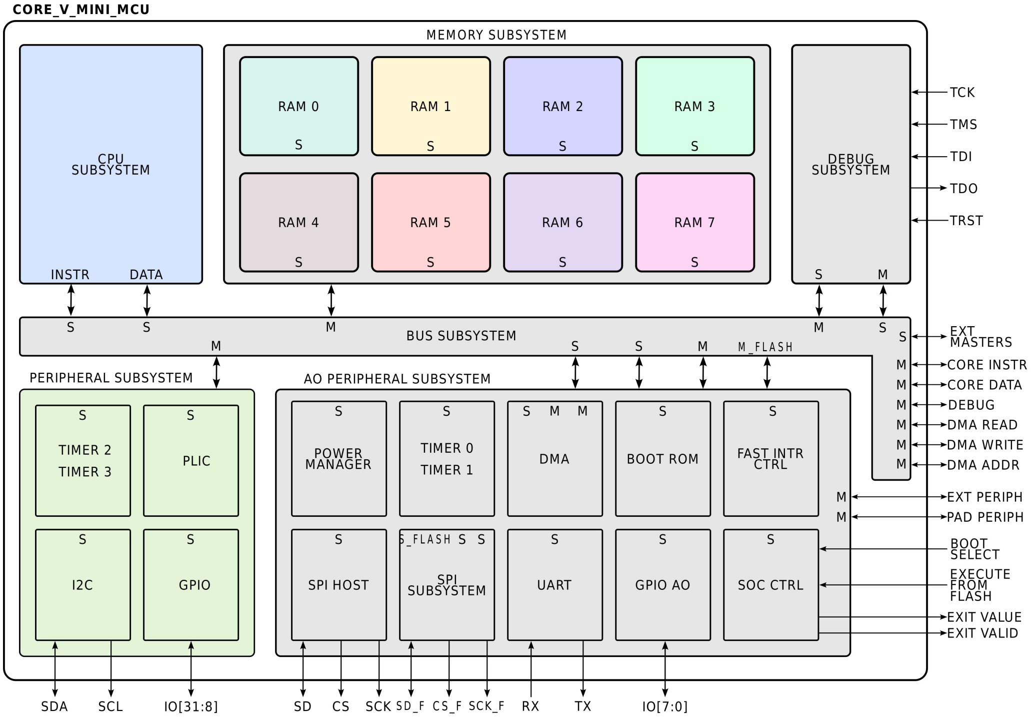


Figure 2.8: The architecture diagram of X-Heep. Adapted from [45]

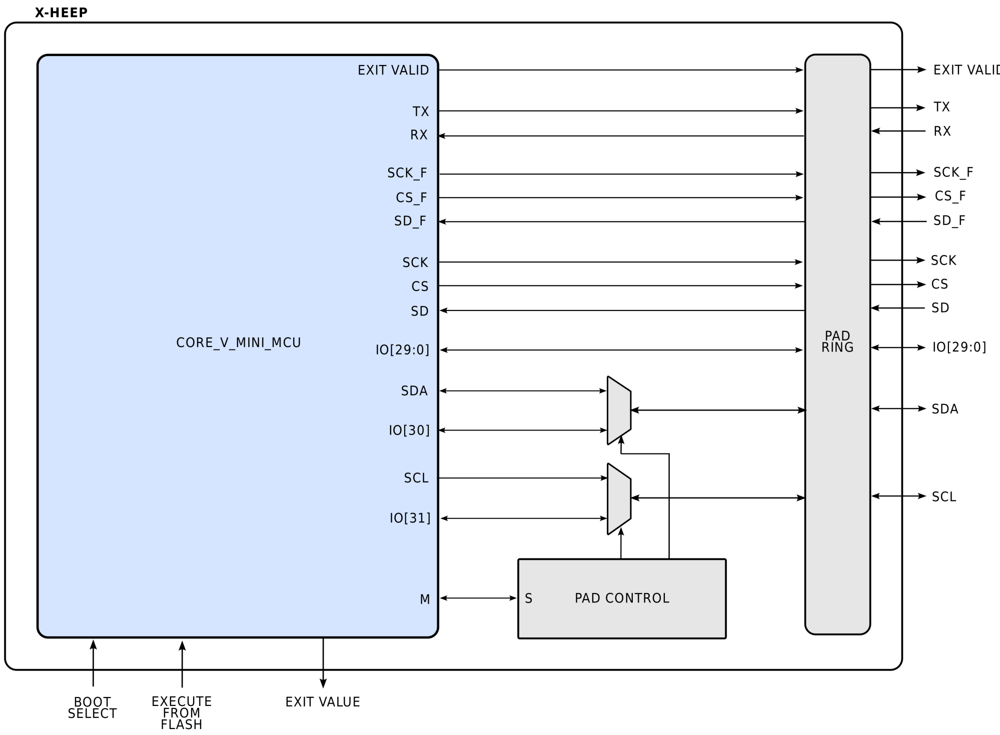


Figure 2.9: The padframe diagram of X-Heep. Adapted from [45]

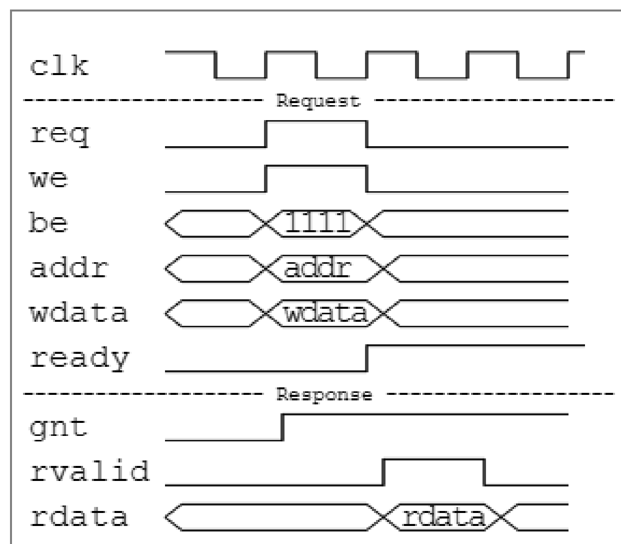


Figure 2.10: One basic request and response transaction between manager and subordinate in the OBI protocol. Modified from [44]

X-Heep Internal Communication

To enable communication between the CPU and other modules in X-Heep, a bus is required to connect the CPU to other devices. The Open Hardware group has developed the OBI bus [44] to connect these CPUs to other modules. The OBI protocol, which is used for point-to-point bus interface on the CV32E40* CPU and related infrastructure components, utilizes a 32-bit bus interface. This protocol is based on request-grant transactions. All modules connected to the OBI bus are assigned to either the manager or the subordinate, where the manager initiates the request and the subordinate returns the response. A typical basic transaction between the manager and subordinate is illustrated in Figure 2.10. The request part of one transaction goes as follows:

- The manager indicates the validity of the request transaction by setting the request (req) signal high, together with other transaction-related signals: write-enable (we), byte-enable (en), address (addr), write data (wdata), etc.
- The subordinate indicates that it can accept the request from the manager by setting the grant (gnt) signal high.
- The request of the transaction starts on the clock rising edge when both req and gnt are high

The response part of one transaction goes as follows:

- The subordinate indicates the validity of the response transaction by setting the response-valid (rvalid) high after the gnt is set high.
- The manager indicates that it can accept the response from the subordinate by setting the ready signal high
- The response of the transaction (rdata) starts on the clock rising edge when both rvalid and ready are high.

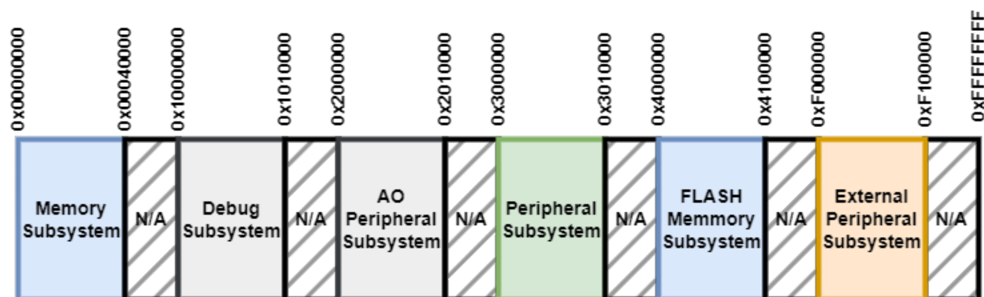


Figure 2.11: The memory map of X-Heep platform. Inspired from [44]

Based on the OBI bus, X-Heep uses Memory-Mapped I/O (MMIO) for communication between the CPU and peripherals. This means that all the components within X-Heep are mapped to specific addresses, allowing the CPU to access all registers and memory space. The memory map for X-Heep is displayed in Figure 2.11, which shows how different subsystems are mapped to distinct addresses. Therefore, the CPU can access the corresponding address to configure all control registers and subsequently configure the identified subsystems as needed. Furthermore, X-Heap offers ample address space to facilitate design customization.

3
Heterogeneous Neuromorphic SoC

In this chapter, we present the design details of the neuromorphic SoC, encompassing hardware and software designs. Employing the concept of software-hardware co-design, we initially trained an SNN model based on the work of [15], and then simplified the scheme for hardware implementation to enable low-cost computation on hardware. Following this, we developed an SNN coprocessor grounded on the pre-trained SNN model. Finally, the SNN co-processor was integrated into X-Heep, thus creating the proposed neuromorphic SoC. This enabled us to demonstrate a proof of concept for hardware-in-the-loop learning.

To gain a deeper understanding of the software design, we will first present the initial work of [15] as preliminary information. Subsequently, we will detail our proposed contributions to the software design, the hardware design, and the final SoC design.

3.1. Preliminaries

In this section, we will discuss the prerequisite knowledge required for software design. The prerequisite knowledge comprises three parts, namely: encoding of the input dataset using the TTFS encoding, the SNN neuron model, and the training algorithm for SNNs based on the TTFS encoding [15].

3.1.1. Time-to-First-Spike Encoding on MNIST Dataset

To utilize an SNN model with the standard MNIST dataset of handwritten digits, TTFS encoding is applied during preprocessing. This is because the input of SNNs is based on spikes, which is different from the data format of the standard MNIST dataset. The encoding procedure modifies the format of a standard MNIST sample to spikes, as shown in Figure 3.1. Each sample consists of 28x28 pixels, with grayscale intensity represented by each pixel assigned a byte value (0-255). A byte value of 255 represents black, while a byte value of 0 represents white. After encoding with TTFS, each pixel's value is mapped to the time of one spike, with larger values indicating earlier spikes.

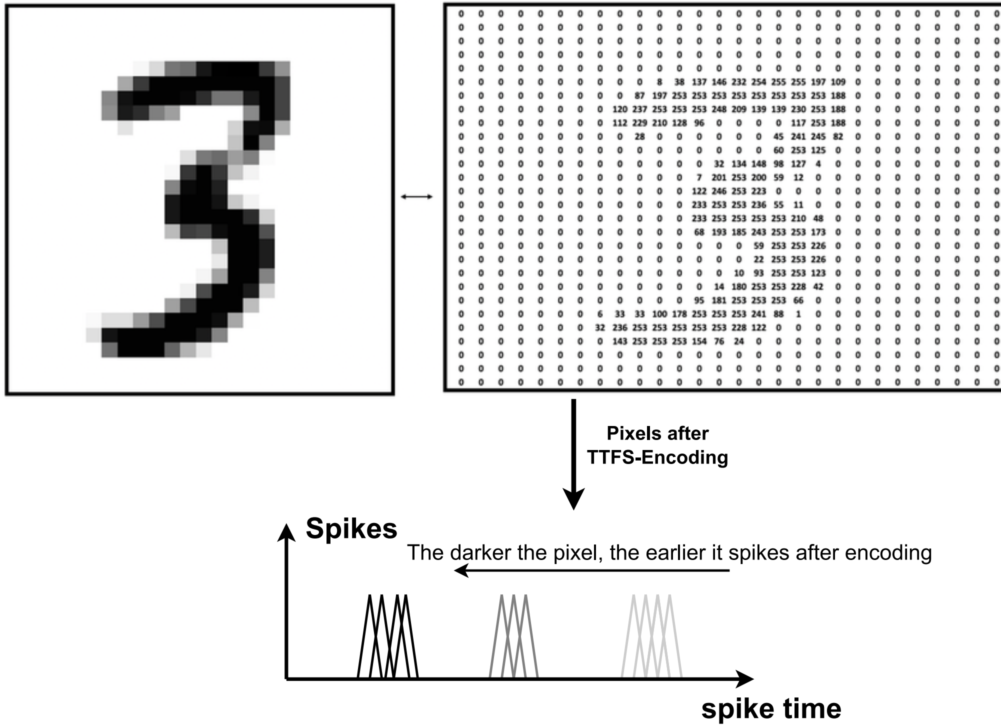


Figure 3.1: A sample "3" in the MNIST dataset and its matrix representation is encoded using the TTFS method. The encoding results in darker pixels spiking earlier, with the color of the spike indicating the grayscale of the corresponding pixel. Modified from [5].

$$T_{spike} = (-\frac{V_{pixel}}{255} + 1.0) * 0.9 + 0.1 \tag{3.1}$$

The TTFS encoding function is presented in Equation 3.1. Following TTFS encoding, T_{spike} denotes the spike time of the neuron for the corresponding pixel with a value of V_{pixel}. The encoding equation can be explained as follows: the dataset is divided by 255 to normalize it within the 0-1 range, and then the outcomes are flipped and scaled to the range of 0.1-1, as demonstrated in Figure 3.2. The range of 0.1 to 1 is selected for ease of computation and following quantization, it corresponds to an arbitrary time unit. This is because we utilize Pytorch for software training, and in Pytorch, the inputs undergo regularization to the 0-1 range. Moreover, we purposefully avoid setting the spike time to 0 after the pixel's encoding, as our intention is for the spike to be generated subsequent to the start of the inference process, rather than coinciding with the start of the inference process. This facilitates the subsequent quantization operation.

3.1.2. Spiking Neuron Model

In the background section, the equations of the basic LIF model (Equation 2.5) were presented. However, in the basic LIF model, spikes generated early do not have a dissimilar effect compared to those generated later, despite temporal differences. For this reason, the authors in [15] applied the LIF model with current-based (Cuba) synapses. The dynamics of this Cuba-LIF model are outlined in Equation 3.2 [15].

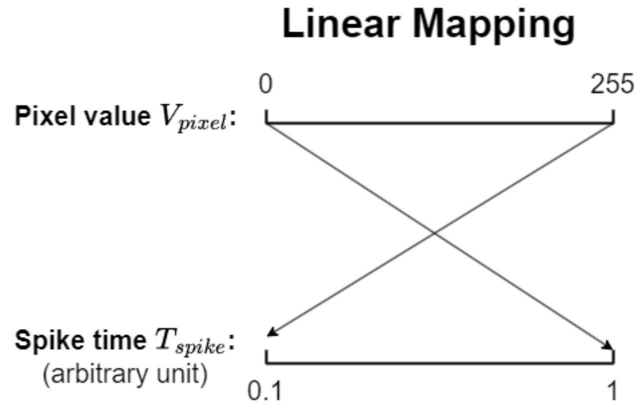


Figure 3.2: Linear mapping between the pixel value and the spike time of the TTFS-encoding of equation 3.1

In contrast to the basic model, the left-hand side of the Equation 3.2 no longer portrays the membrane potential; rather, it displays the derivative of the membrane potential. The Cuba-LIF model's right-hand side incorporates a leakage term as well as a novel element, the synaptic current, which denotes the temporal aggregation of all spikes originating from the preceding layer upon the neuron's membrane potential.

According to Equation 3.2, it is evident that in the Cuba-LIF model, alterations in membrane potential depend on the relation between the magnitude of the leakage function and the synaptic current. Specifically, when the synaptic current surpasses the leakage function for a given Cuba-LIF neuron, the membrane potential increases; otherwise, it decreases. Changes in synaptic currents are dependent on presynaptic spikes. When the membrane potential is above the threshold voltage, the neuron generates a spike and then resets the membrane potential to V_{reset}, which is the same as the basic LIF model. Figure 3.3 illustrates a comparison of these two types of neuron models.

$$\dot{V}_{membrane}(t) = \underbrace{\frac{g_l[E_l - V_{membrane}(t)]}{C_m}}_{Leakage\ \ Function:\ V_{leak}(t)} + \underbrace{\frac{1}{C_m}\sum_i w_i \sum_{t_i} \theta(t - t_i) exp(-\frac{t - t_i}{\tau_s})}_{Synaptic\ \ Current} \tag{3.2}$$

In Equation 3.2, $V_{membrane}$ denotes the membrane potential of the neuron, g_l, E_l, C_m and τ_s are all constants. All symbols are defined as follows:

- g_l is the leak conductance,
- E_l is the leak potential,
- C_m is the membrane capacitance,
- τ_s is the synaptic time constant and we regard C_m/g_l as membrane time constant τ_m,
- w_i indicates the synaptic weight from input neuron i,
- θ function is the Heaviside step function,

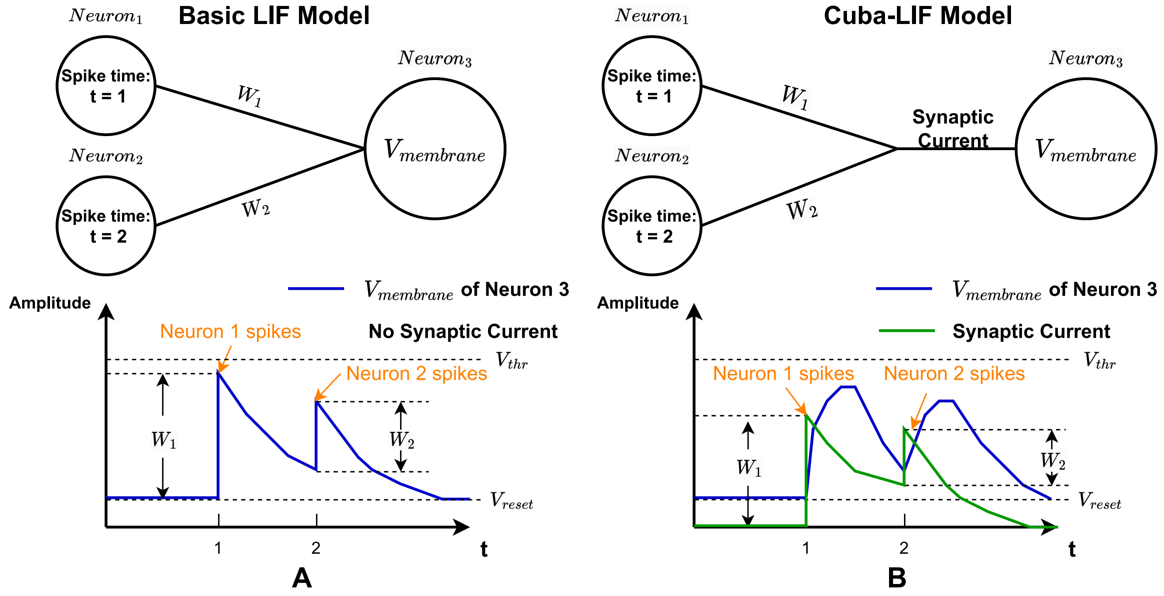


Figure 3.3: The dynamics of $V_{membrane}$ and synaptic currents differ between the basic LIF model and the Cuba-LIF model. Panel A depicts the basic LIF model, where the membrane potential depends on the input spikes. The membrane potential increases when the pre-synaptic neurons spike, and leaks through time. Panel B displays the Cuba-LIF model, in which each neuron has its own synaptic current that increases after a pre-synaptic neuron spikes. The membrane potential changes depend on the relationship between synaptic current and leakage rate.

