A Configurable Digital Neuromorphic Hardware Generator for Heterogeneous Computing

by

Jinhuang Lin

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Thursday December 21, 2023 at 10:00 AM.

Student number:	5512549	
Project duration:	December 1, 2022 – December 1, 2023	
Thesis committee:	Prof. Dr. K. Makinwa,	TU Delft, supervisor
	Dr. C. Frenkel,	TU Delft
	Dr. R. van Leuken,	TU Delft

This thesis is confidential and cannot be made public until December 21, 2025.

Cover:	Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Recent trends in machine learning (ML) have placed a strong emphasis on power- and resource-efficient neural networks, as well as the development of neural networks on edge devices. Spiking neural networks (SNNs), due to their event-based nature, are one of the most promising types of neural networks for low-power applications. To accelerate and ease the deployment of SNNs on edge devices, this thesis presents a configurable digital neuromorphic hardware generator for heterogeneous computing that is capable of generating resource-efficient SNN processing cores. The proposed hardware generator is developed using SpinalHDL, a high-level hardware description language (HDL), which provides a high level of flexibility in hardware generation. Our generator supports the configuration on various parameters and is capable of generating a tree-structured multi-core architecture of heterogeneous cores. The generator is deployed in a sensor-fusion hand-gesture classification use case, for which the configurability of our hardware generator is a key enabler.

Contents

Nomenclature

If a nomenclature is required, a simple template can be found below for convenience. Feel free to use, adapt or completely remove.

Abbreviations

Abbreviation	Definition
AI	Artificial Intelligence
ML	Machine Learning
ANN	Artificial Neural Network
SNN	Spiking Neural Network
FNN	Feedforward Neural Network
RNN	Recurrent Neural Network
BP	Back-propagation
SGD	Stochastic gradient descent
HDL	Hardware Description Language
DVS	Dynamic Vision Sensor
EMG	Electromyography

Notations

Notation	Definition
$x, y, \cdots$	Notation of scalars (i.e. numbers).
$\boldsymbol{x}, \boldsymbol{y}, \ldots$	Notation of vectors.
$\boldsymbol{X}, \boldsymbol{Y}, \cdots$	Notation of matrices.
∇	$\nabla_{\boldsymbol{x}} y = \left[\frac{\partial y}{\partial x_1} \ \frac{\partial y}{\partial x_2} \ \cdots \ \frac{\partial y}{\partial x_n}\right]$, the gradient of scalar y w.r.t. vector $\boldsymbol{x}$.
$\odot$	$\boldsymbol{a} \odot \boldsymbol{b} = [a_1 b_1 \ a_2 b_2 \ \cdots \ a_n b_n]$, the Hadamard product of two vectors or matrices of the same shape.
$\otimes$	$\boldsymbol{a} \otimes \boldsymbol{b} = [a_1 \boldsymbol{b} \ a_2 \boldsymbol{b} \ \cdots \ a_n \boldsymbol{b}]$, the Kronecker product of two vectors or matrices.

1 Introduction

Thanks to the development of artificial intelligence (AI), human life has never been so convenient and automated. With the help of AI, especially machine learning (ML) technology, people today benefit from efficient gesture recognition, speech recognition, face recognition, and many other AI applications. To perform these AI tasks, one of the methods is to first provide various inputs and desired labels and then find a mapping from these inputs to the labels through a variety of learning algorithms. This method is called supervised learning [61]. There are different models for supervised learning, for example, artificial neural networks (ANNs), support vector machines (SVMs), and decision tree. ANNs, as one of the models for supervised learning, which is inspired by biological neural networks, are now commercially important. ANNs consist of many connected units called neurons, each of which receives activations produced by weighted connected neurons or sensors perceiving the environment. The network can be trained to find the weights that exhibit the desired output, such as recognizing a face [92, 59]. Traditionally, ANN training is data-consuming and hardware-consuming in many cases. Therefore, many ANNs run on cloud servers to meet computational requirements, which require a low-latency and high-bandwidth Internet connection when accessing these networks. However, in many practical issues, off-cloud AI is preferred for lower latency and better privacy [20]. In these cases, ANNs should be able to run on edge devices, such as edge servers, smartphones, and smart wearable devices (shown in Figure 1.1), where the hardware resources that can support an ANN are limited, and power consumption might be a critical factor. For example, according to the computation in [106], running a neural network model (OverFeat [94]) for image recognition on a Google Glass may use up its 2.1Wh battery in 25 minutes. Therefore, to deploy ANNs on edge devices and improve user experience, more energy-efficient models are required.

To solve this problem, a new-generation neural network model has been proposed, which is biologically more similar to biological neural systems than conventional ANNs. Spiking neural networks (SNNs), mimic the spike behaviors of a neuron cell and are potentially more power-saving because spike events that consume energy can be made sparse in time [101]. With SNNs and event-based sensors, taking advantage of the address-event representation (AER) protocol, it is also easier to merge data from different sensors without any transformation. This merging paradigm is called sensor fusion, which improves the accuracy and robustness of machine learning applications, such as recognition tasks. However, universal hardware (e.g., CPU or GPU) is not designed for SNNs, resulting in a performance bottleneck when running SNNs. Therefore, specifically designed neuromorphic hardware is required for SNNs.

Different network structures might be demanded for specific applications. The hardware resources consumed by the networks vary as the network varies. It would be efficient for hardware developers if there were a universal hardware generator that could automatically generate hardware with the same architecture but different parameters. Therefore, the main purpose of this project is to build a *configurable neuromorphic hardware generator for heterogeneous computing*, which generates hardware capable for SNNs. As high-level hardware description languages (HDLs) are better at describing a parameterized and flexible hardware architecture than conventional HDLs (e.g., Verilog and VHDL), high-level HDLs could be a favorable option to speedup and simplify the hardware development flow.

The main contributions of this project include:

- investigating and comparing several high-level HDLs,

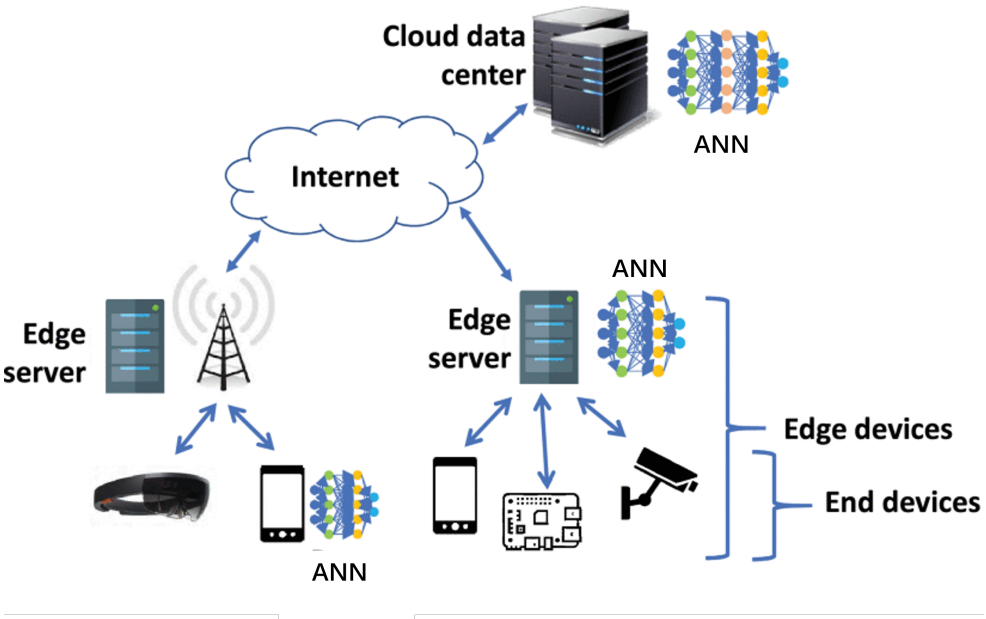

Figure 1.1: ANNs can execute on edge devices (i.e., end devices and edge servers) and on cloud data centers. End devices include: smart glasses, smart phones, micro computers such as Raspberry Pi, smart cameras, etc. Figure adapted from [20].

- building a configurable hardware generator for SNN computing with selected high-level HDLs,
- verifying the configurability of the hardware generator with a field programmable gate array (FPGA).

The hardware generator we propose is highly configurable, providing users with both on-chip and hardware-level configurability, which is also capable for scaling up with a multi-core structure.

In Chapter 2, preliminaries of neuromorphic hardware and its possible applications will be provided. In Chapter 3, a method to generate configurable digital neuromorphic hardware will be introduced in detail. In Chapter 4, neural network structures for gesture recognition and the corresponding hardware structures will be introduced. In Chapter 5, we will introduce the verification process of deployed networks. The network performance on the generated hardware will be analyzed. Finally, in Chapter 6, we give the conclusion about the proposed hardware generator.

2

Background of Neuromorphic Hardware

In this chapter, the background of neuromorphic hardware will be introduced, including preliminary knowledge of SNNs, related work on SNN hardware, and SNN training. Applications of neuromorphic hardware will also be introduced.

2.1. Overview of Neuromorphic Hardware

In this section, we will first introduce basic principles of spiking neural networks, including prerequisite knowledge on the biological neural system, artificial neural system. Then, related work on neuromorphic hardware systems and training frameworks for SNN will be introduced.

2.1.1. Basic Principles of Spiking Neural Networks (SNNs)

To understand the neuron models used in SNNs, we will first introduce the biological neurons, neuron models in different neural networks, network architectures, spike encoding methods in SNN and learning algorithms.

2.1.1.1. Biological neurons

A typical biological neuron consists of three parts: the *dendrites*, the *soma*, and the *axon*, as shown in Figure 2.1a. Dendrites are where signals from other neurons enter the soma. The soma, also known as the neuron body, acts as a ”processor” that performs important nonlinear processing. An *action potential* is generated if the total inputs arriving at the soma exceed a certain threshold. The generated action potential is transmitted along the axon and delivered to other neurons. By placing an electrode on the axon, the transmission of action potentials can be observed [44].

These action potentials generated by a neuron are also known as spikes. A series of identical spikes emitted by a single neuron is called a spike train. Since the spikes in a spike train are identical in amplitude (about 100mV) and duration (1$\sim$2ms), the information is carried by the intervals between the spikes, that is, the number and timing of the spikes [44].

The junction between two connected neurons is called a synapse. A signal is sent from one neuron to another across the synapse. The neuron that sends the signal is called the *presynaptic cell/neuron* (neuron *j* in Figure 2.1b), and the receiving neuron is known as the *postsynaptic cell/neuron* (neuron *i* in Figure 2.1b). The most common synapse in the vertebrate brain is the chemical synapse [44]. At a chemical synapse, the presynapstic cell and the postsynapstic cell come very close to each other, leaving a tiny gap between their cell membranes. When an active potential is transmitted to a synapse, a complex chain of biochemical processing steps is triggered and neurotransmitter molecules are released from the presynaptic cell to the gap. These molecules will later activate the receptors on the postsynaptic cell membrane, causing ions to influx from the channels on the cell membrane. These ions will change the potential difference across the membrane of the postsynaptic neuron and, in this way, the action potential modulates the postsynaptic neuron [44]. The potential difference across the membrane is called the membrane potential. The membrane potential will accumulate if multiple spikes arrive in a short period of

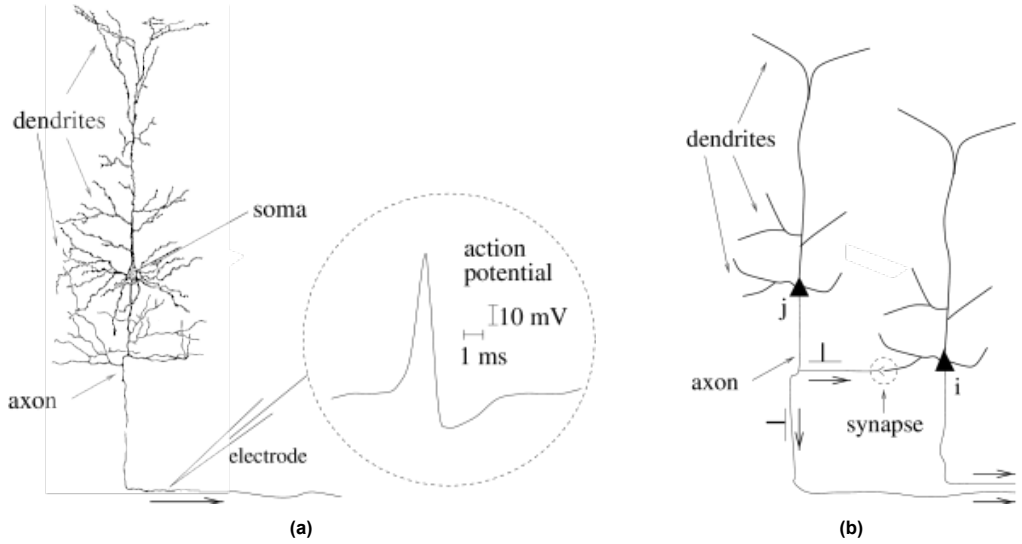

Figure 2.1: (a) A single neuron (by Ramón y Cajal), and a schematic of a neuronal action potential: the action potential is a short voltage of 1-2 ms and an amplitude of about 100mV. (b) A simplified schematic showing the signal transmission from a presynaptic neuron *j* to a postsynaptic neuron *i*. Both figures are taken from [44].

time. When this potential exceeds a threshold, a new spike will be generated and after that the membrane potential is reset to a low level. This process is shown in Figure 2.2.

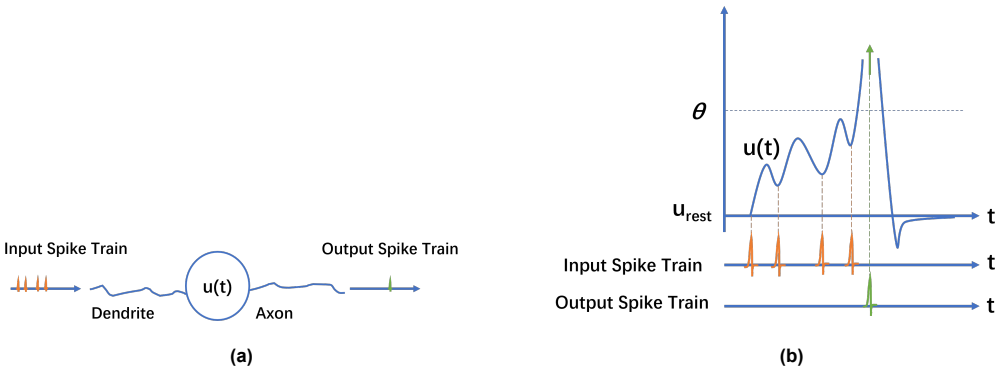

Figure 2.2: (a) A neuron receives a input spike train. (b) The membrane potential curve of the neuron depicted in (a). u(t) is the membrane potential curve. u_{rest}, the resting potential, is the membrane potential after reset. θ is the threshold value. As spikes are received, u(t) accumulates before reaching the threshold θ. When u(t) reach the value θ, this neuron generates an output spike. Then, u(t) resets to u_{rest}. Figures are adapted from [44].

A synapse can be characterized by a parameter w, called *synaptic weight*, *synaptic strength*, or *synaptic efficiency*. Taking the synapse shown in Figure 2.1b as an example, we can use w_{ij} to represent the relationship between the amplitude of the response of the postsynaptic neuron *i* to the arrival of spikes from the presynaptic neuron *j*. If this relationship is positive, the synapse is said to be *excitatory*, otherwise, this synapse is *inhibitory* [44]. In an intuitive sense, an excitatory synapse has a positive weight, and an inhibitory synapse has a negative one.

2.1.1.2. Neuron models in ANNs

As neural networks consist of neurons, the artificial neuron models are first introduced before we look into the connections between these neurons, that is, network architectures. Although inspired by biological neurons, the ”neurons” in a contemporary ANN are usually characterized by a single static continuously valued activation, instead of discrete spikes [101].

As shown in Figure 2.3, a typical **single** neuron in an ANN (a feedforward one, will be explained in Section 2.1.1.3 - **Network Architectures**) takes a set of inputs $\{x_1, x_2, \cdots x_n\}$, where n is the number of presynaptic neurons to which this neuron is connected and $x_i,\ i \in \{1, \cdots n\}$ is the input from the

presynaptic neuron i. These inputs are multiplied by a set of synaptic weights $\{w_1, w_2, \cdots w_n\}$. Here, $w_i,\ i \in \{1, \cdots n\}$ is the adjustable synaptic weight of the input i. The weighted sum of these inputs is computed, and this sum could include a bias, indicated as b in Figure 2.3.

In 1943, McCulloch and Pitts proposed a binary-threshold artificial neuron model [71]. The output of the proposed neuron model can be expressed as [59]:

$$y = \Theta\left(\sum_{i=1}^{n} w_i x_i - u\right). \tag{2.1}$$

In Equation 2.1, Θ is the Heaviside step function, where:

$$\Theta(x) = \begin{cases} 1, & x \geq 0 \\ 0, & x < 0 \end{cases}, \tag{2.2}$$

and u is a threshold value. The McCulloch-Pitts neuron can be generalized, using activation functions instead of the step function in Equation 2.1. Also, in Equation 2.1, we often consider the term $-u$ as a bias. Therefore, the generalized form of a neuron could be expressed as

$$y = f(b + \boldsymbol{wx}), \tag{2.3}$$

where $f(\cdot)$ is the activation function, such as hyperbolic tangent (tanh) or rectified linear unit (ReLU), as shown in Figure 2.3. Compared to the step function, these generalized activation functions are usually differentiable, making it easier to perform learning algorithms (details in Section 2.1.1.6 - **Learning Algorithms**). ***x*** and ***w*** are vector representations of inputs and weights, and

$$\boldsymbol{wx} = [w_1\ w_2\ \cdots\ w_n] \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \sum_{i=1}^{n} w_i x_i.$$

It is possible to find a mapping between input ***x*** and desired output y by adjusting the weight vector ***w*** in Equation 2.3. The process of adjusting weights or other learnable parameters is called *learning*, which will be introduced later in Section 2.1.1.6 - **Learning Algorithms**.

Today, most ANNs are second-generation networks. Together with the high computing power of accessible advanced graphics processing units (GPU) and advanced regularization methods, ANNs have become deeper and deeper with a better ability to generalize hidden data, resulting in overwhelming performance[101].

2.1.1.3. Network architectures

Before we introduce in detail the model of a single neuron in a spiking neural network, knowledge of network architectures will be helpful for understanding key features of neurons in SNNs.

Network architectures are the patterns of neurons' connections. For a single neuron, as mentioned above, it is possible to find a mapping from a set of inputs ***x*** to a single output variable y by adjusting the synaptic weights of the neuron. As neurons are connected in the form of networks, it is also possible to perform mappings between a set of inputs and a set of outputs. However, for different network architectures, the properties of these mappings are also different.

ANNs can be viewed as weighted directed graphs in which neurons are nodes and directed edges are connections between neuron outputs and neuron inputs [59]. Generally, ANNs can be classified into two categories, which are:

- *feedforward* neural networks and
- *recurrent* neural networks (RNNs).

In a feedforward network, loops are not allowed, whereas in a recurrent one, loops occur and form feedback connections. Feedforward networks are considered static and memoryless, that is, the output responses of a feedforward network are independent of historical inputs assuming the neurons maintain no state. On the other hand, recurrent networks are dynamic and outputs may depend on previous inputs because of loops that exist in these networks [59].

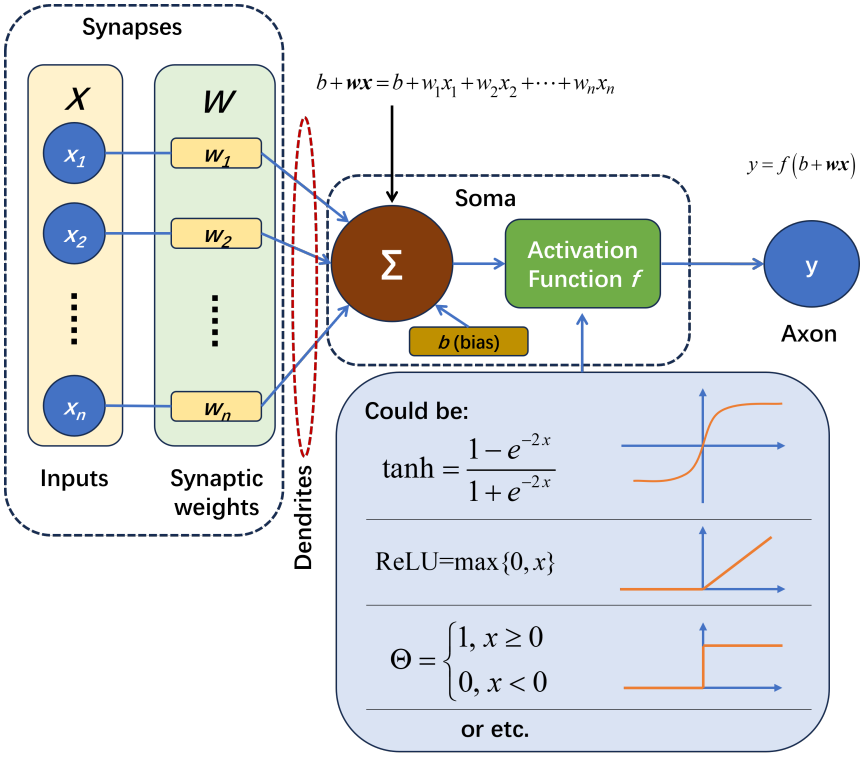

Figure 2.3: A typical neuron model used in ANNs. Associated counterparts of a biological neuron is also shown in this figure: inputs from other neurons or the environment are received with the synapses and then transmitted to the soma through dendrites. The soma processes the inputs and sends the output y to the axon.

One of the most commonly used feedforward networks is called the *multilayer perceptron* (MLP). In a multilayer perceptron, neurons are organized into layers that have unidirectional connections between them [59]. Figure 2.4 illustrates the general concept of a feedforward network, the concept of a single-layer network, and the concept of an MLP. Each node in Figure 2.4 is a neuron, except for the input layers (a neural network can receive input from the environment). Compared to a single-layer network (i.e. perceptron), an MLP includes hidden layers between its input layer and output layer, which make them capable to learn more complex functions. For simplification purposes, we assume that the layers in an MLP are fully connected, i.e., each neuron in the previous layer is connected to all neurons in the next layer and vice versa. In this case, if there are n neurons in the previous layer and m neurons in the next layer, there will be $n \times m$ connections (i.e. $n \times m$ synapses).

Taking the MLP case in Figure 2.4c, the weights of a layer can be expressed as: $w_{ji}, i \in \{1, \cdots n\}, j \in \{1, \cdots m\}$, where w_{ji} is the weight that connects x_i and y_j. For each y_j, the corresponding input weight vector is $\boldsymbol{w}_j = [w_{j1}\ w_{j2}\ \cdots\ w_{jn}]$. From Equation 2.3, assuming that neurons in this layer share the same activation function $f(\cdot)$ and a uniform bias b, we can get the following.

$$y_j = f\left(b + \boldsymbol{w}_j \boldsymbol{x}\right) \ . \tag{2.4}$$

If we rewrite Equation 2.4 with matrices and vectors, we have the following:

$$\boldsymbol{y} = f\left(\boldsymbol{b} + \boldsymbol{W}\boldsymbol{x}\right) \ , \tag{2.5}$$

where:

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}, \ \boldsymbol{W} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix},$$

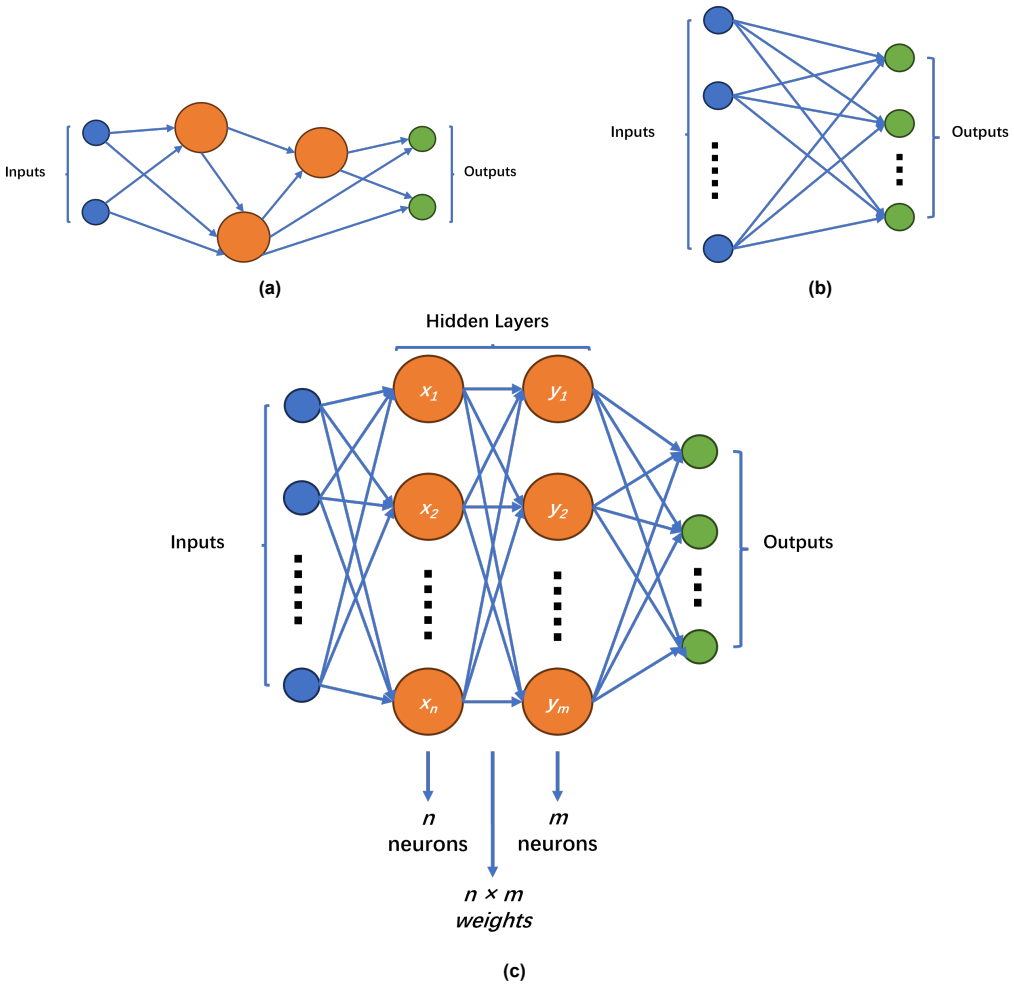

Figure 2.4: (a) A concept graph of a feedforward network: there are no loops, the direction of a edge points from the output of a neuron to the input of another neuron. (b) A concept graph of single-layer fully-connected perceptron: all input units are directly connected to all output neurons. (c) An MLP concept graph: in an MLP, there are hidden layers between the input layer and the output layer. In this graph, the MLP has 2 hidden layers. Assuming that the second layer is fully connected and has m neurons and n inputs, there will be $n \times m$ connections.

$$\boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \ (b_k = b, k \in \{1, \cdots m\}) \ .$$

Equation 2.4 can be applied to each single layer in an MLP and then the matching between the inputs and outputs of the MLP can be derived.

An important application of MLPs is the multi-class classification problem in which the goal is to correctly classify input vectors into a number of possible classes space[10]. Usually, the number of neurons in the output layer is the maximum number of classes that this network can recognize, with one output neuron representing a single class.

There are other feedforward models proposed, such as the convolutional neural network (CNN) [41] and the radial basis function nets (RBF) [12]. These models will not be introduced as they are not used in this project.

As mentioned above, loops are allowed in an RNN, which makes it possible to memorize past states of the hidden layers. Due to this memorizable feature of RNNs, RNNs are usually used to deal with tasks with temporal tasks (e.g., audio recognition / classification), outputs in sequence (e.g., predicting the next word of a set of words) or both (e.g., generating future frames with previous video frames) [4].

2.1.1.4. Neuron model in SNNs

The Hodgkin-Huxley model, proposed by Hodgkin and Huxley, is one of the early attempts to model neuron behaviors[49]. The Hodgkin-Huxley model is an accurate model in biophysics and contains extensive biological details. However, it is also complex with four nonlinear ordinary differential equations [38] and is therefore difficult to use at present due to its high demand for computational resources [33]. Other SNN models include: the spike response model (SRM) [60], the Izhikevich neuron model [57], and the leaky integrate-and-fire model (LIF) [33]. Among these models, the LIF model is extremely popular because it strikes a good balance between biological plausibility and practicality [33]. Therefore, we will focus on the LIF model.

In the LIF model, the basic assumption is that the spikes are always the same (i.e., the same amplitude and the same duration). Based on this assumption, the shape of the spikes is no longer important, and the spikes are reduced to ”events” that occur at a precise moment in time [44]. As introduced above, the behavior of a postsynaptic neuron is closely related to the ion flux across the membrane. Figure 2.5 shows how a neuron is modeled on the basis of its electrical properties. In the LIF model, the voltage across the membrane is U, the flow of ions across the membrane is modeled with a current I_{in}, the membrane is modeled with a capacitor and the channel in the membrane is modeled with a resistor R, as it acts as a pathway for the flow of ions, which charges the capacitor C [33].