- i denotes the index of presynaptic neurons,
- t_i denotes the presynaptic spike times of presynaptic neuron i.

The correlation between τ_m and τ_s (from $\tau_m \ll \tau_s$ to $\tau_m \gg \tau_s$) influences the behaviour of the neurons. In [15], the authors present the computation for various scenarios and show that the case where τ_m=τ_s is simpler than the others (we refer the reader to [15] for details). For ease of implementation on hardware, we prioritized this scenario for our project and make the assumption that τ_m and τ_s are equivalent.

3.1.3. TTFS-based Training

In this TTFS encoding, the neuron with the earliest spike has the highest value, which means that the first spiking neuron in the output layer indicates the result inferred by the network. Therefore, the SNN is trained to prioritize the target neuron to spike first among the output layer neurons.

To realize this, [15] took the cross-entropy function as the loss function. The loss function, L, is shown in Equation 3.3.

$$L[t^{(N)}, n_*] = dist(t^{(N)}_{n^*}, t^{(N)}_{n\neq n^*}) = log[\sum_n exp(-\frac{t^{(N)}_{n\neq n^*} - t^{(N)}_{n^*})}{\xi\tau_s})] \tag{3.3}$$

All symbols in Equation 3.3 are defined as:

- N is the number of neurons in the output layer.
- $t^{(N)}$ is the spike times of the neurons in the output layer.

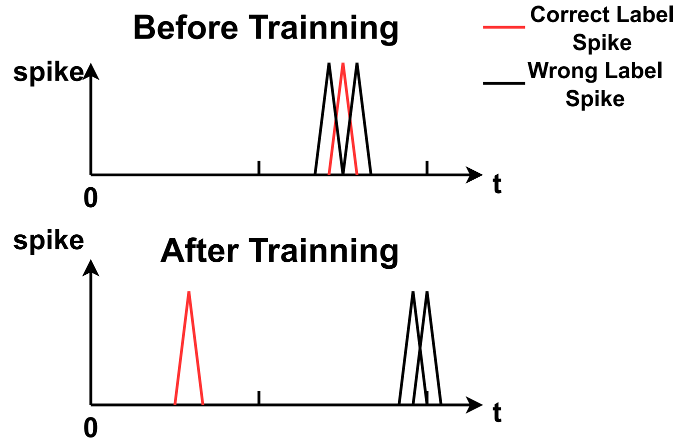


Figure 3.4: The spike times of neurons in the output layer compared before and after training. Prior to training, the spike times of all neurons in the output layer show little variability and were indistinguishable. However, after training, the neuron related to the correct label produces the earliest spike, resulting in a significant temporal difference between its spike and those of the other neurons.

- n_* is the index of the correct label neuron in the output layer.
- ξ is a constant used to scale the result.

The cross-entropy loss function is chosen to maximize the time difference between the spike of the output neuron associated with the correct label and all other output spikes, while facilitating the generation of spikes by the correct target neuron as early as possible. The impact of this training is illustrated in Figure 3.4.

For the computation of the loss function, it is imperative to ascertain the spike times of each neuron in the output layer. These spike times can be directly derived through mathematical calculations, as outlined below. In Equation 3.2, the differential equation of $V_{membrane}(t)$ is given. We can determine the spike time, denoted T, by solving this differential equation and then equating it to the threshold voltage V_{th}, whose result is shown in Equation 3.4 (we refer the reader to [15] for the full derivation):

$$T = \tau_s \left\{ \frac{b}{a_1} - W \left[-\frac{C_m V_{th}}{\tau_s a_1} exp(\frac{b}{a_1}) \right] \right\} \tag{3.4}$$

where

$$a_1 = \sum_{i \in C} w_i exp(\frac{t_i}{\tau_s}) \qquad b = \sum_{i \in C} w_i \frac{t_i}{\tau_s} exp(\frac{t_i}{\tau_s}) \tag{3.5}$$

and W is the Lambert function, defined as:

$$zW'(z) = \frac{W(z)}{W(z) + 1} \tag{3.6}$$

The spike time T is being obtained and now we need to update the weights using the backpropagation algorithm. First, we must compute the derivative of the output spike time with respect to weights ($\frac{\partial T}{\partial w}$) to update them. Additionally, we must compute the

derivative of output spike time with respect to input spike time ($\frac{\partial T}{\partial t_{input}}$) to apply the chain rule, as discussed in Section 2.1.2. According to Equation 3.4, both derivatives can be computed as:

$$\frac{\partial T}{\partial w_i}(w,t,T) = -\frac{1}{a_1}\frac{1}{W(z)+1}exp(\frac{t_i}{\tau_s})(T-t_i) \tag{3.7}$$

$$\frac{\partial T}{\partial t_i}(w,t,T) = -\frac{1}{a_1}\frac{1}{W(z)+1}exp(\frac{t_i}{\tau_s})\frac{w_i}{\tau_s}(T-t_i-\tau_s) \tag{3.8}$$

where

$$z = a_1 exp\frac{b}{a_1} \tag{3.9}$$

Here only the results are listed for the sake of clarity. We again refer the reader to [15]. for the full derivations.

3.1.4. Simplification of the Training Algorithm

The computation of these two derivatives is challenging and difficult to implement in hardware due to the Lambert function and the exponential operation. To simplify Equations 3.7 and 3.8, [15] proposes to approximate computationally expensive parts with a constant λ as per Equations 3.10 and 3.11:

$$\frac{\partial T}{\partial w_i} \approx -\lambda(T-t_i) \tag{3.10}$$

$$\frac{\partial T}{\partial t_i} \approx -\lambda\frac{w_i}{\tau_s}(T-t_i-\tau_s) \tag{3.11}$$

where

$$\lambda = -\frac{1}{a_1}\frac{exp\frac{t_i}{\tau_s}}{W(z)+1} \approx 0.0192 \tag{3.12}$$

It is worth noting that these approximation derivations now rely solely on spike times and weights added and multiplied together. When we attempted to implement this simplified approach during the training of the [15] network on the MNIST dataset, we observed a decrease in accuracy of approximately 5%. from 97.2% with the original training algorithm to around 92.1 with the simplified training algorithm.

3.2. Software Design

The prerequisites for TTFS-encoded SNNs have been introduced in the previous section, encompassing key aspects such as TTFS encoding applied to the MNIST dataset, the Cuba-LIF model, and the associated training algorithm. An important challenge in hardware implementation arises due to the computational complexity

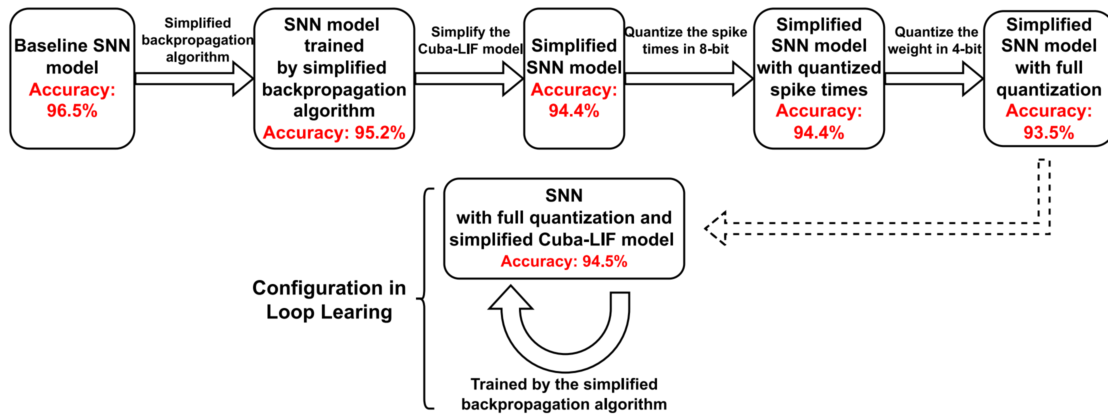


Figure 3.5: The accuracy under each step of software design

of those equations mentioned above, notably the Lambert function and exponential functions.

To address these issues, this section presents three contributions regarding software-hardware codesign. We began by simplifying the backpropagation algorithm, followed by simplifying the Cuba-LIF model. Using the SNN resulting from these two simplifications, we quantized the SNN to obtain an SNN model, that is suitable for loop learning on hardware. Finally, we simulate the software workflow of loop learning to demonstrate the hardware feasibility of loop learning. The work in this section is organized as shown in Figure 3.5.

3.2.1. Baseline

To demonstrate the effect of our work objectively, we present a baseline of the SNN model first.

The MNIST dataset's original sample size is 28×28, leading [15] to employ a 784-350-10 SNN architecture, consisting of 784 neurons in the input layer, 350 neurons in the hidden layer, and 10 neurons in the output layer.

Due to hardware resource limitations discussed in Section 3.3, the total number of neurons in our SNN cannot exceed 256. This is insufficient to accommodate the full MNIST dataset, which features 28×28 pixels, in the input layer. To overcome this, we selectively chose the center pixels of each sample (outlined in red in Figure 3.6) to match the input layer size. After testing various network architectures, we ultimately selected the 144-100-10 model due to its low accuracy drop. For this architecture, we specifically utilized 12×12 pixel central area. The SNN model's architecture and the mapping of the MNIST dataset to the input layer are depicted in Figure 3.6. Although we lost 81% pixels by cropping the sample, the accuracy only decreased from 97.2% to 96.5% compared to the original network architecture, which is acceptable. Therefore, in the next sections, we regard the baseline accuracy as 96.5%.

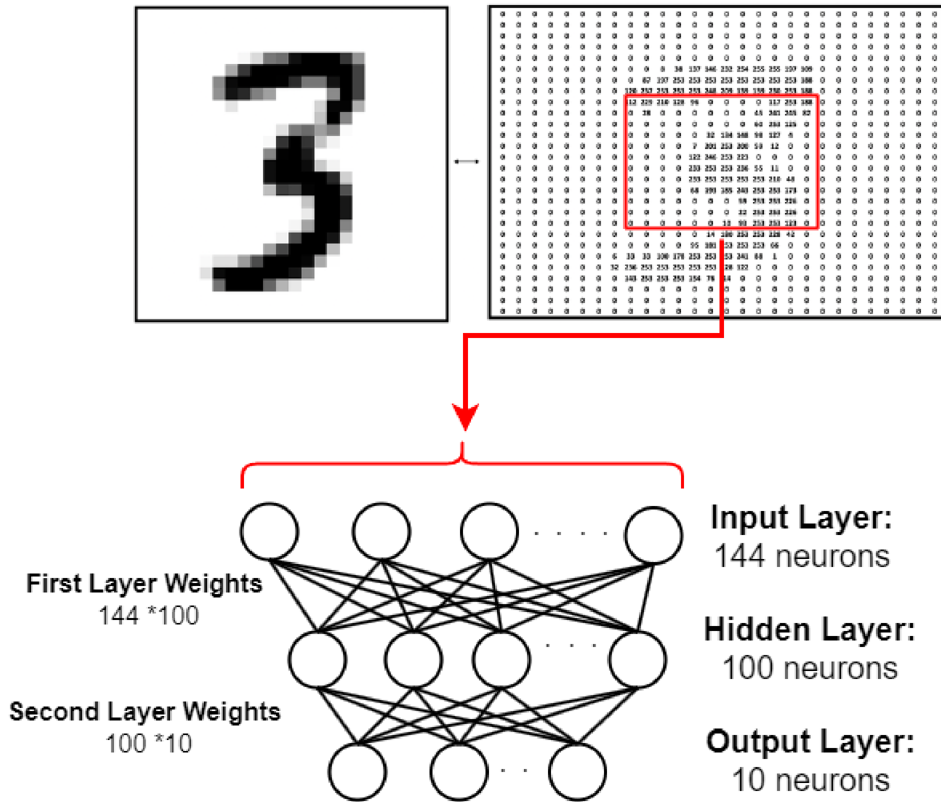


Figure 3.6: The 144-100-10 SNN architecture with cropped MNIST dataset mapped to the input layer. The red-framed pixel points corresponded to the SNN network's input layer in a tiled pattern. For example, the first row's 12 pixels correspond to the first 12 neurons of the input layer, and the second row's 12 pixels correspond to the 13_{th}-24_{th} neurons of the input layer, and so on.

3.2.2. Simplified Backpropagation Algorithm

In Section 3.1, we outlined the backpropagation calculation for TTFS-Encoding. In the loop learning setup on hardware, the RISC-V CPU will perform the backpropagation task. However, this computation places a significant burden on the CPU due to the exponential functions and the Lambert function. While [15] presents a simplification method, as detailed in Section 3.1.4, it is important to note that this simplification results in a significant drop in accuracy, which is mentioned in Section 3.1.4. To address this, we present six approaches (from Method 1 to Method 6) for simplifying the algorithms and comparing their outcomes with the baseline.

Method 1

The method proposed in [15] that simplifies the backpropagation algorithm is discussed in Section 3.1.4. In this method, the original algorithm's term $-\frac{1}{a_1}\frac{exp\frac{t_i}{\tau_s}}{W(z)+1}$ is replaced with a constant value of λ=0.0192 (Equation (3.10) and Equation (3.11)). We implemented this simplification to our SNN with the 144-100-10 architecture while updating the weights of both layers in the network during backpropagation. However, this simplified method resulted in a 5.4% drop in accuracy compared to the baseline after training, achieving only 91.1%.

Although Method 1 produced a significant decrease in accuracy, the strategy of

substituting the intricate function with straightforward expressions shows potential because it reduces the algorithm to fundamental additions and multiplications. This simplification is easily applicable to the RISC-V CPU.

Method 2

Since the objective of simplification is to substitute a complex function in the original equations with a more straightforward expression, we propose in Method 2 to simplify Lambert's function via Taylor expansion to avoid solving its differential equation.

$$W(z) \approx z - z^2 \tag{3.13}$$

This simplification proposed in Equation 3.13 achieves 95.9% accuracy, only 0.6% less than the baseline accuracy, without using the Lambert function. However, calculating z in Equations 3.5 and 3.9 requires multiple exponential calculations.

Method 3

To eliminate the computation of exponential operations entirely, we expanded on Method 1. As shown in Equation 3.14 (reminder of Equation 3.12, where z is a function of w_i and t_i), λ can be considered a function of the input spike times and weights without the need to replace the Lambert function with a Taylor expansion. To better illustrate the relationships, we created a 3D scatterplot depicting the variability of λ in relation to input spike times and weights used in backpropagation. The scatterplot reveals differing amplitudes of λ between the two layers. The range of λ in the first layer appears to be larger compared to the second layer.

$$\lambda = -\frac{1}{a_1} \frac{exp\frac{t_i}{\tau_s}}{W(z) + 1} \tag{3.14}$$

Therefore, unlike Method 1, which employs a single fixed λ for both layers, we implemented Method 3 using two λs in two layers, layer 1 and layer 2, for backpropagation. The two λs were obtained by averaging all points separately, where $\lambda_{Layer1} = -0.26$ is set when updating the weights of the first layer and $\lambda_{Layer2} = -0.048$ is set when updating the weights of the second layer, as shown in Table 3.1. With Method 3, the model can achieve an accuracy of 92.6%.

Table 3.1: The different λs for two layers

Layer	λ
First Layer	-0.26
Second Layer	-0.048

Method 4

Starting with the assignment of different values of λ for two layers, Figure 3.8 shows the scatterplot of λ against the weight. This scatterplot revealed that λ approached 0 when weight was nonzero. When weight equalled 0, λ fluctuated between 0 to -8 and 0 to -0.12 in layer 1 and layer 2 respectively.