First, ignoring the generation of spikes and the reset mechanism at this stage, it is possible to express U, the voltage across the membrane, with I_{in}, R and C. Based on this model, we can get the following equation (for the derivation in detail [44, 33, 32], please refer to Appendix A.1).

$$\tau \frac{\mathrm{d}U(t)}{\mathrm{d}t} = RI_{in}(t) - U(t) \ . \tag{2.6}$$

In Equation 2.6, τ is the time constant and $\tau = RC$. This equation is in continuous-time form and can be rewritten in a discrete form with a timestep Δt ($\Delta t \ll \tau$):

$$U(t + \Delta t) = \left(1 - \frac{\Delta t}{\tau}\right) U(t) + \frac{\Delta t}{\tau} RI_{in}(t). \tag{2.7}$$

We would have to derive a discrete-time expression of Equation 2.7 to adapt it to a digital system. As long as $\Delta t \ll \tau$, we can take Δt as a unit time step in Equation 2.7. We also use β to replace the term $\left(1 - \frac{\Delta t}{\tau}\right)$, which can be called the *decay rate* of the membrane potential. Then, we can get a discrete-time form of Equation 2.7, which is

$$U\left[t + 1\right] = \beta U\left[t\right] + \left(1 - \beta\right) RI_{in}\left[t\right] \ . \tag{2.8}$$

In Equation 2.8, $\left(1 - \beta\right) RI_{in}\left[t\right]$ is the input term at time t. However, in machine learning, the input is always in the form of the product of an input vector and a weight matrix. Notice that the terms $RI_{in}\left[t\right]$ and $U[t + 1]$ are of units [Voltage], if we assume that input and output should have the same units, it is reasonable that we assume that the input vector $\boldsymbol{x}\left[t\right]$ should be of units [Voltage]. Here, we assume that

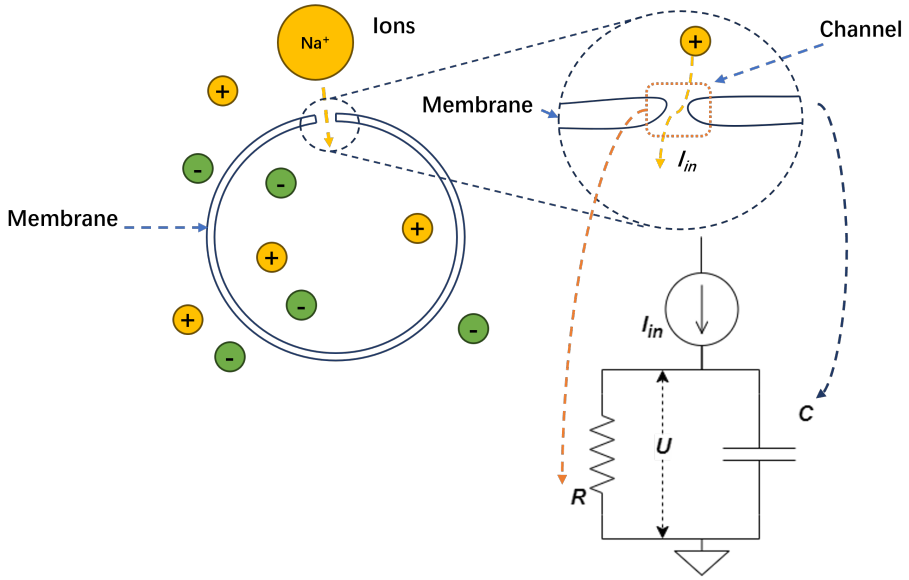

Figure 2.5: Electrical properties of neurons: from biological neuron to LIF model. This figure is inspired by [44] and [33].

we have a resistor of 1 [Ohms], then we can get $1 \cdot I_{in}[t]$ of units [Voltage]. We also need to scale the term $(1-\beta)\,R$ with this 1-Ohm resistance and it becomes $\left((1-\beta)\,\frac{R}{1}\right)$, which is a unitless factor. In this way, we can rewrite the term $(1-\beta)\,RI_{in}\,[t]$ as follows.

$$(1-\beta)\,RI_{in}\,[t] = \left((1-\beta)\,\frac{R}{1}\right)(1 \cdot I_{in}\,[t]) = \boldsymbol{wx}\,[t]\ . \tag{2.9}$$

As a practical neuron should include multiple synapses, it is reasonable to expand the input $I_{in}[t]$ into an input vector. Therefore, in Equation 2.9, the term $1 \cdot I_{in}$ is replaced by the input vector $\boldsymbol{x}\,[t]$, which is of units [Voltage]. And correspondingly, the term $\left((1-\beta)\,\frac{R}{1}\right)$ is replaced by the weight vector $\boldsymbol{w}$, which is learnable. As a result, Equation 2.8 can be rewritten as:

$$U\,[t+1] = \beta U\,[t] + \boldsymbol{wx}\,[t]\ . \tag{2.10}$$

Now, we include the generation of spikes and the reset mechanism in our model. As introduced above, in a biological neuron, a spike is generated as the membrane potential exceeds the threshold. This process is also known as fire. We have:

$$S\,[t] = \Theta\left(U\,[t] - U_{thr}\right) \tag{2.11}$$

and

$$U\,[t+1] = \beta U\,[t] + \boldsymbol{wx}\,[t] - S\,[t]\,\mathcal{R}\,[t]\ . \tag{2.12}$$

Here, $S\,[t]$ is the output spike at t, U_{thr} is the firing threshold, which is learnable. Θ is the Heaviside step function, which is expressed in Equation 2.2. When $S\,[t] = 1$, an output spike is generated. The term $S\,[t]\,\mathcal{R}\,[t]$ is the reset term: when $S\,[t] = 1$, the membrane potential at the next timestep is reset to a resting potential. Assuming that the membrane potential is always reset to zero, we have

$$\mathcal{R}\,[t] = \beta U\,[t] + \boldsymbol{wx}\,[t]\ . \tag{2.13}$$

Equations 2.11, 2.12 and 2.13, together, define a discrete-time LIF model, which is illustrated in Figure 2.6. Notice that in Equation 2.11, $S\,[t]$ is either 0 or 1. As all inputs and outputs are normalized to $\{0, 1\}$, a neuron is activated only when it receives spikes. In this way, a neuron can remain silent if there is no input, which can be exploited to reduce dynamic power consumption. Furthermore, the computation of $\boldsymbol{wx} = \Sigma_{i=1}^{n} w_i x_i,\ i \in \{1, \cdots n\}$ will be less computationally intensive, as the multiply-accumulate (MAC) operation needed to perform this calculation can be greatly simplified with a binary and sparse input vector [87].

Another thing worth noting is that in Equations 2.12 and 2.13, β should be in $(0, 1)$. If $\beta = 1$, the leaky integrate-and-fire model degrades to an integrate-and-fire (I&F) model because there is no leakage in the membrane potential $U\,[t]$. If $\beta = 0$, the model reduces to a stateless artificial neuron in Equation 2.1.

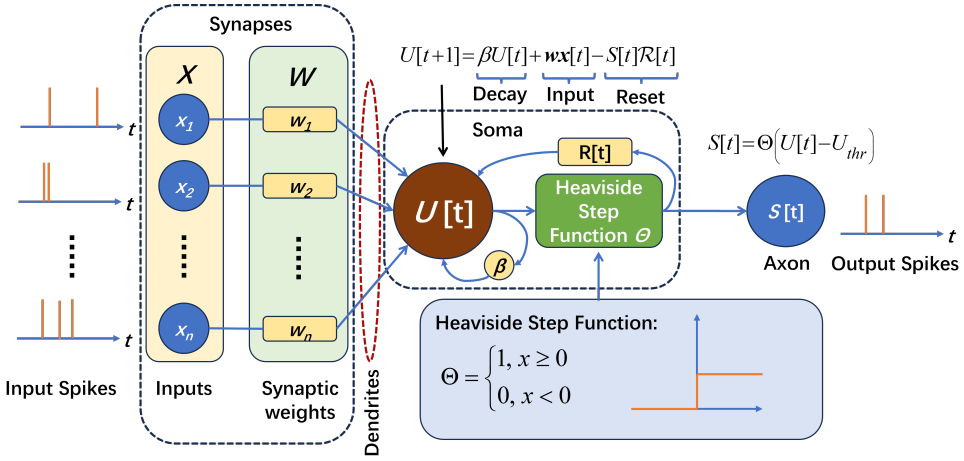

Figure 2.6: A LIF neuron model used in SNN, with n inputs.

2.1.1.5. Spike encoding and decoding in SNNs

As introduced in 2.1.1.1, in a biological neural system, the information is carried by the intervals between spikes. However, in practical applications, inputs are usually not intrinsically in the form of spikes. For example, inputs could be the intensity of pixels in an image or the amplitude in an audio recording. Therefore, unless the sensor emits spikes, we have to convert raw inputs into spikes before feeding them to the network. As outputs from an SNN are also sequences of spikes, it is also necessary to arrange the spikes in a way that is recognizable. These processes are called input encoding and output decoding:

1. **input encoding:** Conversion of input data into spikes which is then passed to a neural network;
2. **output decoding:** Processing of output spikes in a meaningful and informative way [32].

Figure 2.7 shows some encoding and decoding strategies. For input encoding, we have the following options.

1. **Rate coding** converts the input intensity into a firing rate or spike count. As shown in Figure 2.7, after rate coding, the high-intensity input is expressed with a dense spike sequence and the low-intensity input is converted into a sparse spike sequence.
2. **Latency (or temporal) coding** converts the input intensity into the order in which the spikes occur. For example, in Figure 2.7, an input with higher intensity is encoded into a spike that occurs earlier than the spike corresponding to an input with lower intensity.
3. **Delta-modulation** converts a temporal change in input intensity into spikes [32]. As shown in Figure 2.7, spikes occur when the input signal increases. However, this is a basic version of delta-modulation, in which the decreases in a signal are not properly represented. To overcome this problem, we may need to choose another variant of the delta-modulation scheme, which will be discussed in Section 2.2.1.
4. It is also possible that input data can be passed to an SNN without any conversion.

For output decoding, we can choose from the following.

1. **Rate coding** decodes the output value with a firing rate or spike count. For classification problems, as shown in Figure 2.7, the output neuron with the highest firing rate or spike count is chosen as the predicted class.
2. **Latency (or temporal) coding** decodes the output value with a time-relevant spike output. For the classification problem, usually, as shown in Figure 2.7, the output neuron that fires first is chosen as the predicted class [32].

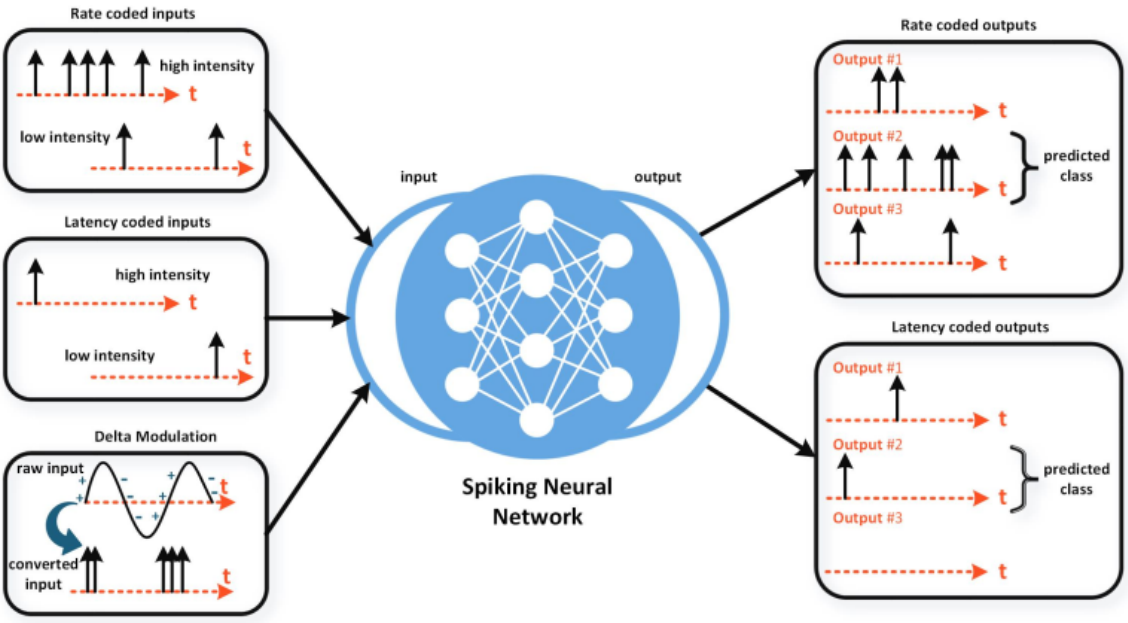

Figure 2.7: Different input encoding strategies and output decoding strategies. For a network, the input encoding and output decoding strategies are independent. [32]

Usually, delta-modulation is used to convert time-varying inputs, such as audio signals. Latency coding or rate coding can be used to convert static inputs, such as an image, into spike sequences. Compared to the latency coding scheme, rate coding provides better robustness and is easier for learning processes as more spikes are provided or generated. However, since more spikes are processed by the SNN under the rate coding scheme, using latency coding is faster and more power-efficient for an SNN [32]. In this project, for input encoding, delta-modulation will be applied because the sensor and interface used in this project are based on this scheme (details in Section 2.2.1). For output encoding, we choose the rate coding scheme, which is simple to implement on both hardware and software.

2.1.1.6. Learning algorithms

As mentioned above, learning is the process of finding the desired match between inputs and outputs by adjusting the weights (or other learnable parameters) in a neural network. In other words, learning is the optimization process of a neural network. During this process, the *learning rules* determine how the weights are updated. A *learning algorithm* refers to a procedure in which learning rules are used to update weights or other parameters [59]. Based on the information available for the models during the learning process, we can classify a learning process into two main paradigms, namely *unsupervised learning*, *supervised learning*.

In supervised learning, the network is provided with a set of input patterns and the corresponding ”correct” output for every single input pattern. Weights are updated to make the outputs of the network as close as possible to the provided output patterns. On the contrary, a ground truth for output patterns is not required for unsupervised learning. Unsupervised learning uses the underlying structure in the data, or correlations between input patterns, to organize them [59]. For example, unsupervised learning is usually used to group input data into several clusters.

Supervised learning is widely used in classification problems. Because we will mainly deal with classification problems in this project, we will introduce learning algorithms on the basis of the supervised learning scheme. Unless specified otherwise, further discussion on learning in the remainder of this thesis refers to supervised learning.

In supervised learning, the provided dataset of input patterns and the corresponding outputs is called the training set. We can define a training set as $\mathbb{T} = \left\{ \left(\boldsymbol{x}^{(1)}, \boldsymbol{d}^{(1)} \right), \left(\boldsymbol{x}^{(2)}, \boldsymbol{d}^{(2)} \right), \cdots \left(\boldsymbol{x}^{(p)}, \boldsymbol{d}^{(p)} \right) \right\}$, which is a set of p training patterns (input-output pairs), where $\boldsymbol{x}^{(i)} \in \mathbb{R}^n$ is the input and $\boldsymbol{d}^{(i)} \in \mathbb{R}^m$ is the output ground truth. Here, m represents a maximum of m classes in classification problems [59].

A common way to optimize a network is to first define a loss function $\mathcal{L}$, which quantifies the distance between the expected output and the actual output of a neural network [59]. Some of the commonly used loss functions include the mean squared error (MSE) loss and the cross-entropy (CE) loss. Then, the goal of the learning algorithms is to minimize the loss function in the training process. In these training processes, especially for training an MLP, two types of algorithm are commonly used: one is the back-propagation algorithm and the other one is one of the various optimization algorithms.

The backward propagation of error (back-propagation or BP in short) algorithm is probably the most well known algorithm in the field of machine learning. When we use a feedforward neural network, the process of giving an input $\boldsymbol{x}$ and eventually getting an output $\boldsymbol{y}$ is called *forward propagation*, in which the information propagates from the input to the output through the hidden units of a neural network. On the contrary, backward propagation allows the error gradient information to flow backwards from the output to the input, using the chain rule for derivatives shown in Equation 2.14 [46].

$$\frac{\partial y}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial y}{\partial u_j} \frac{\partial u_j}{\partial x_i} \,. \tag{2.14}$$

In Equation 2.14, we assumed that y is a scalar, $y = f(\boldsymbol{u})$, $\boldsymbol{u} = [u_1\ u_2\ \cdots\ u_m]$, $u_j = g_j(\boldsymbol{x})$ and $\boldsymbol{x} = [x_1\ x_2\ \cdots\ x_n]$. If $m = 1$, we can get a simplified version of equation 2.14, that is,

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x_i} \,. \tag{2.15}$$

With the chain rule mentioned above, we can use BP to calculate the gradients with respect to all the required parameters in the network, across all layers of the network (e.g., an MLP) [59, 10].

As an example, the processes from forward propagation to back-propagation in a fully-connected MLP are described in Algorithms 1 and 2.

Algorithm 1: Forward propagation through a typical fully-connected MLP, adapted from [46].

Require : L: Network depth
Require : $\boldsymbol{W}^{(i)}, i \in \{1, 2, \cdots L\}$: The weight matrices of each layer
Require : $\boldsymbol{b}^{(i)}, i \in \{1, 2, \cdots L\}$: The bias parameters of each layer
Require : $\boldsymbol{x}$: The input to the network

$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
for $k = 1, 2, \cdots L$ **do**
  $\boldsymbol{a}^{(k)} = \boldsymbol{b}^k + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$ ($\boldsymbol{a}^{(k)}$ is the activation of the k^{th} layer)
  $\boldsymbol{h}^{(k)} = f\left(\boldsymbol{a}^{(k)}\right)$ ($\boldsymbol{h}^{(k)}$ is the output of the k^{th} layer)
end
$\boldsymbol{y} = \boldsymbol{h}^{(L)}$ ($\boldsymbol{y}$ is the output of the MLP)

Algorithm 2: Back-propagation through a typical fully-connected MLP after a feedforward process, based on the results from Algorithm 1, adapted from [46].

Require : $\boldsymbol{d}$: Desired output pattern
Require : $\mathcal{L}$: The loss function

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{y}} \mathcal{L} = \nabla_{\boldsymbol{y}} \mathcal{L}(\boldsymbol{y}, \boldsymbol{d})$
for $k = L, L-1, \cdots 1$ **do**
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} \mathcal{L} = \boldsymbol{g} \odot f'\left(\boldsymbol{a}^{(k)}\right)$ (Convert the gradient on the layer's outputs into a gradient on the activations)
  $\nabla_{\boldsymbol{b}^{(k)}} \mathcal{L} = \boldsymbol{g}$ (Calculate the gradients on the biases of the k^{th} layer)
  $\nabla_{\boldsymbol{W}^{(k)}} \mathcal{L} = \left(\boldsymbol{g} \otimes \boldsymbol{h}^{(k-1)}\right)^{\mathsf{T}}$ (Calculate the gradient on the weights of the k^{k} layer)
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} \mathcal{L} = \boldsymbol{g}\boldsymbol{W}^{(k)}$ (Propagate the gradient w.r.t. the next lower-level hidden layer's activations)
end

In Algorithm 2, we can see that the gradient computation process starts from the output layer and goes backward to the first hidden layer. See Appendix A.2 for details of how the gradients are calculated layer by layer.

Algorithms 1 and 2 show how to compute gradients on $\mathcal{L}$ based on a single input-output pair. In practical use, we usually use a small batch of data, rather than a single input-output pair. A batch is a subset of the entire training set. A batch with p' samples can be expressed as:

$$\mathbb{B} = \left\{ \left(\boldsymbol{x}^{(1)}, \boldsymbol{d}^{(1)} \right), \left(\boldsymbol{x}^{(2)}, \boldsymbol{d}^{(2)} \right), \cdots \left(\boldsymbol{x}^{(p')}, \boldsymbol{d}^{(p')} \right) \right\}, \ \mathbb{B} \subseteq \mathbb{T}.$$

In this case, the loss function should cover all samples in a batch. Therefore, the loss function is written as follows:

$$\mathcal{L}_B = \frac{1}{p'} \sum_{i=1}^{p'} \mathcal{L} \left(\boldsymbol{y}^i, \boldsymbol{d}^i \right) \ . \tag{2.16}$$

Gradients are now expressed as

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_B = \frac{1}{p'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{p'} \mathcal{L} \left(\boldsymbol{y}^{(i)}, \boldsymbol{d}^{(i)} \right) \ , \tag{2.17}$$

where $\boldsymbol{\theta}$ is the learnable parameter in the network (e.g. $\boldsymbol{W}$ or $\boldsymbol{b}$).

The BP algorithm provides a way to compute the gradients, and optimization algorithms, on the other hand, use these gradients to update the parameters in the network and minimize the loss. The stochastic gradient descent (SGD) algorithm is one of the widely used and important optimization algorithms. Many optimization algorithms are developed from SGD, including a popular algorithm called *Adam*, whose name is derived from the phrase "adaptive moments" [46]. The Adam algorithm is shown in Algorithm 3.

Algorithm 3: The Adam Algorithm, adapted from [46] and [62].

Require : α: Step size (suggested default: 0.001)
Require : $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for moment estimates (suggested default: 0.9 for β_1 and 0.999 for β_2)
Require : ϵ: A small constant for numerical stability (suggested default: 10^{-8})
Require : $\boldsymbol{\theta}$: Initial parameters

1 $\boldsymbol{m} \leftarrow \boldsymbol{0}$ (Initialize 1^{st} moment vector)
2 $\boldsymbol{v} \leftarrow \boldsymbol{0}$ (Initialize 2^{st} moment vector)
3 $t \leftarrow 0$ (Initialize time step) **while** *stopping criterion not met* **do**
4 Get a new minibatch $\mathbb{B}$ from the training set $\mathbb{T}$.
5 $\boldsymbol{g} \leftarrow \frac{1}{p'} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{p'} \mathcal{L} \left(\boldsymbol{y}^{(i)}, \boldsymbol{d}^{(i)} \right)$ (Compute gradients using Algorithm 2)
6 $t \leftarrow t + 1$
7 $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + (1 - \beta_1) \, \boldsymbol{g}$ (Update biased first moment estimate)
8 $\boldsymbol{v} \leftarrow \beta_1 \boldsymbol{v} + (1 - \beta_2) \, \boldsymbol{g} \odot \boldsymbol{g}$ (Update biased second moment estimate)
9 $\hat{\boldsymbol{m}} \leftarrow \frac{\boldsymbol{m}}{1 - \beta_1^t}$ (Compute bias-corrected first moment estimate)
10 $\hat{\boldsymbol{v}} \leftarrow \frac{\boldsymbol{v}}{1 - \beta_2^t}$ (Compute bias-corrected second moment estimate)
11 $\Delta \boldsymbol{\theta} = -\alpha \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{v}}} + \epsilon}$ (Compute update, $\sqrt{\hat{\boldsymbol{v}}}$ is applied element-wise)
12 $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$ (Apply update)
13 **end**

As shown in Algorithm 3, Adam sets an update rule and updates the parameters based on the gradients acquired in Step 5. α is also known as the learning rate, which is a parameter controlling the step length of updates. β_1 and β_2 are parameters that adjust the learning rate in the learning process. $\boldsymbol{m}$ and $\boldsymbol{v}$ are moment vectors. These vectors are introduced to accelerate the learning process.

Generally, Adam is regarded as a robust algorithm [46], and also performs well on recurrent networks in addition to feedforward networks [33].

Quantization

Quantization is the process of reducing the number of bits of the parameters, including synapse weights, in a neural network model. Reducing the bit precision of the parameters allows for a significant improvement in memory utilization and power efficiency [96]. For example, it is common to quantize 32-bit floating-point parameters into integers with lower bitwidth.

There are two main methods of quantization, namely quantization-aware training (QAT) and post-training quantization (PTQ). QAT involves the quantized network model directly in the training process, and PTQ quantizes an existing network, without retraining [109]. In most cases, QAT results in a better accuracy than PTQ.

To quantize a value, a quantization function mapping high-precision values to lower ones must be provided. A common quantization function that maps a floating-point value r to an integer is

$$Q(r) = Int(r/S) - Z \ . \tag{2.18}$$

In Equation 2.18 [109], S is a floating-point scaling factor and Z is an integer zero point [45]. r is the parameter (e.g.$\sim$weight) to be quantized. $Int(\cdot)$ is the function that assigns the scaled r to an integer, such as a simple ceiling function. The quantization function in Equation 2.18 performs *uniform quantization* as all r values are scaled by the same factor S so that the distance between quantized values is the same. *Non-uniform quantization*, on the contrary, will result in non-uniform distances between quantized values.

The scaling factor in Equation 2.18 can be expressed as

$$S = \frac{\beta - \alpha}{2^b - 1} \ , \tag{2.19}$$

where β and α are the upper bound and lower bound of the range of quantized values and b is the quantization bit width [109]. If $\alpha = \beta$, the quantization is symmetric and Z is 0.

2.1.2. Related Work

In this sub-section, we will first introduce the key event-based interface widely used in neuromorphic hardware systems. Then, some state-of-the-art SNN hardware will be introduced. In this part, we will also focus on the reconfigurability of these hardware systems. At the end, a brief introduction to the available training framework for SNN will be provided.

2.1.2.1. Event-based interface

Before we introduce the state-of-the-art hardware for SNNs, we would like to first introduce the key interface used in SNN hardware.

Address-event representation (AER) is an inter-chip point-to-point communication protocol. This protocol is designed based on the characteristic of SNNs: events (spikes) are sparse in time, and computation is only required when an event (spike) is generated or received [87]. Whenever a neuron fires, its address is output as a packet to the inter-chip data bus. Based on the assumption that the interval between the events (spikes) in a neuromorphic system is much longer than the time needed to transmit an address, addresses can be multiplexed on a digital bus. The receiver chip then can decode the addresses into corresponding events [86]. An AER packet can also include other types of information (e.g., the time of events or accompanying instructions) along with the addresses [110].

Each communication cycle of the AER protocol involves a *four-phase handshake* [11] mechanism. This four-phase handshake is controlled by two handshake signals, namely `request` from the sender and `acknowledge` from the receiver. The AER communication channel and the timing between a sender and a receiver are shown in Figure 2.8.

In many cases, a purely point-to-point communication protocol might be too simple to handle the communication between the cores, and hence limit the design of more complex architectures. In this case, other routing methods (e.g., multicast) are required, as well as an extended or modified AER protocol, specifically the packet structure [110].

2.1.2.2. SNN hardware

Unlike non-spiking ANNs, which can be greatly accelerated by GPUs with powerful tensor computation ability, the sparse event-driven and stateful nature of SNNs is ill-suited for GPUs or CPUs, and hence needs specifically designed hardware. To improve the efficiency of SNNs, neuromorphic hardware systems optimized for SNNs have been developed over the past decade [79]. These efficient hardware systems enable the deployment of SNNs on edge devices, including mobile phones and internet-of-things (IoT) devices. Some of these systems also support online learning, allowing learning on the device with dedicated synaptic weight update circuitry, in addition to inference with trained networks.

In general, these neuromorphic systems can be classified into *analog/mixed-signal* and *digital* systems. Although memristor-based neuromorphic hardware has shown great potential in terms of area and energy

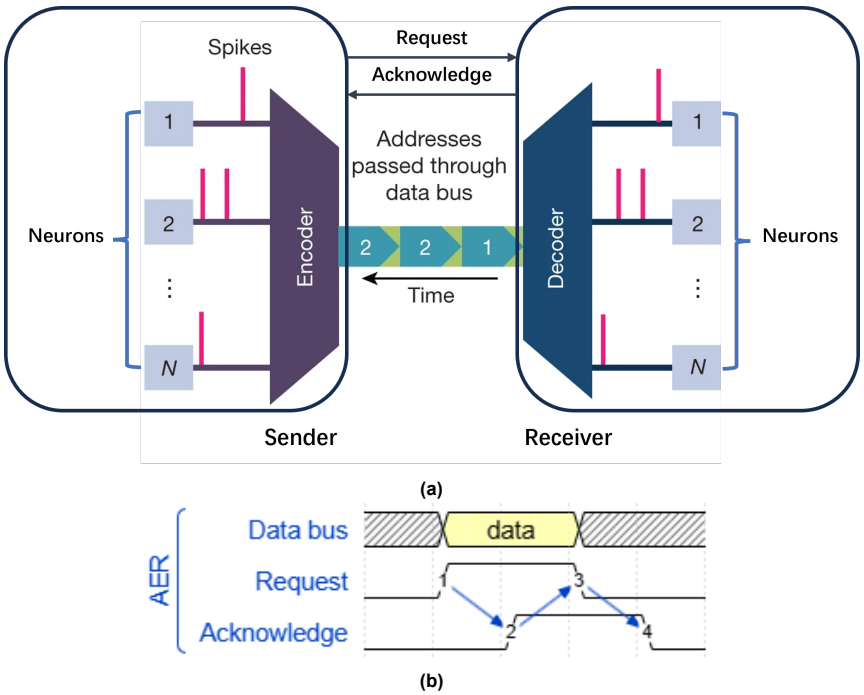

Figure 2.8: (a) Schematic of an AER communication channel. In this scheme, a sender and a receiver are connected with a data bus and a `request` line from the sender, and an `acknowledge` line from the receiver. A spike from neuron 1 of the sender is first coded and sent to the data bus, followed by two events from neuron 2. The receiver then receives and decodes the events from different neurons in exactly the same order. The sender and receiver could be different cores of the same chip, different chips connected together, or a sensor output interface and the input of a processor. Figure adapted from [87]. (b) Timing diagram of an AER communication cycle. A communication cycle includes four phases: the `request` signal from the sender is first pulled high as the data are sent to the bus (edge 1); then the receiver responds with a high signal on the `acknowledge` line after acquiring the data (edge 2); once the sender receives the acknowledgment, it pulls down the `request` (edge 3); finally, as the receiver receives a low signal on the `request` line, the `acknowledge` is pulled down and a communication cycle is accomplished (edge 4).

efficiency [86], we will not discuss these systems because they are not yet mature. Besides these two categories, we also want to emphasize the *configurability* and *scalability* of the hardware systems. An overview of the neuromorphic hardware systems based on the two categories and the two specific topics is given below.

Analog/mixed-signal neuromorphic processors

In an analog/mixed-signal scheme, the subthreshold or weak-inversion circuit not only saves power but also provides biological emulation for neuron behavior [86], since the current flow in the MOS transistor follows a diffusion mechanism, identical to that of the ion flow through an ion channel of a neuron cell [38].

The reconfigurable online learning spiking neuromorphic processor (ROLLS) [84] has 128k analog synapses and 256 adaptive exponential integrate-and-fire (AdExp[1]) neurons. Its scaled-up version, the 180-nm dynamic neuromorphic asynchronous processors (DYNAPs) [72] chip, is a quad-core system with 256 analog AdExp neurons implemented in each core. The digital communication logic of DYNAPs is fully asynchronous. The dynamic neuromorphic asynchronous processor with scalable and learning (Dynap-SEL) is an improved version of DYNAPs with an advanced 28-nm fully depleted silicon on insulator (FDSOI) process [83, 102]. Dynap-SEL can be integrated into an array of up to 256 chips, which forms a large system. Neurogrid [9] is another mixed-signal multichip neuromorphic system designed to perform biological emulation of biological neural networks. The whole system consists of 16 Neurocore chips connected in a binary tree structure on a printed circuit board (PCB). Each Neurocore chip supports 64k neurons with its 180-nm MOS transistors operating in the subthreshold regime [9, 42].

Another kind of analog/mixed-signal circuit emulates the mathematical models of neurons or synapses. In this case, the MOS transistors operate in the above-threshold regime [38].

BrainScaleS [91] is a wafer-scale neuromorphic system, implemented with an above-threshold analog circuit that models 180k AdExp neurons and 40 million synapses on a single 20-cm wafer. This above-threshold analog circuit, together with a high-speed digital communication network, can run at an acceleration factor that ranges from 10^3 to 10^5 compared to biological time constants [91]. Multiple wafers can also be connected together through inter-wafer connections to form a cluster [42]. A second-generation BraubScaleS has also been designed [73], which uses a more advanced 65-nm process.

Digital neuromorphic processors

Compared to analog/mixed-signal circuits, fully digital circuits are much less sensitive to non-ideal effects, such as mismatch, noise, and PVT (process, voltage, and temperature) variations. Digital circuits also provide high programmability.

Following a distributed von Neumann architecture, SpiNNaker [77] is designed to model and simulate large-scale SNNs. This system is composed of a large array of nodes, each containing a chip multiprocessor (CMP) with 18 ARM968 processing cores. Each processing core in the CMP is capable of simulating up to 1000 neurons. The full-size system contains more than 1 million processors, organized in a globally asynchronous and locally synchronous (GALS) architecture [77].

Another GALS design is the IBM TrueNorth [16, 2]. This fully digital 28-nm chip contains 4096 neurosynaptic cores. Each core implements a 256$\times$256 crossbar connecting 256 neurons, each with 256 synaptic inputs [16, 42].

Loihi [27] , developed by Intel, is a 14-nm digital neuromorphic chip. It is a fully asynchronous digital design that supports online learning. It consists of 128 neuromorphic cores and a total of 130k neurons.

Darwin [69] is a neuromorphic co-processor targeted for embedded applications [110]. By using a time multiplexing scheme, Darwin can simulate up to 32k LIF neurons with only 8 physical neurons on the chip [69]. As a co-proccessor, it stores the information of synapses in off-chip memory.

The online-learning digital spiking neuromorphic processor (ODIN) [40] is based on a 28-nm FDSOI CMOS process. Its 256 neurons support both the Izhikevich model and the LIF model. These 256 neurons are organized under a 256$\times$256 time-multiplexed architecture. The tinyODIN is a simplified version of ODIN without online-learning capability. It uses the same crossbar structure as ODIN, which is able to simulate 256 LIF neurons [37].

MorphIC [39] is a 65-nm quad-core neuromorphic processor organized with a hierarchical routing scheme. Each core comprises 512 LIF neurons in a crossbar array. It also supports online learning.

[1]Nomenclature: this model is first introduced in [14] as "aEIF". Other abbreviations include "AEI&F" [29], "AdExp-I&F" [110] or just "AdExp" [38].

Wenquxing 22A [108] is a neuromorphic processor based on a general-purpose RISC-V CPU. By integrating SNN units into the architecture of a CPU, the general-purpose CPU can handle neuromorphic tasks based on the LIF model, including online learning.

SENECA [100] is a digital neuromorphic architecture that offers great configurability with a hierarchical-controlling system. In this architecture, a RISC-V controller and a loop buffer are integrated, providing flexible pipeline controls and memory controls. This architecture supports various algorithms and can scale up into a many-core structure.

Reconfigurable neuromorphic hardware

Reconfigurable hardware provides users with high configurability to adjust the connections between neurons, the parameters of neurons and synapses, and even adjust the structure or composition of hardware itself. Here, to better compare the configurability, we divide these reconfigurable hardware systems into two categories: *on-chip reconfigurable hardware* and *reconfiguable hardware generators*.

On-chip reconfigurable hardware

On-chip reconfigurable hardware here refers to hardware systems that allow users to program connections in the network or the parameters of neurons and synapses. Most neuromorphic hardware systems provide users with on-chip configurability and allow them to define the connections of the network, for example:

- ROLLS provides configurable synapse properties, network topology, and neuron properties [84]. That is, the activation modes of synapses, the connectivity between synapses and neurons, and the behavior of neurons can be reconfigured.
- The TrueNorth architecture [16] is configurable with various parameters. By configuring these parameters, users can choose from a range of neuron behaviors: 2 synapse modes, 2 leak modes, 4 leak direction modes, 3 threshold modes, and 6 reset modes. Based on this, it is also possible to perform various functions with a single neuron or multiple neurons. For example, various neuron encoding schemes, including rate coding and population coding, and various arithmetic functions, including division, square root, etc.
- Loihi [27] offers high configurability for users, including configurable online learning rules, variable synaptic weight resolution (could be signed or unsigned) and delay, configurable spike filtering window for online learning, etc.
- DARWIN uses a time-multiplexing architecture with a reconfigurable memory array. This allows users to choose between a higher number of neurons and a higher time resolution of synapses. That is, with fewer neurons, there will be more space in the memory array for more fine-grained spike timing resolution and vice versa. This is defined as a configurable degree of time-division multiplexing [69].
- ODIN provides two optional neuron models to choose from, that is, the Izhikevich model and the LIF model. The parameters for these models are also programmable [40].
- SENECA offers great on-chip configurability. That is, it supports various types of neuron and synapse model because no specific models are built. Instead, different neuron/synapse/learning models are realized with the instructions running on the controllers. With its flexible memory structure, it also supports different network models, such as MLPs, spiking CNNs, and recurrent spiking neural networks (RSNNs) [100].

Reconfigurable hardware generators

The field programmable gate array (FPGA) is a type of hardware that allows prototyping arbitrary digital logic through its programmable interconnects. In this way, it is possible to prototyping application-specific neural network architectures and adapt them when needed.

The reconfigurable architecture for neuromorphic computing (RANC) [70] is a software-hardware ecosystem that can generate FPGA implementations of neuromorphic processors with software-specified parameters. These parameters include number parameters (e.g., the number of axons per core, the number of neurons per core, etc.) and bitwidth parameters (e.g., the bitwidth of a weight, the bitwidth of a potential value, etc.).

Scalability of neuromorphic hardware systems

Scalability is important for a neuromorphic hardware system. Scaling up a system enables it to handle neural networks with higher complexity, while scaling down a system reduces its power consumption and resource usage.

For silicon-based neuromorphic hardware systems, scalability here means the ability to build larger systems using individual cores, chips, or platforms. For example, multiple BrainScaleS wafers can be meshed together and form a computing platform of the EU Human Brain Project [42]. Scaling up a system in this way will be a great challenge for the routing system, which should ensure that the communication between chips/systems is reliable.

In FPGA implementations, scaling can be done by configuring the parameters of corresponding units or memory. For example, by simply decreasing the number of neurons and the number of axons per neuron, RANC can generate a smaller core that occupies less look-up tables (LUTs) and block memory in an FPGA [70]. In this sense, a reconfigurable system has a higher potential for scalability. An FPGA-based system can also be scaled by increasing or decreasing the number of cores in a multi-core system. In this case, a routing method and a corresponding router architecture are necessary.

2.1.2.3. SNN training framework

In practice, a neural network is typically trained through a machine learning framework, such as PyTorch and TensorFlow, which are capable of loading data in the workflow, creating models, optimizing model parameters, and saving trained models [82].

Despite the fact that there are many frameworks available, they are by default not intended for SNN tasks. There are frameworks specifically designed for SNN, such as *SpikingJelly* [35] and *snnTorch* [33]. As the training framework is not the main concern of this project, we will only briefly introduce one of the key SNN training frameworks: snnTorch.

snnTorch

snnTorch is a Python package to perform gradient-based learning with spiking neural networks, which applies back-propagation rules to SNNs. It extends the capabilities of PyTorch, taking advantage of its GPU-accelerated tensor computation and applying it to networks of spiking neurons [33]. snnTorch has integrated some predesigned neuron models, including the LIF model. Taking a network of LIF neurons, we will explain how snnTorch applies the learning algorithms we discussed previously.

The LIF model provided by snnTorch is a modified version of the one defined in Equations 2.11, 2.12 and 2.13. The three equations that define the LIF model (assuming that the membrane potential is reset to zero) in snnTorch are listed as follows:

$$S\left[t\right] = \Theta\left(U\left[t\right] - U_{thr}\right) \ , \tag{2.20}$$

$$U\left[t+1\right] = \beta U\left[t\right] + \boldsymbol{wx}\left[t+1\right] - S\left[t\right]\mathcal{R}\left[t+1\right] \ , \tag{2.21}$$

and

$$\mathcal{R}\left[t+1\right] = \beta U\left[t\right] + \boldsymbol{wx}\left[t+1\right] \ . \tag{2.22}$$

The modification in orange is made to map to an RNN representation, without any change in performance [33]. We can easily recognize the term $\boldsymbol{wx}[t+1]$ in Equation 2.21 as the current input and the terms $\beta U[t]$ and $S[t]R[t+1]$ as influences from past states. In this way, this discrete and recursive form of the LIF model can take advantage of developments in the training of RNNs and sequence-based models [33].

The spiking neuron model defined by Equation 2.20, 2.21 and 2.22 can also be unrolled like an RNN model. Figure 2.9 illustrates an unrolled graph of a spiking neuron. Here, $I_{in}[t] = \boldsymbol{wx}[t]$.

By unrolling an SNN into a computational graph in Figure 2.9, it is possible to calculate the outputs, losses, and gradients based on the flow of information over time. With this computational graph, β can be treat as a learnable parameter instead of a hyperparameter.

Although we have a clear computational graph, there is still a problem when we try to apply back-propagation to the network: the Heaviside step function $\Theta(\cdot)$ is not differentiable. When we try to get the partial derivative of $\mathcal{L}$ w.r.t.∼a weight w, $\frac{\partial \mathcal{L}}{\partial w}$, we are supposed to get the following equation with the chain rule [33]:

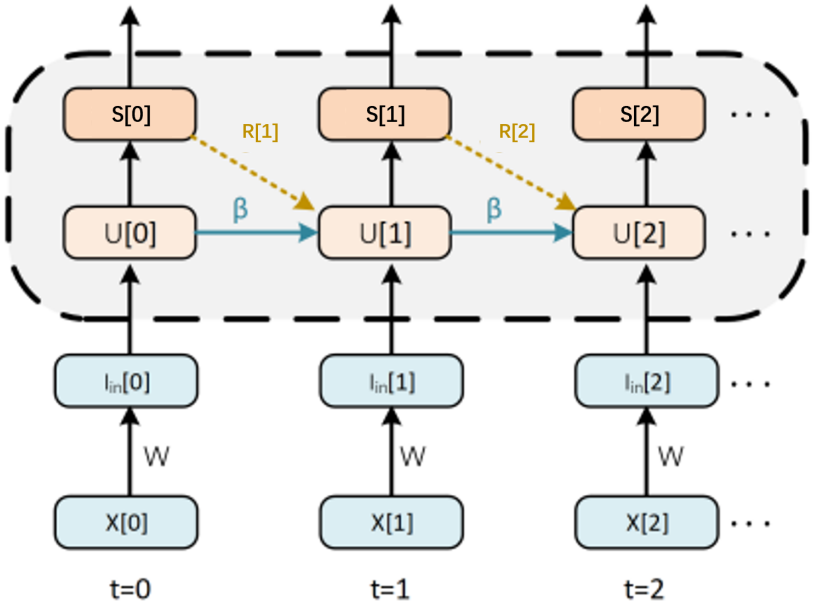

Figure 2.9: Equivalent spiking neuron illustrated with an unrolled computational graph, adapted from [33].

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial S}\frac{\partial S}{\partial U}\frac{\partial U}{\partial w} \,. \tag{2.23}$$

Here, due to the non-differentiability of the step function, the partial derivative of a spike w.r.t.∼a membrane potential, $\frac{\partial S}{\partial U}$, is either 0 or ∞. Therefore, in the backward pass, the Heaviside step function is replaced by a differentiable function. For example, we can choose the shifted arc-tan function and use $\frac{\partial \tilde{S}}{\partial U}$ to replace $\frac{\partial S}{\partial U}$ [33], where

$$\frac{\partial \tilde{S}}{\partial U} = \frac{1}{\pi}\frac{1}{1+(\pi(U-U_{thr}))^2} \,. \tag{2.24}$$

Here, $\tilde{S} = \frac{1}{\pi}\arctan\left(\pi\left(U - U_{thr}\right)\right)$. In this way, Equation 2.23 becomes [33]

$$\frac{\partial \mathcal{L}}{\partial w} \simeq \frac{\partial \mathcal{L}}{\partial S}\frac{\partial \tilde{S}}{\partial U}\frac{\partial U}{\partial w} \,. \tag{2.25}$$

This replacement method is called the surrogate gradient approach, which is only applied in the back-pass process. The default surrogate gradient method in snnTorch is the arc-tan surrogate gradient, as shown in Equation 2.24 [32].

With the unrolled computational graph and the surrogate gradient, snnTorch is able to perform the back-propagation algorithm. snnTorch provides several loss functions to choose from, and optimization algorithms, including Adam, are also integrated in this PyTorch-based framework. The entire learning process of an SNN can be thus performed with snnTorch.

Quantization tool: Brevitas

snnTorch itself cannot involve weight quantization in the training process, although it can perform quantization on the membrane potentials. Therefore, we need to introduce a quantization tool to perform weight quantization.

Brevitas is a suitable tool that supports uniform quantization. Brevitas supports various quantization schemes and can cooperate with various frameworks under a single unified application programming interface (API) [78]. It has been proven that Brevitas can work well with snnTorch [33] and an example can be found in [31]. Brevitas is also capable of performing both QAT and PTQ techniques in the learning process.

2.2. Application of Neuromorphic Hardware in Heterogeneous Computing

With neuromorphic hardware designs and suitable network models, it is possible to conduct a number of machine learning applications on a neuromorphic hardware system, e.g. robotic, biomedical, or computer vision applications, to edge-cutting neuroscience research such as the human brain project (HBP) [54], and the SyNAPSE program [2].

In real-world applications, sensors are often included in the system to collect information and feed it to neural networks running on neuromorphic processors. Therefore, in this section, we will first introduce event-based sensors and event-based encoding interfaces. Then, we will introduce sensor-fusion. Finally, we will introduce a specific real-world application based on sensor-fusion, which will be used as a benchmark for this project.

2.2.1. Event-based sensors

Event-based sensors record information as a sequence of events, similarly to biological neural systems. Therefore, event-based sensors are naturally compatible with SNNs and are a promising avenue for low-power computing [87]. These sensors usually work asynchronously and rely on the AER protocol (as discussed in Section 2.1.2.1) to communicate with a neuromophic processing system.

Event-based vision sensors

An event-based camera is a bio-inspired vision sensor that acquires visual information in an event-based manner. These cameras can output a stream of events that can signal the intensity, local spatial contrast, or even temporal changes in the brightness[2] of each pixel.

The DVS [67], which stands for *dynamic vision sensor*, is probably the most well-known event-based camera. It outputs events signaling brightness changes of each pixel with a differencing circuit. The pixel-level circuit and the working principle of the DVS are shown in Figure 2.10.

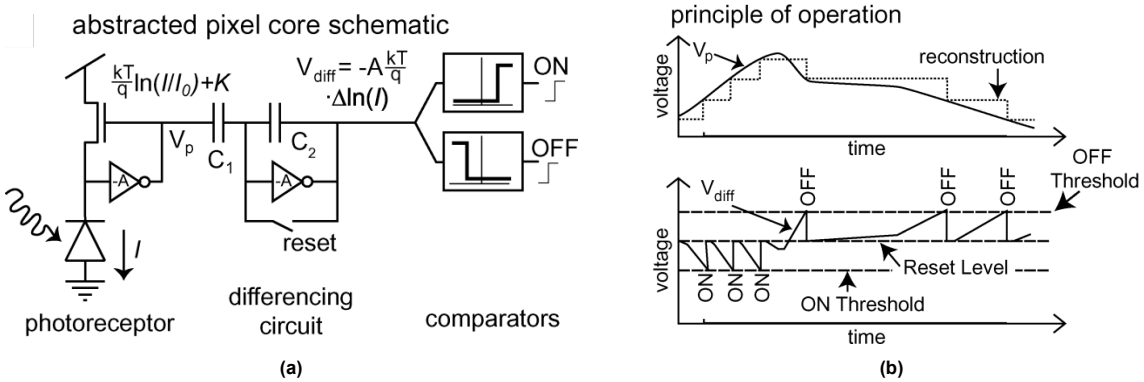

Figure 2.10: The pixel-level schematic and working principle of DVS. (a) The pixel-level schematic. Changes in intensity lead to a log change in V_{diff}. (b) The working principle of DVS. When changes in V_{diff} reach a threshold, an on or off spike is triggered depending on the sign of V_{diff}. Then, V_{diff} is reset to zero. This process is asynchronous. Both figures taken form [67].

Because of its working principle, a DVS is able to output events asynchronously. As DVS pixels are only sensitive to changes in brightness, given a steady scene illumination (i.e., steady background), the events output from a DVS usually indicate movement of objects in the field of view, without background information. The event-based output scheme of the DVS offers: high temporal resolution ($\sim 1\mu$s), low latency (each pixel outputs independently), low power (power only used to process changing pixels), and high dynamic range [43]. Figure 2.11 compares the output of a frame-based camera and a DVS [32].

Other event-based sensors include: DAVIS240 [13], ATIS [80], DVS-Gen4 [99], etc.

Event-based auditory and olfactory sensors

Event-based sensors are also available for auditory or olfactory sensing tasks.

[2]The term ”brightness” refers to log intensity [43].

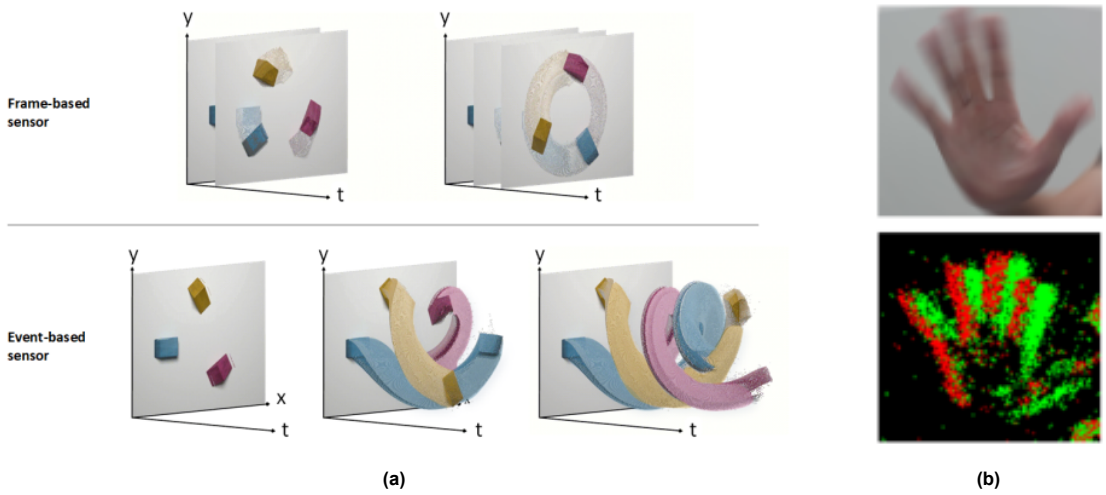

Figure 2.11: Comparison between a frame-based camera and a DVS. (a) Outputs from a frame-based sensor (above): The movements of the three moving objects are recorded together with the background at a fixed frame rate. Motion blur can occur due to a limited shutter speed. Outputs from a DVS (below): Only the movements of these objects are recorded continuously and asynchronously, since pixels only react to a change in brightness. (b) Illustration of a frame-based camera and a DVS for a waving hand. Output from a frame-based camera (above): an image includes a background and a blurred hand. Output from a DVS in a period of time (below): green dots and red dots represent events of different polarities (the black background is only for illustration, the information of steady background without any changes in brightness will not be output).Figures are from [32].

For auditory tasks, a good example is the AER EAR [19], which is a matched silicon cochlea pair with an AER output interface. This sensor is able to provide output over 32 channels with its cascaded low-pass filters, each channel being responsive to an adjustable frequency range. Other event-based auditory sensors include [66] and [68].

For olfactory sensors, also known as electronic noses (ENs), some event-based solutions are discussed in [65], [75] and [52]. However, compared to event-based vision and auditory sensors, event-based olfactory sensors are less investigated, resulting in a lack of benchmarks for their implementation and performance evaluation [105].

Event-based interfaces for sensory applications

In case an event-based sensor is not available for certain sensory applications, event-based encoding can be applied to translate non-event-based signals into event-based representations, as discussed in Section 2.1.1.5 - **Spike encoding and decoding in SNNs**. With a proper conversion, non-event-based signals can be interfaced with neuromorphic systems.

Corradi and Indiveri reported on an event-based recording system in [26]. Integrated as part of ROLLS [84], this system serves as an interface that converts non-event-based signals from sensors into event-based signals before feeding them to the neuromorphic cores. Some practical applications have been proposed using this delta-modulator ADC algorithm. One of the applications discriminates electromyography (EMG) signals, a commonly used biomedical signal, for example, in hand-gesture recognition, with an SNN [29]. In this application, the full system can perform EMG recognition with low power consumption (0.05mW) [29]. The working principle of delta-modulation is equivalent to the one shown in Figure 2.10b [26].

2.2.2. sensor-fusion

sensor-fusion is a sensing paradigm that combines sensors measuring a physical phenomenon or process in different ways to obtain complementary information, thereby improving performance on specific tasks.

sensor-fusion has been applied in various domains, including autonomous vehicle applications [36], robotic applications [3], emotion recognition [111], medical diagnosis [50], etc.

For event-based sensors, it is even easier to perform sensor-fusion as all readouts from the sensors use an AER interface. This allows for a straightforward fusion by an SNN without the need for any transformation in the raw data.

2.2.3. Using SNNs in Gesture Recognition Applications

A typical machine learning application for the edge is gesture recognition. To enhance the recognition performance, the sensor-fusion can be used.

Ceolini et al. proposed a fully neuromorphic sensor-fusion scheme for hand-gesture recognition [18]. This system integrates vision data and eletromyography signals from muscles to realize an accurate and robust hand-gesture recognition. An SNN is designed and deployed on neuromorphic chips to process the data from the sensors. The neuromorphic system is capable of attaining a gesture-recognition accuracy comparable to that of a conventional machine learning model running on a GPU system, but is faster and more energy efficient.

This system collects vision data and electromyography data of gestures with an event-based camera (DVS) and an EMG armband sensor called Myo. The DVS sensor has been introduced in 2.2.1. As for the Myo armband, it is a wearable device embedded with eight equally spaced noninvasive electrodes, each collecting signals from the forearm muscles at a 200Hz frequency [18], which are converted to spikes. To verify this system, a dataset involving samples of five gestures is acquired, which is described in [17]. The setup of the sensors and the five gesture categories are shown in Figure 2.12.

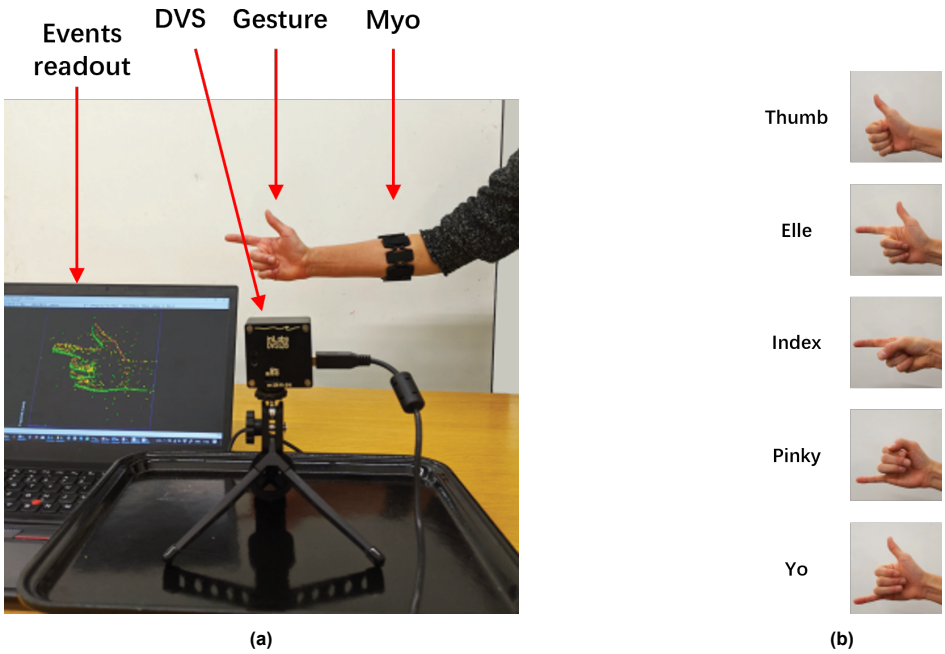

Figure 2.12: Sensor setup and involved gestures. (a) The sensor setup in the hand-gesture recognition application. A DVS is used to collect vision data and a Myo band is used to collect skin electromyography data. Something worth noting is that the output of Myo is not event-based, the output of Myo is converted to events with the delta-modulation approach, as discussed in 2.1.1.5 and 2.2.1. (b) Five gestures involved in the dataset. Figures are adapted from [18].

In order to benchmark our configurable hardware generator, which is proposed in 3.2, we will reproduce this application with the dataset provided in [17].

3

Generation of Configurable Digital Neuromorphic Hardware

In this chapter, the method used to generate a configurable neuromorphic digital hardware and the insight of the proposed hardware will be provided.

3.1. High-Level Hardware Description Languages (HDLs)

Hardware description languages (HDLs) are languages that model hardware logic. These languages are human-readable and support the development, verification, synthesis, and testing of digital hardware design [56, 55]. Among these HDLs, Verilog and VHDL have been used in the industry for decades. Both Verilog and VHDL are standardized by the IEEE [56, 55] and are well supported by mainstream hardware development tools. However, as hardware becomes more complex, more efficient design tools are required. As part of the design toolchain, HDLs could be more scalable by improving their reusability. Traditional HDLs, such as Verilog and VHDL, were initially developed for simulation purposes and are generally not flexible for large and complex circuits (e.g., configurable multi-core neuromorphic hardware) by making it hard to reuse and customize components, leading to low designer productivity [8, 5]. Therefore, in this project, it is necessary to introduce a higher-level HDL that can better describe configurable hardware.

By applying modern software programming language features, such as object-oriented and functional programming, polymorphism, and abstract data types, the reuse problem of HDLs starts to be addressed as the abstraction level increases [104, 107, 88]. At this moment, there are promising HDLs with higher abstraction levels, the most famous one being Chisel (Chisel3 in particular) [8]. Other outstanding high-level HDLs include SpinalHDL [98], Cλash [5] and MyHDL [107].

To verify the capability of the HDLs mentioned above in building configurable neuromorphic hardware, a reconfigurable integrate-and-fire neuron written in Verilog is re-implemented using these languages, and the low-level HDL (i.e. Verilog) files generated by these languages are verified with a testbench. Results show that all these languages can generate a basic building block of a complex neuromorphic circuit. In order to choose from these HDLs, we analyze and compare these languages in five metrics:

- modularity,
- high-level circuit modeling flexibility,
- readability of the generated lower-level HDL,
- quality of the documentation,
- long-term support.

With these criteria, it is possible to choose an optimal high-level HDL for the development of a neuromorphic hardware generator in this project.

In the following subsections, we will first give a brief overview of the selected high-level HDLs. These high-level HDLs will then be discussed in detail on the basis of the five metrics mentioned above. Finally, the results will be given in the form of a comparison table, clearly showing the reason for choosing SpinalHDL in this project.

3.1.1. Overview of High-Level HDLs

3.1.1.1. Chisel

Chisel, which stands for the Constructing Hardware in a Scala Embedded Language, can describe circuits at the register-transfer level, where each operation processes the contents of one or more source registers and assigns the result to a target register [23, 95]. Base on the Scala programming language, it incorporates features such as object orientation, functional programming, parameterized types, and type inference, which are often found in modern programming languages [8]. More specifically, Chisel is a library of class definitions, predefined objects, and usage conventions that are necessary to express digital circuits within Scala [93]. Chisel can generate fast cycle-accurate C++ simulators for a design, or generate low-level Verilog or VHDL for synthesis [8]. Chisel works with FIRRTL (Flexible Intermediate Representation for RTL), which transforms target-independent RTL into technology-specific RTL and performs optimizations of Chisel-generated circuits [23, 58]. Figure 3.1 shows the workflow from Chisel to Verilog. Chisel source code is first compiled and results in **.class** files that run on the Java Virtual Machine (JVM) with the Scala build tool (SBT) and the Scala compiler (Scalac). Then, the JVM runs the **.class** files and generates circuits represented with intermediate representations (e.g. FIRRTL). In the final step, these intermediate representations (IRs) are compiled to generate the required Verilog files.

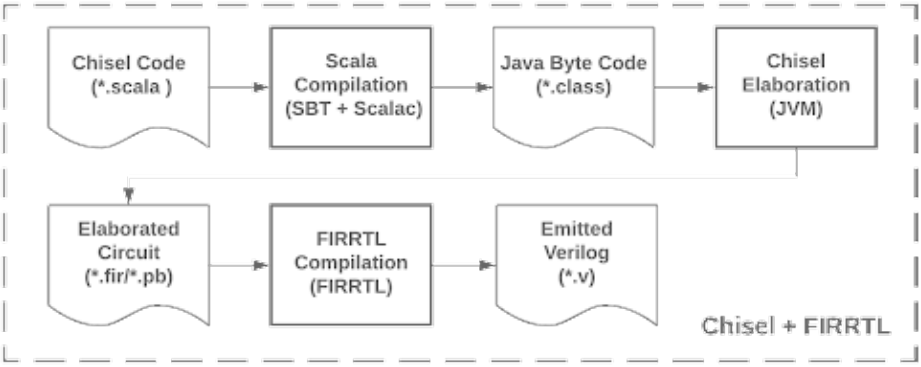

Figure 3.1: Workflow from Chisel to Verilog [63].

Chisel is one of the most successful open source high-level HDLs, as it owns a large community and has already been used to develop company projects, such as Edge TPU, an AI inference accelerator developed by Google [22].

3.1.1.2. SpinalHDL

SpinalHDL is another Scala-based language [98]. Therefore, SpinalHDL provides features similar to Chisel, such as object-oriented and functional programming. Parameterization and hardware code generation are also well supported by Scala's features [8]. Compared to Chisel, SpinalHDL provides better clock domain support and modeling functions, which will be discussed in the following subsections. Figure 3.2 shows how SpinalHDL generates Verilog or VHDL files. As a Scala-based HDL, SpinalHDL also works with SBT and JVM, which is similar to Chisel. Unlike Chisel, SpinalHDL generates Verilog or VHDL from Scala source files directly without generating intermediate representations (i.e. FIRRTL), which is faster than Chisel when generating Verilog or VHDL files. It is easier to access the netlist when using SpinalHDL, as the netlist can only be accessed through FIRRTL in a Chisel setup.

3.1.1.3. Cλash

Computer Architecture for Embedded System (CAES) Language for Synchronous Hardware (Cλash, read as Clash) is a functional hardware description language originally developed by the University of Twente [5]. As a library of Haskell, the development can be done with the Haskell compiler. Based on a purely functional programming language, Cλash models combinational logic with functions and sequential logic with stream data structures and recursive equations [104]. The basic working principles include: 1) every function is translated to a component, 2) every function argument is translated to an input port, 3) the result value of a function is translated to an output port, and 4) function applications are translated to component instantiations [5]. Since Haskell supports parametric polymorphism, functions can be used

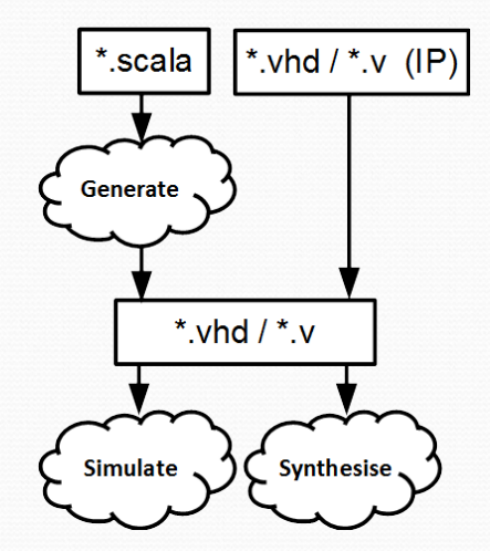

Figure 3.2: Workflow from SpinalHDL to Verilog [97].

without specifying actual types. This feature, which improves the abstraction level, will be discussed in subsection 3.1.2 [5].

Figure 3.3 shows the workflow of the Cλash compiler. In this workflow, the original Haskell description is first translated into core, a small typed functional language, with the Glasgow Haskell Compiler (GHC). Then, the core description is transformed into a normal form which corresponds directly to hardware [5]. Then, back-ends will generate VHDL or Verilog on demand.

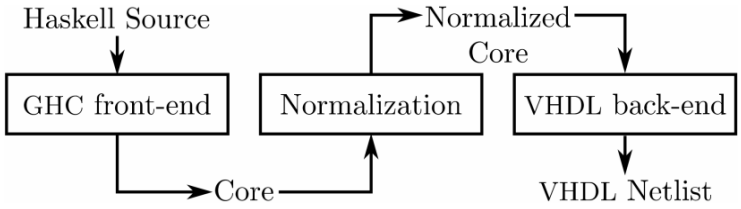

Figure 3.3: Workflow of the Cλash compiler [5].

3.1.1.4. MyHDL

MyHDL is a Python-based hardware description language. The idea of MyHDL is to model hardware modules with Python generators and decorators [107]. MyHDL generators are similar to always blocks in Verilog and processes in VHDL [76]. Figure 3.4 shows the scheme for generating Verilog and VHDL with MyHDL. A Python source file is first elaborated without convertibility limitations to obtain a hierarchy of Python generators. In the second stage, the code generator will analyze this hierarchy using the Python introspection application programming interface (API) to generate VHDL or Verilog [107].