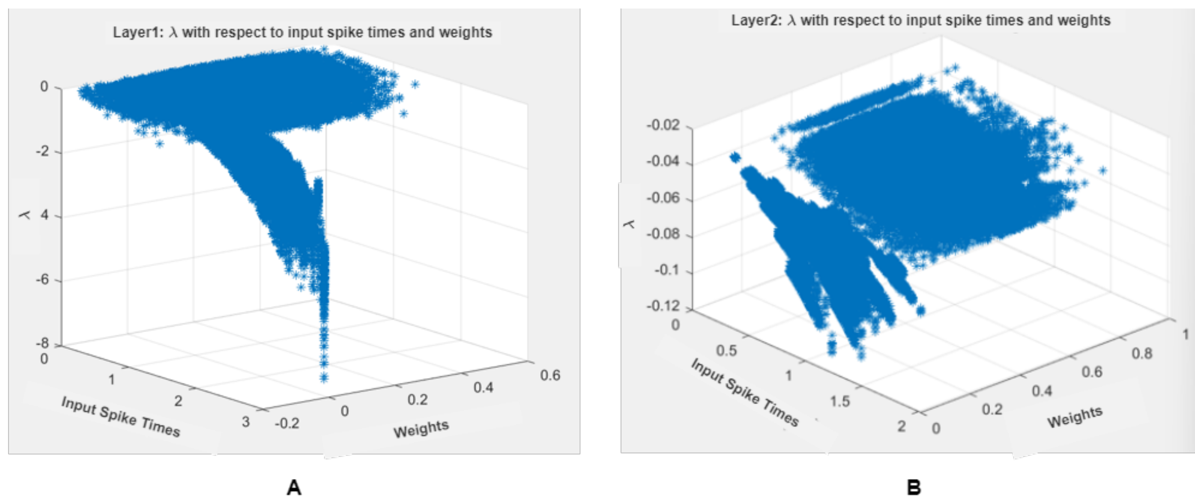


Figure 3.7: Scatterplot of λ distribution under the original algorithm with respect to the input spike times and weights in the two layers Panel A is the λ obtained when updating the first layer weights and panel B is the λ obtained when updating the second layer weights.

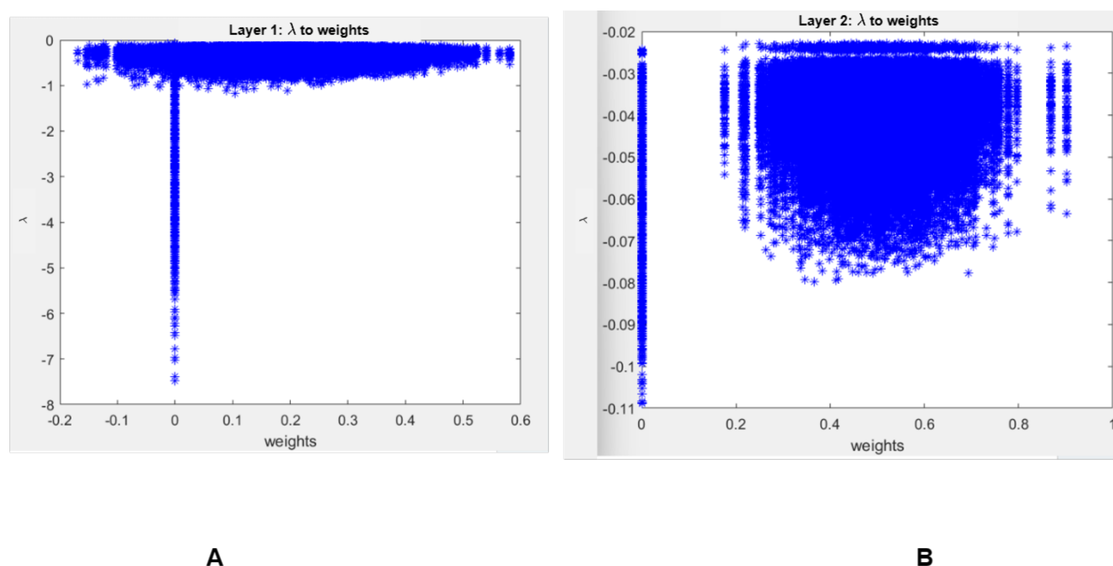


Figure 3.8: Scatterplot of λ under the original algorithm with respect to the weights in the two layers. Panel A is the λ from the first layer and panel B is the λ from the second layer.

Therefore, in order to fit this distribution, for cases where weights were nonzero, the λ value was selected as the average of all of the λ values. Similarly, when the weight equals 0, we use the same procedure to select a different λ value. This process represents Method 4, where we choose four λ values in four cases and tune them to determine the optimal performance. Upon testing various combinations of λs, we achieved a 95.2% accuracy rate using the respective λ values depicted in Table 3.2.

Table 3.2: The different λs under four cases

Layer	λ for zero weights	λ for nonzero weights
First Layer	-0.0482	-0.26
Second Layer	-0.016	-0.048

Method 5 and 6

Expanding on this observation, it becomes evident that the range of λ values exhibits significant variation when the weight is equal to zero. To further illustrate this, we drew a scatterplot illustrating the relationship between λ and the input spike timing in cases where the weights are zero, as displayed in Figure 3.9.

To capture this correlation, we fitted both a linear model (Method 5) and a quadratic polynomial model (Method 6), as presented in Tables 3.3 and 3.4, respectively. Method 5 attained 94.6% accuracy and Method 6 attained 95.0% accuracy. It is important to note that $polyfit$, a built-in function in Matlab, is used to fit lambda based on input spike. A brief scan of the coefficients of each function resulting from this process was conducted. It should be noted, however, that this method does not yield the most optimal fit. We would have expected these methods to achieve a higher level of accuracy than Method 4 since both methods locate lambda closer to its true value. However, the results demonstrate that both methods actually generate worse results instead. It is our belief that this is due to the fact that the best-fitting function as previously mentioned was not found. Due to time constraints, we did not extensively attempt to find the most suitable fit. In addition, we opted not to use higher-order polynomials to avoid unnecessarily complex computations that would detract from the original aim of the simplification.

Table 3.3: The different λ under four cases with linear approximation

Layer	λ for zero weights	λ for nonzero weights
First Layer	-0.0482	$-0.64t_i - 0.15$
Second Layer	-0.016	$-0.04t_i - 0.03$

Table 3.4: The different λ under four cases with quadratic polynomial approximation

Layer	λ for zero weights	λ for nonzero weights
First Layer	-0.0482	$-0.36t_i^2 - 0.06t_i - 0.22$
Second Layer	-0.016	$-0.01t_i^2 - 0.02t_i - 0.03$

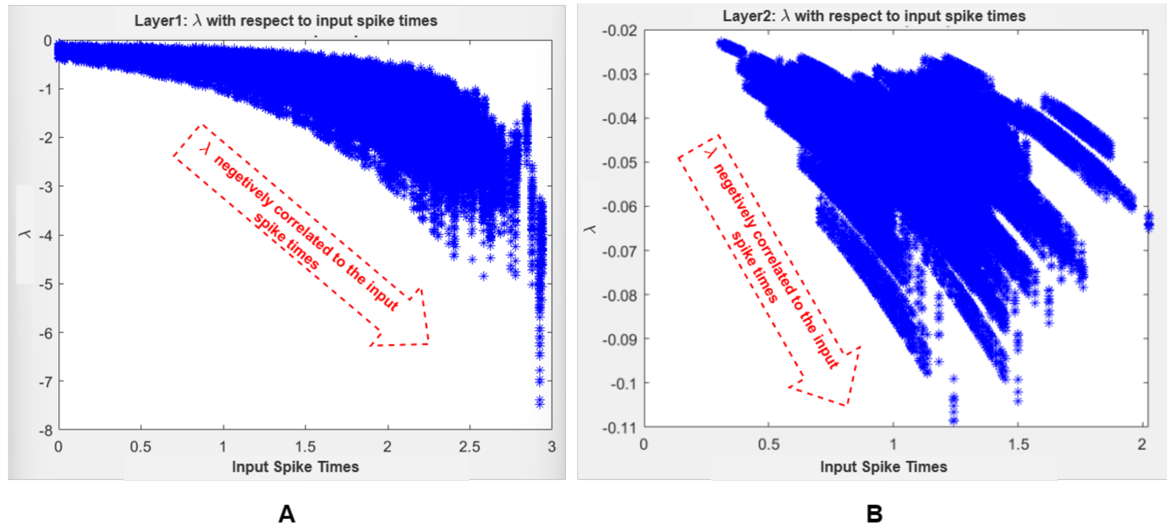


Figure 3.9: Scatterplot of λ under the original algorithm with respect to the time of input spikes when weights equal 0 in the two layers. Panel A is the λ from the first layer and panel B is the λ from the second layer.

Comparison of different Methods

To facilitate comparison between the different methods, we compiled the results of in Table 3.5.

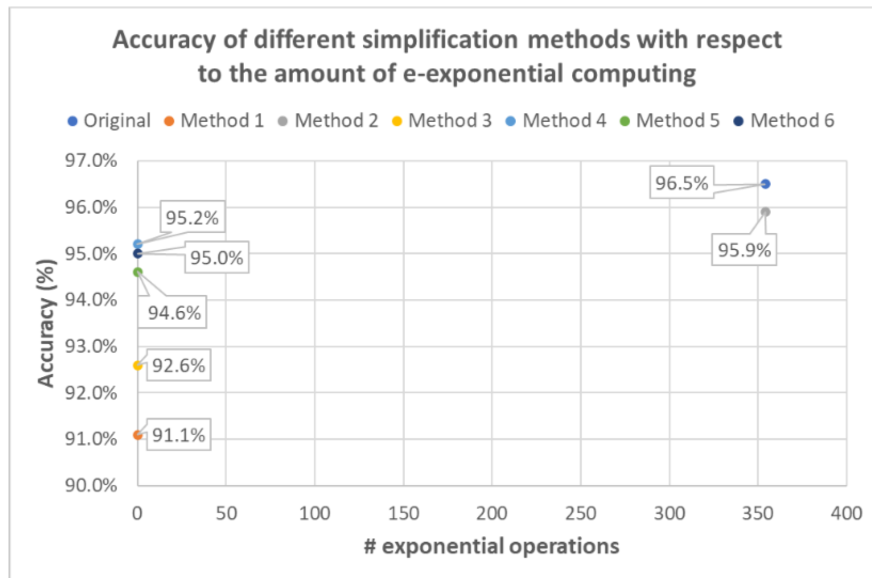
Table 3.5: Accuracies of the SNN under different simplification choices

Accuracy	Method	Simplification Methods
96.5%	Original	Without any simplification
91.1%	Method 1	One fixed λ in all cases [15]
95.9%	Method 2	Talyor expansion of Lambert Function
92.6%	Method 3	Two λs respectively for two layers, independently of the weights
95.2%	Method 4	Four λs for two layers and when weight=0 and weight≠0 respectively
94.6%	Method 5	Two λs respectively for two layers when weight≠0 and linear approximation of λ when weight=0
95.0%	Method 6	Two λs respectively for two layers when weight≠0 and quadratic polynomial approximation of λ when weight=0

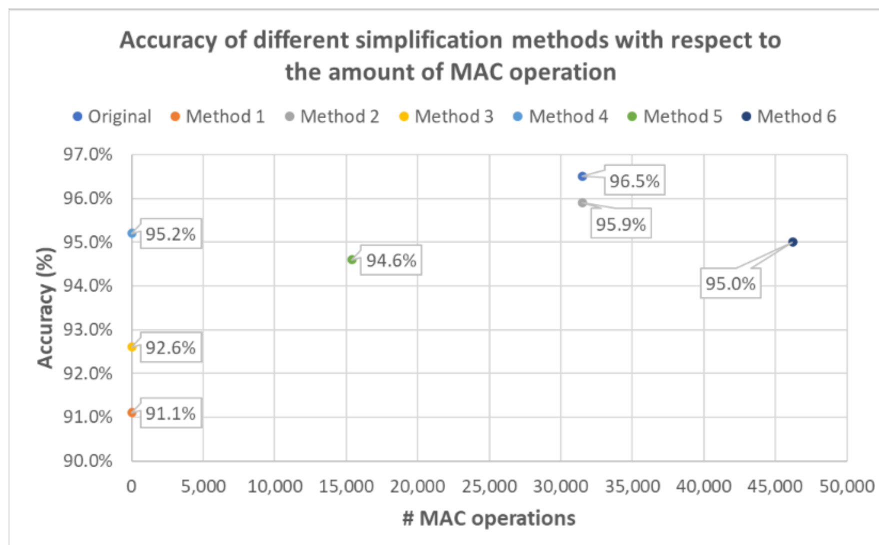
Furthermore, in order to find the balance between the computational burden and the accuracy. We show in Figure 3.10 the tradeoff between accuracy and the number of MAC and exponential operations. This translates to a better tradeoff the closer we are to the top left point in both diagrams. Accordingly, our final choice is Method 4, which does not require computing exponential functions and has the fewest MAC operations with the second-highest accuracy among the six methods with 95.2%, which is close to the baseline of 96.5% at a much reduced computational cost.

3.2.3. Simplified Cuba-LIF Model

In this part, we simplify the original Cuba-LIF model in two steps. First, we impose a constraint that restricts all neurons to firing at most once. Second, we eliminate the leakage feature to streamline the computational process. Subsequently, we apply this simplified Cuba-LIF model to the SNN that was trained in Section 3.2.2. These simplifications are aimed at reducing model complexity and computational overhead while maintaining accuracy.



A



B

Figure 3.10: Scatterplot of the accuracies of different simplification methods with respect to the amount of exponential operations (panel A) and MAC operation (panel B), related to Table 3.5

One-Spike Limitation

In the original Cuba-LIF model, after a neuron generates a spike it can begin to accumulate membrane potential again if there is a non-zero synaptic current. Consequently, it is possible for a single neuron to produce multiple spikes within a single inference, which is not necessary when the TTFS code is used.

Nonetheless, during the inference phase, our aim is to minimize the occurrence of spikes, as fewer spikes translate to reduced computational load. Consequently, we imposed a constraint that limits all neurons in the network to emitting at most one spike during a single inference by disabling them as soon as they spike. Figure 3.11 provides a visual comparison of the membrane potential dynamics with and without this limitation.

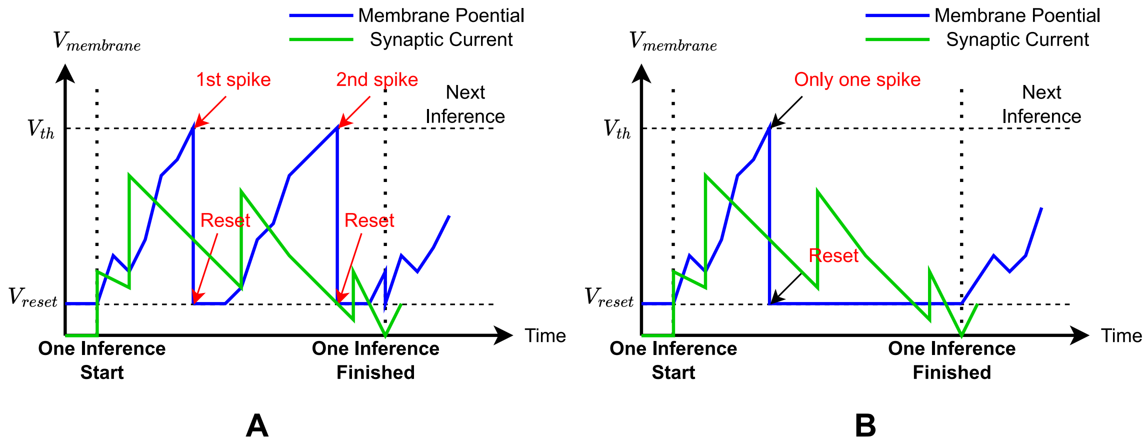


Figure 3.11: The comparison of the membrane potential with (panel A) and without (panel B) the one-spike limitation. Without the limitation, one neuron might spike several times in one inference, with the limitation, one neuron can only spike once in one inference.

The mathematical equation is simplified by the removal of the second sum in Equation 3.2. The updated equation with a one-spike limit is displayed in 3.15. The revised equation eliminates the second sum over t_i times in the synaptic current term as there is only a single t_i value per neuron.

$$\dot{V}_{membrane}(t) = \underbrace{\frac{g_l[E_l - V_{membrane}(t)]}{C_m}}_{Leakage\ \ Function:\, V_{leak}(t)} + \frac{1}{C_m}\underbrace{\sum_i w_i \theta(t - t_i) exp(-\frac{t - t_i}{\tau_s})}_{Synaptic\ \ Current} \tag{3.15}$$

No-Leakage Simplification

In Equation 3.15, two leakage terms are present. The first one, $\frac{g_l[E_l - V_{membrane}(t)]}{C_m}$, is utilized for computing the leakage of the membrane potential. The second one, $exp(-\frac{t-t_i}{\tau_s})$ is utilized for computing the leakage of the synaptic current. These leakages take place continuously within the timeline, requiring significant computing resources to record the process of each computational time step. Here, we eliminate the leakage by setting g_l to 0 and then τ_s to infinity. As a result, the leakage function becomes 0 and

the $exp(-\frac{t-t_i}{\tau_s})$ equals 1, indicating that the synaptic current and membrane potential do not decay over time. The visual comparison is shown in Figure 3.12.