3.1.2. Evaluation of Selected High-Level HDLs

First of all, an integrate-and-fire neuron written in Verilog is re-implemented using all four high-level HDLs. This standard neuron model supports both positive and negative overflow protection and has a configurable resolution. A testbench shows that all the generated Verilog files are functionally correct and equivalent. Although all high-level HDLs are capable of implementing a simple neuron, it is not enough to conclude that these HDLs are capable of implementing a large-scale and complex neuromorphic circuit. Therefore, this section will evaluate and compare the high-level HDLs along the following metrics:

- modularity,
- high-level circuit modeling flexibility,
- readability of the generated lower-level HDL,

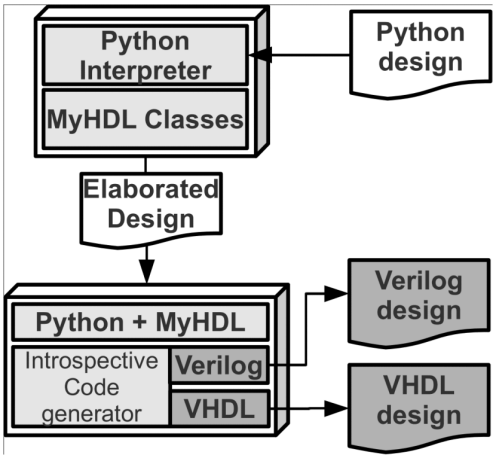

Figure 3.4: Workflow from MyHDL to Verilog and VHDL [107].

- quality of the documentation,
- long-term support.

The final results are shown in Table 3.1. Details of this table will be discussed below.

Table 3.1: Comparison of Selected HDLs

Language		Chisel	SpinalHDL	Clash	MyHDL
Modularity	Parameterization Features	+[1]	+	+	∼
	Programming Paradigm	OOP and FP	OOP and FP	Pure FP	OOP and FP
	Overall	+	+	∼	∼
High-level Circuit Modeling Flexibility	Circuit Structuring	+	+	+	∼
	Clock Domain Configuration	∼	+	+	-
	Signed Representation	+	+	+	-
	Other Modeling Function	∼	+	∼	-
	Overall	∼	+	∼	-
Readability of Generated Verilog	Internal Signals	∼	∼	-	+
	Code Structure	∼	+	-	∼
	Annotations	-	∼	∼	∼
	Overall	∼	+	-	+
Quatlity of Documentation	Manual/ Guide	✓	✓	✓	✓
	Tutorial	✓	✗	✓	✓
	Cheat Sheet	✓	✓	✓	✓
	Overall	✓	✓	✗	✗
Long-term Support	Year of Creation	2010	2014	2009	2009
	Major Releases in the Last 2-year	22	18	9	0
	Stars on GitHub	3.2k+	1.4k+	1.3k+	900+
	Community Events	126	86	61	43
	Overall	+	∼	∼	-
Overall		∼	+	∼	-

[1] + = Good, ∼ = Average, - = Bad. These notes also apply to other tables in this thesis.

3.1.2.1. Modularity

Modularity is the key factor that we will emphasize in this section. Here, modularity indicates the parameterizable features of a language. A language that provides better modularity features will make the codes and components more reusable. As all four high-level HDLs are based on existing modern programming languages, the properties provided by the base language should be carefully considered on top of the HDL libraries or extensions.

The programming paradigm of each high-level HDL will be introduced and discussed first, followed by a discussion on the parameterization features of each HDL. The comparison of modularity is included in Table 3.1.

Programming paradigm

The programming paradigm is a critical factor that affects the parameterization of an HDL. The two programming paradigms used by the selected high-level HDLs are object-oriented programming (OOP) and functional programming (FP), which are discussed hereafter.

The key features of OOP are objects, classes, and inheritance. An object contains a collection of operations, which are called methods, and a state that remembers the effects of these operations. A class can define how objects can be created [74]. In the field of HDLs, hardware can be modeled as objects. As an object has a method interface instead of an RTL-style hardware interface, object-oriented programming makes it easier to manage global variables by encapsulating them into an object [28, 74]. For example, in SpinalHDL, the *resize* method can be applied to a number to extend or truncate it to the required bit width, which can be called with `x.resize(y)`. Here, `x` is a bit vector and *y* is an integer. `x.resize(y)` will return a resized copy of `x`, which is *y*-bit wide [98]. An example in MyHDL is that a register signal and its next state can be represented by `reg_name` and `reg_name.next`. These features can help to manage values and registers in a clearer way. Furthermore, object-oriented programming offers high reusability of components and codes by organizing classes and objects in a systematic way [74].

Functional programming structures a circuit with pure functions [104]. Global variables and commands no longer exist in a functional programming scheme. With these properties, functional programs are naturally parallel [74], avoiding the problem of a value being accessed by different functions or expressions at the same time, which is important when modeling sequential logic. Another feature of functional programming is that since no command is available, recursion is used.

Both of the programming paradigms discussed above provide useful properties for hardware generation. Chisel, SpinalHDL, and MyHDL offer both the OOP and FP paradigms, and Cλash only supports the FP paradigm [8, 88, 5, 107]. In this respect, Cλash is less flexible than the other three HDLs.

Parameterization feature

A powerful feature of these selected programming languages is polymorphism, which allows a function to handle different data types [5, 15].

In Scala, taking Chisel as an example [23], it is allowed to define a function as:

Listing 3.1: Example of a polymorphic function in Scala.

```
1 def Mux[T <: Bits](c: Bool, con: T, alt: T):
2     T = { /*function body*/ }
```

In this definition, the function *Mux* takes arguments *c* of *Bool*, *con* and *alt* of type *T*, and returns type *T*. Defined by the espression in the [*T* <: *Bits*], type *T* can be any subclass of *Bits*, including *SInt* (signed integer) and *UInt* (unsigned integer) [23, 90]. In this way, the type *T* of arguments becomes a parameter of this function so the polymorphic function can support different types, including bit vectors with different widths, with the same interface [93], and returns a result corresponding to the given *T*. Therefore, these polymorphic functions are more powerful than functions simply parameterized with bit widths. SpinalHDL supports the same definition style [97].

A similar definition style is also supported by Cλash and MyHDL, which abstract a low-level type to a higher-level type. For example, in Cλash, the following function is supported [5]:

Listing 3.2: A polymorphic function definition in Haskell.

```
1 first :: (a, b) -> a
```

In this *first* function, the notation ::, which can be read ”has type” [53], is followed by the declaration of the types, showing that this function maps inputs of type *a* and type *b* to an output of type *a*, where *a* and *b* can be any type supported by the Cλash compiler. The type is parameterized in *a* and *b*. These definitions with parameterized types greatly improve the reusability of the function. Constructing circuits with this kind of function will therefore improve modularity.

MyHDL is different, as Python is a dynamically typed language. Therefore, polymorphism in MyHDL is not constrained by static check, and error-checking is deferred to the latest possible time for maximal flexibility [1]. For example, in MyHDL, the signal types of a circuit module are not defined or checked until the circuit module is instantiated, which is shown in the following code:

Listing 3.3: Example of the polymorphism of Python.

```
1 @block
2 def inc(count, enable, clock, reset):
3     """Circuit description of inc block"""
4
5 def convert_inc(hdl):
```

```
6     """Convert inc block to Verilog or VHDL."""
7     count  = Signal(modbv(0)[8:])
8     enable = Signal(bool(0))
9     clock  = Signal(bool(0))
10    reset = ResetSignal(0, isasync=True)
11    """Signal type definition for block inc"""
12    inc_1 = inc(count, enable, clock, reset)
13    inc_1.convert(hdl=hdl)
```

Here, all signal types are defined (lines 7 to 10) when converting this circuit block to Verilog or VHDL, which offers great flexibility. However, these dynamic features may cost more effort when debugging.

A comparison of the modularity of these languages can be made based on the discussion above. As Cλash only supports FP, it is not as competitive as the other three high-level HDLs in the *programming paradigm* category. The parameterization features of these languages are similar, but statically typed languages such as Haskell-based and Scala-based languages are preferred as they make debugging easier.

3.1.2.2. High-level circuit modeling flexibility

In this subsection, the problem we are focusing on is building a complex circuit with packed parameterized blocks. Even though a building block or module can be well parameterized, building a complex circuit with basic blocks is not straightforward. Therefore, high-level HDLs should provide some structuring tools to simplify this process.

To evaluate the support for building a complex circuit, circuit structuring will be first discussed. Then, the support for clock domains, signed representation, and modeling for other circuit components in a complex circuit, including buses and communication interfaces, will be discussed.

Circuit structuring

Ideally, a high-level HDL should easily generate a structured circuit given basic blocks, for example, generating an adder chain with basic adders.

Scala-based HDLs provide a useful method called *reduce*. Given a list of elements, the *reduce* method can take a function (or an anonymous function) and apply that function to successive elements in the same list [90]. This function applied to two neighboring elements can be any function that takes two inputs and gives a single output, such as "add". An example with an FIR filter implementation is provided on the Chisel website [23]. Figure 3.5 shows the structure of an FIR filter, where

$$y[i] = \sum_j x[j] \cdot b[i-j] \ . \tag{3.1}$$

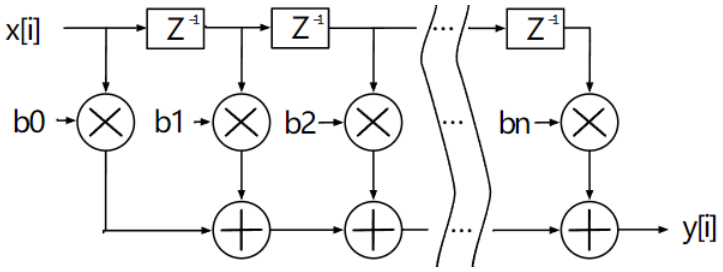

Figure 3.5: Structure of an FIR Filter [23].

In Chisel, the *reduce*$(\_ + \_)$ method is equivalent to the operation $\sum$ and an FIR can be modeled as [23]:

Listing 3.4: Code of an FIR in Chisel3 [23].

```
1  // Do the multiplications
2  val products = VecInit.tabulate(bs.length)
3                 (i => zs(i) * bs(i))
4
```

```
5   // Sum up the products
6   io.out := products.reduce(_ + _)
```

Here `products` is a list of the internal products of an FIR, and `products.`*reduce*`(_+_)` forms an adder chain that sums up these internal products. The method *tabulate* generates a list of the same length as `bs`. Each element in this list is calculated by `zs(i)*bs(i)`. The SpinalHDL impelementation is similar.

An example of generating an FIR filter with Cλash is given in [5]. The equivalent FIR can be implemented with Cλash using *zipWith* and *fold*. In Cλash, *zipWith* provides an even simpler way to get multiple results, that is, *zipWith*(*) [1; 2][3; 4] becomes [1*3; 2*4]. This method basically does the same work as *tabulate* in Scala, which maps the elements in two lists with a given function and generates a new list. The method *foldl1* reduces a vector using a chain structure from left to right. For example, folding a list *xs* with function *f* (i.e. *foldl1 f xs*) results in the circuit in Figure 3.6, where *x0*, $\cdots$ *xn* are the elements of *xs*. To generate chain structures in other forms, other "fold" methods can be used. For example, the method *fold* generates a tree-like structure.

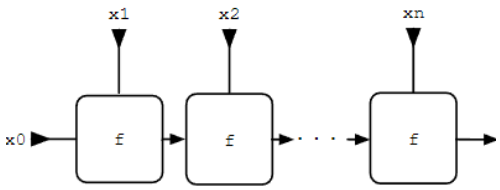

Figure 3.6: Circuit layout generated by *foldl1 f xs* in Cλash [25].

The same FIR filter as in Figure 3.5 can be implemented by Cλash as [5]:

Listing 3.5: Code of an FIR in Cλash

```
1   products = foldl1 (+) (zipWith (*) as bs)
```

Python also provides a similar *reduce* method to perform reduction calculations [81]. However, the *reduce* method is not fully supported by MyHDL: it can only be used in simulation and cannot generate a corresponding circuit structure in Verilog. This problem is explained in [51]. Therefore, generating an FIR or complex structured circuits with MyHDL takes more effort than with the other three high-level HDLs.

To conclude, with methods such as *reduce* in Chisel or SpinalHDL [90], or *foldl1* in Cλash [25], a chain structure can be generated. A tree structure can also be modeled by *reduceTree* in Chisel [93] or *fold* in Cλash [25]. To map two lists, *tabulate* in Chisel or SpinalHDL [90], or *zipWith* in Cλash [25] can be applied. With these methods, it is easier to automate complex circuit connections. It is thus important in the evaluation of complex circuit modeling. As these mentioned methods are not fully supported by MyHDL, MyHDL is less competitive in this category.

Clock domain configuration

Both SpinalHDL and Cλash support a direct definition of clock domains. These clock domain definitions make it more convenient to use the same module in different clock domains.

In SpinalHDL, a configuration of a clock domain can be defined as [98]:

Listing 3.6: Example of the configurations of an clock domain in SpinalHDL [98].

```
1 val defaultCC = ClockDomainConfig(
2   clockEdge        = RISING,
3   resetKind        = ASYNC,
4   resetActiveLevel = HIGH
5 )
```

In this configuration, the clock edge, reset kind, and reset level can be defined. SpinalHDL also offers clock domain crossing support, as it provides a simple function to generate a clock domain crossing buffer [98]. These features are important when designing a complex system with multiple clock domains.

Cλash also supports a similar way to construct a clock domain. In fact, all signals in Cλash should be stated under a clock domain [25].

These clock domain constructs are not supported in Chisel. Furthermore, in Chisel, the reset and clock are implicit by default. An asynchronous reset can only be obtained with an explicit definition that includes clock and reset signals, such as [23]:

Listing 3.7: Explicit definition of clock and reset in Chisel3 [23].

```
val resetAgnosticReg = withClockAndReset(clk, rst)(RegInit(0.U(8.W)))
```

where the explicit reset signal can be set as asynchronous.

As for MyHDL, it does not provide any dedicated feature for clock definition. The way it defines clocks and resets is like the way Verilog does, which is not as straightforward and high-leveled as what SpinalHDL does [28].

Signed representation

Signed arithmetic is supported by Chisel, SpinalHDL and Cλash [23, 98, 25]. Many methods or functions are also provided to perform both signed and unsigned arithmetic. An outstanding feature of SpinalHDL is that fixed-point and floating-point representations are supported (although still under test) [98].

MyHDL represents signed numbers in a different way. The key point is that the authors of MyHDL wanted to perform arithmetic like with normal integers, which however allows for non-standard usages. An example of signed signal definition in MyHDL is shown as follows [28]:

Listing 3.8: Definition of a signed signal in MyHDL [28].

```
a = intbv(6, min=-13, max=7)
```

In this definition, *a* is a 5-bit signed integer bit vector (*intbv*) with the value of 6 and the allowed range of *a* is -13 to 7. Though such an unconventional definition style is allowed, it is misleading and may cause difficulty for hardware engineers if the arithmetic part needs debugging.

Other modeling functions for circuit components

All these high-level HDLs provide methods for modeling memory and finite-state machines (FSMs). However, SpinalHDL is more powerful as it can also model other circuit components, such as

Buses

SpinalHDL provides many templates for different bus protocols, including the APB3 and the AXI4. With these templates, a bus protocol can be easily deployed in a complex design [98].

Communication interfaces

SpinalHDL also provides templates for communication interfaces such as UART and USB [98].

3.1.2.3. Readability of the generated Verilog

The Verilog files generated by the four languages for the same integrate-and-fire neuron model are compared in terms of readability. To compare the readability more objectively, the following factors are taken into consideration:

- internal signal names,
- code structure,
- annotations.

Table 3.2 shows the detailed readability comparison of Verilog generated by different high-level HDLs. For reference, the full source files can be accessed on the GitHub repository[1].

Internal signal names

Take signal `sign_ext_syn_weight` in the original Verilog file as an example, which corresponds to a gated sign extension operation. The following part shows the different assignment expressions of the exact same signal in different codes:

[1] `https://GitHub.com/BlackJackJam/HDL_For_Neuromorphic`

Table 3.2: Readability of each HDL

Languages	Chisel	SpinalHDL	Cλash	MyHDL
Internal signals	$\sim$	$\sim$	-	+
Code structure	$\sim$	+	-	$\sim$
Annotations	-	$\sim$	$\sim$	$\sim$
Overall	$\sim$	+	-	+

Listing 3.9: Original Verilog code.

```
assign sign_ext_syn_weight = spike_in ?
           $signed({{(N-S){syn_weight[S-1]}}, syn_weight})
           : $signed('d0);
```

Listing 3.10: Equivalent Verilog code generated by Chisel

```
wire [5:0] sign_ext_syn_weight = io_spike_in ?
           $signed(_sign_ext_syn_weight_T_4)
           : $signed(6'sh0);
```

Listing 3.11: Equivalent Verilog code generated by SpinalHDL

```
assign sign_ext_syn_weight = (io_spike_in ?
                 _zz_sign_ext_syn_weight :
                 _zz_sign_ext_syn_weight_1);
```

Listing 3.12: Equivalent Verilog code generated by Cλash

```
assign c$case_scrut =
               ($\sim$((((c$bv[6-1]))) == 1'b1))
               & (((((c$bv_0[6-1]))) == 1'b1)
               & ((((c$bv_1[4-1]))) == 1'b1));
```

Listing 3.13: Equivalent Verilog code generated by MyHDL

```
always @(syn_weight, spike_in) begin:
  if (spike_in) begin
      sign_ext_syn_weight = syn_weight;
  end
  else begin
      sign_ext_syn_weight = 0;
  end
end
```

From the different assignment expressions, it is obvious that Cλash fails to reproduce the original signal name when generating Verilog, which causes inconvenience when debugging. Chisel and SpinalHDL generate intermediate signals with names similar to the final one, which is good for debugging. Because the coding style of MyHDL is similar to Verilog, the signal names typed in MyHDL are just copied to the generated Verilog file.

Code structure

The code structure generated by SpinalHDL is the clearest.

Chisel and Cλash generate code with multiple *@always* blocks for sequential logic instead of grouping them. MyHDL generates codes with *@always* blocks that can be misleading, as they use a specific sensitivity list of combinational signals (also visible in the example for MyHDL above).

There are numerous duplicated signals in the file generated by Cλash.

Annotations
As opposed to SpinalHDL, Cλash and MyHDL, the Verilog file generated by Chisel includes numerous pre-compile annotations, i.e., more than 50% of the generated file. These annotations make the generated code hard to read. However, some of them can help trace back to the original line of code and are useful when debugging.

3.1.2.4. Quality of documentation

Given that the selected high-level HDLs are not yet the mainstream choices for hardware generation, the quality of their documentation is important. Good documentation will help developers to transfer from Verilog or VHDL to a new high-level HDL. However, the evaluation of documentation quality could be subjective because it relies on the user's personal preference. Hence, we will try to describe the quality of documentation in a more objective way, such as the types of documents or references that we can access.

The results are shown in Table 3.3.

Table 3.3: Quality of Documentation of each HDL

Language	Chisel	SpinalHDL	Cλash	MyHDL
Manual / Guide	✓	✓	✓	✓
Tutorial	✓	✗	✓	✓
Project Examples	✓	✓	✓[1]	✓
Cheat Sheet	✓	✓	✗	✗
Overall	+	∼	∼	∼

[1] Third-party examples with links provided on the official site.

Basically, all of these languages provide well-structured manuals and come with some examples to help users understand how they work.

The highlight of Chisel is that it comes with published open-access guidebooks in four different languages (English, Chinese, Japanese, and Vietnamese) [93] and an official cheat sheet for quick reference [64]. However, it is not yet mature, as some facts are not stated in the main manual. An example is the bitwise reduction operation. Although the bitwise reduction operation is stated to support both signed and unsigned signals in Chisel, such an operation can actually only support unsigned signals: a signed signal has to be casted as an unsigned signal beforehand [23, 64].

SpinalHDL also provides a concise cheat sheet, and the examples are also of high quality [98].

Cλash provides detailed documentation, references, and tutorials. However, reading these documents will be challenging for developers unfamiliar with Haskell. Simple example projects can be found in the repository [48]. Links to more complicated third-party examples are also given on Cλash's website [24]. Printed or electronic books on Cλash are also available but come with a charge [30].

Although the MyHDL manual is short, it is clear and of high quality. Six examples are provided on the official site, from a basic "Hello World" to complex examples such as a Cordic-based sine computer [34]. All these examples include detailed comments and instructions, which are friendly to new users.

Compared to SpinalHDL, Chisel provides tutorials including a printed book that gives instructions for beginners step by step. Compared to Cλash and MyHDL, Chisel provides a cheat sheet for quick reference. Therefore, Chisel offers all types of documentation for users of different levels, and it should be the winner in terms of the quaity of documentation.

3.1.2.5. Long-term support

As part of the hardware development environment, it is important that a high-level HDL benefits from long-term support. Update frequency is a key factor, as a fast-updating language is more likely to follow the latest features and trends, have an up-to-date development environment, and benefit from quick bug fixes.

Unlike Verilog and VHDL, which are regulated by IEEE standards [56, 55], the only guarantee of long-term support of open-source HDLs is the community. An active community usually means that the language can be better monitored, maintained, and improved. Therefore, an active community is a plus when evaluating the long-term support of a high-level HDL. A high number of active users is a good indicator, together with community events, if any.

To conclude, to evaluate the long-term support of an HDL, these factors are investigated:

- update frequency,
- number of active user,
- community events.

Table 3.4 shows the factors and results for the evaluation of the long-term support of each high-level HDL.

Table 3.4: Long-term support of each HDL

Language	Chisel	SpinalHDL	Cλash	MyHDL
Year of creation[1]	2010	2014	2009	2009
Major releases in the last 2-year period[2]	22	18	9	0
Stars on GitHub[3]	3.2k+	1.4k+	1.3k+	900+
Contributors on GitHub[4]	126	86	61	43
Community Events	✓	✗	✗	✗
Overall	+	∼	∼	-

[1] Data sources: [7], [98], [6] and [85]
[2] From 25 October 2021 to 25 October 2023
[3, 4] Until October 2023

Update frequency
Table 3.4 shows the number of major releases on the GitHub repositories of each high-level HDL [21, 89, 48, 103] in the last 2-year period. It is clear that the development of MyHDL appears to be suspended, and it is not likely to provide major updates as frequently as the other three languages in the future. However, the issue tracker and commits history on GitHub prove that MyHDL is still maintained and minor amendments are still uploaded [103].

Number of active users
The numbers shown in Table 3.4 indicate that Chisel is the most popular and fast-growing high-level HDL among the selected ones. SpinalHDL and Cλash communities are also active. Created in 2014, SpinalHDL already got 1.4k stars on GitHub, which shows that the SpinalHDL community is also growing fast. The community of MyHDL is far less active compared to the others.

Community events
Community events are a great plus because they gather both developers and users. The Chisel community holds an annual event called the Chisel Community Conference (CCC) and its recordings are available on the Internet [23]. Unfortunately, there are no such community events for the other three languages.

In conclusion, Chisel attracts the most contributors and followers among these four HDLs. It also maintains a high update frequency. Community events are also held regularly by the Chise community. Therefore, Chisel has the best long-term support among these HDLs. As the latest one created, SpinalHDL has the second largest community and frequency of updates. Therefore, we can say that SpinalHDL is a promising HDL.

3.1.3. Conclusion
As shown in Table 3.1, although SpinalHDL is less competitive than Chisel in terms of quality of documentation and long-term support, it is still the most suitable high-level HDL for our project because it performs the best in terms of high-level circuit modeling flexibility and readability of generated Verilog.

3.2. Proposed Configurable Digital Neuromorphic Hardware Generator
In this section, we will first explain our design choices and discuss the trade-offs we made in our design process. Then, we will show the architecture and workflow of the proposed neuromorphic hardware generator, followed by details in single cores and the multi-core generation process.

3.2.1. Design Choice

In this sub-section, we will first discuss the trade-offs of the designs before presenting the whole workflow and architecture of the generator.

As introduced in Section 2.1.2.2, ODIN is a good example for power- or resource-constrained applications, such as low-power gesture recognition. ODIN demonstrates impressive power-efficiency, with only 12.7 pJ per synaptic operation (SOP), and the chip only takes an area of 0.086mm^2 in 28 nm CMOS [40]. As a simplified version of ODIN, tinyODIN [37] is expected to be more power- and resource-efficient. Hence, we select tinyODIN as the fundamental processor model for our hardware generator, which aims to generate low-resource hardware as required.

3.2.1.1. Trade-off between on-chip configurability and generator flexibility

As discussed in Section 2.1.2.2, some neuromorphic hardware systems offer good on-chip configurability. However, achieving good on-chip configurability requires more resources. For example, supporting different neuron models necessitates the implementation of different neuron update logic. To support both the Izhikevich and LIF models, ODIN includes an Izhikevich update logic along with an LIF update logic [40]. On the other hand, a flexible generator allows easy switching among different hardware architectures, by changing modules or logics within the generated platform.

To keep the energy- and resource-efficiency of tinyODIN in our hardware, we maintain the necessary on-chip configurability provided by tinyODIN. These programmable parameters of tinyODIN are necessary for specifying network connections. Other configurations will be implemented at the generator level. This approach allows us to generate hardware systems that are both energy- and resource-efficiency for specific applications.

3.2.1.2. Trade-off between single-core and multi-core architectures

The time-multiplexing scheme of ODIN and tinyODIN includes only one physical neuron with 256 synapses [40, 37]. This physical neuron is multiplexed into 256 neurons, forming a 256$\times$256 time-multiplexed crossbar. Both the axons and neurons in this crossbar are time-multiplexed, making it area-efficient [100]. However, a large core results in significant resource consumption because the synapse memory grows proportionally to n^2, n being the number of neurons. Therefore, while our generator is capable of generating arbitrary large single cores, using large n is not recommended.

A solution to perform more efficient calculation is a multi-core architecture. In a multi-core structure, neurons from different layers of an MLP can be allocated to different cores. This allows neurons from different layers to operate in parallel, with each core handling a relatively small number of neurons.

Another advantage of multi-core architectures is the ability of having different weight and neuron resolutions with different cores. In this way, we may further reduce the resources for targeted applications, as some applications do not require high-resolution operations. As reported in [18], the DVS data classification is less sensitive to weight resolution, but requires large resources of neurons and synapses, while the EMG data classification relies on a higher weight resolution due to its low input dimensionality. Therefore, it is reasonable to build a multi-core architecture for the sensor-fusion hand-gesture recognition application in this project.

3.2.2. Workflow and architecture of the generator

The workflow for generating an FPGA-based prototype for a Xilinx FPGA is shown in Figure 3.7[2]. The file structure of this generator is shown in Figure 3.8, and the dependencies of the files are also indicated. This workflow allows for simple generation of a multi-core architecture with only a few lines of code in SpinalHDL. The detailed generation process for such a multi-core architecture will be introduced in Section B.2.

The general process is as follows: first, we specify the hardware *config* classes and the core connections in *main.scala*. The *config* classes include different classes for specifying and calculating the hardware parameters, while *main.scala* includes some compiling instructions. Then, using the standard Scala compile tool, *sbt*, we can generate the corresponding Verilog files. In a *.sbt* file, the Scala version, the dependencies (i.e., the included libraries), and the versions of these included libraries (e.g., spinal.lib) are specified. For example, in this project, we use Scala 2.12.16, with essential SpinalHDL libraries such as *spinal.lib*. The version of the SpinalHDL core library, *Spinal.lib*, is 1.8.0.b.

During the compile process, the compiler will check for design errors, including combinational loops, so that the generated Verilog files are free from syntax errors. SpinalHDL also supports user-defined

[2]Extension of this workflow to an ASIC flow is also straightforward.

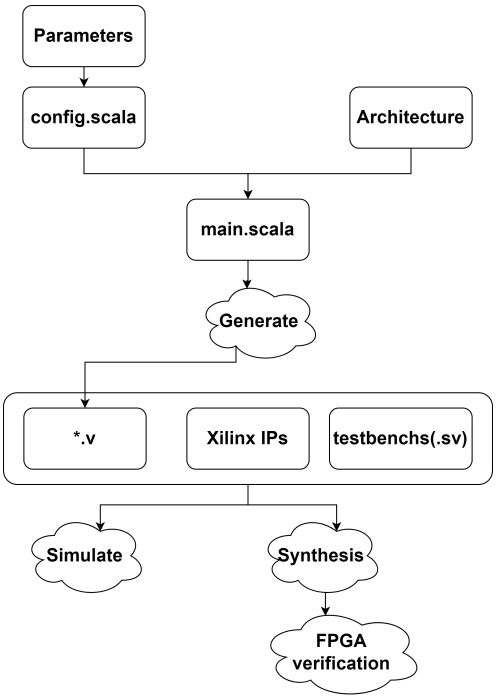

Figure 3.7: The workflow of hardware generation. First, the hardware parameters are specified with the *config* classes. Then the multi-core architecture is defined in *main.scala*. Using specified parameters and architecture, the standard SpinalHDL compile process (as introduced in Section 3.1.1.2) can be run, generating the corresponding Verilog file (*MultiCore_Gen.v* in Figure 3.8). This file, along with necessary Xilinx IPs and testbenches can be used for simulation, synthesis and FPGA verification.

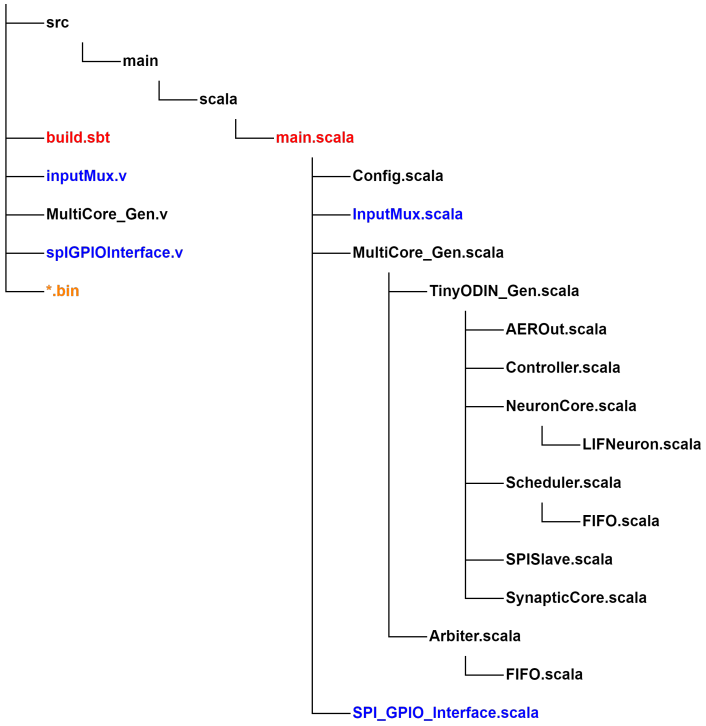

Figure 3.8: File structure of the SpinalHDL project. The structure level starts from *main.scala* and also indicates the dependencies of the files. The files in red are the necessary components for the Scala compile process. The files in blue are source files and the generated Verilog files included in verification process, details of the verification setup can be found in Section 5.1. The files in orange are the files for memory initialization, when a register-based memory model is used for simulation purpose. Among the listed files, *MultiCore_Gen.scala* is the top-level file of the entire multi-core generator, and *TinyODIN_Gen.scala* is the top-level module of a single tinyODIN-based core.

error or constraint checks. In this project, we defined some constraints, such as the relationship among different bitwidths, which will be explained later in Section 3.2.3.1.

3.2.3. Configurable Core Architecture

As discussed previously, we have chosen tinyODIN as the base processor model for our generator. To fit tinyODIN to networks with various complexity, scaling the processor by adjusting the number of neurons is a straightforward method. Therefore, the number of neurons must be reconfigurable. Additionally, to optimize memory resources, the memory array should also be reconfigurable to accommodate for different numbers and resolutions of neurons and synapses. Though different configurations can be made, our core still follows the crossbar architecture. Hence, the core architecture we implemented is a parameterized architecture based on tinyODIN and is presented hereafter.

3.2.3.1. Intra-core parameters

Before we introduce the parameters in detail, we need to first introduce a function, which is

$$A(n) = \lceil \log_2 n \rceil \ . \tag{3.2}$$

This function provides a straightforward expression for the required address width for an array of n elements and will be used throughout the thesis.