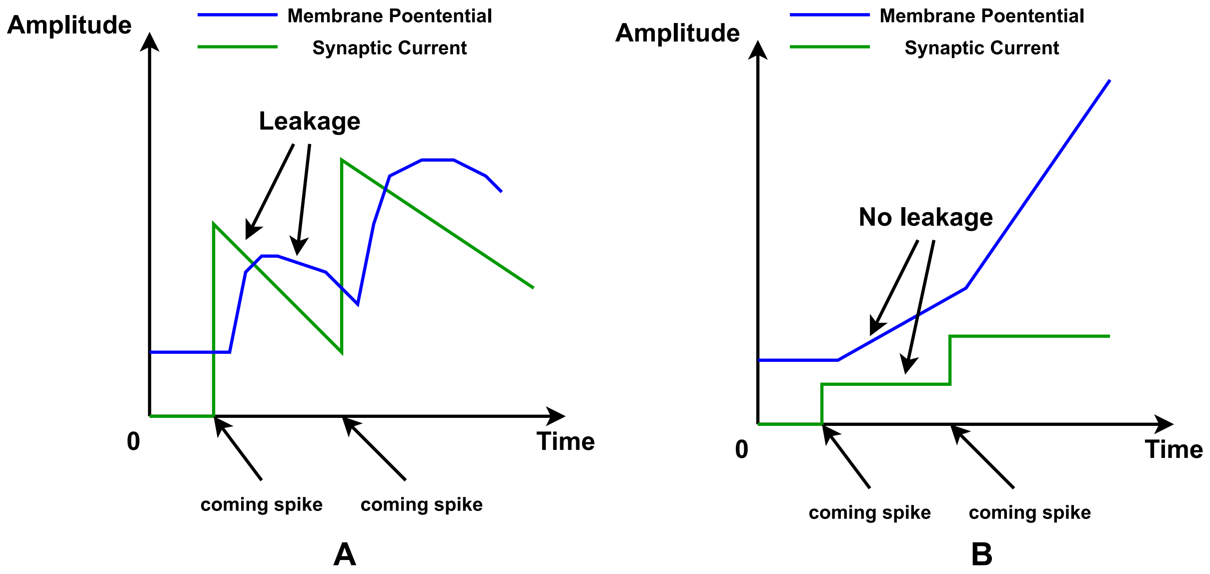


Figure 3.12: The comparison of the membrane potential and synaptic current with and without leakage. Panel A is with the leakage and panel B is without leakage. With the leakage, the synaptic current decays with time if there is no new spike coming and the membrane potential decays if the synaptic current is lower than the leakage speed, but without the leakage, the synaptic and membrane potential will not decay.

With this restriction, the equation for the membrane potential is now expressed in Equation 3.16. The leakage function is set to 0, and the exponential calculation within the synaptic current item has been eliminated. This simplified formula is more computationally efficient compared to the original Equation 3.2.

$$\dot{V}_{membrane}(t) = \underbrace{\frac{1}{C_m}\sum_i w_i \theta(t-t_i)}_{Synaptic\ \ Current} \tag{3.16}$$

Mathematical expressions for the simplified Cuba-LIF model

Using Equation 3.16, we can derive an equation for the membrane potential as expressed in Equation 3.17. Here, $V_{membrane}(t_0)$ represents the initial membrane potential value, equivalent to the value of V_{reset}.

$$V_{membrane}(t) = \underbrace{V_{membrane}(t_0)}_{V_{reset}} + \frac{1}{C_m}\sum_i w_i(t-t_i)\theta(t-t_i) \tag{3.17}$$

This expression signifies that the membrane potential equals the cumulative sum of all presynaptic spikes integrated over time. This simplified formula is intuitively comprehensible, and in Section 3.3, we will provide a detailed elaboration on how to simulate this model and its computational process through hardware design.

Results

In this section, we introduce two simplification techniques for streamlining the basic Cuba-LIF model: the one-spike limitation and the removal of leakage. These lead to a simplified membrane potential calculation represented by Equation 3.17.

We applied this simplified Cuba-LIF model to replace the original Cuba-LIF model in the SNN obtained in Section 3.2.2. The SNN model achieved an accuracy of 94.4%, with an accuracy drop of 0.8% compared to that with the original Cuba-LIF model. The accuracy results are presented in Table 3.6. It is worth noting that the network has not been retrained following the substitution of the original Cuba-LIF model with the simplified version. Retraining will be undertaken after the completion of the quantization operation.

To summarize the progress thus far, we integrated the simplified Cuba-LIF model with the backpropagation algorithm, as outlined in Figure 3.5.

Table 3.6: The accuracies of the SNN with the original Cuba-LIF model and with the simplified Cuba-LIF model

SNN Configuration	Original Cuba-LIF model and simplified backpropagation algorithm	Simplified Cuba-LIF model and simplified backpropagation algorithm
Accuracy	95.2%	94.4%

3.2.4. Quantization of the SNN

In the software design phase, the SNN model is trained using PyTorch, which utilizes 64-bit double-precision floating-point numbers to represent all numerical values. However, in order to mitigate computational and memory demands in the subsequent hardware design phase, we implemented PTQ on the SNN model obtained in Section 3.2.3.

With PTQ, we quantized the spike times and the weights within the SNN. This quantization process leads to the corresponding quantization of the membrane potential and synaptic current.

Quantization of Spike Times

In Section 3.1, after applying TTFS encoding, neurons in the input layer generate spikes within the time interval of t=0.1 to t=1. These initial spikes subsequently stimulate neurons in the subsequent layers, resulting in a cascade of spikes. Therefore, it is necessary to first define the time range in order to quantize the spike times accurately.

To this end, we conducted 10,000 inferences on the test dataset using the SNN model outlined in Section 3.2.3 and visualized the spike times of neurons in the output layer in a graph presented in Figure 3.13. The graph demonstrates that more than 85% of neurons in the output layer spike prior to t=2.5. This observation implies that the SNN has effectively produced the inference result by t=2.5. Consequently, we established a simulation time window ranging from t=0 to t=2.5, during which spiking neurons beyond t=2.5 are considered to spike at t=2.5.

We quantized the spike time by mapping the floating-point value 0 to integer value 0 and the floating-point value 2.5 to the maximum integer value, in the same way as the

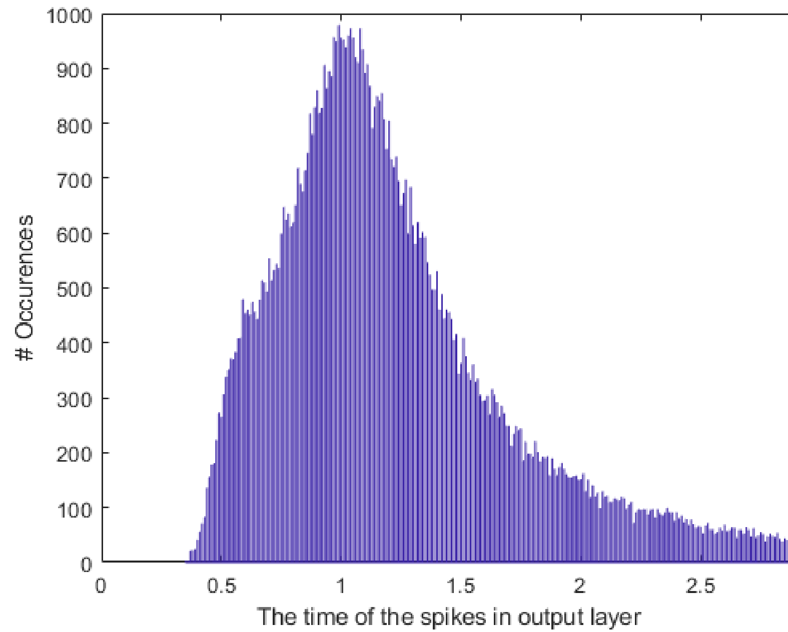


Figure 3.13: Spike times of neurons in the output layer and their corresponding counts.

example in Figure 2.6. This implies that S=2^N-1 and Z=0 in Equation 2.7, where N is the resolution in bits. We compared the accuracies achieved by quantizing spike times to various resolutions of bits, and the outcomes are displayed in Figure 3.14.

Our experimentation has revealed that when 8 bits are utilized to quantize spike times based on the SNN model obtained in Section 3.2.3, the accuracy of the model reaches 94.4%. However, when fewer than 8 bits are employed, a considerable loss of performance is observed. Conversely, increasing the bit precision beyond 8 bits does not yield any improvement in accuracy. Based on these findings, we decided to adopt an 8-bit representation for quantizing spike times.

Quantization of Weights

For the same reason that Pytorch defaults to a 64-bit double-precision floating-point value for weights, we also quantized the weights to a lower bit resolution. It is worth highlighting that this quantization process already takes into account the previous quantization of spike times in the network.

To quantize the weights, it is necessary to first define the weight range. The decision on the truncated range is based on the fact that the larger the range, the more bits are required for quantization at the same precision. Consequently, we aim to minimize the range while ensuring precision. We recorded the weights of the SNN model in Section 3.2.3. Subsequently, the weight distribution is plotted according to Figure 3.15, with the weights on the x-axis and the corresponding occurrence on the y-axis.

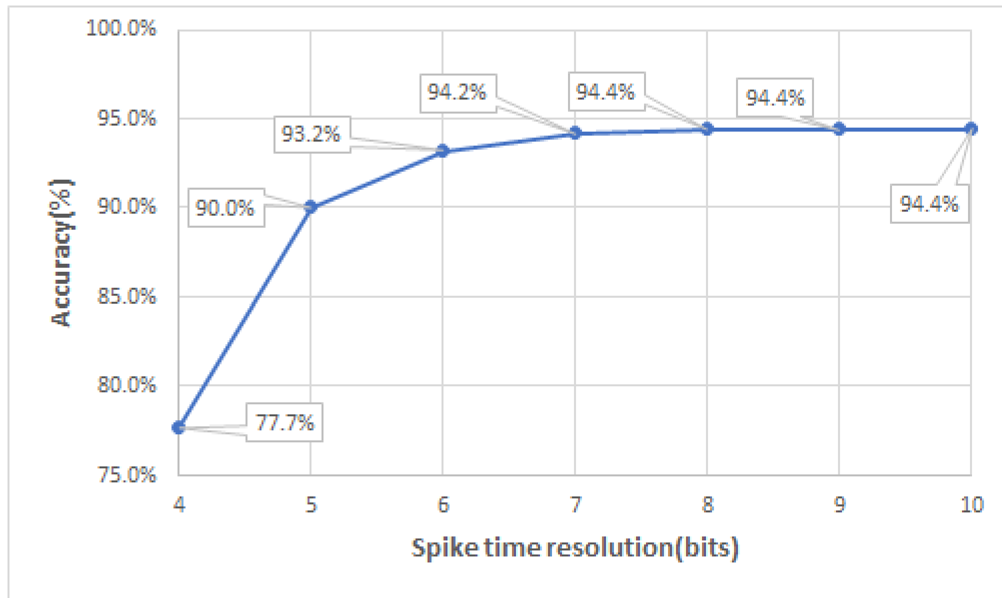


Figure 3.14: Accuracies at different input quantization resolutions, after 8bits, the accuracy doesn't have clear increases.

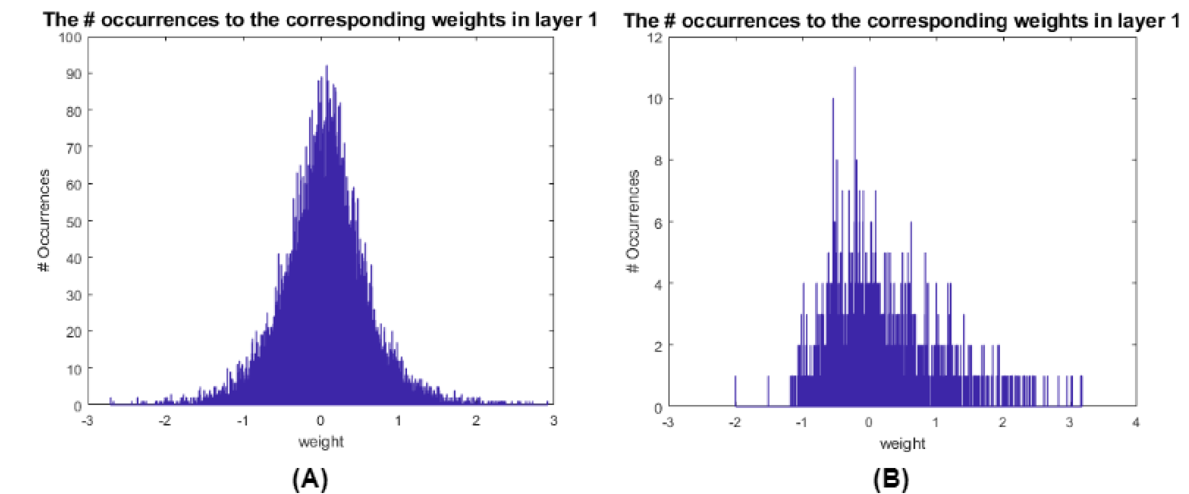


Figure 3.15: The distribution of weights in the SNN obtained in Section 3.2.3. (A) Weights of the first layer. (B) Weights of the second layer.

From Figure 3.15, it is apparent that the weights in this SNN are primarily concentrated around 0, roughly in the -3 to 3 range.

The weight truncation range, [-w_{clip}, +w_{clip}], was determined through a comparison of accuracy using various truncation ranges. Here the symmetric truncation was chosen to prevent the introduction of zero offset, thus facilitating hardware design. Evaluating the accuracy for different weight quantization resolutions necessitates analyzing various ranges of weight truncation. Consequently, we compared the accuracies of varying quantization and truncation ranges. The results are shown in Figure 3.16, which shows that when w_{clip} is set to 2 and 4 bits are used for quantization, the accuracy can reach 93.5%. However, using more bits for quantization only results

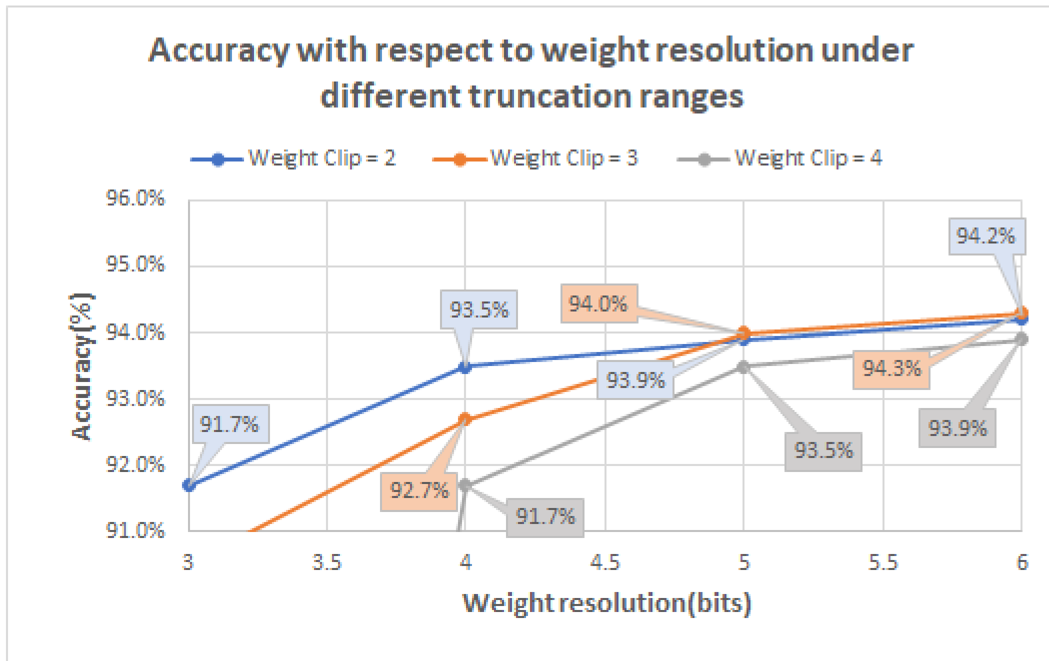


Figure 3.16: Accuracy of the SNN model under different truncation ranges and weight quantization resolutions.

in a minor increase in accuracy, while requiring at least 20% more memory resources on the hardware. Thus, to balance resource consumption and accuracy, we chose w_{clip} as 2 and 4 bits to quantize the weights.

Other Quantization Configurations

After quantizing the weights and spike times, the model attained 93.5% accuracy. We therefore included quantizing the synaptic current and membrane potential.

We accomplished synaptic current quantization by using an 8-bit resolution, which produced satisfactory outcomes without compromising accuracy. We did not examine other resolutions as this is not the dominant factor in our design: SNN's 15,400 weights (4-bit weights) but only 154 synaptic currents (8-bit synaptic currents).

The same principle also applies to the quantization of membrane potential, which is quantized with a 12-bit resolution without compromising accuracy.

Result of Quantization

In the quantization phase, we apply an 8-bit quantization for spike time, 4-bit for weights, 8-bit for synaptic current, and 12-bit for membrane potential for the SNN model obtained in Section 3.2.3. As a result, the accuracy decreases from 94.4% to 93.5%, compared to the SNN model prior to quantization (see Figure 3.5), with the required memory resources reduced by around 16 times. The accuracy so far is as outlined in Figure 3.5.

3.2.5. Loop Learning in Software

After simplifying the Cuba-LIF model and PTQ, we developed a software-based emulation of the loop learning work model to forecast its outcomes on hardware.

During hardware implementation of loop learning, the SNN coprocessor carries out the inference stage to quantize all inference values for resource conservation. The RISC-V CPU carries out backpropagation, maintaining 32-bit precision for accuracy.

The software-based emulation of the loop learning work model is shown in Figure 3.17. All the steps in red will be carried by the SNN coprocessor and the steps in blue will be carried by the RISC-V core. In contrast to the original process outlined in Section 3.1.3, the simplified backpropagation algorithm adjusts the weights based on the simplified Cuba-LIF model and quantized spike times and weights. As a result, the network can learn to adapt to these features during training.

With loop learning, the SNN model achieves an accuracy of 94.5%. The accuracy variation during training is shown in Figure 3.18. It was observed that the accuracy decreases with excessive training time. This is due to the simplification of the backpropagation algorithm, leading to errors in the calculated gradient that diverge from the true gradient, ultimately impeding the training from converging. The Early Stopping technique was used to address this issue. It involves stopping training when accuracy begins to decline, as shown in the figure. As a result, we achieved a training accuracy rate of 94.5% on Loop Learning, demonstrating the method's theoretical feasibility for hardware implementation.

3.2.6. Results of Software Design

To summarize, in the software design module, we simplified the SNN network by the following three steps to reduce the computational amount on the subsequent hardware while maintaining accuracy:

- Simplifying the equations for calculating $\frac{\partial T}{\partial w_i}$ and $\frac{\partial T}{\partial t_i}$ in backpropagation by using four λs to replace the complex term in Equation 3.2 in four cases.
- Simplifying the neuron model by limiting each neuron to produce at most one spike and removing the neuron's leakage.
- Quantizing the spike time, weight, synaptic current, and membrane potential of the SNN using 8, 4, 8, and 12 bits, respectively.