Generator parameters

Generator parameters are the parameters that can be configured during the hardware generation process. These parameters cannot be configured once the hardware is implemented with an ASIC or an FPGA (unless recompiled). The configurable parameters are listed in Table 3.5. Internal parameters that are defined as constants or automatically generated from user-specified parameters, are listed in Table 3.6. The inter-core parameters correspond to multi-core structuring and will be introduced in Section 3.2.4. The parameters for first-in-first-out memories (FIFOs) are listed in Table 3.7.

On-chip configurable parameters

The on-chip configurable parameters are listed in Table 3.8. As the configurable parameters provided by tinyODIN are insufficient for our project, we also introduced additional configurable parameters to enable more flexible network configuration within a core, such as *neur_output* and *neur_reschedule*. These parameters are configurable through the SPI configuration (details in Section 3.2.3.3). The parameters are categorized into two groups: global parameters that control the behavior of the core, and neuronal parameters that control each neuron. Details of the neuronal parameters will be introduced in Section 3.2.3.4.

3.2.3.2. Architecture

Crossbar architecture

The parameterized crossbar architecture of a single core based on tinyODIN [37] is shown in Figure 3.9.

From Figure 3.9, we can see that in a parameterized core, the 256×256 crossbar is replaced by an $n \times n$ crossbar. In this way, we can manage the utilization of hardware resources with the parameter n. The input AER becomes A_{AER} bits and the output AER becomes $A_{n_{out}}$ bits, to fit the number of neurons. Details of the AER interfaces will be introduced in Section 3.2.3.3.

Block diagram

The block diagram of a single core and some of the corresponding Scala files are shown in Figure 3.10.

From this diagram, we can see that the neuron and synapse memories as well as the LIF update logic, are parameterized and configurable (see definitions in Tables 3.5 and 3.6).

The function of the blocks are briefly introduced as follows:

- AER input: The event-based input interface of the core.
- AER output: The event-based output interface of the core.
- SPI slave: The SPI interface of the core, which also stores the global configurations in Table 3.8.
- Controller: Controls the pushing and popping of events and accesses to memories.
- Neuron memory: Memory that stores the states and parameters of neurons.
- Synapse memory: Memory that stores synaptic parameters (weights).

Table 3.5: Configurable parameters for a single core (class *CoreConfig* in *Config.scala*).

Category		Type	Prameter	Definition	Note
intra-core	number	Int	n	Number of neurons	
		Int	D	Depth of FIFO	**Default**: $D = 128$
		Int	n_{out}	Number of outputs, cannot exceed the total number of neurons n	$n_{out} \leq n$
	bitwidth	Int	B_n	Bitwidth of neuron	
		Int	B_s	Bitwidth of synapse, cannot exceed the length of a synapse word B_{wordS} or the bitwidth of a neuron B_n	$B_s \leq B_{wordS}$, $B_s < B_n$
		Int	B_l	Bitwidth of leakage, cannot exceed the bitwidth of a neuron B_n	$B_l < B_n$
		Int	B_{byte}	Bitwidth of a byte	**Default**: $B_{byte} = 8$
	control	Int	*leakMode*	Neuron leakage behavior control	**0 (default)**: LIF (leak by subtraction); **1**: I&F (leakage disabled in the LIF module)
		Int	*resetMode*	Neuron reset behavior control	**0 (default)**: reset to zero; **1**: reset by subtraction
		Boolean	*blkBox*	Memory generation control	**true (default)**: blackbox memory interface; **false**: register array
		Boolean	*virturalCore*	Virtual core generation control	**true:** only generate core interfaces (virtual core); **false (default):** generate the whole core
inter-core	number	Int	*idx*	The index of this core	
	bitwidth	Int	B_{SPI}	Bitwidth of SPI, should be able to hold all necessary addresses	$B_{SPI} > max(A_n + A_{byteN}, A_{m'} + A_{byteS})$
	connection	List[Int]	*inputCore*	A list indicating the indices of cores that connect to the input of this core	Empty list indicates that this core is directly connected to the AER interface to the chip
		Boolean	*extrernalAER*	External input source to this core	**true**: this core connects to an external AER input in addition to other outputs of other cores; **false (default)**: this core only connects to outputs of other cores

[1.] The constraints on the parameters are automatically checked by the Scala compiler (as introduced in Section 3.2.2). If these constraints are not met, the compilation process fails with error information and no Verilog file is generated.

Table 3.6: Pre-defined (not exposed to users) parameters for a single core (class *CoreConfig* in *Config.scala*).

Category		Type	Prameter	Definition	Note
intra-core	bitwidth	Int	*lineScale*	Scaling factor of the bitwidth of a word in the neuron memory	**Default**: *lineScale* = 8, which is the bitwidth of a byte
		Int	B_{wordN}	Word size in the neuron memory	$B_{wordN} = \lceil (2B_n + B_l + 3)/\text{lineScale} \rceil \times \text{lineScale}$
		Int	B_{wordS}	Word size in the synapse memory	**Default**: $B_{wordS} = 32$
	number	Int	m	Total number of synapses	$m = n^2$
		Int	n_{ByteN}	Number of bytes in a neuron word	$n_{ByteN} = B_{wordN}/B_{byte}$
		Int	n_{ByteS}	Number of bytes in a synapse word	$n_{ByteS} = B_{wordS}/B_{byte}$
		Int	m_{wordS}	Number of synapses in a synapse word	$m_{wordS} = \lfloor B_{wordS}/B_s \rfloor$
		Int	n_{wordS}	Number of synapse words in a block	$n_{wordS} = \lceil n/m_{wordS} \rceil$
		Int	m'	Number of words in the synapse memory	$m' = n_{wordS} \times n$
		Int	SPIMaxCycle	Number of cycles to complete an SPI transmission	SPIMaxCycle$= 2B_{SPI}$
	address width	Int	A_{byteN}	Address width of a neuron word	$A_{byteN} = A(n_{byteN})$
		Int	A_{byteS}	Address width of a synapse word when accessed in bytes	$A_{byteS} = A(n_{byteS})$
		Int	A_{wordS}	Address width of a synapse word when accessed in synapses	$A_{wordS} = A(m_{wordS})$
		Int	A_n	Address width of the neuron memory	$A_n = A(n)$
		Int	$A_{m'}$	Address width of the synapse memory	$A_{m'} = A(m')$
		Int	$A_{n_{out}}$	Address width of the outputs	$A_{n_{out}} = A(n_{out})$
inter-core	address width	Int	A_{AER}	Input AER address width of this core	$A_{AER} = max(A_n + 2, B_s + 4)$
		Int	A_{AERO}	Output AER address width of this core	$A_{AERO} = A_{n_{out}} + 2$

Table 3.7: Parameters of an FIFO (class *FIFOConfig* in *Config.scala*).

Category	Type	Parameter	Definition	Note
bitwidth	Int	w	Bitwidth of the FIFO	
number	Int	d	Depth of the FIFO	
address width	Int	A_d	Address width of the FIFO	$A_d = A(d)$

Table 3.8: On-chip configurable parameters, adapted from [37].

Category	Name	Width	Description
Global	SPI_GATE_ACTIVITY	1-bit	Gates the network activity and allows the SPI to access the neuron and synapse memories for programming and readback.
	SPI_OPEN_LOOP	1-bit	Prevents spike events generated locally by the neuron array from entering the scheduler, they will thus not be processed by the controller and the scheduler only handles events received from the input AER interface. Locally-generated spike events can still be transmitted via the output AER interface if the SPI_AER_SRC_CTRL_nNEUR configuration register is de-asserted.
	SPI_AER_SRC_CTRLn_NEUR	1-bit	Defines the source of the AER output events when a neuron spikes, either directly from the neuron when the event is generated (0) or from the controller when the event is processed (1). This distinction is of importance, especially if SPI_OPEN_LOOP is asserted.
	SPI_MAX_NEUR	A_n-bit	Defines the maximum neuron index (0 to $n-1$) to be processed, i.e. the crossbar array size. This parameter is useful to avoid processing dummy synaptic operations in case the neuron resources actually being used are small.
Neuronal	neur_disable	1-bit	Disables the neuron.
	neur_output	1-bit	Allows events generated by this neuron to enter the output AER interface.
	neur_reschedule	1-bit	Allows events generated by this neuron to enter the scheduler.
	leak_str	B_l-bit	Defines the leakage strength of a neuron.[1]
	threshold	B_n-bit	Defines the firing threshold: a spike is issued and the neuron is reset when the membrane potential reaches this value.
	core_state	B_n-bit	Contains the current value of the membrane potential. This membrane potential is signed and can be initialized through the SPI configuration.
Synaptic	*w*	B_s-bit	Defines the synaptic weight. This weight is signed.

[1.] By default, the generator generates update logic for the LIF model. For I&F update logic, this leakage strength is not used. For the default LIF update logic, this unsigned value is subtracted from the membrane potential until it reaches zero in case of time reference / leakage events.

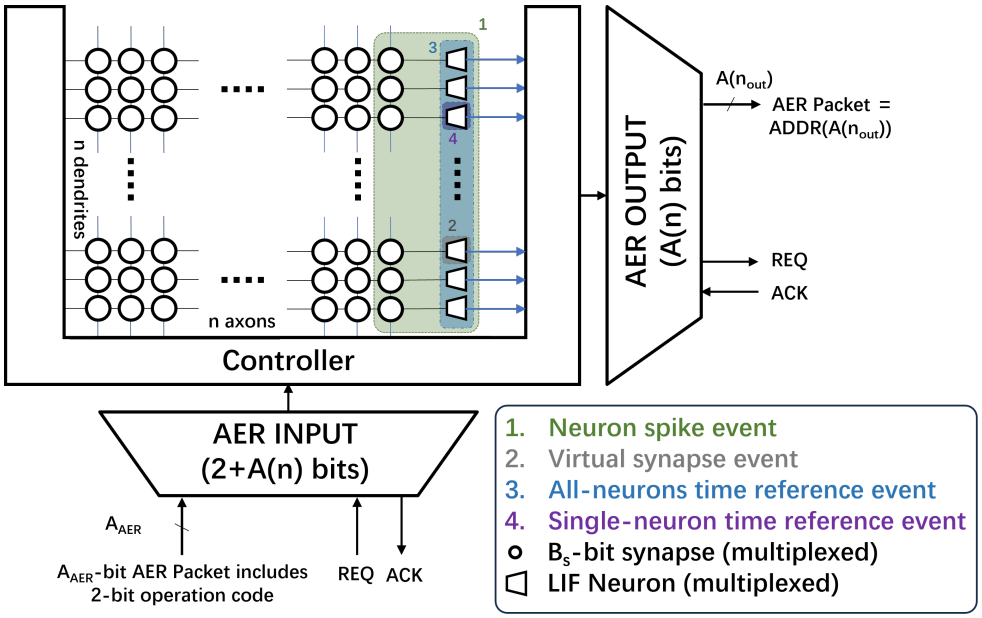

Figure 3.9: Crossbar structure of a parameterized core. The highest 2-bit of AER input is operation code, which indicates the types of events. Adapted from [40] and [37].

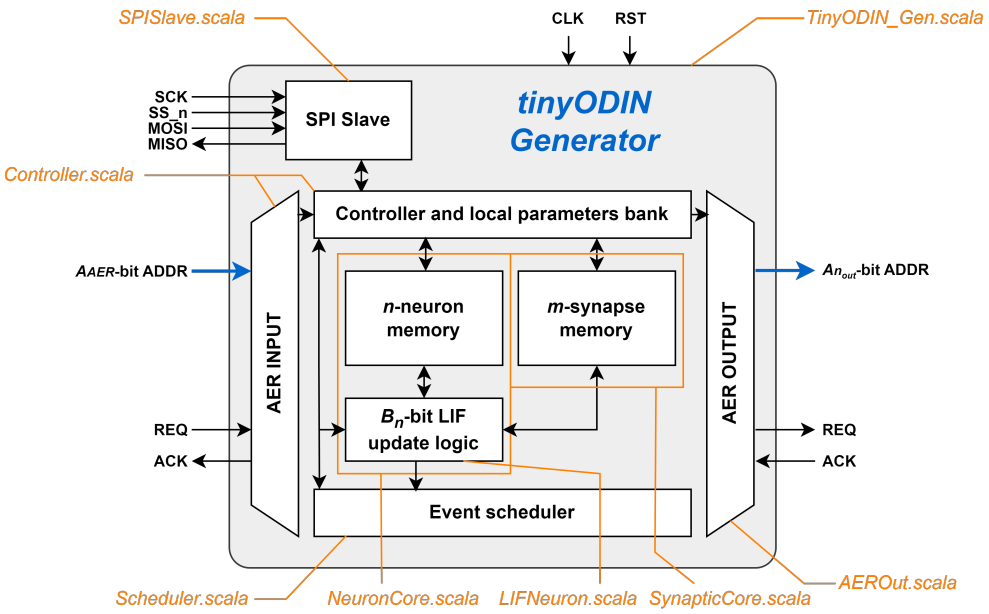

Figure 3.10: Architecture of the parameterized core. Adapted from [40] and [37].

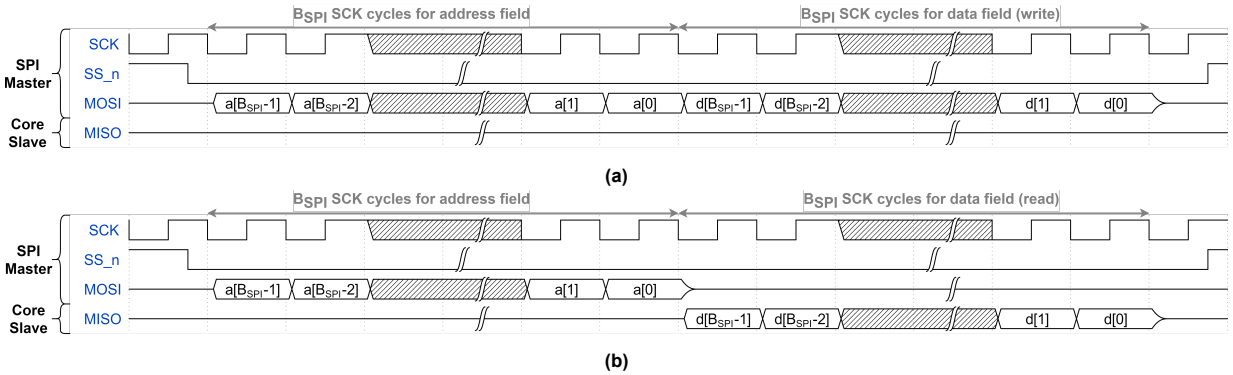

Figure 3.11: SPI timing diagram for a single core. (a) Write operations. (b) Read operations. `SCK`: the SPI clock generated by the SPI master, `SS_n`: an active low chip-select signal for the slave. `MOSI`: master output, slave input. `MISO`: master input, slave output. Figures are adapted from [37].

- LIF update logic: Logic that models an LIF neuron.
- Event scheduler: Events from the AER input or the LIF update logic are scheduled in its FIFO. Events are popped following requests from the controller.

3.2.3.3. Interfaces and operations

The architecture includes two types of interfaces, namely *serial peripheral interface* (SPI) and AER.

SPI

The serial peripheral interface, commonly known as SPI, is a widely used serial communication protocol. In our system, the SPI interface is used to program global configurations and initialize neurons and synapses of a core. Each core works as a slave on the SPI bus and is activated by the chip-select signal `SS_n`. The timing diagrams of the SPI write and read operations are shown in Figures 3.11a and 3.11b, respectively.

During an SPI transmission cycle, the chip-select signal `SS_n` for the destination slave is first de-asserted. At the same time, the master transmits a B_{SPI}-bit address to the bus. The activated slave then registers the address before data associated with this address are sent by the master in a write operation or received from this slave in a read operation. SPI operations are controlled by the address field *a*, the details of different operations are listed in Tables 3.9 and 3.10.

The SPI bus structure for multi-core operations will be introduced in Section 3.2.4.3.

AER

The AER interface of the proposed hardware system is based on the AER protocol introduced in Section 2.1.2.1. Therefore, the timing diagram for an AER communication is the same as shown in Figure 2.8.

The input AER interface comprises a 1-bit request signal `REQ`, a 1-bit acknowledge signal `ACK`, and an input AER packet. The input AER packet includes a 2-bit operation code and an A_n-bit address, together form a A_{AER}-bit AER bus (as shown in Figures 3.9 and 3.10). The output AER packet does not contain operation information and is a pure address of $A_{n_{out}}$ bits. Details of AER operations will be introduced in 3.2.3.4.

3.2.3.4. Neuron and synapse operations

Memory Structure

In this project, a configurable memory structure is implemented to minimize resource utilization. This structure allows for a flexibility in the word length of a neuron word, based on user-specified parameters. The word length of a synapse word is pre-defined in the generator, but it can accommodate for a variable number of synaptic parameters per word. As a result, the total number of synapse words can also vary.

The memory structure of the neuron memory is shown in Figure 3.12 and the content of each neuron word is shown in Figure 3.13.

As shown in Figure 3.12 and Table 3.6, a neuron word has a length of B_{wordN}, which contains a threshold of B_n bits, a current neuron state of B_n bits, a leakage strength of B_l bits and 3 1-bit on-chip

Table 3.9: Contents of SPI address field *a*, adapted from [37].

R(*a*<B_{SPI}-1>)	W(*a*<B_{SPI}-2>)	cmd(*a*<B_{SPI}-3:B_{SPI}-4>)	addr<B_{SPI}-5:0>	Description
N/A	N/A	00	{*conf_addr*(*a*<B_{SPI}-5:0>)}	Configuration register write at address *conf_addr*.
1	0	01	{n/a,*byte_addr*<A_{ByteN}-1:0>,*word_addr*<A_n:0>}	Read to the neuron memory. Byte *byte_addr* from word *word_addr* is retrieved
0	1	01	{n/a,*byte_addr*<A_{ByteN}-1:0>,*word_addr*<A_n:0>}	Write to the neuron memory. Byte *byte_addr* from word *word_addr* is written.
1	0	10	{n/a,*byte_addr*<A_{ByteS}-1:0>,*word_addr*<$A_{m'}$-1:0>}	Read to the synapse memory. Byte *byte_addr* from word *word_addr* is retrieved
0	1	10	{n/a,*byte_addr*<A_{ByteS}-1:0>,*word_addr*<$A_{m'}$-1:0>}	Write to the synapse memory. Byte *byte_addr* from word *word_addr* is written.

Table 3.10: Contents of SPI data field *d*, adapted from [37].

n/a(*d*<B_{SPI}-1:2B_{byte}>)	mask<B_{byte}-1:0>(*d*<2B_{byte}-1:B_{byte}>)	byte<B_{byte}-1:0>(*d*<B_{byte}-1:0>)	Source	Description
N/A	*w_mask*<B_{byte}-1:0>	*w_byte*<B_{byte}-1:0>	SPI Master	SPI Write - Data byte *w_byte*<B_{byte}-1:0> masked by *w_mask*<B_{byte}-1:0> (1=masked, 0=non-masked) is written to the target memory location.
N/A	N/A	*r_byte*<B_{byte}-1:0>	SPI Slave	SPI Read - Data byte *r_byte*<B_{byte}-1:0> associated to the target memory location is retrieved from a core.

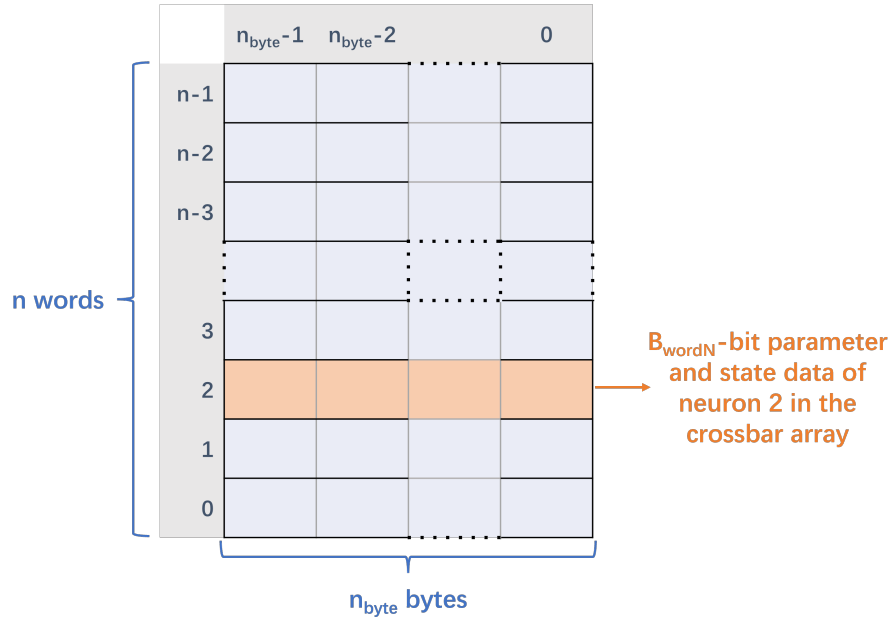

Figure 3.12: Neuron memory structure.

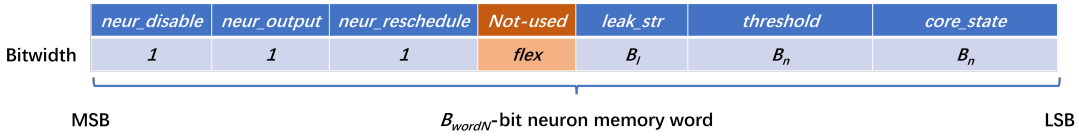

Figure 3.13: Contents of a word in the neuron memory.

configurable parameters. To align with byte-level access to neuron memory through SPI read/ write operations, a scaling factor, *lineScale* is introduced. By default, *lineScale* is defined as the width of a byte in this system.

The structure of the synapse memory is shown in Figure 3.14.

The entire synapse memory is divided into n blocks, each associated with a pre-synaptic neuron address. Each block contains n synaptic parameters distributed among n_{wordS} 32-bit synapse words. Within each synapse word, there are m_{wordS} synaptic parameters. In this way, we can minimize the size of synapse memory while maintaining convenient access to the synaptic parameters. The synapse memory is also accessed by byte via SPI.

The neuron and synapse memories are generated as blackboxes with interfaces when using memory IPs such as the Xilinx block memory generator by setting *blkbox* = *true*. Alternatively, if *blkbox* = *false*, the genrated hardware will implement register arrays as memory, which simplifies the process of behavioral simulation.

Among the parameters stored in the neuron and synapse memory, *leak_str* is stored and operated as an unsigned integer, and *threshold*, *core_state*, and *w* are stored and operated as signed integers.

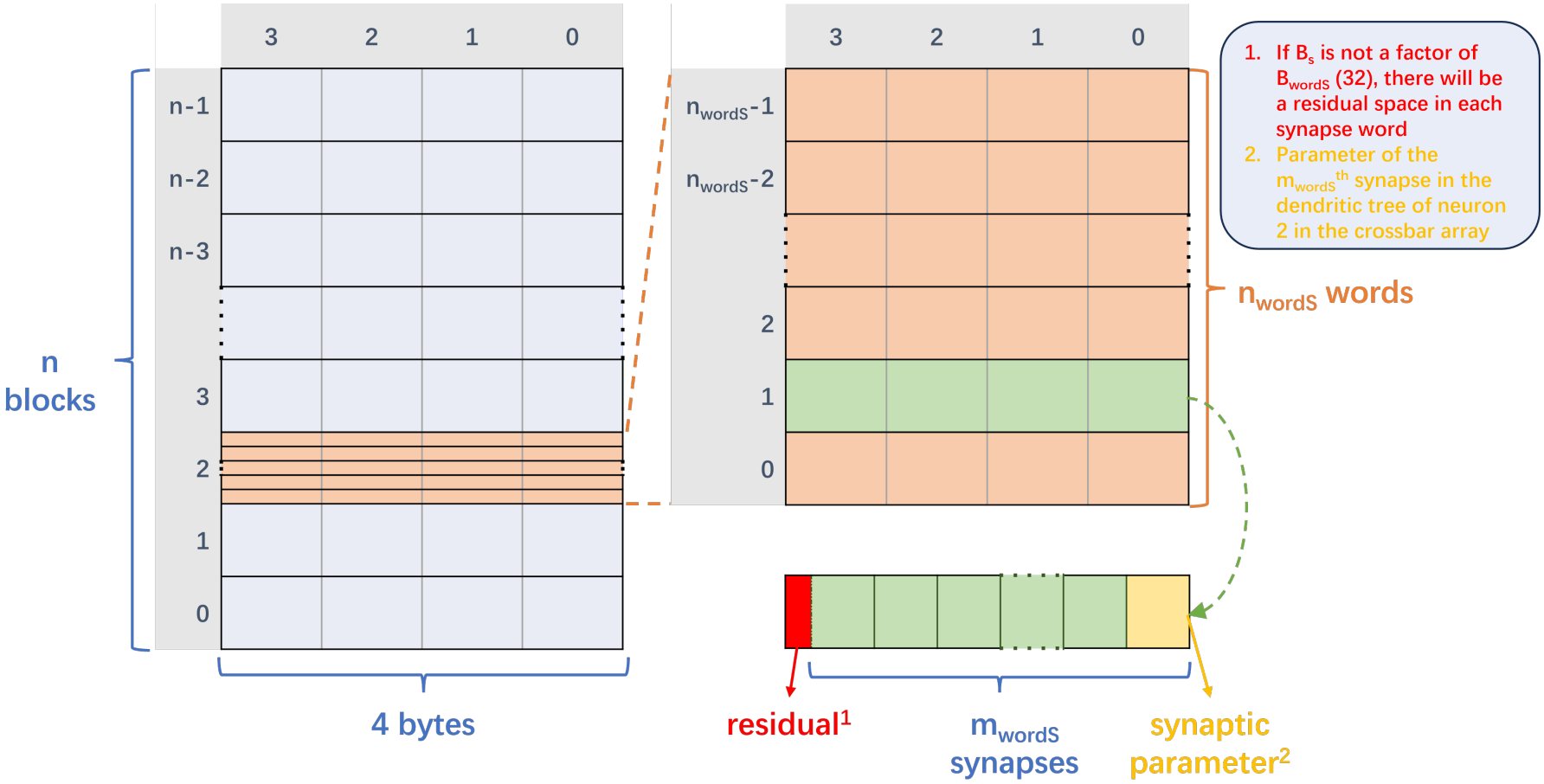

Figure 3.14: Synapse memory structure and contents.

Working principles

A single core in this architecture shares working principles similar to those of tinyODIN. As a parameterized architecture, the cycles needed for particular operations are also parameterized. As introduced in Section 3.2.3.3, each core can operate on four types of events. An event is pushed into the scheduler (if the scheduler FIFO is not full) once an AER packet is received. Events are distinguished by the contents of an AER packet, which are described in Table 3.11. The details of these events are introduced below.

Table 3.11: Contents of an input AER packet, adapted from [37].

ADDR<A_{AER}-1>	ADDR<A_{AER}-2>	ADDR<A_{AER}-3:0>	Event type
1	0	{*w*<B_s-1:0>,*neur*<A_{AER}-B_s-3:0>}	Virtual event
0	1	{(A_{AER}-2){1'b1}}	All-nerons time reference event
0	1	*neur*<A_n-1:0>	Single-neuron time reference event
0	0	*pre_neur*<A_n-1:0>	Neuron spike event

Virtual event

When a virtual event is received, the neuron specified by *neur*<A_{AER}-B_s-3:0> is stimulated with a signed weight *w*<B_s-1:0>, without activating a physical synapse. Virtual events go through the scheduler and it takes 3 cycles to handle a virtual event (1 cycle for a push to the scheduler and 2 cycles fore processing when the event is popped from the scheduler) [37].

The reason why the AER packet has a width of A_{AER} bits is that the AER input packet must hold a virtual synaptic weight of B_s bits together with a neuron address to access specified by *neur*<A_{AER}-B_s-3:0> in the case of a virtual event. To validate *neur*<A_{AER}-B_s-3:0>, $A_{AER} - B_s - 3$ should be at least 0, otherwise no neuron can be accessed. Considering that a virtual event is a testbench-type event [39] and does not need full access to neurons, the width of the AER packet is expressed as: $A_{AER} = max(A_n + 2, B_s + 4)$. In an extreme case with a large B_s, a virtual event can still access neurons in a 2-bit address field.

All-neurons and single-neuron time reference events

A single-neuron time reference event activates a time reference event (leakage) for neuron *neur*<A_n-1:0> and an all-neurons time reference event activates a time reference event (leakage) for all activated neurons, as defined by SPI_MAX_NEUR in Table 3.8 [37].

A single time reference event takes two clock cycles: read current neuron state and corresponding leakage strength in the first cycle; write the state updated by the LIF update logic back to the neuron memory in the second cycle. As an all-neurons time reference event applies leakage to all activated neurons, it takes 2(SPI_MAX_NEUR+1) cycles in total [37].

Neuron spike event

When a neuron spike event occurs, it stimulates all activated neurons, as defined by SPI_MAX_NEUR, with the synaptic weight associated with the pre-synaptic neuron *pre_neur*<A_n-1:0> [37]. Therefore, a neuron event takes SPI_MAX_NEUR+1 SOPs, where each SOP lasts two clock cycles, to update all associated neurons. As another cycle is required for pushing the event to the scheduler, the total number of cycles required for a neuron spike event is 2(SPI_MAX_NEUR+1)+1.

The timing diagram of a spike event associated to pre-synaptic neuron *i* is shown in Figure 3.15. In the first cycle of an SOP, the parameters and current state of the destination post-synaptic neuron *j* are retrieved from the memory and updated by the LIF update logic. In the second cycle, the current state of neuron *j* and the associated synaptic weight corresponding to the source neuron *i* are provided to the LIF update logic, the updated state of neuron *j* is then written back to the neuron memory [40]. To minimize memory accesses, m_{WordS} synapses are handled at a time, where $m_{WordS} = \lceil 32/B_s \rceil$ is the largest possible number of synapses that can be contained in a 32-bit synapse word.

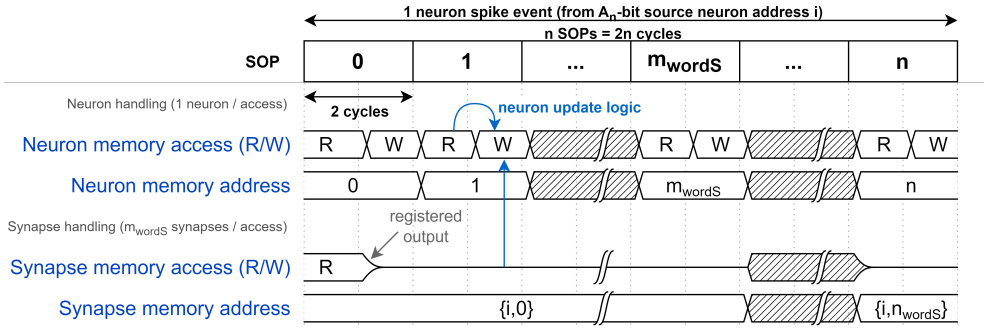

Figure 3.15: Timing diagram of a neuron event from neuron address *i*. This diagram depicts the case where SPI_MAX_NEUR is set as $n-1$. With a smaller SPI_MAX_NEUR, the process of a neuron event will terminate after the neuron specified by SPI_MAX_NEUR is updated. The figure is adapted from [40].

In a practical application, neuron spike events come from an input sensor or from a neuron in the network, whereas time reference events are user-defined events. Therefore, by default, the events that can be communicated among cores are neuron spike events.