The final SNN model achieves an accuracy of 93.5% after simplification and quantization, which is a 3% drop from the original model's accuracy of 96.5%. Nevertheless, there is an order-of-magnitude decrease in computing and hardware resources needed on hardware.

To predict the performance of loop learning in hardware, the feasibility of loop learning on hardware is verified by running a software-based simulation setup shown in Figure 3.17, resulting accuracy is 94.5% thanks to early stopping (Figure 3.18). The accuracy summary of each step of the software design is presented in Figure 3.5.

3.3. Hardware Design

In this chapter, we developed a neuromorphic SoC that implements the loop learning setup. First, we present the overall architecture of this neuromorphic SoC and then delve into the components. Specifically, we designed a coprocessor for the SNN called

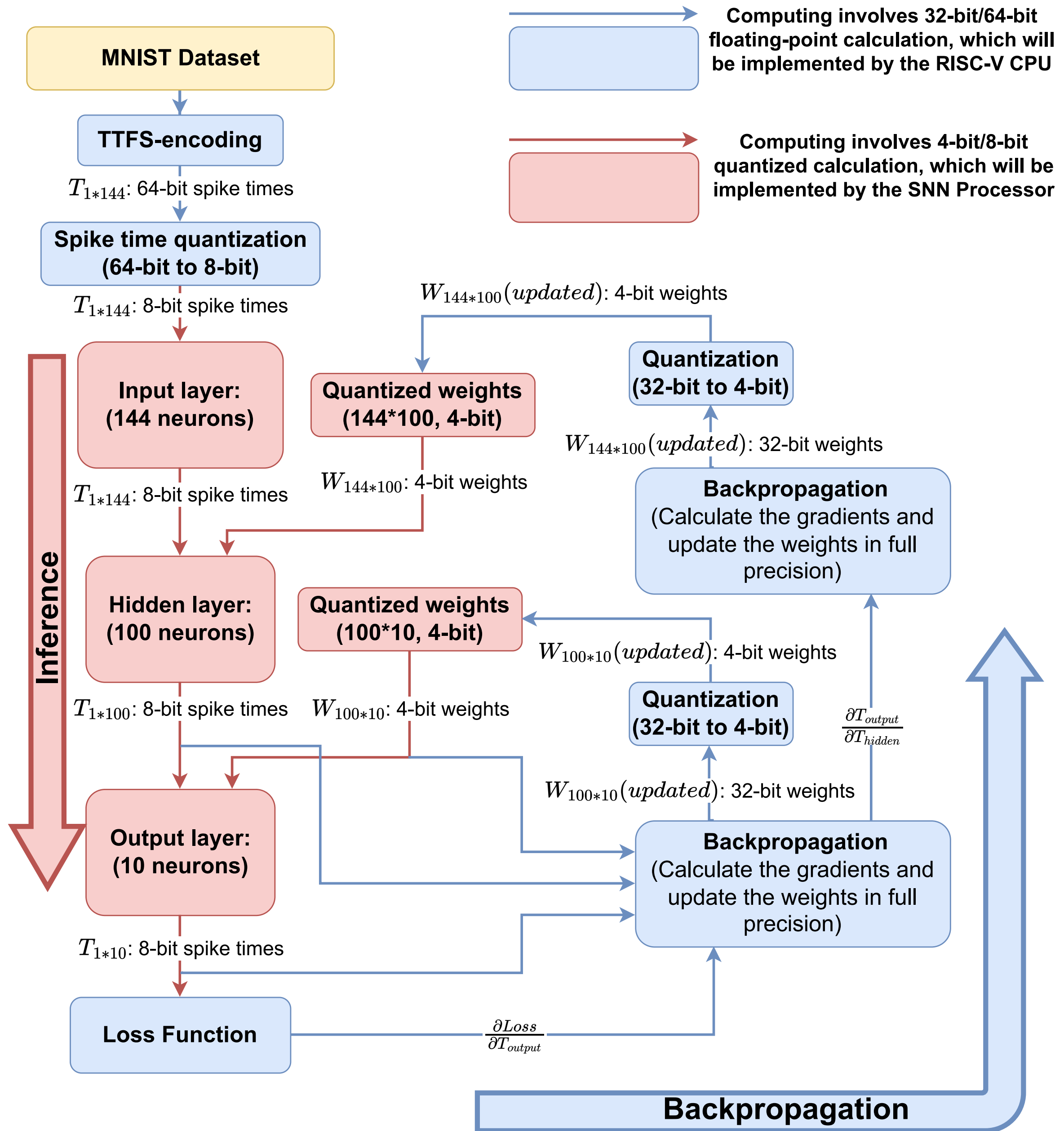


Figure 3.17: Workflow of the simulation of loop learning in software. All the modules in red will be carried out by the RISC-V core in software, and all the modules in blue will be carried out by the SNN coprocessor.

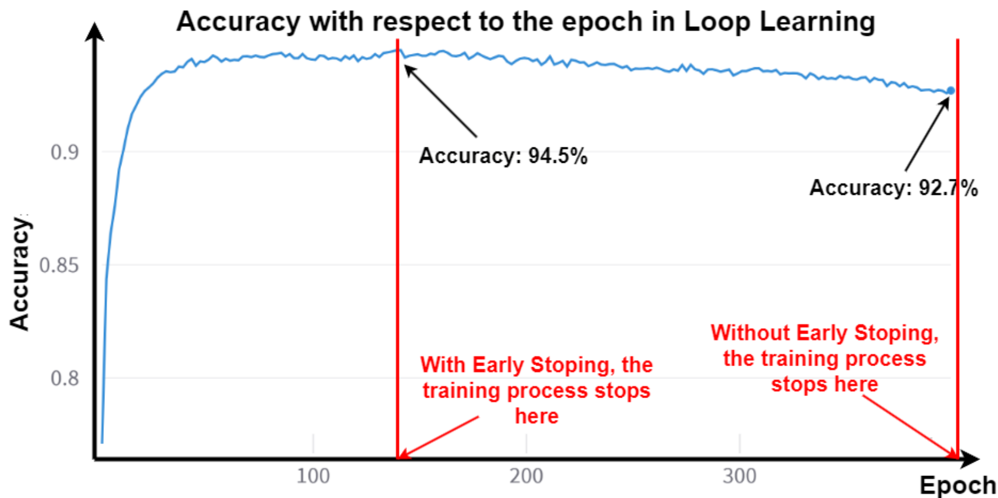


Figure 3.18: Accuracy with respect to the epoch in Loop Learning

TTFS-tinyODIN. We subsequently integrated TTFS-tinyODIN into X-Heep, an MCU that uses a RISC-V CPU to enable the configuration and control of TTFS-tinyODIN via the RISC-V CPU. Subsequently, we created the software driver for TTFS-tinyODIN, which facilitates CPU configuration. Finally, we confirmed the successful operation of the loop learning setup between the CPU and TTFS-tinyODIN.

3.3.1. Neuromorphic SoC Architecture

The architecture diagram of the proposed neuromorphic SoC is shown in Figure 3.19. The SoC is realized by combining X-Heep [15] with the TTFS-tinyODIN and a self-boot module through the OBI bus.

The RISC-V CPU serves as the central component of this SoC, accessing all other modules via the OBI BUS. The TTFS-tinyODIN block implements the inference process for the SNN trained in the software design portion, receiving input and configurations from the RISC-V CPU. The self-boot module is an OBI bus manager that includes a memory block storing the binary code of the runtime system. Upon each SoC boot, the binary code is written to a specific address in the memory subsystem, and the self-boot module prompts the CPU to read the same address, thereby initializing the SoC.

3.3.2. SNN Processor: TTFS-tinyODIN

In this section, we introduce the SNN coprocessor, TTFS-tinyODIN. TTFS-tinyODIN is a highly versatile processor capable of mapping a variety of fully connected neural networks. We will utilize TTFS-tinyODIN to map the SNN model acquired in Section 3.2 and to carry out the inference process in this project.

Architecture

The TTFS-tinyODIN architecture utilizes a crossbar setup to establish connections between all neurons in an all-to-all pattern, as depicted in Figure 3.20. The crossbar design was purposefully selected, influenced by [12], due to its inherent ability to facilitate direct connections between any two neurons, allowing for great flexibility in

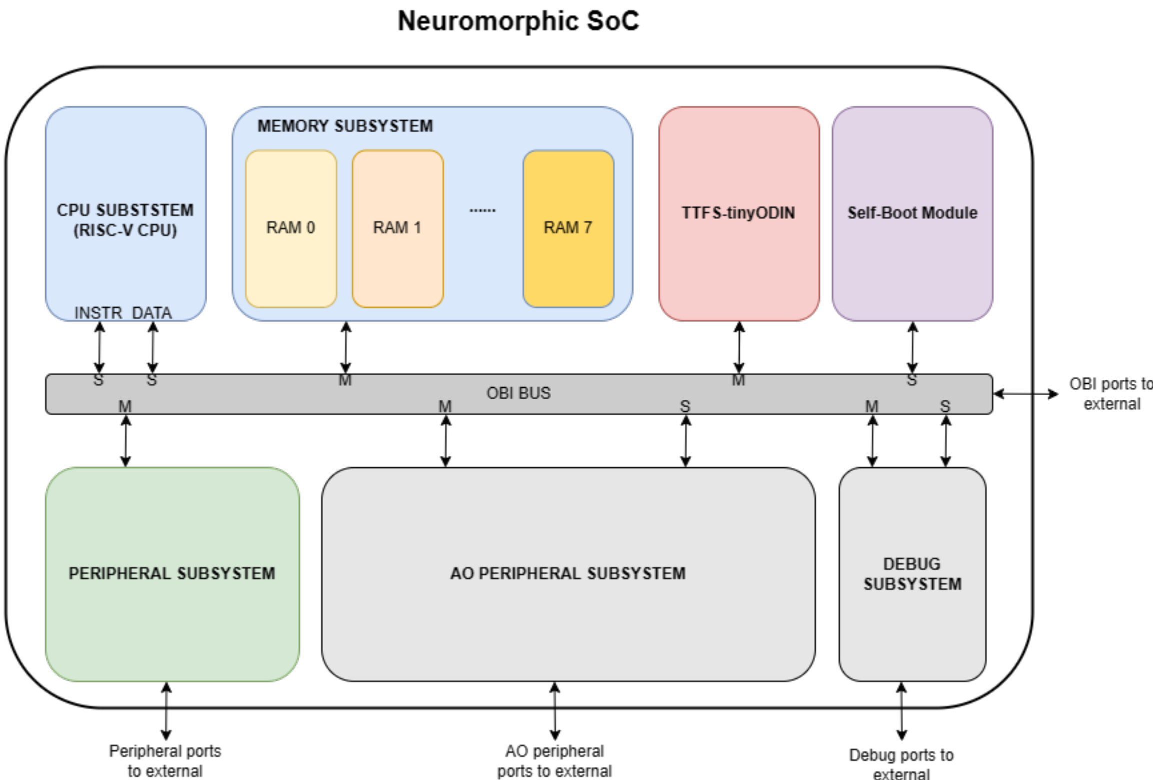


Figure 3.19: Architecture of the final neuromorphic SoC, based on X-Heep, all the internal submodules in the peripheral subsystem, Always on (AO)-peripheral subsystem and those ports to the external are omitted in.

mapping any SNN to this coprocessor.

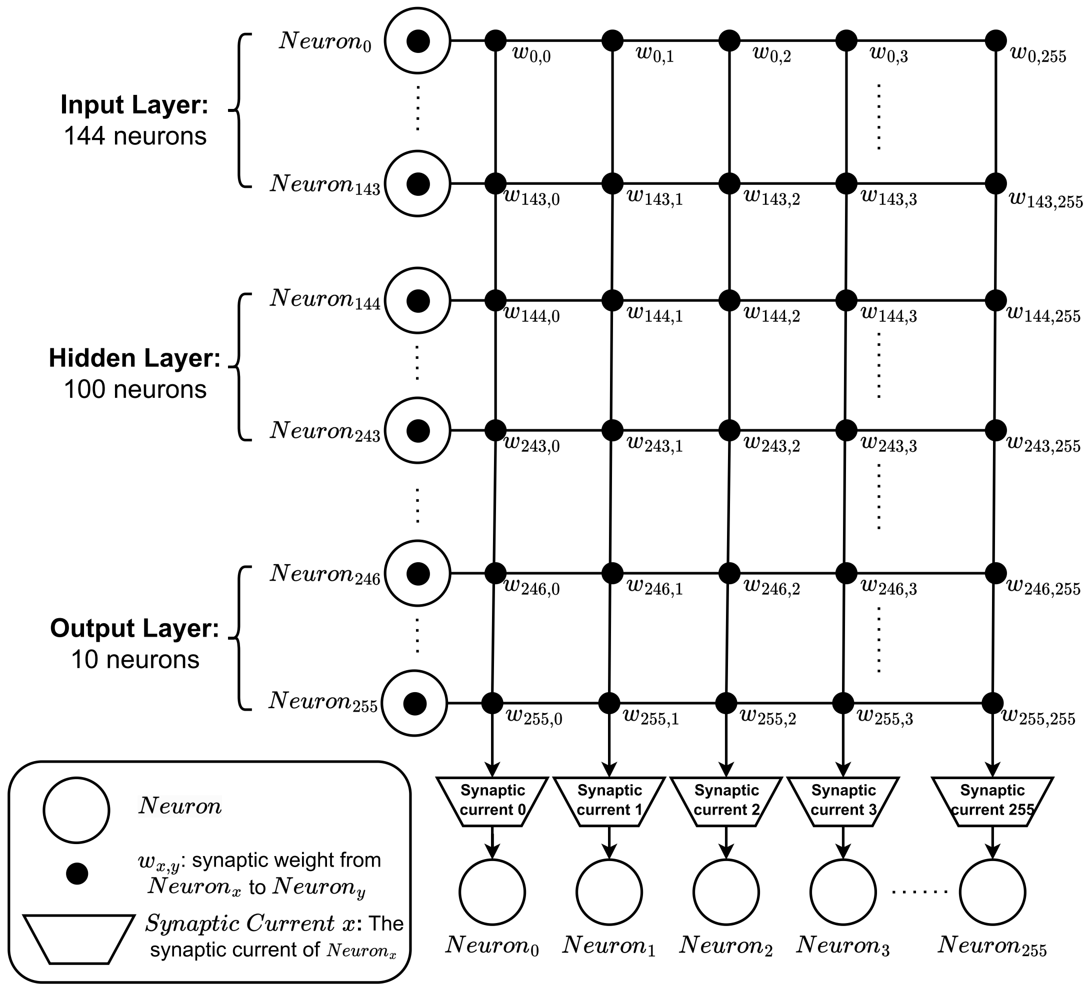


Figure 3.20: The architecture of the SNN implementation in TTFS-tinyODIN

The crossbar structure described in [12] includes 256 neurons and a corresponding set of 65,536 synaptic weights (256x256). We chose to maintain this configuration as adding more neurons would result in quadratic growth of synaptic weights, ultimately leading to increased hardware resource consumption. Conversely, reducing the number of neurons would make the coprocessor inadequate for precisely mapping the SNN model, as our SNN model contains 254 neurons.

Within Figure 3.20, each row represents 256 output synaptic weights originating from $Neuron_x$, while each column signifies 256 input synaptic weights directed towards $Neuron_y$. The input synaptic weights to $Neuron_y$ will accumulate in the $Synaptic\ Current\ x$, leading to changes in the membrane potential of $Neuron_y$.

In accordance with this architecture, when the synaptic weight $w_{x,y}$ is equal to zero, it signifies the absence of a synaptic connection between the neurons $Neuron_x$ and $Neuron_y$. Conversely, when the synaptic weight is a non-zero value, it indicates the presence of a synaptic connection linking these two neurons.

As depicted in Figure 3.20, the mapping of the SNN model discussed in Section 3.2,

from its software representation to the hardware implementation, is represented. We executed the following mapping scheme:

1. The input layer of our SNN was mapped to $Neuron_0$ through $Neuron_{143}$ of the TTFS-tinyODIN architecture.
2. The hidden layer of our SNN was mapped to $Neuron_{144}$ through $Neuron_{243}$ of the TTFS-tinyODIN architecture.
3. The output layer of our SNN was mapped to $Neuron_{246}$ through $Neuron_{255}$ of the TTFS-tinyODIN architecture.

Note that $Neuron_{244}$ and $Neuron_{245}$ were excluded from this mapping process and remained unused in the current configuration.

Within our network, 15,400 weights are utilized out of a total capacity of 65,536 in the crossbar structure. The remaining 50,136 weights are inactive and set to zero. Although a significant amount of weight resources is left unused, this allocation is a deliberate and necessary aspect of the coprocessor's design. It guarantees that the coprocessor has the flexibility necessary to map arbitrary neural network models.

Module Diagram

The top-level module diagram of TTFS-tinyODIN is represented in Figure 3.21. The interfaces to X-Heep, which will be introduced in the next section, are omitted for clarity. It is also noteworthy that TTFS-tinyODIN is designed for integration into the X-Heep platform and is intended to facilitate communication with a 32-bit RISC-V core. In alignment with this integration, all SRAMs and registers that are intended to be accessed by the RISC-V CPU have a width of 32 bits. This design choice ensures compatibility and enables communication between TTFS-tinyODIN and the 32-bit RISC-V core. The specific data arrangement within the SRAM is explained in detail below.

To optimize the utilization of hardware resources, a time multiplexing approach[50] is employed within each module. This means that all neurons operate sequentially, based on shared update logic, a concept that will be elaborated upon in subsequent paragraphs.

Next, we will introduce the internal structure and workflow of the four modules of TTFS-tinyODIN, namely the spike core, synaptic core, neuron core, and control module.

- Spike core: The spike core module, which is enclosed within the yellow dashed box in Figure 3.21, consists of several key components, as shown in Figure 3.22. This module includes a 256B spike core SRAM, a spike checker module and a 32-stage 8-bit-width spiked neuron first-in-first-out (FIFO) queue.
  - Spike core SRAM: The spike core SRAM is responsible for storing spike time information of 256 neurons. The data is stored in a specific format, utilizing 64 32-bit words. Within each 32-bit segment, the 8-bit spike times of four neurons are encoded. For instance, the 32-bit value located at address 0 holds the spike times of $neuron_0$ to $neuron_3$, while the 32-bit value at address 1 contains the spike times of $neuron_4$ to $neuron_7$, etc.

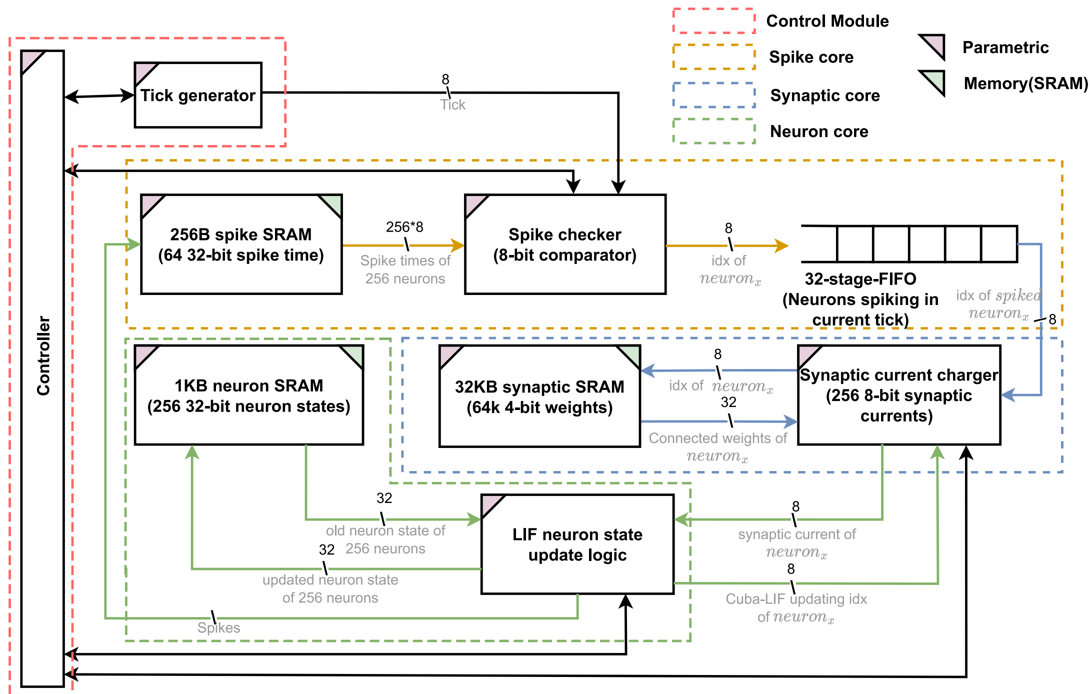


Figure 3.21: Architecture of TTFS-tinyODIN. The yellow box indicates the spike core module. The blue box indicates the synaptic core module. The green box indicates the neuron core module. The red box indicates the control module.

  - Spike checker: The spike time sequencer is responsible for traversing the spike core SRAM sequentially and converting the 32-bit spike time information into four 8-bit spike times. The comparator compares the spike times with the current time (referred to as "tick"). When a match is found, indicating that a neuron spikes at the current time, the index of the spiking neuron will be pushed into the spiked neuron FIFO. This process is based on time multiplexing, in every clock only one 8-bit spike time is compared. Therefore, 256 clock cycles are needed to check the spike times of 256 neurons.
  - Spiked Neuron FIFO: The FIFO stores the indexes of the neurons spiking at the current time

- Synaptic core: This module is denoted by the blue dash box of Figure 3.21. The detail of this module is shown in Figure 3.23. This module contains a 32KB synaptic core SRAM and a synaptic current charger.
  - Synaptic Core SRAM: This 32KB synaptic core SRAM stores all the 64 4-bit weights on the crossbar. This SRAM represents the crossbar structure mapping. Thus, the SRAM size is 8192*32 bits. In this SRAM, each 32-bit row contains eight 4-bit weights. The 32-bit value in row 0 represents the eight weights leading from $neuron_0$ to $neuron_{0-7}$, and rows, ranging from address 0 to 31 represent the 256 weights from $neuron_0$ to all 256 neurons, etc.
  - Synaptic current charger: Based on the index in the spiked neuron FIFO,

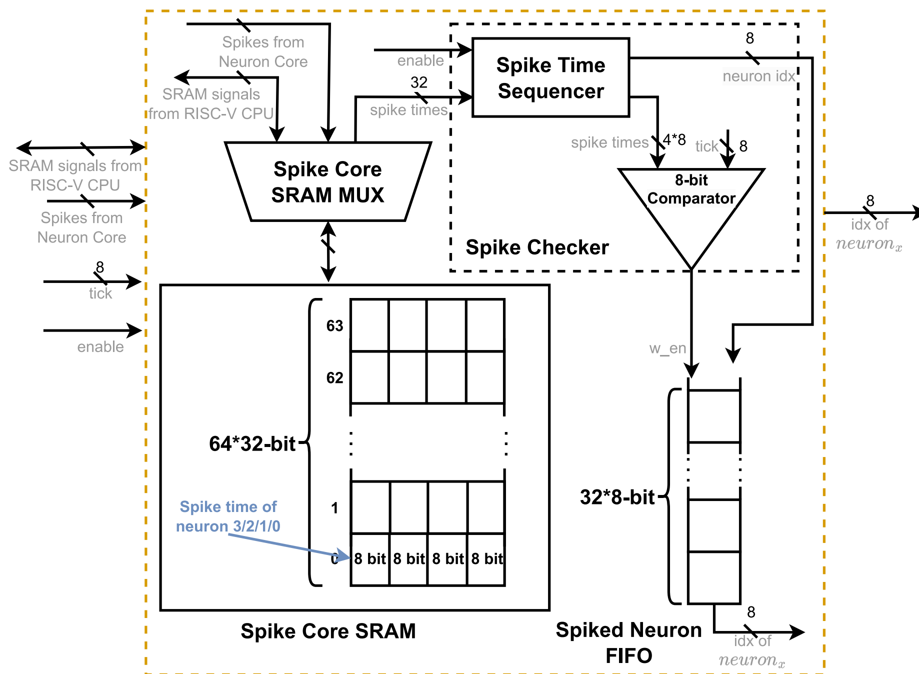


Figure 3.22: Internal components and dataflow of the spike core module

the synaptic current charger controller accesses the synaptic core SRAM and sequentially reads the 256 post-synaptic weights of that neuron. These weights are subsequently used to update the synaptic currents of the 256 neurons via time multiplexing. In every clock cycle, the synaptic current charger Controller reads 8 4-bit weights from the synaptic core SRAM and adds them to the corresponding 8 synaptic currents. Updating the synaptic currents of 256 neurons therefore requires 64 clock cycles. The updated synaptic current will then be stored in the synaptic current registers, which store the synaptic currents of 256 neurons.

- Neuron core: This module is the part in the green dash box of Figure 3.21. The detail of this module is shown in Figure 3.24. The part contains a 1KB dual-port neuron core SRAM and the LIF neuron state update module.
  - Neuron core SRAM: The neuron SRAM stores the states of 256 neurons, where each neuron state has 32 bits (Figure 3.25), with 12 bits for the threshold voltage, 13 bits for the membrane potential, and 1 bit for the enable. We also reserved 6 bits in order to ensure that the length is 32 bits. Furthermore, this is a dual-port SRAM, enabling concurrent read and write operations to neuron states, which is explained in the next paragraph.
  - Neuron state updating module: This module updates the state of 256 neurons and implements Equation 3.17. The neuron state updating controller reads each neuron's state sequentially and retrieves the corresponding synaptic currents from the synaptic core module. This information is subsequently transmitted to the Cuba-LIF model, comprising an adder and a comparator. The adder updates the membrane potential, while the comparator compares the membrane potential to the threshold voltage. The Cuba-LIF model

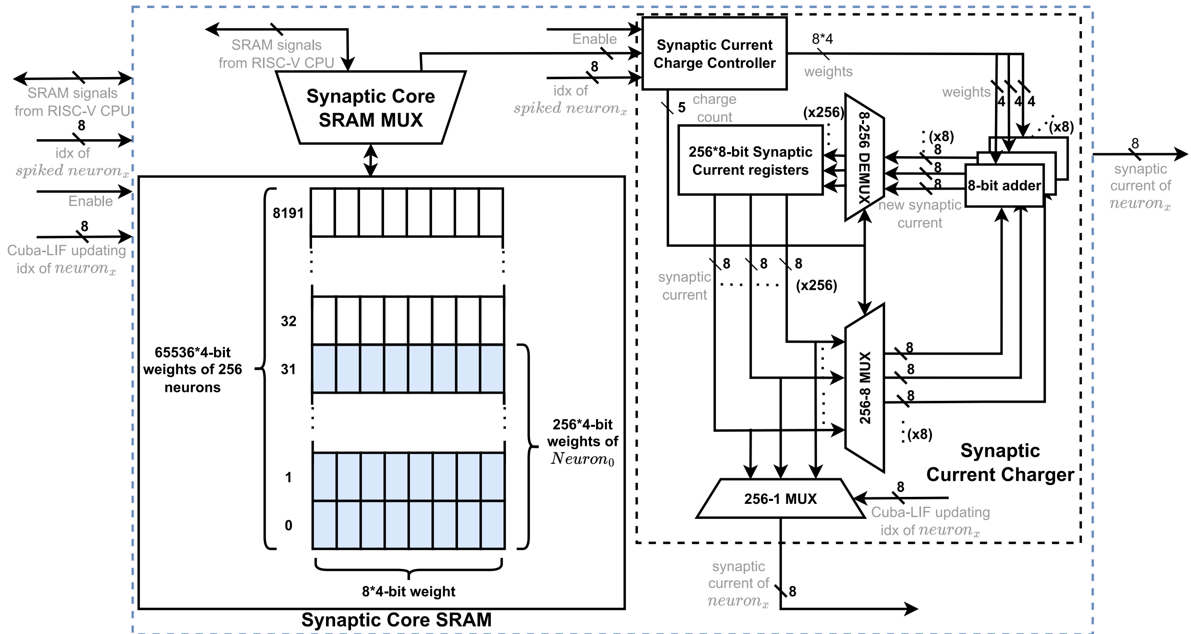


Figure 3.23: Internal components and dataflow of synaptic core module

updates the membrane potential by accumulating the synaptic current to the previous membrane potential and then writing the updated value to the neuron core SRAM. If the neuron's membrane potential surpasses its threshold voltage, the Cuba-LIF model produces a spike and writes the corresponding spike time to the same spike core SRAM. To decrease inference latency, we utilize the dual-port SRAM which allows for read and write operations to occur simultaneously, ensuring that the updated neuron state is written at the same clock cycle as the next neuron state is read. This updating process is shown in Figure 3.26. This process uses time multiplexing to update only one neuron per clock cycle. 257 clock cycles are required to update the state of 256 neurons.

- Control module: This module is the part in the red dash box of Figure 3.21. The control module consists of a controller and a tick generator. The controller enables and disables the three cores throughout different inference phases, and a finite state machine (FSM) executes the control process, which will be elaborated on in detail in the dataflow paragraph below. In addition, a 32-bit control register that the CPU can access facilitates the control process. The control register is shown in Figure 3.27. It consists the number of neurons in the input layer of the SNN network, the total neuron count in the network, and the start and end flag bits for inference. The timing of inference in this co-processor is controlled by the tick generator, which will be introduced in the next section.

Dataflow

In this section, we will present the detailed dataflow in an inference of TTFS-tinyODIN. We simulated the membrane potential of the simplified Cuba-LIF model over time and reproduced the calculation of Equation 3.17 in hardware design. Based on this, we replicated the inference process on software in X-Heep through hardware acceleration

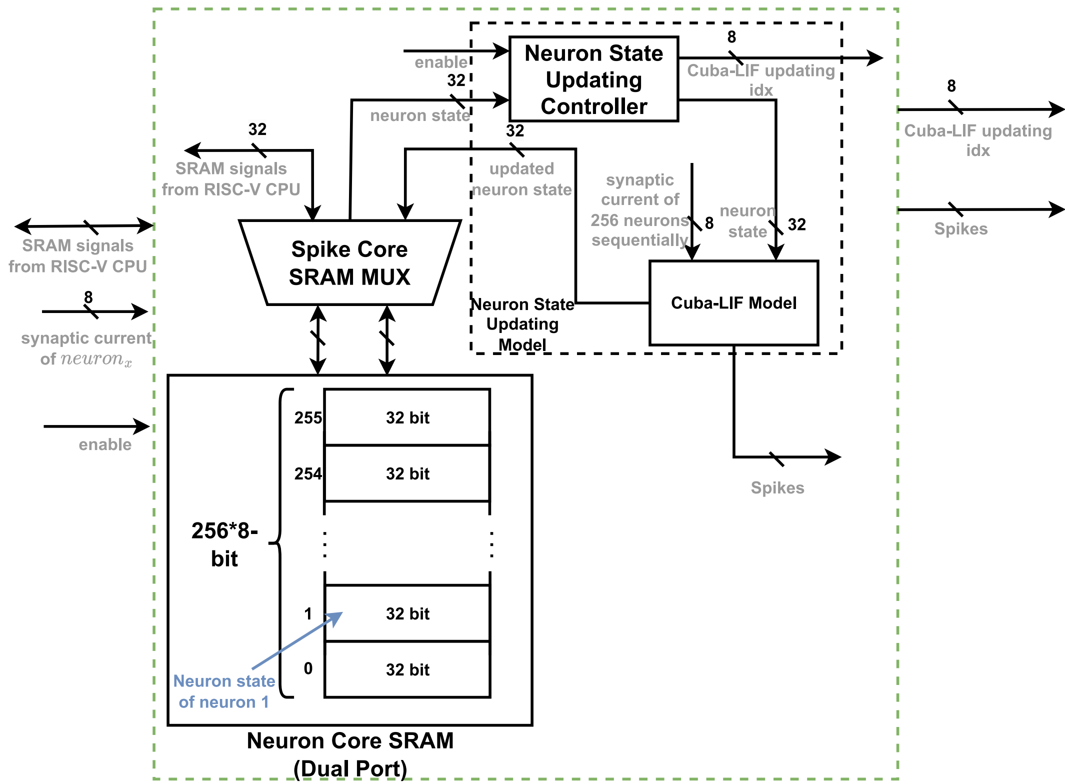


Figure 3.24: Internal components and dataflow of neuron core module

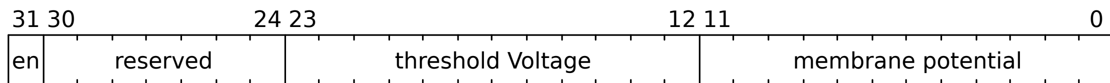


Figure 3.25: The 32-bit neuron state per neuron in the neuron core SRAM: 12 bits for threshold voltage, 13 bits for membrane potential, 1 bit for enable, and 6 bits for parameters.

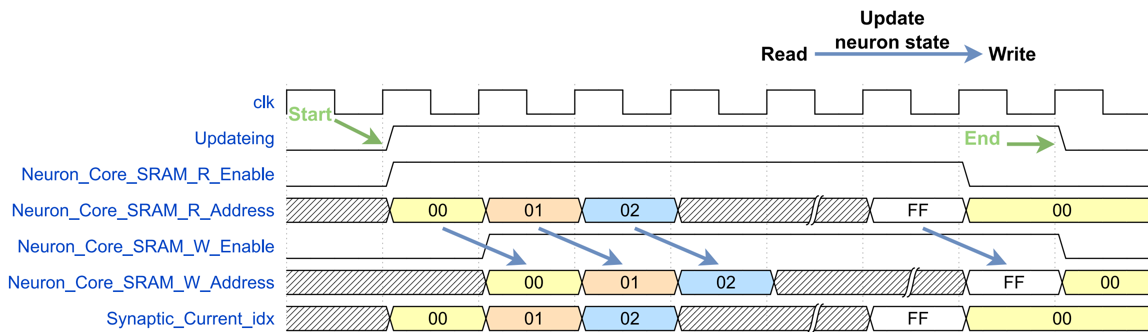


Figure 3.26: Updating the membrane potential of 256 neurons. The neuron SRAM is a dual-port SRAM that reads the membrane potential of the k_{th} neurons at the i_{th} clock cycle and writes back the membrane potential of the neuron after updating; it simultaneously reads the membrane potential of the $(k+1)_{th}$ neuron at the $(i+1)_{th}$ clock cycle.

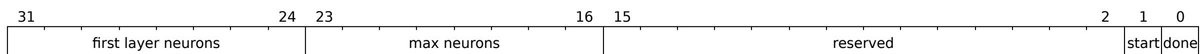


Figure 3.27: The register map of the control register. 8 bits for the number of neurons in the first layer, 8 bits for the number of neurons of the whole SNN, 14 bits for the parameter, 1 bit for the start which indicates the start of one inference and 1 bit for the done which indicates the finish of one inference.