3.2.4. Multi-core Architecture for sensor-fusion Applications

As discussed in Section 3.2.1.2, our generator should be able to generate multi-core architecture. In our specific sensor-fusion application as introduced in Section 2.2.3, our system classifies hand-gestures. In this situation, we only need a single output channel. Then, we can arrange the cores in a tree structure, where the output core is the root node of the tree. In this way, data in this architecture is always flowing from a child node to its parent node. An example of such a tree-structured core connection is shown in Figure 3.17. The generation process of such a multi-core architecture, the details of the arbiter, and the structure of the SPI bus will be introduced hereafter.

3.2.4.1. Generation process of a multi-core architecture

With the powerful collection methods provided by Scala (for the details of these methods, refer to Appendix B.1), we can generate cores, arbiters, and IOs of the entire multi-core architecture using the module *MultiCore_Gen* with the workflow shown in Figure 3.16.

The definitions of the functions and variables that appear in Figure 3.16 are listed in Table 3.12.

Generally, our multi-core generator is capable of generating tree-structured multi-core architectures like the one shown in Figure 3.17. An example with more detailed parameters is provided in Appendix B.2. Details of IO definitions and how connections are generated are provided in Appendix B.3.

3.2.4.2. Arbiter Design

Arbiters are required to connect different cores in our tree-structured architecture. To fit various multi-core structures, these arbiters are also parameterized and scalable. The parameters used in an arbiter are listed in Table 3.13. Figure 3.18 shows that an arbiter takes L AER inputs and provides only one AER output. As mentioned in Section 3.2.3.3, the output AER packet of a core only includes address information. As we suppose that a neuron event can only trigger another neuron event, the AER packets from preceding cores are expanded with the neuron event operation code, which is ”00”. However, there is also an option for an arbiter to directly take an external AER input, allowing users to specify other events through this external AER input. The connection of external AER inputs is also shown in Figure 3.17.

To increase throughput and reduce the time overhead for arbitration, pipelining is introduced in AER communication between different stages [11]. This is achieved by implementing a small FIFO in each arbiter, whose depth depends on the number of inputs and is defined by D_a. Two different AER communication patterns (with and without a FIFO) are shown in Figure 3.19. When the FIFO is not full, the arbiter can push a request from a preceding core into the FIFO and immediately acknowledge the request once the it is stored in the FIFO. This helps prevent stalling a core in the preceding stage due to the busy AER bus.

The arbiter includes a parameterized round-robin scheduler which can be scaled as required. The round-robin scheduler is realized in a one-line code with a round-robin method *OHMasking.roundRobin()* provided by SpinalHDL [98, 89], as shown in Listing 3.14.

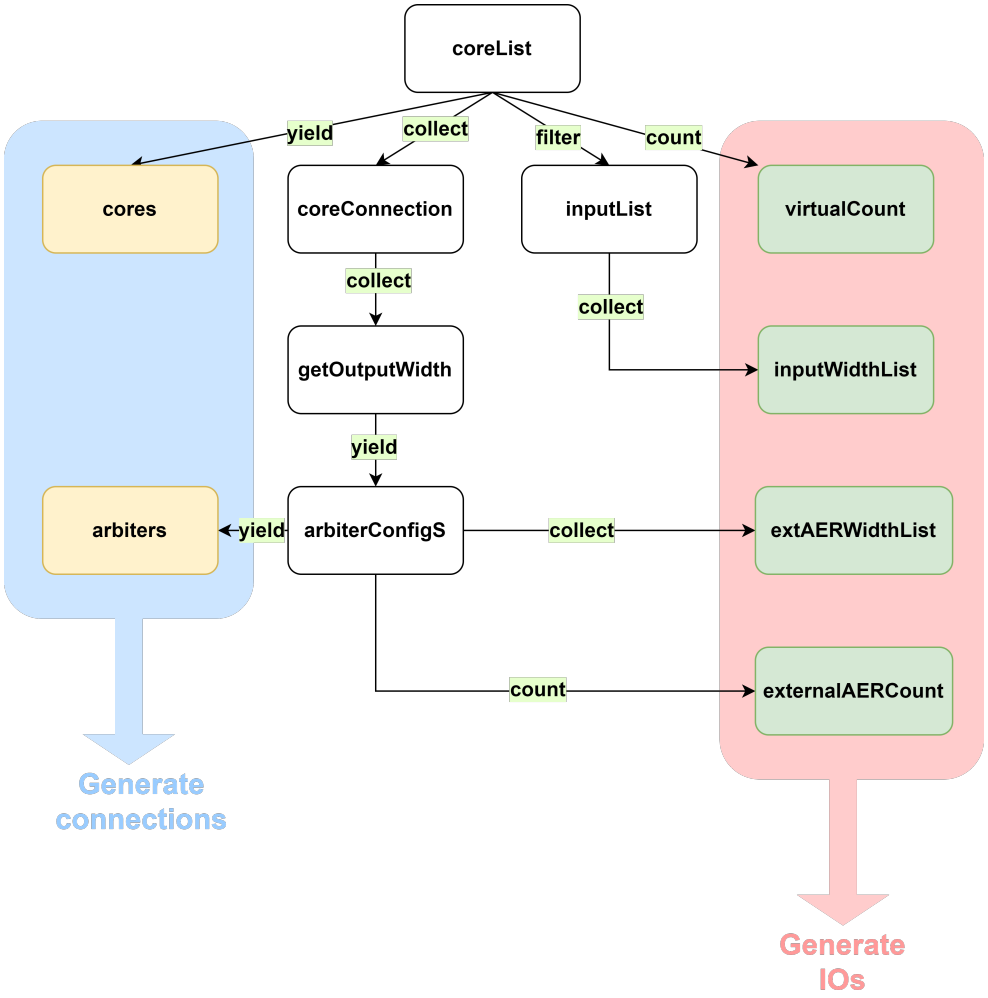

Figure 3.16: Workflow of *MultiCore_Gen*. To generate a muti-core architecture, we first define the parameters of each core with the *CoreConfig* class. Then we input a list of *CoreConfig* definitions as a parameter to the module *MultiCore_Gen*. Using collection methods, cores and arbiters can be generated from the parameters in the *coreList*. Variables for IO definitions can also be derived from the *coreList* with collection methods. Then, the connections in the architecture are generated.

Table 3.12: Functions and variables for multi-core generation.

Name	Type	Description
coreList	List[*CoreConfig*]	A *List* of *CoreConfig*s, includes parameters of each core and connection among different cores
coreConnection	List[(Int, List[Int],Boolean)]	A *List* of tuples that contains the core *idx*, *inputCore*, and *externalAER* of the *CoreConfig*s with non-empty *inputCore* in *coreList*
getOutputWidth	List[Int]	A function, given the *inputCore* of a *CoreConfig*, it gathers the A_{nout} of the cores that connect to the specified core and returns in a *List*.
arbiterConfigS	List[*ArbiterConfig*]	A *List* of *ArbiterConfig*s, the parameters of each are generated by applying *getOutputWidth* to the elements in *coreConnection*.
inputList	List[*CoreConfig*]	A *List* of *CoreConfig*s with empty *inputCore*, the cores generated from these *CoreConfig*s are directly connected to the inputs of the multi-core architecture.
virtualCount	Int	The number of virtual cores. These cores are only implemented with AER interfaces.
inputWidthList	List[Int]	A *List* containing the *AERWidth* of *CoreConfig*s with empty *inputCore*.
extAERWidthList	List[Int]	A *List* containing the widths of external AER channels
externalAERCount	Int	The number of external AER channels

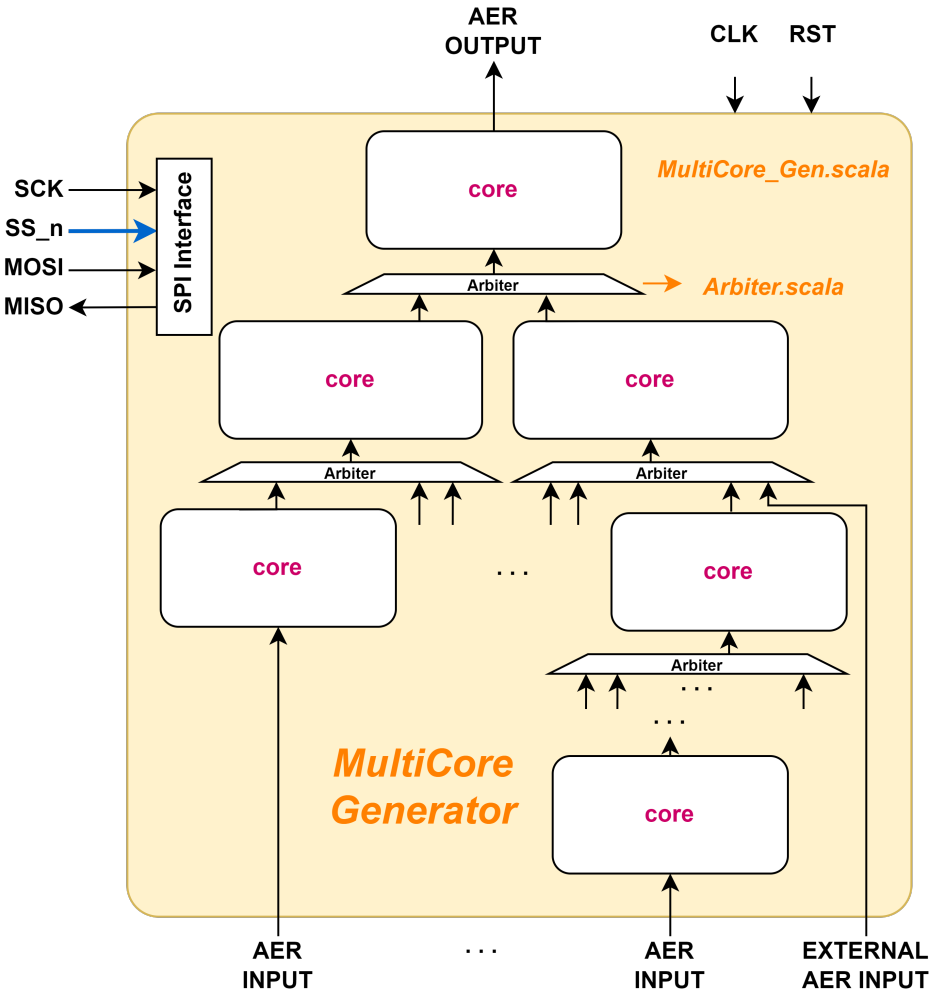

Figure 3.17: An illustration of a tree-structured mullti-core architecture.

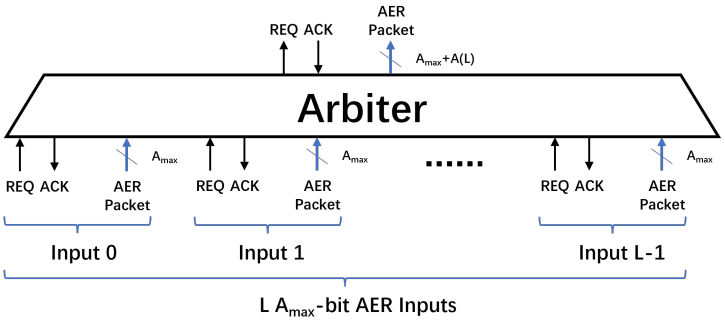

Figure 3.18: Input and output interfaces of an arbiter.

Listing 3.14: Example of round-robin logic in SpinalHDL

```
1     val maskArb = OHMasking.roundRobin(AERInReqSync , lastArb)
```

Here, we define a round-robin logic *maskArb* with *OHMasking.roundRobin()*. *AERInReqSync* contains the input requests in bit vector form and *lastArb* is the one-hot priority. The round-robin scheduler starts looking for inputs from the position specified by the one-hot priority [98].

The block diagram of an arbiter and its round-robin scheduller are shown in Figures 3.20a and 3.20b. To control the input interfaces and the output interface separately, two FSMs are implemented in the controller. The states of the FSMs are shown in Figure 3.21.

Table 3.13: Parameters of an arbiter (class *ArbiterConfig* in *Config.scala*).

Category	Type	Parameter	Definition	Note
connection	List[Int]	N	List of the widths for each input	
control	Boolean	*externalAERin*	The same as *externalAER* defined in coreConfig	
number	Int	L	Length of input list, including the optional external AER input	
number	Int	D_a	Depth of the FIFO in this AER	$D_a = \begin{cases} 2, & (L \leq 4) \\ 2^{A(L)-1}, & (L > 4) \end{cases}$
address width	Int	A_{max}	Maximum input bitwidth	

3.2.4.3. SPI connection

To interface multiple cores with a single SPI interface, a chip-select mechanism is used. As an example, let k be an arbitrary integer, the block diagram of this SPI configuration that connects the $k+1$ cores is shown in Figure 3.22. In this structure, only one core (slave) is activated at a time.

When not selected, each core sets its `MISO` to low, so that a bitwise OR logic can output the `MISO` signal from the activated core.

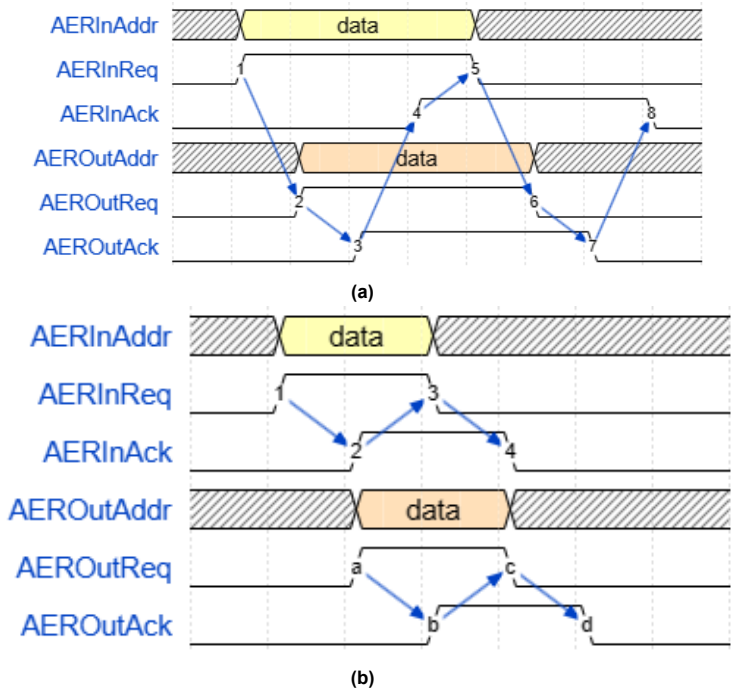

Figure 3.19: Two different AER communication patterns between two stages. `AERInAddr`, `AERInReq` and `AERInAck` are the input AER interface of an arbiter, which connects to an output AER interface of a preceding core. `AEROutAddr`, `AEROutReq` and `AEROutAck` are the output AER interface of an arbiter, which connects to an input AER interface of a succeeding core. With a pipelining mechanism, the acknowledge to the preceding core is independent from the acknowledge from the succeeding core and improves the throughput. (a) Without pipelining. (b) With pipelining.

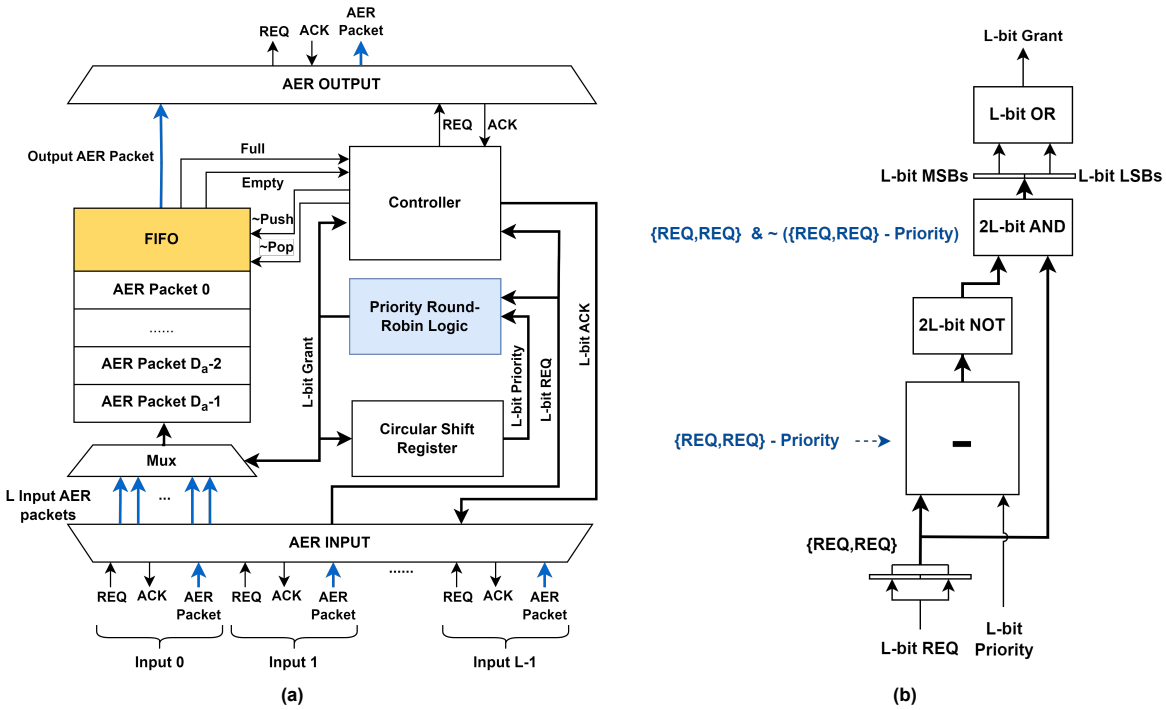

Figure 3.20: The block diagram of: (a) an arbiter; (b) the round-robin scheduler.

Figure 3.21: States of the FSMs in an arbiter. (a) States controlling input interfaces. `IDLE`: idle state; `WAIT_FIFO`: waits for FIFO and pushes an input AER event once FIFO is not full; `WAIT_REQDN`: waits for the AER request line to be pulled down as the corresponding acknowledge is asserted when entering this state; `1`: transition when input request exists but FIFO is full; `2`: transition when input request exists and FIFO is not full; `3`: transition when FIFO is not full; `4`: transition when AER request is pulled down. (b) States controlling output interfaces: `IDLE`: idle state; `POP`: pops an event from the FIFO to the output AER interface, a request is raised at the same time; `WAIT_ACK`: waits for the acknowledge from the succeeding core; `WAIT_ACKDN`: waits for the acknowledge to be de-asserted; `1`: transition when FIFO is not empty; `2`: transition when acknowledge is asserted; `3`: transition when acknowledge is de-asserted.

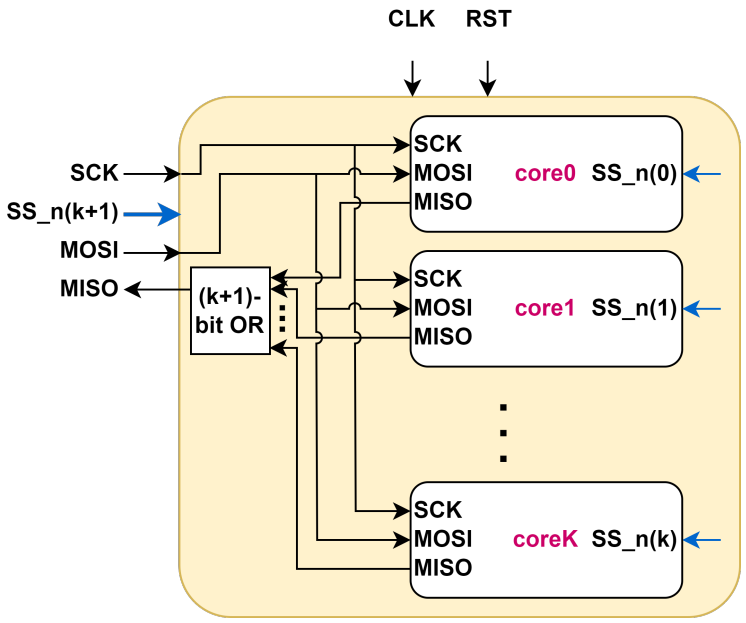

Figure 3.22: SPI interfacing.

4

Deployment of Gesture Recognition Application

In this chapter, we will first introduce the baseline model we chose for the hand-gesture recognition application. Then, the proposed hardware configurations for this application will be introduced.

4.1. Machine Learning Baseline

In this section, we will first introduce more details on the dataset, including how we handle this dataset to meet our requirements. In general, we used an existing neuromorphic baseline, we modified the network architecture by sweeping the configurable parameters so that we can optimize the baseline for our hardware.

To train these models, we use snnTorch [33] as our training framework and Brevitas [78] as a quantization tool. *Adam* [62] was chosen as the optimizer with default parameters except for the learning rate.

4.1.1. Details of dataset

As introduced in Section 2.2.3, the sensor-fusion dataset we used in this project includes the DVS and EMG data of 5 gestures. Specifically, it consists of 1575 recordings, with each recording lasting for around 2 seconds. The 1575 recordings are obtained from 21 subjects, where each subject performs 5 gestures in each of the 3 sessions, and each gesture is repeated 5 times [18]. The reasonable response time for hand-gesture recognition is 200 ms as it matches the time required to perform low-latency prosthesis control [18]. Therefore, we can slice the dataset into smaller pieces to obtain more samples for training [18]. Each $\sim$2 s recording is equally sliced into 10 slices, together forming 15750 samples. To accommodate for the varying length of the recordings, slices shorter than 200 ms are discarded. By doing this, we avoid generating empty recordings and get more samples for training.

In the training process, the temporal resolution of each recording is reduced to 500 time bins to reduce the number of simulation steps. Although this method reduces temporal precision [33], it will reduce simulation time and data size during training.

4.1.1.1. DVS

The DVS data in the dataset is in a resolution of 40×40 pixels. Each of the recordings is obtained from a 128×128 DVS sensor [18]. This modification is made mainly to reduce the number of neurons we need in the input layer [18].

4.1.1.2. EMG

The EMG recording in this dataset has been converted to events, we refer the reader to [18] for details on retrieving and processing EMG data.

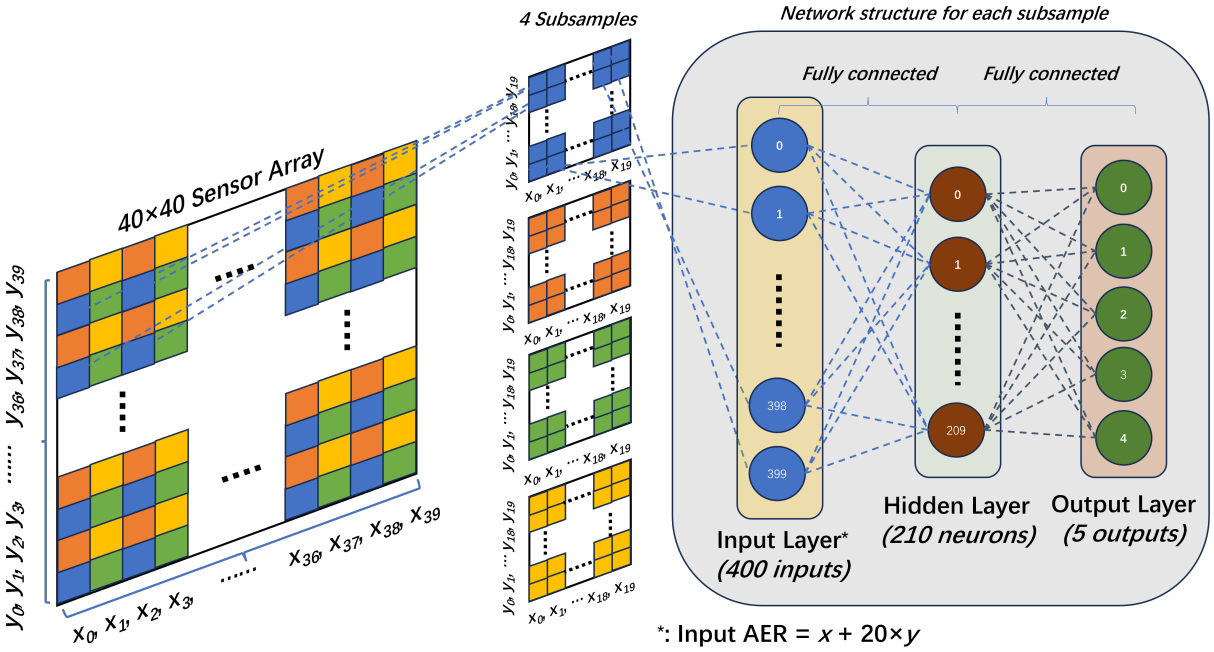

Figure 4.1: MLP architecture for DVS classification.

4.1.2. Neural network for vision data classification

In this subsection, we will introduce the network architecture used for DVS classification, the hyperparameter sweeping process, and the baseline model.

4.1.2.1. Network architecture

The network architecture for DVS classification is shown in Figure 4.1. It contains 4 2-layer sub-MLPs, each handles a 20×20 sub-sample of the recording with a 400-210-5 network structure. The outputs from each of the sub-MLPs are simply added up to obtain the final classification result. This network was proposed as a baseline for neuromorphic computing in hand-gesture recognition in [18]. In [18], this network architecture is deployed on a MorphIC processor with 1-bit synaptic weights [39].

Coding Scheme

For the output coding scheme, we choose the rate coding scheme as it is the most widely used coding scheme in SNN applications [47], and snnTorch provides good support for it.

For input encoding, as our hardware is not capable of handling inputs with two polarities, we only considered positive events in the recordings. Throughout our training and verification process, we show that single-polarity inputs already yield good accuracy for DVS classification.

Loss function

We use a cross-entropy loss function, hence the loss function we used in SNNs for DVS classification is *ce_count_loss()* in snnTorch. This function calculates the cross-entropy loss based on the accumulated number of spikes for each output neuron throughout the entire simulation time range. The expression of the cross-entropy loss function is

$$l_n = -\log \frac{e^{spike\_count_{y_n}}}{\sum_{c=1}^{C} e^{spike\_count_c}}, \quad \mathcal{L} = \sum_{n=1}^{N} \frac{1}{N} l_n \ . \tag{4.1}$$

In Equation 4.1, the loss $\mathcal{L}$ is computed for a batch that includes N samples. The loss of the n^{th} sample is l_n. $spike\_count_c$ denotes the accumulated number of spikes from the output neuron c. y_n is the target output neuron of the n^{th} sample. If the $spike\_count$ of the neuron y_n is small, then it results in a larger l_n and hence in a larger $\mathcal{L}$. This loss function aims to encourage the target output neuron to spike while suppressing the firing of other neurons, as illustrated in Figure 4.2.

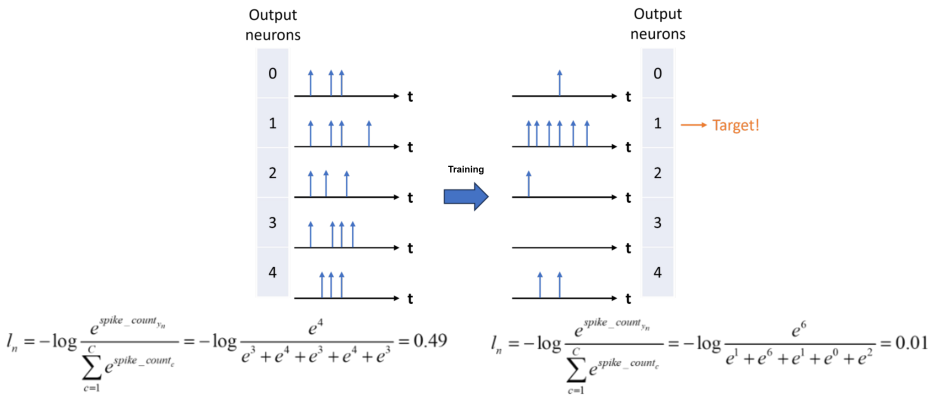

Figure 4.2: An interpretation on the training process. In this figure, the effect of the training process encouraged neuron 1 to fire and suppresses spikes from other neurons.

Quantization

We use a QAT scheme provided by Brevitas [78]. By default, it is a symmetric quantization process. Quantized weights, scaling factors, and zero points can be read directly from the training framework.

4.1.2.2. Hyperparameter sweeping

For simplicity and fast iteration, we divided the whole dataset into 20% as a test set, 70% as a training set, and 10% as a validation set.

Due to the higher complexity of the network architecture for DVS classification, the training speed is significantly slower than the training of the network for EMG classification. Therefore, for the hyperparameters, we focus on the bitwidth of the synaptic weights (B_s).

In the architecture in [18], a 1-bit resolution is used. So, we sweep B_s from 1 to 3 bits, with B_n adapted accordingly. The hyperparameters are listed in Table 4.1. Here, *N* stands for the number of neurons in the hidden layer.

Table 4.1: Hyperparameters swept for DVS classification.

Parameter	Value
B_s	{1-3}
N	210
Learning rate	0.0005
Epochs	20
Decay	0.9999
Batchsize	200
Sweep method	Grid

In each run, the network is trained for 20 epochs (each sample in the training set is accessed 20 times).

Figure 4.3 shows the classification results for the validation set, each line in the figure is the average result for runs with the same B_s. It is clear that higher B_s results in better accuracy. However, a higher B_s leads to higher memory usage, and we need to make a trade-off between accuracy and resources. For this reason, we choose $B_s = 2$.

4.1.2.3. Baseline result

As we choose $B_s = 2$, the hyperarameters used in the baseline model are listed in Table 4.2.

The results on the validation set and the test set based on 3 runs are listed in Table 4.3.

4.1.3. Neural Network Setup for Electromyography (EMG) Data Classification

In this subsection, we will introduce the network architecture we used for EMG classification, the hyperparameter sweeping process, and the baseline model.

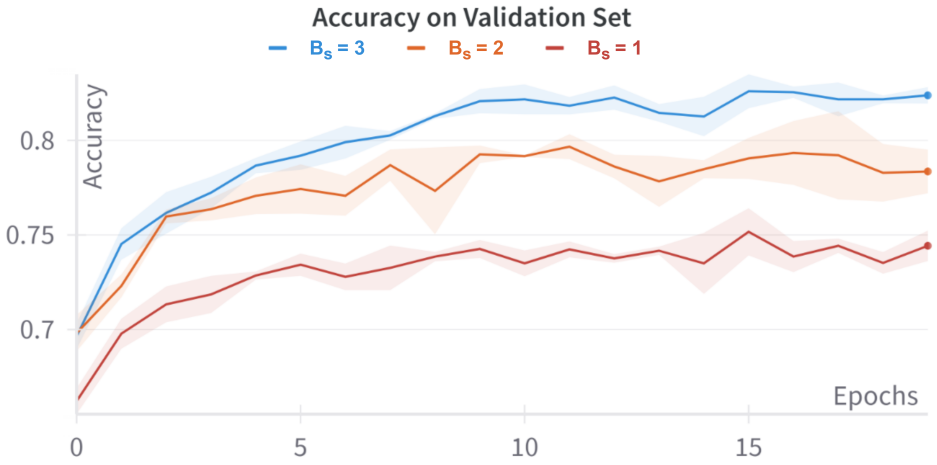

Figure 4.3: Accuracy of DVS networks on the validation set with different B_s, average of 3 runs.

Table 4.2: Hyperparameters of the baseline model for DVS classification.

Parameter	Value
B_s	2
N	210
Learning rate	0.0005
Epochs	20
Decay	0.9999
Batchsize	200

Table 4.3: Baseline results of DVS classification.