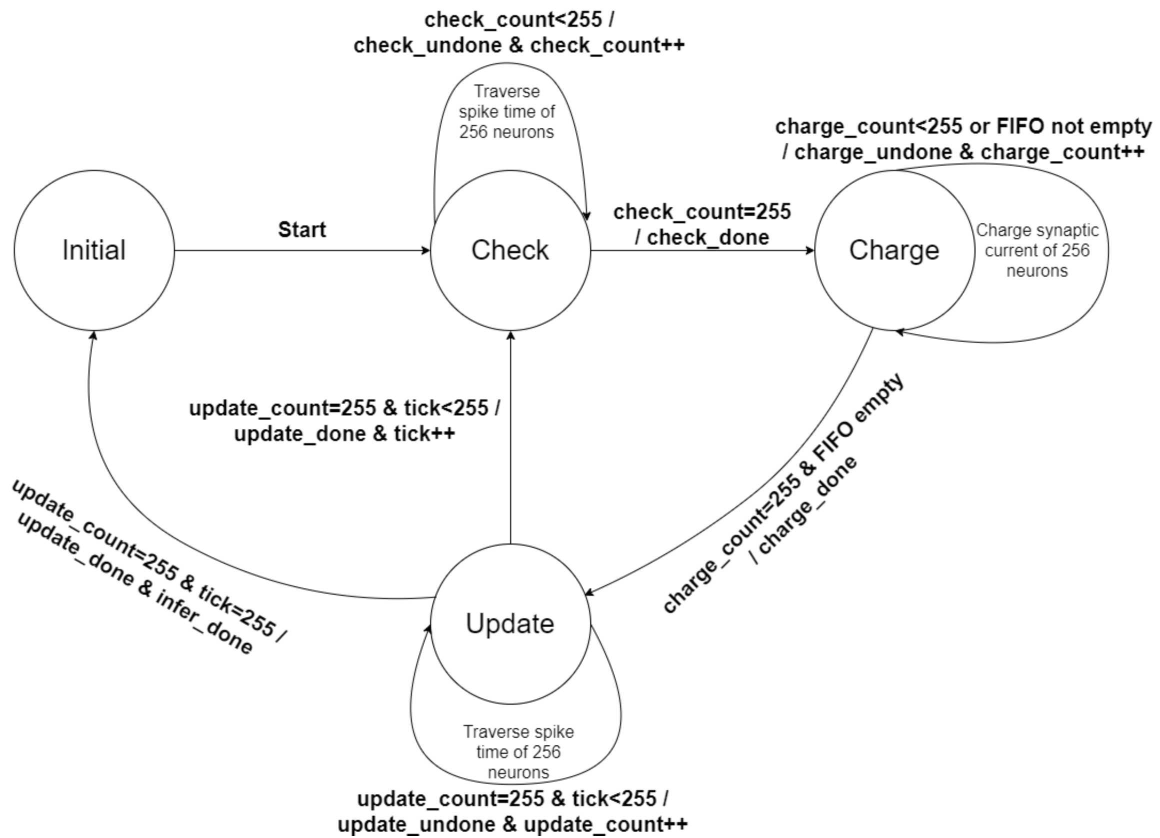


Figure 3.28: The state transfer diagram of the FSM in inference in TTFS-tinyODIN.

in the TTFS-tinyODIN coprocessor

In the software component, we confine the simulation time range between t=0 and t=2.5, subsequently quantizing it into 8 bits. Therefore, in the hardware implementation, we establish a resolution of 2.5/256, which we refer to as a "tick." This implies that within the hardware framework, we employ 256 ticks to emulate the processes occurring in the software from t=0 to t=2.5. Consequently, the entire inference process within this coprocessor operates based on these ticks, starting at tick=0 and concluding at tick=255.

The controller drives the inference process with one FSM, which is described in the following states and Figure 3.28:

1. Initial: The process begins by writing the weights into the synaptic core SRAM. This establishes the architecture of the network. Next, the threshold voltage and initial membrane potential values for all neurons are written to the neuron core SRAM. Finally, the TTFS-encoded pattern is written to the spike core SRAM to serve as the input for this coprocessor. The initialization process is simulated to be executed in the testbench through writing by the RISC-V core.
2. Check: After the $start$ bit in the control register is set to high. The inference starts with tick=0, the spike checker reads the spike time of 256 neurons stored in the spike core SRAM, compares the spike time with the current tick, and stores the neuron indices that match the tick into the spiked neuron FIFO. After checking

all 256 spike times of 256 neurons, the $Check$ phase ends and the $Charge$ phase begins.

3. Charge: Based on the neuron index read from the FIFO, the corresponding 256 weights in the synaptic core SRAM are accessed and added to the latest value of the 256 synaptic currents, thereby updating them. In this step, the synaptic current charger reads the values in the spiked neuron FIFO in sequence. The synaptic current charger repeats this process of reading until the FIFO is empty.
4. Update: At this phase, the LIF neural state update module updates the membrane potential of 256 neurons based on their synaptic currents. The updated current membrane potentials are calculated by retrieving the latest value of the membrane potentials from the neuron core SRAM and adding the corresponding synaptic currents, which implements Equation 3.17. After updating the membrane potentials of all 256 neurons, a tick is completed and the process is repeated with the next tick and continues to the $Check$ phase. Notably, the Cuba-LIF model generates a spike and writes the tick+1 to the spike core SRAM when the updated membrane potential of a neuron exceeds its threshold voltage. For example, if $Neuron_{10}$ generates a spike at tick=5, the Cuba-LIF model will write tick=6 to address 10 (the address of $Neuron_{10}$) in the spike core SRAM. This allows the $Check$ stage to capture all spikes generated during the current tick on the next tick. If tick=255 is detected, the inference is completed, marking the end of the state and returning to the $Initial$ state.

Verification

To verify the computational process of TTFS-tinyODIN, a post-implementation simulation was conducted on the Vivado platform to observe the waveforms of one inference. The simulation frequency was set at 100MHz.

In this testbench, the TTFS-tinyODIN coprocessor is configured by initializing the weights, membrane potentials, and threshold voltages according to the SNN model employed during software training. To achieve this, we reset the membrane potentials of all 256 neurons to 0 and set the threshold voltage for each neuron to a uniform value of 340 by writing to the neuron core SRAM. We initialize the spike times of the neurons in the hidden and output layers to 255 by writing to the spike core SRAM

Subsequently, we input a 12×12 sample input image into the SNN co-processor, following TTFS encoding and quantization, by writing the first 144 spike times to the spike core SRAM. This aligns with the input in the software, and then we start the inference process. It is important to note that during the testing process, the inputs were already encoded and quantized, which will be executed by the RISC-V CPU in the final SoC. Once the tick count reaches 255, the inference process concludes. The resulting neuron spike times originating from the output layer for a '7' handwritten digit are then showcased in Figure 3.29, illustrating that output neuron 7 spikes first, thereby proving correct inference.

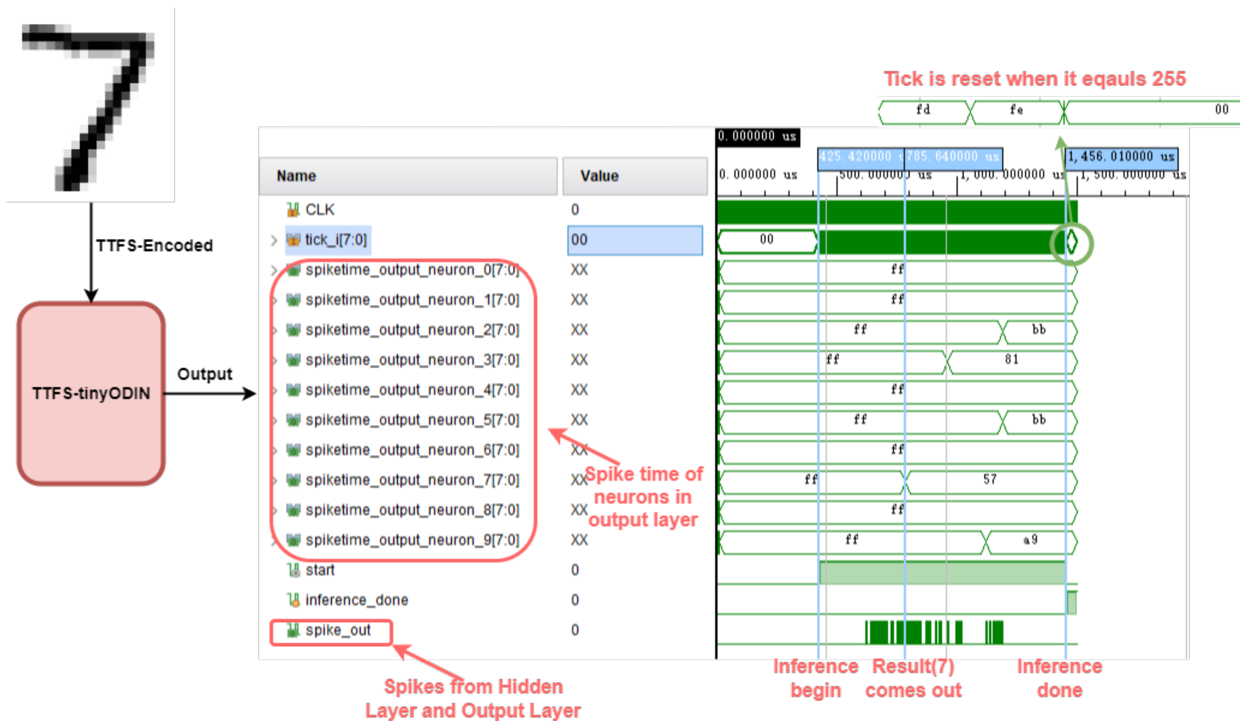


Figure 3.29: Simulation results with respect to an input sample of a "7" digit. In the output layer, the 1_{st} to 10_{th} neurons indicate the inference result as 0-9 respectively. Output neuron 7 is the first to generate a spike at tick = 57, which indicates that this inference result is 7, proving that the inference is correct. This inference finishes at tick = 255.

Note that for faster inference, it is possible to halt the inference process once any neuron in the output layer spikes, without waiting for tick=255.

Additionally, we conducted random tests on 100 samples using the same testing methodology, and the inference results matched those produced by the software consistently. Based on these consistent outcomes, we conclude that the coprocessor is capable of effectively achieving the same results as the software for the inference process.

It is important to acknowledge that, due to time constraints within the project, we were unable to validate these test results on field-programmable gate arrays (FPGAs). Consequently, all test results presented here are based on simulation on a limited number of samples.

3.3.3. Cooperation between X-Heep and TTFS-tinyODIN

To develop a comprehensive neuromorphic SoC, we integrated TTFS-tinyODIN into X-Heep with the OBI bus interface. We also designed and implemented the necessary software driver to facilitate seamless communication between the RISC-V CPU and TTFS-tinyODIN. Following this integration, we successfully established the loop learning setup within the SoC, enabling the combined operation of these components for efficient and adaptive neuromorphic processing.

TTFS-tinyODIN Integration

To configure TTFS-tinyODIN within the X-Heep platform, we memory-mapped it to a specific address within X-Heep's memory space, specifically at address 0x50000000. In addition, we established an interface to facilitate communication between TTFS-tinyODIN and the OBI bus.

This interface includes the multiplexing of the OBI's 32-bit address bus to enable access to the three SRAMs and control registers within TTFS-tinyODIN, as illustrated in Figure 3.30. Selection is accomplished by utilizing the 22_{nd} and 21_{st} bits of the address, allowing for controlled access to TTFS-tinyODIN's components.

It is crucial to emphasize that X-Heep employs a word-based addressing scheme, wherein addresses increment by units of 0x04 bytes. In contrast, TTFS-tinyODIN employs a byte-based addressing system, with addresses incrementing by 0x01 bytes. To bridge this addressing difference, we integrated a shift register into the system. This shift register effectively aligns addresses between the two systems, ensuring compatibility and accurate addressing between X-Heep and TTFS-tinyODIN.

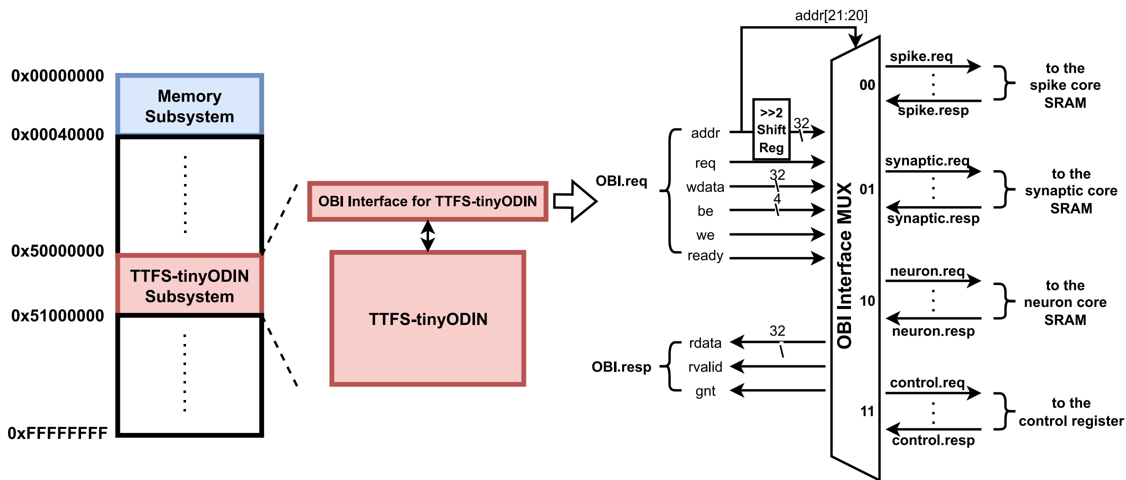


Figure 3.30: The TTFS-tinyODIN address locations in the X-Heep correspond to the interface that it exposes to the OBI bus. The three SRAMs and control registers are presented to the OBI bus holistically as a subordinate unit via a 4-1MUX. The chip selects signals for the req's address line using bits [21:20].

Configuration of TTFS-tinyODIN

The RISC-V CPU can configure TTFS-tinyODIN by accessing its SRAMs and control registers:

- Spike core SRAM: The CPU is able to write and read the spike time of 256 neurons through this spike core SRAM. Before inference, the RISC-V CPU defines the inputs of the SNN by initializing the spike times of neurons in the input layer to the spike core SRAM. After inference, the RISC-V CPU reads the spike times of neurons in the output layer from the spike core SRAM and chooses the neuron that spiked first as the inference result.
- Neuron core SRAM: The CPU is able to write and read the neuron states of 256 neurons through this neuron core SRAM. Before inference, the CPU enables all neurons mapped to the SNN by setting their en bit to 1, resetting their membrane

potential, and setting their threshold voltage by writing their 32-bit neuron state in the SRAM. This step initializes all the neuron states before inference.

- Synaptic core SRAM: The CPU is able to write and read the 64k weights between 256 neurons through the synaptic core SRAM. Before inference, by writing the values of all weights, the CPU can determine how the neurons are interconnected, enabling TTFS-tinyODIN to map various neural networks with different architectures. After the inference process, the CPU can retrieve these weights from the synaptic core SRAM. Then, the CPU updates them based on the simplified backpropagation algorithm detailed in Section 3.2.2. Subsequently, the CPU writes these updated weight values back to the SRAM.
- Control Register: The CPU is able to control the beginning and end of inference through this control register. By setting the *start* bit of the control register to a high state, the CPU signals TTFS-tinyODIN to initiate the inference process. Subsequently, the CPU can monitor the *done* bit within the control register to determine precisely when the inference has been successfully completed.

Loop learning implementation within the SoC
In this section, we demonstrate the utilization of TTFS-tinyODIN in conjunction with the RISC-V CPU to implement a loop learning setup. The workflow involves TTFS-tinyODIN performing the inference process, while the RISC-V CPU executes the backpropagation algorithm to update the weights. The following sections offer a comprehensive breakdown of the loop learning process, and Figure 3.31 illustrates the corresponding schematic diagram.

- Initialization: The TTFS-tinyODIN is first initialized by the RISC-V CPU by writing the initial weights and neuron states to the synaptic core SRAM and neuron core SRAM respectively.
- Configuration: The RISC-V CPU is configured to receive external input data, for example, from the GPIO ports of X-Heep. These inputs are pre-processed using TTFS encoding, quantized, and subsequently written to the spike core SRAM of TTFS-tinyODIN.
- Input: The CPU initiates inference in TTFS-tinyODIN by raising the control register's *start* bit and then continuously monitors the control register's *done* bit to detect its high state which implies the completion of the inference.
- Backpropagation: The RISC-V processor obtains spike times of neurons in both the hidden and output layers by reading data from the spike core SRAM. Concurrently, it accesses the weights stored in the synaptic core SRAM. The RISC-V CPU subsequently executes the back-propagation algorithm, utilizing the retrieved spike times and weights to iteratively update the synaptic weights.
- Write back: The updated weights are written by the RISC-V CPU back to the synaptic core SRAM of TTFS-tinyODIN before returning to the *Initialization* step.