Set	Validation	Test
Accuracy	78.36%±1.15%	76.78%±0.85%

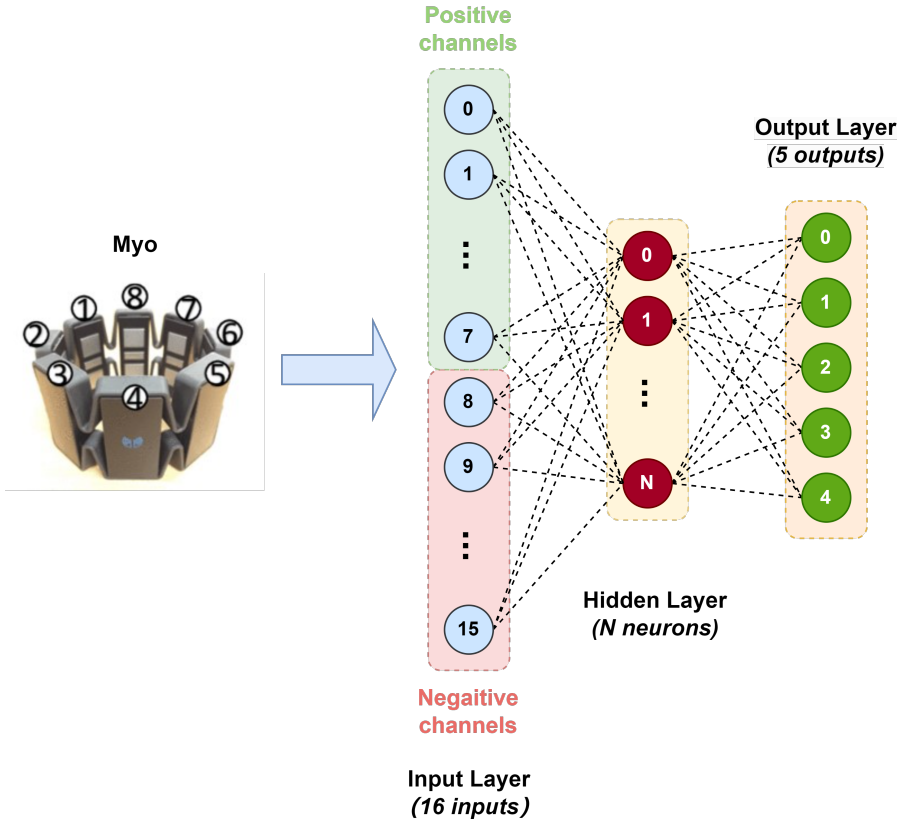

Figure 4.4: MLP architecture for EMG classification. The figure of Myo is taken from [29].

4.1.3.1. Network architecture

The network architecture for the EMG classification is shown in Figure 4.4. This 2-layer MLP is structured as 16-*N*-5, where *N* is the number of neurons in the hidden layer, which will be determined by sweeping. A baseline model for EMG classification is proposed in [18], which is a 16-230-5 network and is deployed on an ODIN processor with 3-bit synaptic weights. In this work, we modified *N* so that it could be deployed on our hardware, taking as little resources as possible.

Coding scheme

Due to the low dimensionality of the 8-channel EMG data, both positive and negative spikes are used in this architecture. Because our hardware cannot distinguish the polarity of an input event, we use 16 input neurons, 8 taking the positive events, while the other 8 handling the negative ones.

For the output coding scheme, we also use the rate coding scheme as in the DVS classification.

Loss function

The loss function we used for the EMG classification is the same cross-entropy loss function as we used in the DVS task.

4.1.3.2. Hyperparameter sweeping

For EMG classification, we use the same dataset division as in the DVS task. As the network architecture for EMG classification is less complex, we can sweep in a wider scope and perform more tuning. Therefore, B_s and *N* are searched during hyperparameter sweeping. We first set *N* = 230, which is the same as the model in [18], and we search for B_s from 2 to 5. Then we choose a B_s based on the results and sweep for *N*. B_n is adapted accordingly in the process. The hyperparameters for B_s sweeping are listed in Table 4.4. The accuracy on the validation set during the sweeping process is shown in Figure 4.5.

From Figure 4.5, we can see that using 2-bit or 3-bit resolution in the network results in $\sim 3\%$ accuracy loss compared to 4-bit resolution. The difference in the accuracy of using 4- and 5-bit synapses is within $\sim 1\%$. Hence, we chose $B_s = 4$ in both our network architecture and hardware design and searched for *N* based on this choice. The parameters for *N* sweeping are listed in Table 4.5.

Table 4.4: B_s sweeping for EMG classification.

Parameter	Value
B_s	{2-5}
N	230
Learning rate	0.0005
Epochs	20
Decay	0.9999
Batchsize	200
Sweep method	Grid

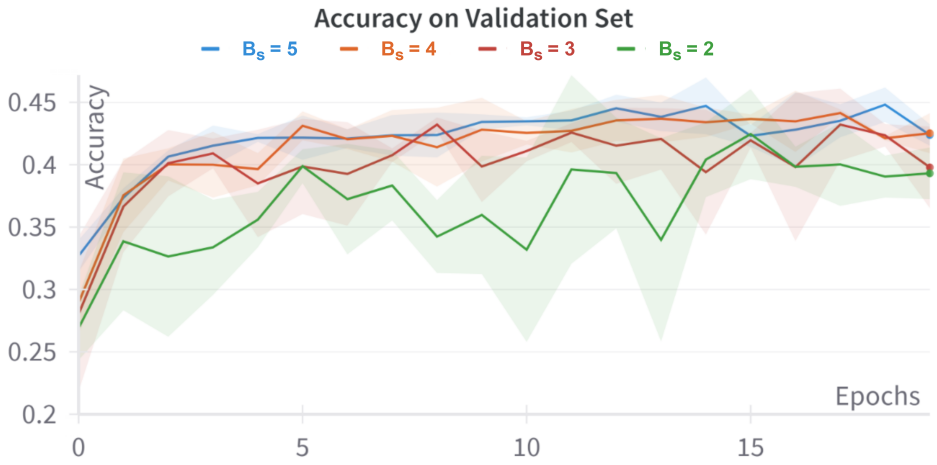

Figure 4.5: Accuracy of EMG networks on the validation set with different B_s, average of 3 runs.

Table 4.5: *N* sweeping for EMG classification.

Parameter	Value
B_s	4
N	{110, 140, 170, 200, 230}
Learning rate	0.0005
Epochs	20
Decay	0.9999
Batchsize	200
Sweep method	Grid

Figure 4.6 shows the accuracy on the validation set with different *N*. From this figure we can conclude that different values (in the range of our sweeping) of *N* have a trivial affect on accuracy (within $\sim 1\%$).

As we wish to use smaller hardware at the same time, achieving acceptable performance, we chose *N* = 110 in our baseline model and hardware implementation. Compared to [18], which uses a 256-neuron ODIN processor for EMG classification, we can deploy the network on a 128-neuron core with 50% less neuron memory and 75% less synapse memory in an ideal case.

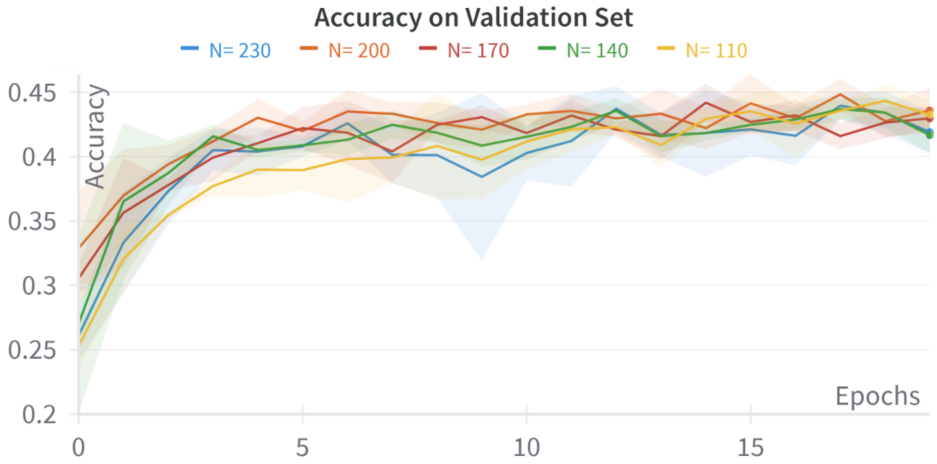

Figure 4.6: Accuracy of EMG networks on the validation set with different *N*, average of 3 runs.

4.1.3.3. Baseline result

The final parameters for EMG-classification are listed in Table 4.6. With these parameters, we are able to obtain the accuracy on validation sets and test set listed in Table 4.7.

Table 4.6: Hyperparameters of the EMG classification network.

Parameter	Value
B_s	4
N	110
Learning rate	0.0005
Epochs	20
Decay	0.9999
Batchsize	200

Table 4.7: Baseline results of EMG classification.

Set	Validation	Test
Accuracy	43.26%±1.14%	44.02%±1.64%

4.1.4. Neural Network Setup for sensor-fusion

4.1.4.1. Network Architecture

The sensor-fusion network combines the DVS 4 sub-MLPs and the EMG MLP. The hidden layers of these networks are concatenated as the input of the output layer. Therefore, the output layer is 950-5. This fusion network is also proposed in [18] and since we use a smaller MLP for EMG classification, the fusion network is also smaller. The network architecture is shown in Figure 4.7.

To save time, in the training process, we kept the trained parameters in the 4 sub-MLPs and the EMG MLP and only train the output layer.

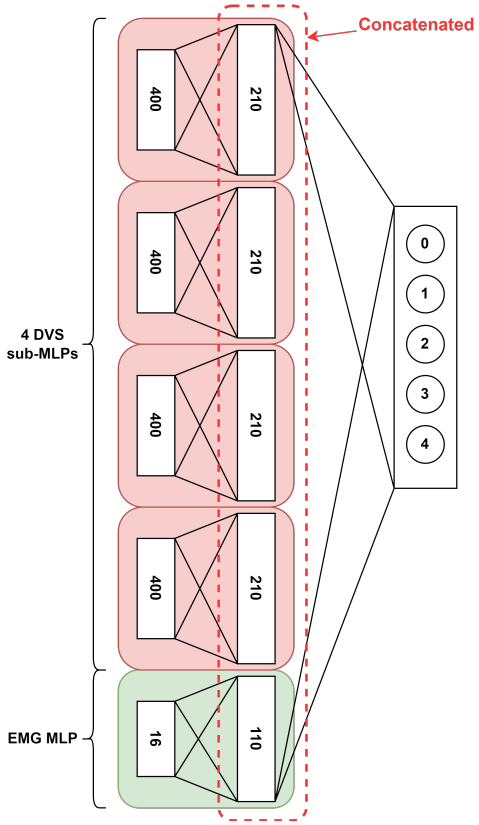

Figure 4.7: Network architecture of the fusion network to classify 5 hand-gesture classes.

Coding scheme and loss function
We use the same coding scheme and loss function as in DVS and EMG networks.

4.1.4.2. Baseline results
We kept the B_s in the EMG MLP and we trained with the parameters in Table 4.8. After 20 epochs in total, we are able to obtain the results listed in Table 4.9.

Table 4.8: Parameters of the sensor-fusion network.

Parameter	Value
Bs	4
N	950
Learning rate	0.001
Epochs	20
Decay	0.9999
Batchsize	200

4.2. Mapping on Proposed Hardware
In this section, we will introduce how we deploy the networks on the proposed hardware for DVS classification and EMG classification.

4.2.1. Hardware configuration for vision data classification
The hardware for DVS classification is depicted in Figure 4.8a. This specific multi-core configuration used for DVS classification is referred to as the DVS-core.

Table 4.9: Baseline results of fusion network on hand-gesture classification.

Set	Validation	Test
Accuracy	81.38%±5.88%	82.08%±5.25%

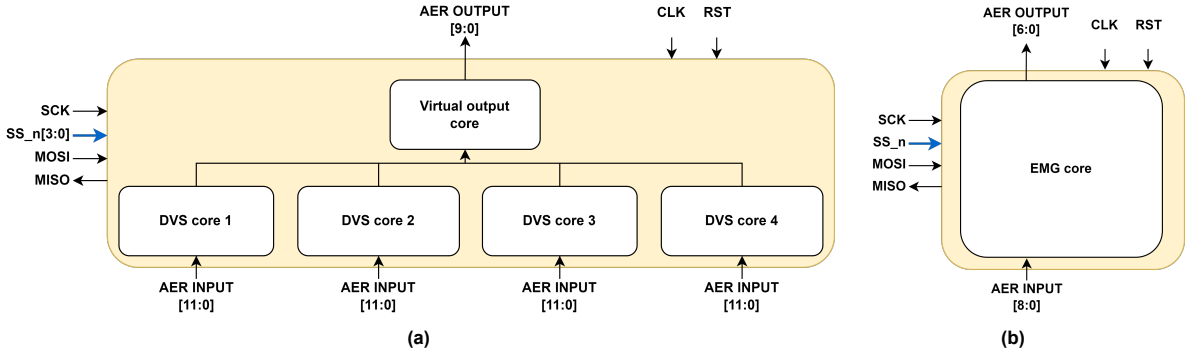

Figure 4.8: The hardware architecture for DVS classification and EMG classification. (a) Architecture for DVS classification, the arbiter and the SPI interface are omitted from the figure. (b) Architecture for EMG DVS classification, the SPI interface is omitted from the figure. Combining these hardware architecture, we can get the hardware architecture for the sensor-fusion network in a straightforward way.

The DVS-core is a quad-core architecture, each core comprises 640 5-bit neurons, each neuron word is 16-bit in length, and the synapse resolution is 2-bit. The outputs from the cores are sent to a virtual core, which truncates the address and retains only the lower address field of a single core. In this way, the results from the cores are simply added as the final classification output.

The neurons in the network are mapped to the crossbar architecture of the DVS-core following the pattern illustrated in Figure 4.9.

4.2.2. Hardware configuration for electromyography (EMG) data classification

The hardware for EMG classification is shown in Figure 4.8b. This architecture is referred to as the EMG-core.

It is a single core architecture, and the core comprises 128 7-bit neurons, each neuron word is 32-bit in length, and the synapse resolution is 4-bit. The neurons are mapped to the crossbar in the way shown in Figure 4.10.

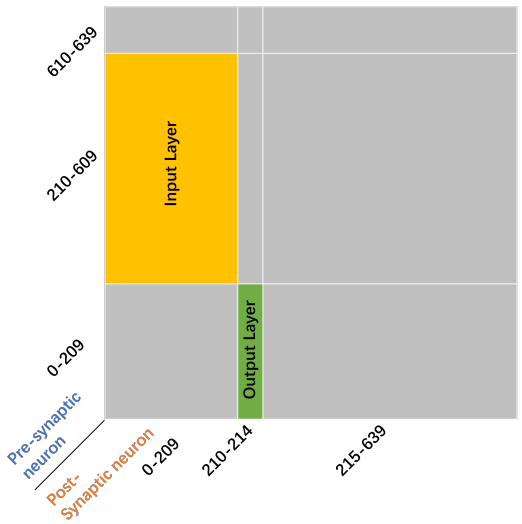

Figure 4.9: The neuron mapping of each core in the DVS-core.

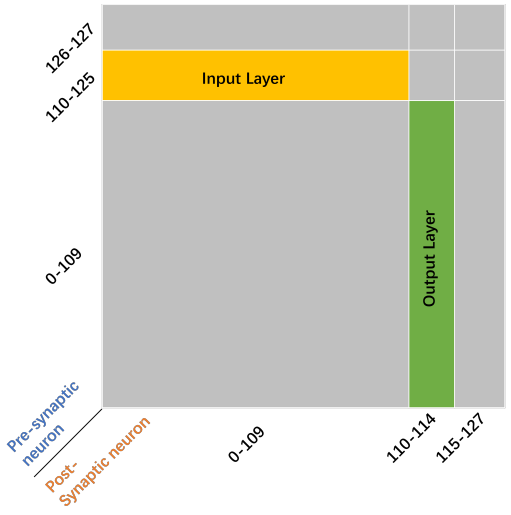

Figure 4.10: The neuron mapping of each core in EMG-core.

5 Verification

5.1. Hardware Verification Setup

In this section, we will introduce our hardware setup for verification.

5.1.1. Target FPGA platform

The generated cores are synthesized using Xilinx Vivado 2023.1 and implemented on a PYNQ-Z2 board (shown in Figure 5.1), which comprises a Xilinx Zynq-7000 series SoC. The Zynq-7000 series SoC is equipped with a dual-core ARM Cortex-A9 processor and programmable logic (equivalent to Artix-7 FPGAs). The SoC is thus divided into two parts: the processing system (PS) and the programmable logic (PL). The communication interface between the PS and the PL is an AXI-based GPIO interface. Through the Xilinx software portal, Vitis 2023.1, we can directly program the PS and control the core implemented in the PL through GPIO interfaces, which is helpful for the verification process. Most interfaces on our generated hardware system in the PL can be directly connected to GPIO interfaces. However, since the SPI is a serial interface, we need to convert the parallel GPIO signal into the serial SPI signal, which is the purpose of the SPI_GPIO interface (*SPI_GPIO_Interface.scala* and *SPI_GPIO_Interface.v* in Figure 3.8). To save GPIO interfaces, we have also introduced an input demultiplexer for AER interfaces, allowing multiple cores to be interfaced using only one GPIO interface.

5.1.1.1. Generated modules for verification

As mentioned in Section 3.2.2, two interface modules are generated for verification purposes.

GPIO-SPI interface

This interface bridges the parallel GPIO interface of the PS and the serial SPI interface of the multi-core architecture implemented in the PL. It is also parameterized so that we can easily generate interfaces for different SPI widths.

The timing diagrams of this interface are shown in Figure 5.2.

Input demux

To save GPIO resources, we use a single GPIO to output address events, which are distributed to different cores in our multi-core architecture through a parameterized input demultiplexer, which is generated by *InputMux.scala*. In this scheme, the highest 2-bit address specifies the destination core, and the rest of the bits are sent to the cores.

The entire setup for verification

Because the estimated hardware resources required exceed the resources on our target SoC, we only implement the DVS-core and the EMG-core on the FPGA. The block diagram of the entire system is shown in Figure 5.3. As shown in the figure, the test is performed automatically with the board. Using Vitis and Vivado, the testing program and the hardware bitstream can be programmed to the board through USB-JTAG connection. The test set is stored on an SD card. In the verification process, the SoC reads the test

[1] https://www.tulembedded.com/fpga/ProductsPYNQ-Z2.html

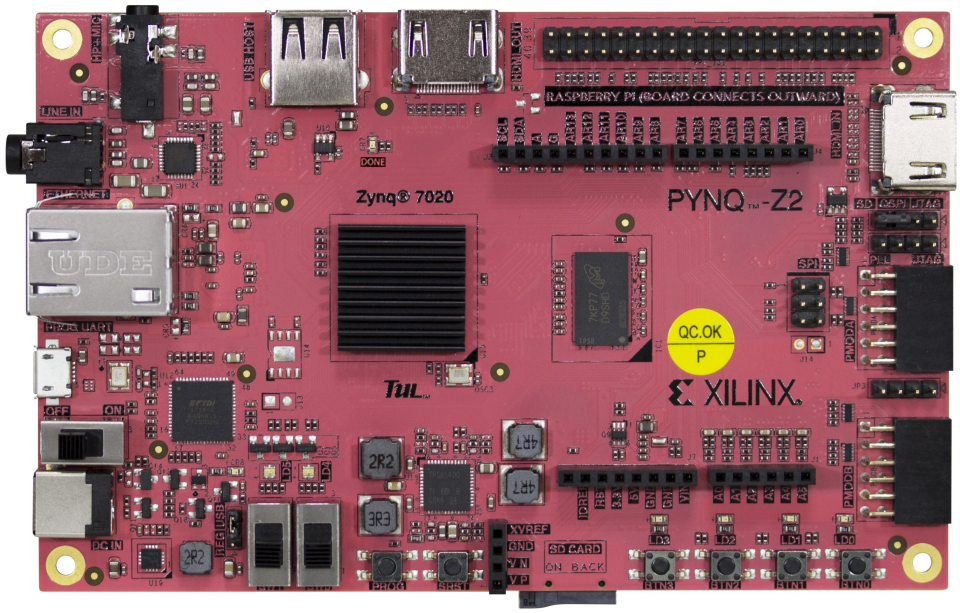

Figure 5.1: PYNQ board with the target SoC platform[1].

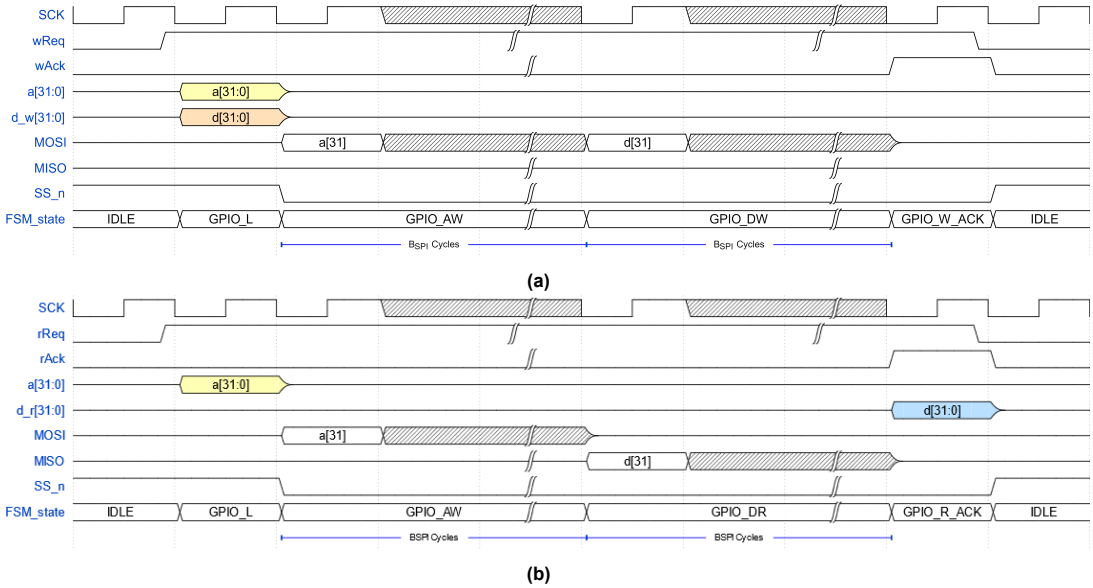

Figure 5.2: The timing diagram of the GPIO-SPI interface, $B_{SPI} = 32$. `SCK`: the clock of the SPI interface; `wReq`: write request from the PS; `rReq`: read request from the PS; `wAck`: write acknowledge to the PS, will be asserted after all B_{SPI} (32) bits of data are transmitted; `rAck`: read acknowledge to the PS, will be asserted after all B_{SPI} (32) bits of data are received; `a[31:0]`: address field of SPI communication; `d_W[31:0]`: data to be write through the SPI; `d_r[31:0]`: data read from the SPI; `SS_n` SPI chip-select signal; `FSM_state`: the states of the FSM of this interface.

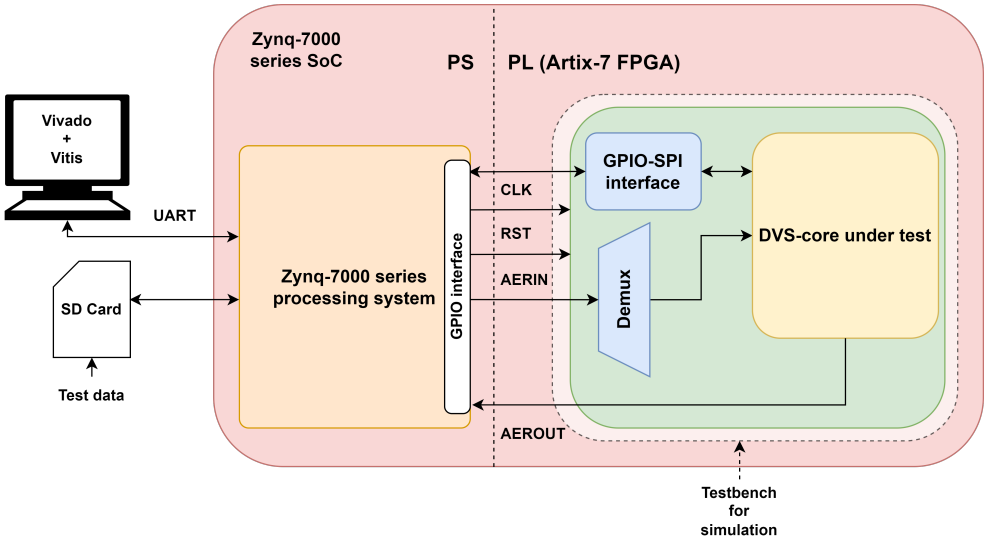

Figure 5.3: Hardware verification setup.

set data from the SD card and writes the results to the SD card. The verification process can be monitored through Vitis with USB-UART connection.

The setup for the EMG-core is similar. Since the EMG-core is a single-core architecture, the input demux has been removed.

Both verification setups run at a clock frequency of 25 MHz, and a SCK frequency of of 5 MHz. The PS runs at 650 MHz. In the verification setup, the samples are handeled in real-time without any acceleration.

5.1.1.2. Simulation

Here, we present the simulation results for DVS classification. In the simulation and verification process, as we set a decay rate β = 0.9999, and a sample is divided into 500 time bins, the length of a bin is less then 400 μs. Assuming a temporal resolution of 1 μs, as $\beta^{400} = 0.9999^{400} \simeq 0.96$, we can ignore the neuron leakage in the entire process. Hence we do not need to add time reference events in our deployment.

Without time reference events, we can also send the input events in a spike-by-spike manner, and hence accelerate the recognition process. As shown in Figure 5.4, events in a $\sim$200 ms scope can be compressed into a $\sim$20 ms scope in this configuration. This result indicates that the processing of the spike-based information can be accelerated with our hardware.

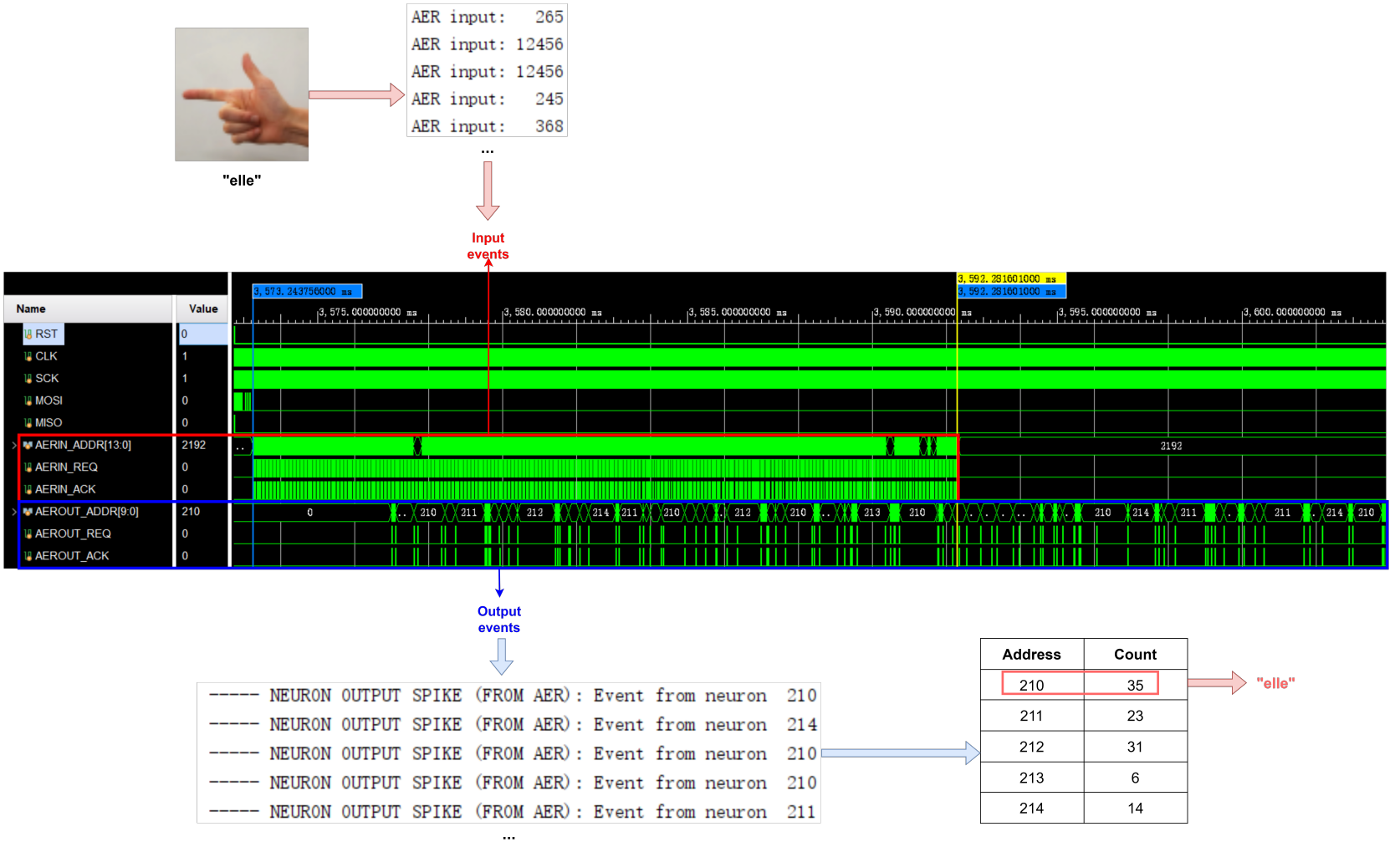

Figure 5.4: Simulation process of a DVS classification. The figure of ”elle” is taken from [18].

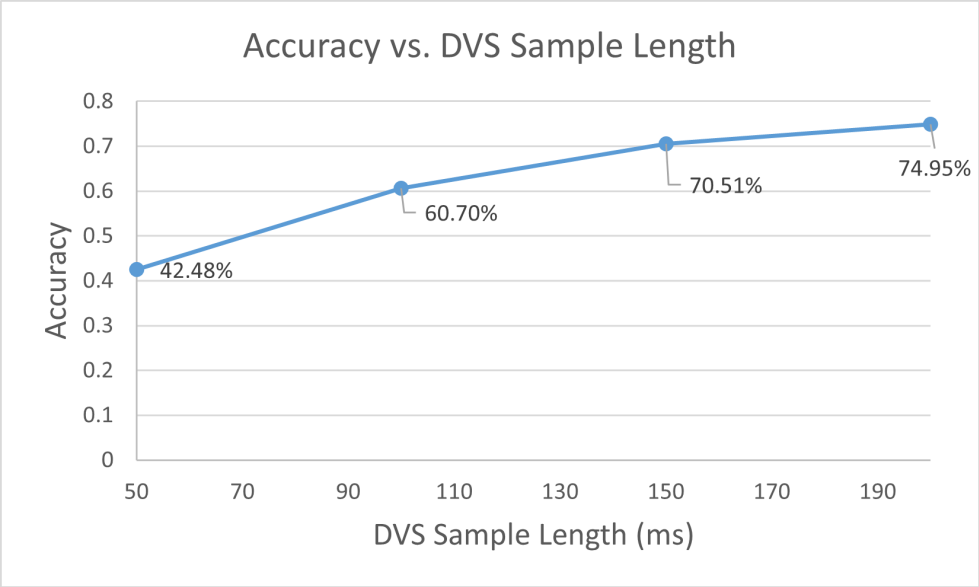

Figure 5.5: Accuracy versus DVS sample length.

5.2. Results from the FPGA Deployment

In this section, results of the DVS sub-MLPs and the EMG MLP deployed on our verification setup will be presented.

5.2.1. Test results

The test results are obtained on the same test sets as we used in Sections 4.1.2 and 4.1.3.

5.2.1.1. DVS classification result

We achieved an accuracy of 74.95%. Although with an accuracy drop compared to our baseline, the results show that our DVS-core is able to perform classification of DVS data.

5.2.1.2. EMG classification result

We achieved an accuracy of 17.36%. This result does not match our baseline. The analysis of possible reasons is presented in Appendix C and their resolution is left for future work.