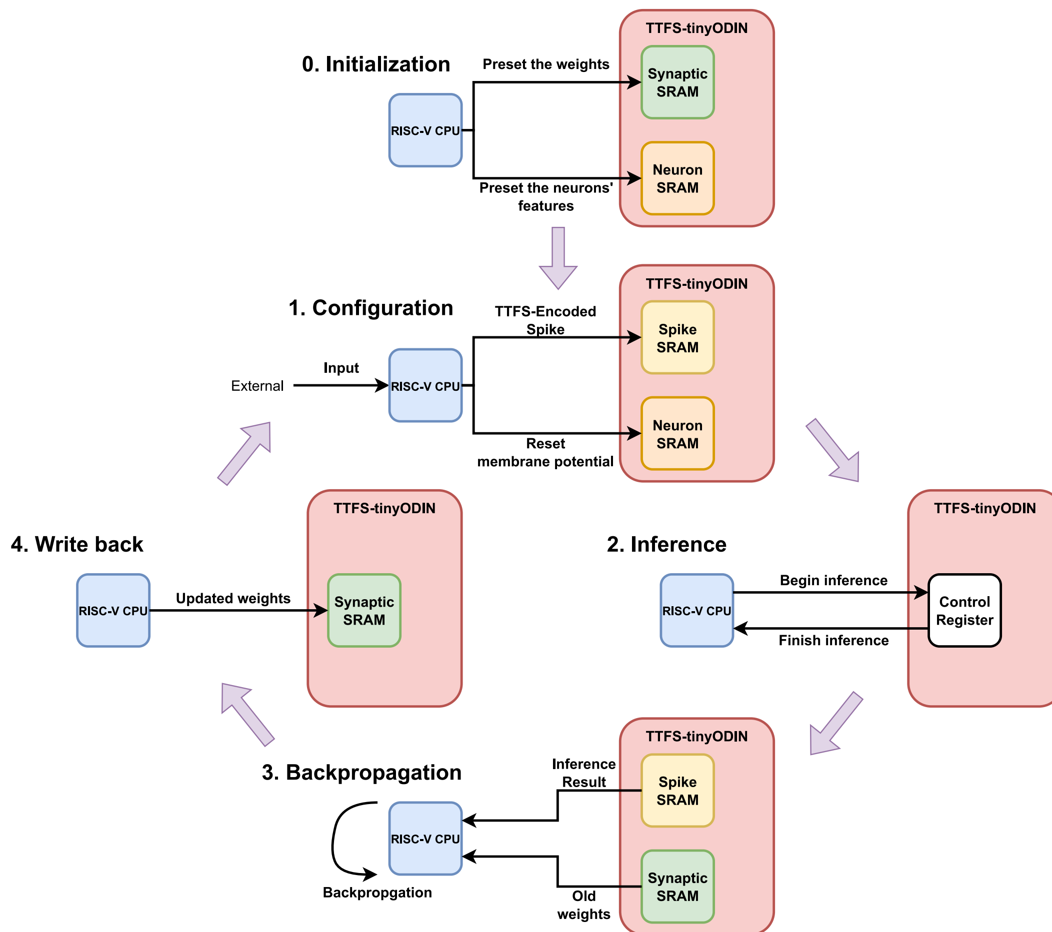


Figure 3.31: Workflow of the loop learning model.

3.3.4. Results and Fallback Plan

The designs discussed earlier were initially based on the assumption that the RISC-V CPUs on X-Heep could handle floating-point arithmetic seamlessly. However, at the end of the design process, we found that X-Heep was unable to properly handle floating-point results, which posed challenges to the project. Simulations on the SoC revealed that floating-point computation was currently neither well supported on X-Heep nor well documented. Therefore, we needed to find an alternative approach to avoid floating-point computation.

To avoid restarting the software design from scratch, we had to work within the existing design and transition to using fixed-point arithmetic for computation. To achieve this, we converted floating-point numbers to fixed-point numbers by scaling the original number by a factor of 10 and then discarding the fractional part. A factor-10 scaling was found to minimize the accuracy loss.

In the original algorithm, both the loss function (cross-entropy) and the weight update (Adam optimization algorithm) are heavily based on precise floating-point operations, including square roots. To address this and use fixed-point arithmetic, we replaced the cross-entropy loss function with the MSE loss function. and the Adam optimization algorithm by the SGD algorithm, which is computationally simple.

We successfully simulated the backpropagation algorithm using fixed-point arithmetic. However, when attempting to update the weights of the first layer in the SNN during our simulation, we encountered memory limitations on X-Heep, preventing us from computing the gradient of all the weights concurrently. Given that this project aims to provide proof-of-concept for a loop learning model connecting a RISC-V CPU and an SNN coprocessor, we chose to update only the weights of the second layer, while the first layer used a pre-trained model. This decision was based on the fact that the second layer has only 100*10 weights, while the first layer has 144*100 weights. By focusing on updating the second layer weights, we were able to bypass the memory limitation.

As a result of this training setup, the achieved accuracy stands at 92.2%, which was validated with our software model. As the hardware implementation is restricted to a post-implementation simulation setup (Section 3.3.2), validating a full hardware loop learning setup over several epochs would lead to intractable simulation time. Therefore, we have checked that the weight update process implemented in hardware is in one-to-one correspondence with our software model, which we report in more detail in Appendix A. We thus predict that the accuracy of 92.2% should be successfully obtained with a dedicated hardware implementation running, for example, on an FPGA, which we discuss in Section 4.2.

4

Conclusion and Future Work

In this chapter, we summarize the results of this project and suggest some key points for future work.

4.1. Conclusion

In this project, we successfully co-designed the software model training and hardware implementation to create a neuromorphic SoC that combines a RISC-V CPU with an SNN coprocessor. This SoC demonstrates a proof of concept for loop learning on dedicated hardware.

In the software design phase, we began with an SNN neuron model and the backpropagation algorithm for TTFS encoding from a previous work [15]. We established a baseline SNN with an architecture of 144-100-10, achieving an accuracy of 96.5% on the TTFS-encoded MNIST dataset. To adapt this model to hardware, we simplified the backpropagation algorithm by introducing four λ values for four distinct cases. We also simplified the Cuba-LIF model by implementing a one-spike limitation and removing the leakage. These simplifications resulted in a limited 2.1% drop in accuracy compared to the baseline.

Next, we quantized the SNN model, using an 8-bit resolution for spike times and a 4-bit resolution for weights, the synaptic current and membrane potential are quantized accordingly. This quantization led to a further 0.9% decrease in accuracy, resulting in a final accuracy of 93.5%. Finally, we conducted software simulations of the loop learning process with early stopping, achieving an accuracy of 94.5%.

In the hardware design phase, we created an SNN coprocessor called TTFS-tinyODIN to execute the inference process of the SNN. TTFS-tinyODIN is highly configurable and can accommodate various SNN architectures. Simulations confirmed that it produced the same results as the software model.

We then integrated TTFS-tinyODIN into a RISC-V-based MCU, X-Heep, to realize a neuromorphic SoC. Within this SoC, the RISC-V CPU can configure TTFS-tinyODIN by accessing different modules inside TTFS-tinyODIN through the OBI bus. Finally, we demonstrated the feasibility of loop learning on hardware with communication

between the RISC-V CPU and TTFS-tinyODIN. Despite X-Heep limitations that forced resolving to a fixed-point backpropagation, the expected accuracy reaches 92.2%.

4.2. Future Work

There are several works that can be realized in the future:

- Different dataset: In this project, we only encoded the MNIST dataset with TTFS. To prove the generality of our work, more complex image datasets like CIFAR-10 [7] can be encoded with TTFS and applied to this SNN.
- Backpropagation algorithm: In the software design part, we simplified the Cuba-LIF model but didn't modify the backpropagation algorithm accordingly. Since the backpropagation algorithm is based on the neuron model equations, once the equation of the Cuba-LIF model changes, the backpropagation algorithm should change accordingly, to minimize the accuracy loss.
- Quantization config: When we tried loop learning in software design, we already defined the quantization configuration of spike times and weights. However, those quantization configurations are obtained from the model trained under full precision computing, so the best quantization configuration in loop learning might differ since the resolution is different.
- Sparsity handling: In TTFS-tinyODIN, a lot of weights are 0, which is called sparsity. These weights do not contribute to the results, but TTFS-tinyODIN still takes them into account. Leveraging this sparsity and skipping those 0s can further reduce the computing load, the inference latency and the power consumption.
- FPGA Implementation: This proof of loop learning is achieved based on the simulation with ModelSIM: the whole SoC should be implemented in the FPGA for verification.

References

[1] July 2023. URL: https://en.wikipedia.org/wiki/History_of_artificial_neural_networks#cite_note-robbins1951-21.

[2] E. D. Adrian and Yngve Zotterman. "The impulses produced by sensory nerve-endings". In: *The Journal of Physiology* 61.2 (1926), pp. 151–171. DOI: https://doi.org/10.1113/jphysiol.1926.sp002281. eprint: https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.1926.sp002281. URL: https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1926.sp002281.

[3] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[4] Seunghwan Bang et al. "An Energy-Efficient SNN Processor Design based on Sparse Direct Feedback and Spike Prediction". In: *2021 International Joint Conference on Neural Networks (IJCNN)*. 2021, pp. 1–8. DOI: 10.1109/IJCNN52387.2021.9534107.

[5] Soha Boroojerdi and George Rudolph. "Handwritten Multi-Digit Recognition With Machine Learning". In: May 2022, pp. 1–6. DOI: 10.1109/IETC54973.2022.9796722.

[6] "Bursts as a unit of neural information: selective communication via resonance". In: *Trends in Neurosciences* 26.3 (2003), pp. 161–167. ISSN: 0166-2236. DOI: https://doi.org/10.1016/S0166-2236(03)00034-1. URL: https://www.sciencedirect.com/science/article/pii/S0166223603000341.

[7] *CIFAR-10 (Canadian Institute for Advanced Research)*. https://www.cs.toronto.edu/~kriz/cifar.html. Accessed on September 15, 2023. 2010.

[8] Matthieu Courbariaux et al. "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1". In: *arXiv preprint arXiv:1602.02830* (2016).

[9] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[10] Jane Doe. "Reduced Instruction Set Computer (RISC) Architecture". In: *Computer Architecture Journal* 25.4 (1995), pp. 30–35.

[11] Erik Engheim. *Erik Engheim*. 2022. URL: https://itnext.io/risc-vs-cisc-microprocessor-philosophy-in-2022-fa871861bc94.

[12] Charlotte Frenkel et al. "A 0.086-mm^2 12.7-pJ/SOP 64k-Synapse 256-Neuron Online-Learning Digital Spiking Neuromorphic Processor in 28-nm CMOS". In: *IEEE Transactions on Biomedical Circuits and Systems* 13.1 (2019), pp. 145–158. DOI: 10.1109/TBCAS.2018.2880425.

[13] Wulfram Gerstner. *Neuronal Dynamics : From Single Neurons to Networks and Models of Cognition*. 2014.

[14] Tim Gollisch and Markus Meister. "Rapid Neural Coding in the Retina with Relative Spike Latencies". In: *Science* 319 (2008), pp. 1108–1111. URL: https://api.semanticscholar.org/CorpusID:1032537.

[15] Julian Göltz et al. *Fast and energy-efficient neuromorphic deep learning with first-spike times*. 2021. DOI: https://doi.org/10.1038/s42256-021-00388-x.

[16] Margaret A. Goralski and Tay Keong Tan. "Artificial intelligence and sustainable development". In: *The International Journal of Management Education* 18.1 (2020), p. 100330. ISSN: 1472-8117. DOI: https://doi.org/10.1016/j.ijme.2019.100330. URL: https://www.sciencedirect.com/science/article/pii/S1472811719300138.

[17] Samuel Greengard. "Will RISC-V Revolutionize Computing?" In: *Commun. ACM* 63.5 (Apr. 2020), pp. 30–32. ISSN: 0001-0782. DOI: 10.1145/3386377. URL: https://doi.org/10.1145/3386377.

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

[19] Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

[20] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[21] Weihua He. *Spiking Neural Network : Towards Brain-inspired Computing*.

[22] A. L. Hodgkin and A. F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *J Physiol* (1952).

[23] ARM Holdings. "ARM Architecture Reference Manual". In: (2020). URL: https://developer.arm.com/documentation/ddi0487/latest/.

[24] Itay Hubara et al. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017.

[25] Itay Hubara et al. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017.

[26] JayeshBapuAhire. *The Artificial Neural Networks handbook: Part 1*. 2018. URL: https://www.datasciencecentral.com/the-artificial-neural-networks-handbook-part-1/.

[27] John Kane. "MIPS Architecture for Programmers". In: (2006). URL: https://www.mips.com/products/architectures/mips32-2/.

[28] Henry J Kelley. "Gradient theory of optimal flight paths". In: *Ars Journal* 30.10 (1960), pp. 947–954.

[29] Jaehyun Kim et al. “Deep neural networks with weighted spikes”. In: *Neurocomputing* 311 (2018), pp. 373–386. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2018.05.087. URL: https://www.sciencedirect.com/science/article/pii/S0925231218306726.

[30] Sung Eun Kim and Il Won Seo. “Artificial Neural Network ensemble modeling with conjunctive data clustering for water quality prediction in rivers”. In: *Journal of Hydro-environment Research* 9.3 (2015), pp. 325–339. ISSN: 1570-6443. DOI: https://doi.org/10.1016/j.jher.2014.09.006. URL: https://www.sciencedirect.com/science/article/pii/S1570644315000192.

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

[32] Yisong Kuang et al. “An Event-driven Spiking Neural Network Accelerator with On-chip Sparse Weight”. In: *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2022, pp. 3468–3472. DOI: 10.1109/ISCAS48785.2022.9937521.

[33] Louis Lapicque. “Quantitative investigations of electrical nerve excitation treated as polarization”. In: *Biological cybernetics* (1907).

[34] Mingxuan Liang, Jilin Zhang, and Hong Chen. “A 1.13J/Classification Spiking Neural Network Accelerator with a Single-Spike Neuron Model and Sparse Weights”. In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021, pp. 1–5. DOI: 10.1109/ISCAS51556.2021.9401607.

[35] Yang Lu. “Artificial intelligence: a survey on evolution, models, applications and future trends”. In: *Journal of Management Analytics* 6.1 (2019), pp. 1–29. DOI: 10.1080/23270012.2019.1570365. eprint: https://doi.org/10.1080/23270012.2019.1570365. URL: https://doi.org/10.1080/23270012.2019.1570365.

[36] Yifang Ma et al. “Artificial intelligence applications in the development of autonomous vehicles: a survey”. In: *IEEE/CAA Journal of Automatica Sinica* 7.2 (2020), pp. 315–329. DOI: 10.1109/JAS.2020.1003021.

[37] Wolfgang Maass. “Networks of spiking neurons: The third generation of neural network models”. In: *Neural Networks* 10.9 (1997), pp. 1659–1671. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(97)00011-7. URL: https://www.sciencedirect.com/science/article/pii/S0893608097000117.

[38] Advait Madhavan. *Brain-Inspired Computing Can Help Us Create Faster, More Energy-Efficient Devices — If We Win the Race*. 2023. URL: https://www.nist.gov/blogs/taking-measure/brain-inspired-computing-can-help-us-create-faster-more-energy-efficient#:~:text=The%20human%20brain%20is%20an,just%2020%20watts%20of%20power..

[39] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[40] *PaLM: Scaling Language Modeling with Pathways*. 2022. arXiv: 2204.02311 [cs.CL].

[41] Albert Reuther et al. “AI and ML Accelerator Survey and Trends”. In: *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 2022, pp. 1–10. DOI: 10.1109/HPEC55821.2022.9926331.

[42] Herbert Robbins and Sutton Monro. “A stochastic approximation method”. In: *The annals of mathematical statistics* (1951), pp. 400–407.

[43] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. In: *arXiv preprint arXiv:1609.04747* (2016).

[44] Davide Schiavone and Arjan Bink. *Repository that maintain the OpenBus Interface spec*. 2023. URL: https://github.com/openhwgroup/obi.

[45] Davide Schiavone and et al Simone Machetti. *eXtendable Heterogeneous Energy-Efficient Platform based on RISC-V*. 2023. URL: https://github.com/esl-epfl/x-heep.

[46] Pasquale Davide Schiavone et al. “Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX”. In: *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 2018, pp. 1–3. DOI: 10.1109/S3S.2018.8640145.

[47] Farzin Shama, Saeed Haghiri, and Mohammad Amin Imani. “FPGA Realization of Hodgkin-Huxley Neuronal Model”. In: *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 28.5 (2020), pp. 1059–1068. DOI: 10.1109/TNSRE.2020.2980475.

[48] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[49] John Smith. “Complex Instruction Set Computer (CISC) Architecture”. In: *Computer Architecture Review* 20.2 (1990), pp. 10–15.

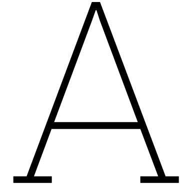
[50] John Smith. “Time Multiplexing Approach: A New Perspective”. In: *Journal of Communication* 25.3 (2000), pp. 123–135.

[51] Christian Szegedy et al. *Going Deeper with Convolutions*. 2014. arXiv: 1409.4842 [cs.CV].

[52] Xiangwen Wang, Xianghong Lin, and Xiaochao Dang. “Supervised learning in spiking neural networks: A review of algorithms and evaluations”. In: *Neural Networks* 125 (2020), pp. 258–280. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2020.02.011. URL: https://www.sciencedirect.com/science/article/pii/S0893608020300563.

[53] Andrew Waterman and Krste Asanovi. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, Dec. 2019.

[54] Shuai Zhao, Frede Blaabjerg, and Huai Wang. “An Overview of Artificial Intelligence Applications for Power Electronics”. In: *IEEE Transactions on Power Electronics* 36.4 (2021), pp. 4633–4658. DOI: 10.1109/TPEL.2020.3024914.



Result of Loop Learning in the SoC

This screenshot shows the workflow for updating the ten weights within TTFS-tinyODIN while performing Loop Learning on a RISC-V CPU.

```
Initializing base address of tinyODIN...Finished :)
Initialize input...Finished :)
Initialize neuron state...Finished :)
Initialize weights...Finished :)
Begin inference...
Inferencing...Finished :)
Fetch the output spike...Finished :)
Fetch the hidden spike...Calculating Loss...Finished :)
Fetch weights of layer 2 from tinyODIN...
The weights of neuron 0 are-6,1,1,-6,-7,-7,-7,-7,-7,-7,Finished :)
Calculating spike diff...Finished :)
Calculating dw...
dw(0) is -8208
dw(1) is -8208
dw(2) is -4944
dw(3) is -2160
dw(4) is -8208
dw(5) is -4944
dw(6) is -8208
dw(7) is -144
dw(8) is -8208
dw(9) is -4080Finished :)
Updating weights...
After updated, weight_l2 0 is 15
After updated, weight_l2 1 is 14
After updated, weight_l2 2 is 10
After updated, weight_l2 3 is 15
After updated, weight_l2 4 is 15
After updated, weight_l2 5 is 15
After updated, weight_l2 6 is 15
After updated, weight_l2 7 is 14
After updated, weight_l2 8 is 15
After updated, weight_l2 9 is 15
weight_writeback_0: ef000000
weight_writeback_1: ffefffaFinished :)
Check updated weights...
The updated weights of neuron 0 are-8,-7,-3,-8,-8,-8,-8,-7,-8,-8,Finished :)
---Exit--
```


Figure A.1: The printf result of loop learning in the SoC