5.2.2. Accuracy-latency trade-off

As the rate coding scheme relies on the accumulation of output spikes over time, with a longer sample length, the network is more likely to give a correct classification. An example showing the relationship between the accuracy of the DVS classification and the length of the sample is shown in Figure 5.5. That is, for each sample in the test set, we measure the accuracy with the events in first 50 ms, first 100, first 150 ms, and the whole 200 ms sample.

5.3. Results Assessment and Discussion

In this section, we will give assessment on the resource utilization, and compare our work with the only existing work that is close to our generator: the RANC [70].

5.3.1. Resource utilization

The resource utilization of the DVS-core and EMG-core are listed in Table 5.1 and Table 5.2.

In a neuromorphic processor, memory always takes an significant part in resource utilization. Therefore, a method to reduce hardware utilization is to reduce the number of neurons and synapses. As we discussed in Section 3.2, our generator can generate user-specified crossbars with configurable memory structure. Table 5.3 is the comparison of memory utilization for our cores and the cores used for the hand-gesture application in [18]. For the DVS-core, our hardware is not as efficient as MorphIC because we

Table 5.1: Resource utilization of DVS-core

Resource	Utilization	Available	Utilization %
LUT	4955	53200	9.31
LUTRAM	62	17400	0.36
FF	9515	106400	8.94
BRAM	96	140	68.67
DSP	4	220	1.82
BUFG	4	32	12.50
MMCM	1	4	25.00

Table 5.2: Resource utilization of EMG-core

Resource	Utilization	Available	Utilization %
LUT	1843	53200	3.46
LUTRAM	62	17400	0.36
FF	3267	106400	3.07
BRAM	2.5	140	1.79
BUFG	4	32	12.50
MMCM	1	4	25.00

use 2-bit weights. For the EMG-core, if we only consider the EMG classification tasks, our hardware uses much less memory.

5.3.2. Reconfigurability

As there are few works that provide hardware reconfigurability, the closest work is RANC [70]. The comparison between our generator and RANC is listed in Table 5.4.

On a hardware level, the configurability of these works are similar. RANC allows configuration on the number of axons per neuron, which can help reduce the memory taken by a core. However, the network parameters of RANC, such as weights, are configured with memory files and are not configurable once implemented on an FPGA. Hence, RANC can only be used as an FPGA platform as the parameters can only be configured by reprogramming the FPGA. Our generator, on the other hand, keeps the configurability of these parameters through an SPI bus and users can configure these parameters post-deployment. This provides the possibility of using our generator for ASIC implementations.

RANC implements a mesh-based network-on-chip architecture, the ”XY” routing scheme allows packet routed to any destination in the network. On the other hand, our generator generates a tree-structured network. Thanks to the high-flexibility of SpinalHDL, our generator can generate heterogeneous cores with different parameters, whereas RANC also supports heterogeneous cores with IP wizard.

On a software level, RANC provides a good workflow from training to simulation and deployment on FPGA. It includes a tick-accurate C++ simulator and the hardware can be implemented on an FPGA with the provided IP wizard. Overall, the software environment is one of the most significant advantages of RANC compared to our work.

In summary, our generator offers more flexibility in generating heterogeneous cores with different parameters. But the lack of a software ecosystem for our generator should be considered as a shortcoming and open for future work.

Table 5.3: Memory size of different SNN processors.

	MorphIC [39]	DVS-core	ODIN [40]	EMG-core
Neuron memory	32KB	5KB	4KB	512B
Synapse memory	256KB	400KB	32KB	8KB

Table 5.4: Comparison between our generator and RANC [70].

	RANC	This work
Configurability		
Number of neurons	YES	YES
Number of axons	YES	NO
Neuron	YES	YES
Synapse	YES	YES
Leakage	YES	YES
On-chip configurability	NO	YES
Network-on-chip		
Structure	mesh-based	tree-structured
Routing scheme	”XY” routing	Arbiter
Heterogeneous cores	YES	YES
Ecosystem		
Software simulation	C++ simulation	Custom setup necessary
FPGA emulation	IP wizard	Manual

6

Conclusion

In summary, we propose a reconfigurable digital neuromorphic hardware generator. Our generator is capable of generating a tree-structured multi-core architecture where each core can be configured with user-defined parameters. This hardware generator is developed using SpinalHDL for high flexibility in hardware generation. We verified our hardware generator with a sensor-fusion hand-gesture recognition application: we first obtained the baseline models for DVS classification, EMG classification, and a sensor-fusion network for hand-gesture classification; then we generated hardware for DVS classification and EMG classification and verified them on a PYNQ board. The results validate our hardware generator in the case of DVS classification. Finally, we compare the memory utilization of different hardware systems for the same application, and the configurability with a similar hardware generator: RANC. We show that our generator offers good configurability for users to generate heterogeneous and resource-efficient hardware that is tailored to specific applications.

The following aspects are left for future work:

- resolution for the mismatch in the accuracy of the EMG-core;
- hardware implementation of the sensor-fusion network;
- a software ecosystem for easy network deployment, including a simulator (possibly using SpinalHDL's simulation API [98]);
- support for more routing schemes and more multi-core architectures (such as the grid network in RANC);
- support for more neuron models (such as the Izhikevich model);
- more configurable parameters, such as the number of physical neurons in a core;
- support for online learning.

References

[1] Beatrice Åkerblom and Tobias Wrigstad. “Measuring Polymorphism in Python Programs”. In: *Proceedings of the 11th Symposium on Dynamic Languages*. DLS 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 114–128. ISBN: 9781450336901. DOI: 10.1145/2816707.2816717. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/2816707.2816717.

[2] Filipp Akopyan et al. “TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (2015), pp. 1537–1557. DOI: 10.1109/TCAD.2015.2474396.

[3] Mary B. Alatise and Gerhard P. Hancke. “A Review on Challenges of Autonomous Mobile Robot and Sensor Fusion Methods”. In: *IEEE Access* 8 (2020), pp. 39830–39846. DOI: 10.1109/ACCESS.2020.2975643.

[4] Md Zahangir Alom et al. “A State-of-the-Art Survey on Deep Learning Theory and Architectures”. In: *Electronics* 8.3 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8030292. URL: https://www.mdpi.com/2079-9292/8/3/292.

[5] Christiaan Baaij et al. “CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell”. In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. 2010, pp. 714–721. DOI: 10.1109/DSD.2010.21.

[6] Christiaan Baaij et al. “Tool Demonstration CLasH From Haskell to Hardware”. In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*. Haskell ’09. Edinburgh, Scotland: Association for Computing Machinery, 2009. ISBN: 9781605585086. DOI: 10.1145/1596638.1667736. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/1596638.1667736.

[7] Jonathan Bachrach. *Chisel*. [Online]. https://jackbackrack.github.io/Projects/chisel/chisel.htm.

[8] Jonathan Bachrach et al. “Chisel: Constructing hardware in a Scala embedded language”. In: *DAC Design Automation Conference 2012*. 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[9] Ben Varkey Benjamin et al. “Neurogrid: A Mixed-Analog-Digital Multichip System for Large-Scale Neural Simulations”. In: *Proceedings of the IEEE* 102.5 (2014), pp. 699–716. DOI: 10.1109/JPROC.2014.2313565.

[10] Chris M Bishop. “Neural networks and their applications”. In: *Review of scientific instruments* 65.6 (1994), pp. 1803–1832.

[11] K.A. Boahen. “Point-to-point connectivity between neuromorphic chips using address events”. In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 47.5 (2000), pp. 416–434. DOI: 10.1109/82.842110.

[12] Janette B. Bradley. “Neural networks: A comprehensive foundation: S. HAYKIN. New York: Macmillan College (IEEE Press Book) (1994). v + 696 pp. ISBN 0-02-352761-7”. In: *Information Processing & Management* 31.5 (1995). Summarizing Text, p. 786. ISSN: 0306-4573. DOI: https://doi.org/10.1016/0306-4573(95)90003-9. URL: https://www.sciencedirect.com/science/article/pii/0306457395900039.

[13] Christian Brandli et al. “A 240 × 180 130 dB 3 µs Latency Global Shutter Spatiotemporal Vision Sensor”. In: *IEEE Journal of Solid-State Circuits* 49.10 (2014), pp. 2333–2341. DOI: 10.1109/JSSC.2014.2342715.

[14] Romain Brette and Wulfram Gerstner. “Adaptive Exponential Integrate-and-Fire Model as an Effective Description of Neuronal Activity”. In: *Journal of Neurophysiology* 94.5 (2005). PMID: 16014787, pp. 3637–3642. DOI: 10.1152/jn.00686.2005. eprint: https://doi.org/10.1152/jn.00686.2005. URL: https://doi.org/10.1152/jn.00686.2005.

[15] Luca Cardelli and Peter Wegner. “On Understanding Types, Data Abstraction, and Polymorphism”. In: *ACM Comput. Surv.* 17.4 (Dec. 1985), pp. 471–523. ISSN: 0360-0300. DOI: 10.1145/6041.6042. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/6041.6042.

[16] Andrew S. Cassidy et al. “Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores”. In: *The 2013 International Joint Conference on Neural Networks (IJCNN)*. 2013, pp. 1–10. DOI: 10.1109/IJCNN.2013.6707077.

[17] Enea Ceolini et al. *EMG and Video Dataset for sensor fusion based hand gestures recognition*. DOI: 10.5281/zenodo.3228845.

[18] Enea Ceolini et al. “Hand-Gesture Recognition Based on EMG and Event-Based Camera Sensor Fusion: A Benchmark in Neuromorphic Computing”. In: *Frontiers in Neuroscience* 14 (2020). ISSN: 1662-453X. DOI: 10.3389/fnins.2020.00637. URL: https://www.frontiersin.org/articles/10.3389/fnins.2020.00637.

[19] Vincent Chan, Shih-Chii Liu, and Andr van Schaik. “AER EAR: A Matched Silicon Cochlea Pair With Address Event Representation Interface”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 54.1 (2007), pp. 48–59. DOI: 10.1109/TCSI.2006.887979.

[20] Jiasi Chen and Xukan Ran. “Deep Learning With Edge Computing: A Review”. In: *Proceedings of the IEEE* 107.8 (2019), pp. 1655–1674. DOI: 10.1109/JPROC.2019.2921977.

[21] *Chisel 3: A Modern Hardware Design Language*. [Online]. https://github.com/chipsalliance/chisel3.

[22] the Chisel/FIRRTL Developers. *Chisel Users Community*. [Online]. https://www.chisel-lang.org/community.html.

[23] the Chisel/FIRRTL Developers. *Chisel/FIRRTL Hardware Compiler Framework*. [Online]. https://www.chisel-lang.org/.

[24] *Clash*. [Online]. https://clash-lang.org/.

[25] *clash-prelude-1.6.4: Clash: a functional hardware description language - Prelude library*. [Online]. https://hackage.haskell.org/package/clash-prelude-1.6.4/docs/Clash-Prelude.html.

[26] Federico Corradi and Giacomo Indiveri. “A Neuromorphic Event-Based Neural Recording System for Smart Brain-Machine-Interfaces”. In: *IEEE Transactions on Biomedical Circuits and Systems* 9.5 (2015), pp. 699–709. DOI: 10.1109/TBCAS.2015.2479256.

[27] Mike Davies et al. “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning”. In: *IEEE Micro* 38.1 (2018), pp. 82–99. DOI: 10.1109/MM.2018.112130359.

[28] Jan Decaluwe. *MyHDL documentation*. [Online]. http://docs.myhdl.org/en/stable/index.html.

[29] Elisa Donati et al. “Discrimination of EMG Signals Using a Neuromorphic Implementation of a Spiking Neural Network”. In: *IEEE Transactions on Biomedical Circuits and Systems* 13.5 (2019), pp. 795–803. DOI: 10.1109/TBCAS.2019.2925454.

[30] Gergő Érdi. *Retrocomputing with Clash: Haskell for FPGA Hardware Design*. 2021.

[31] Jason K Eshraghian et al. “Navigating local minima in quantized spiking neural networks”. In: *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE. 2022, pp. 352–355.

[32] Jason K Eshraghian et al. “Training spiking neural networks using lessons from deep learning”. In: *Proceedings of the IEEE* (2023).

[33] Jason K. Eshraghian. *snnTorch Documentation*. [Online]. https://snntorch.readthedocs.io/en/latest/index.html.

[34] *Examples*. [Online]. https://www.myhdl.org/docs/examples/.

[35] Wei Fang et al. “SpikingJelly: An open-source machine learning infrastructure platform for spike-based intelligence”. In: *Science Advances* 9.40 (2023), eadi1480. DOI: 10.1126/sciadv.adi1480. eprint: https://www.science.org/doi/pdf/10.1126/sciadv.adi1480. URL: https://www.science.org/doi/abs/10.1126/sciadv.adi1480.

[36] Jamil Fayyad et al. “Deep Learning Sensor Fusion for Autonomous Vehicle Perception and Localization: A Review”. In: *Sensors* 20.15 (2020). ISSN: 1424-8220. DOI: 10.3390/s20154220. URL: https://www.mdpi.com/1424-8220/20/15/4220.

[37] Charlotte Frenkel. *tinyODIN*. [Online]. https://github.com/ChFrenkel/tinyODIN.

[38] Charlotte Frenkel, David Bol, and Giacomo Indiveri. “Bottom-Up and Top-Down Approaches for the Design of Neuromorphic Processing Systems: Tradeoffs and Synergies Between Natural and Artificial Intelligence”. In: *Proceedings of the IEEE* 111.6 (2023), pp. 623–652. DOI: 10.1109/JPROC.2023.3273520.

[39] Charlotte Frenkel, Jean-Didier Legat, and David Bol. “MorphIC: A 65-nm 738k-Synapse/mm^2 Quad-Core Binary-Weight Digital Neuromorphic Processor With Stochastic Spike-Driven Online Learning”. In: *IEEE Transactions on Biomedical Circuits and Systems* 13.5 (2019), pp. 999–1010. DOI: 10.1109/TBCAS.2019.2928793.

[40] 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.

[41] Kunihiko Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological cybernetics* 36.4 (1980), pp. 193–202.

[42] Steve Furber. “Large-scale neuromorphic computing systems”. In: *Journal of Neural Engineering* 13.5 (Aug. 2016), p. 051001. DOI: 10.1088/1741-2560/13/5/051001. URL: https://dx.doi.org/10.1088/1741-2560/13/5/051001.

[43] Guillermo Gallego et al. “Event-Based Vision: A Survey”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.1 (2022), pp. 154–180. DOI: 10.1109/TPAMI.2020.3008413.

[44] Wulfram Gerster et al. *Neuronal Dynamics: From single neurons to networks and models of cognition*. Cambrige University Press, 2014.

[45] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference”. In: *arXiv preprint arXiv:2103.13630* (2021).

[46] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[47] 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.

[48] *Haskell to VHDL/Verilog/SystemVerilog compiler*. [Online]. https://github.com/clash-lang/clash-compiler.

[49] A. L. Hodgkin and A. F. Huxley. “A quantitative description of membrane current and its application to conduction and excitation in nerve”. In: *The Journal of Physiology* 117.4 (1952), pp. 500–544. DOI: https://doi.org/10.1113/jphysiol.1952.sp004764. eprint: https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.1952.sp004764. URL: https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.1952.sp004764.

[50] Caroline M. A. Hoeks et al. “Prostate Cancer: Multiparametric MR Imaging for Detection, Localization, and Staging”. In: *Radiology* 261.1 (2011). PMID: 21931141, pp. 46–66. DOI: 10.1148/radiol.11091822. eprint: https://doi.org/10.1148/radiol.11091822. URL: https://doi.org/10.1148/radiol.11091822.

[51] *how to properly create a (synthesized) parity checker function*. [Online]. https://github.com/myhdl/myhdl/issues/378.

[52] Hung-Yi Hsieh and Kea-Tiong Tang. “VLSI Implementation of a Bio-Inspired Olfactory Spiking Neural Network”. In: *IEEE Transactions on Neural Networks and Learning Systems* 23.7 (2012), pp. 1065–1073. DOI: 10.1109/TNNLS.2012.2195329.

[53] Paul Hudak, John Peterson, and Joseph Fasel. *A Gentle Introduction to Haskell 98*. [Online]. https://www.haskell.org/tutorial/haskell-98-tutorial.pdf. 1999.

[54] *Human Brain Project*. [Online]. https://www.humanbrainproject.eu/en/.

[55] "IEEE Standard for Verilog Hardware Description Language". In: *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), pp. 1–590. DOI: 10.1109/IEEESTD.2006.99495.

[56] "IEEE Standard for VHDL Language Reference Manual - Redline". In: *IEEE Std 1076-2019 - Redline* (2019), pp. 1–1118.

[57] E.M. Izhikevich. "Simple model of spiking neurons". In: *IEEE Transactions on Neural Networks* 14.6 (2003), pp. 1569–1572. DOI: 10.1109/TNN.2003.820440.

[58] Adam Izraelevitz et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017, pp. 209–216. DOI: 10.1109/ICCAD.2017.8203780.

[59] A.K. Jain, Jianchang Mao, and K.M. Mohiuddin. "Artificial neural networks: a tutorial". In: *Computer* 29.3 (1996), pp. 31–44. DOI: 10.1109/2.485891.

[60] Renaud Jolivet, Timothy J., and Wulfram Gerstner. "The Spike Response Model: A Framework to Predict Neuronal Spike Trains". In: *Artificial Neural Networks and Neural Information Processing — ICANN/ICONIP 2003*. Ed. by Okyay Kaynak et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 846–853. ISBN: 978-3-540-44989-8.

[61] M. I. Jordan and T. M. Mitchell. "Machine learning: Trends, perspectives, and prospects". In: *Science* 349.6245 (2015), pp. 255–260. DOI: 10.1126/science.aaa8415. eprint: https://www.science.org/doi/pdf/10.1126/science.aaa8415. URL: https://www.science.org/doi/abs/10.1126/science.aaa8415.

[62] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[63] Jack Koenig. *Chisel Breakdown 2*. [Online]. https://docs.google.com/presentation/d/114YihixFBPCfUnv1inqAL8UjsiWfcNWdPHX7SeqlRQc/edit#slide=id.p.

[64] Jack Koenig. *Chisel3 Cheat Sheet*. [Online]. https://inst.eecs.berkeley.edu/~cs250/sp17/handouts/chisel-cheatsheet3.pdf.

[65] Thomas Jacob Koickal et al. "Analog VLSI Circuit Implementation of an Adaptive Neuromorphic Olfaction Chip". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 54.1 (2007), pp. 60–73. DOI: 10.1109/TCSI.2006.888677.

[66] Thomas Jacob Koickal et al. "Design of a spike event coded RGT microphone for neuromorphic auditory systems". In: *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*. 2011, pp. 2465–2468. DOI: 10.1109/ISCAS.2011.5938103.

[67] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. "A 128$\times$ 128 120 dB 15 μs Latency Asynchronous Temporal Contrast Vision Sensor". In: *IEEE Journal of Solid-State Circuits* 43.2 (2008), pp. 566–576. DOI: 10.1109/JSSC.2007.914337.

[68] Shih-Chii Liu et al. "Asynchronous Binaural Spatial Audition Sensor With $2 \times 64 \times 4$ Channel Output". In: *IEEE Transactions on Biomedical Circuits and Systems* 8.4 (2014), pp. 453–464. DOI: 10.1109/TBCAS.2013.2281834.

[69] De Ma et al. "Darwin: A neuromorphic hardware co-processor based on spiking neural networks". In: *Journal of Systems Architecture* 77 (2017), pp. 43–51. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2017.01.003. URL: https://www.sciencedirect.com/science/article/pii/S1383762117300231.

[70] Joshua Mack et al. "RANC: Reconfigurable Architecture for Neuromorphic Computing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.11 (2021), pp. 2265–2278. DOI: 10.1109/TCAD.2020.3038151.

[71] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[72] Saber Moradi et al. "A Scalable Multicore Architecture With Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs)". In: *IEEE Transactions on Biomedical Circuits and Systems* 12.1 (2018), pp. 106–122. DOI: 10.1109/TBCAS.2017.2759700.

[73] Eric Müller et al. *The BrainScaleS-2 Neuromorphic Platform — A Report on the Integration and Operation of an Open Science Hardware Platform within EBRAINS*. Sept. 2023.

[74] K. W. Ng and C. K. Luk. “A survey of languages integrating functional, object-oriented and logic programming”. In: *Microprocessing and Microprogramming* 41.1 (1995), pp. 5–36. ISSN: 0165-6074. DOI: https://doi.org/10.1016/0165-6074(94)00017-5. URL: https://www.sciencedirect.com/science/article/pii/0165607494000175.

[75] Kwan Ting Ng et al. “Characterization of a logarithmic spike timing encoding scheme for a 4×4 tin oxide gas sensor array”. In: *SENSORS, 2009 IEEE*. 2009, pp. 731–734. DOI: 10.1109/ICSENS.2009.5398548.

[76] *Overview*. [Online]. https://www.myhdl.org/start/overview.html.

[77] Eustace Painkras et al. “SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation”. In: *IEEE Journal of Solid-State Circuits* 48.8 (2013), pp. 1943–1953. DOI: 10.1109/JSSC.2013.2259038.

[78] Alessandro Pappalardo. *Xilinx/brevitas*. DOI: 10.5281/zenodo.3333552. URL: https://doi.org/10.5281/zenodo.3333552.

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

[80] Christoph Posch, Daniel Matolin, and Rainer Wohlgenannt. “A QVGA 143 dB Dynamic Range Frame-Free PWM Image Sensor With Lossless Pixel-Level Video Compression and Time-Domain CDS”. In: *IEEE Journal of Solid-State Circuits* 46.1 (2011), pp. 259–275. DOI: 10.1109/JSSC.2010.2085952.

[81] *Python 3.11.0 documentation*. [Online]. https://docs.python.org/3/library/functools.html.

[82] *PyTorch*. [online]. https://pytorch.org/.

[83] Ning Qiao and Giacomo Indiveri. “Scaling mixed-signal neuromorphic processors to 28 nm FD-SOI technologies”. In: *PROCEEDINGS OF 2016 IEEE BIOMEDICAL CIRCUITS AND SYSTEMS CONFERENCE (BIOCAS)*. Biomedical Circuits and Systems Conference. 12th IEEE Biomedical Circuits and Systems Conference (BioCAS), Shanghai, PEOPLES R CHINA, OCT 17-19, 2016. IEEE; IEEE Circuits & Syst Soc; IEEE EMB; IEEE SSC; Shanghai Jiaotong Univ. 2016, pp. 552–555. ISBN: 978-1-5090-2959-4.

[84] Ning Qiao et al. “A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses”. In: *Frontiers in Neuroscience* 9 (2015). ISSN: 1662-453X. DOI: 10.3389/fnins.2015.00141. URL: https://www.frontiersin.org/articles/10.3389/fnins.2015.00141.

[85] *Release news*. [Online]. https://www.myhdl.org/news/releases/.

[86] Garrett S Rose et al. “A system design perspective on neuromorphic computer processors”. In: *Neuromorphic Computing and Engineering* 1.2 (Nov. 2021), p. 022001. DOI: 10.1088/2634-4386/ac24f5. URL: https://dx.doi.org/10.1088/2634-4386/ac24f5.

[87] Kaushik Roy, Akhilesh Jaiswal, and Priyadarshini Panda. “Towards spike-based machine intelligence with neuromorphic computing”. In: *Nature* 575.7784 (2019), pp. 607–617.

[88] Katharina Ruep and Daniel Große. “SpinalFuzz: Coverage-Guided Fuzzing for SpinalHDL Designs”. In: *2022 IEEE European Test Symposium (ETS)*. 2022, pp. 1–4. DOI: 10.1109/ETS54262.2022.9810421.

[89] *Scala based HDL*. [Online]. https://github.com/SpinalHDL/SpinalHDL.

[90] *SCALA BOOK - COMMON SEQUENCE METHODS*. [Online]. https://docs.scala-lang.org/overviews/scala-book/collections-methods.html.

[91] Johannes Schemmel et al. “A Wafer-Scale Neuromorphic Hardware System for Large-Scale Neural Modeling”. In: *2010 IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS*. IEEE International Symposium on Circuits and Systems. International Symposium on Circuits and Systems Nano-Bio Circuit Fabrics and Systems (ISCAS 2010), Paris, FRANCE, MAY 30-JUN 02, 2010. IEEE; CAS; ISEP. 2010, pp. 1947–1950. ISBN: 978-1-4244-5309-2.

[92] Jürgen Schmidhuber. “Deep learning in neural networks: An overview”. In: *Neural Networks* 61 (2015), pp. 85–117. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.003. URL: https://www.sciencedirect.com/science/article/pii/S0893608014002135.

[93] Martin Schoeberl. *Digital Design with Chisel*. Kindle Direct Publishing, 2019. URL: https://github.com/schoeberl/chisel-book.

[94] Pierre Sermanet et al. *OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks*. 2014. arXiv: 1312.6229 [cs.CV].

[95] Moe Shahdad. “An Overview of VHDL Language and Technology”. In: *Proceedings of the 23rd ACM/IEEE Design Automation Conference*. DAC ’86. Las Vegas, Nevada, USA: IEEE Press, 1986, pp. 320–326. ISBN: 0818607025.

[96] Ayan Shymyrbay, Mohammed E. Fouda, and Ahmed Eltawil. “Low Precision Quantization-aware Training in Spiking Neural Networks with Differentiable Quantization Function”. In: *2023 International Joint Conference on Neural Networks (IJCNN)*. 2023, pp. 1–8. DOI: 10.1109/IJCNN54540.2023.10191387.

[97] *SpinalHDL - An alternative to standard HDL*. [Online]. https://spinalhdl.github.io/SpinalDoc-RTD/master/SpinalHDL/Getting%20Started/presentation.html.

[98] *SpinalHDL’s Documentation*. [Online]. https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html.

[99] Yunjae Suh et al. “A 1280×960 Dynamic Vision Sensor with a 4.95-µm Pixel Pitch and Motion Artifact Minimization”. In: *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2020, pp. 1–5. DOI: 10.1109/ISCAS45731.2020.9180436.

[100] Guangzhi Tang et al. “SENECA: building a fully digital neuromorphic processor, design trade-offs and challenges”. In: *Frontiers in Neuroscience* 17 (2023). ISSN: 1662-453X. DOI: 10.3389/fnins.2023.1187252. URL: https://www.frontiersin.org/articles/10.3389/fnins.2023.1187252.

[101] Amirhossein Tavanaei et al. “Deep learning in spiking neural networks”. In: *Neural Networks* 111 (2019), pp. 47–63. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2018.12.002. URL: https://www.sciencedirect.com/science/article/pii/S0893608018303332.

[102] Chetan Singh Thakur et al. “Large-Scale Neuromorphic Spiking Array Processors: A Quest to Mimic the Brain”. In: *Frontiers in Neuroscience* 12 (2018). ISSN: 1662-453X. DOI: 10.3389/fnins.2018.00891. URL: https://www.frontiersin.org/articles/10.3389/fnins.2018.00891.

[103] *The MyHDL development repository*. [Online]. https://github.com/myhdl/myhdl.

[104] Lenny Truong and Pat Hanrahan. “A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity”. In: *3rd Summit on Advances in Programming Languages (SNAPL 2019)*. Ed. by Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi. Vol. 136. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 7:1–7:21. ISBN: 978-3-95977-113-9. DOI: 10.4230/LIPIcs.SNAPL.2019.7. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10550.

[105] Anup Vanarse, Adam Osseiran, and Alexander Rassau. “A Review of Current Neuromorphic Approaches for Vision, Auditory, and Olfactory Sensors”. In: *Frontiers in Neuroscience* 10 (2016). ISSN: 1662-453X. DOI: 10.3389/fnins.2016.00115. URL: https://www.frontiersin.org/articles/10.3389/fnins.2016.00115.

[106] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. “Efficient embedded learning for IoT devices”. In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2016, pp. 308–311. DOI: 10.1109/ASPDAC.2016.7428029.

[107] J.I. Villar et al. “Python as a hardware description language: A case study”. In: *2011 VII Southern Conference on Programmable Logic (SPL)*. 2011, pp. 117–122. DOI: 10.1109/SPL.2011.5782635.

[108] Jiulong Wang et al. “RISC-V toolchain and agile development based open-source neuromorphic processor”. In: *arXiv preprint arXiv:2210.00562* (2022).

[109] Olivia Weng. “Neural network quantization for efficient inference: A survey”. In: *arXiv preprint arXiv:2112.06126* (2021).

[110] Aaron R. Young et al. “A Review of Spiking Neuromorphic Hardware Communication Systems”. In: *IEEE Access* 7 (2019), pp. 135606–135620. DOI: 10.1109/ACCESS.2019.2941772.

[111] Yuanyuan Zhang, Zi-Rui Wang, and Jun Du. “Deep Fusion: An Attention Guided Factorized Bilinear Pooling for Audio-video Emotion Recognition”. In: *2019 International Joint Conference on Neural Networks (IJCNN)*. 2019, pp. 1–8. DOI: 10.1109/IJCNN.2019.8851942.

A

Derivations

A.1. Derivation of the LIF model

In Figure 2.5, we have:

$$I_{in}(t) = I_R(t) + I_C(t) \ , \tag{A.1}$$

where $I_R(t)$ and $I_C(t)$ are the current of resistor R and capacitor C at time t. From Ohm's Law, we have:

$$I_R(t) = \frac{U(T)}{R} \ . \tag{A.2}$$

From the property of capacitors, we can get the charge stored on the capacitor Q:

$$Q = CU(t) \ . \tag{A.3}$$

We also have:

$$\frac{\mathrm{d}Q}{\mathrm{d}t} = I_C(t) = C\frac{\mathrm{d}U}{\mathrm{d}t} \ . \tag{A.4}$$

From Equations A.1, A.2 and A.4, we can get:

$$I_{in}(t) = \frac{U(t)}{R} + C\frac{\mathrm{d}U}{\mathrm{d}t} \Rightarrow RC\frac{\mathrm{d}U}{\mathrm{d}t} = RI_{in}(t) - U(t) \ . \tag{A.5}$$

With $\tau = RC$, Equation A.5 can also be written as:

$$\tau\frac{\mathrm{d}U}{\mathrm{d}t} = RI_{in}(t) - U(t) \ , \tag{A.6}$$

which is the same as Equation 2.6.

A.2. Derivation on getting gradients using BP algorithm

Here, we will give a brief derivation on how to get the gradient w.r.t. the weights in the output layer in Algorithm 2. Assume that:

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{bmatrix}, \ \boldsymbol{a}^{(L)} = \begin{bmatrix} a_1^{(L)} \\ a_2^{(L)} \\ \vdots \\ a_p^{(L)} \end{bmatrix}, \ \boldsymbol{h}^{(L-1)} = \begin{bmatrix} h_1^{(L-1)} \\ h_2^{(L-1)} \\ \vdots \\ h_m^{(L-1)} \end{bmatrix}$$

Our goal is to get:

$$\nabla_{\boldsymbol{W}^{(k)}}\mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_{11}^{(L)}} & \frac{\partial \mathcal{L}}{\partial w_{12}^{(L)}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{1m}^{(L)}} \\ \frac{\partial \mathcal{L}}{\partial w_{21}^{(L)}} & \frac{\partial \mathcal{L}}{\partial w_{22}^{(L)}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{2m}^{(L)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{L}}{\partial w_{p1}^{(L)}} & \frac{\partial \mathcal{L}}{\partial w_{p2}^{(L)}} & \cdots & \frac{\partial \mathcal{L}}{\partial w_{pm}^{(L)}} \end{bmatrix} . \tag{A.7}$$

By applying the chain rule to Equation A.7, we can get:

$$\nabla_{\boldsymbol{W}^{(k)}}\mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial y_1}\frac{\partial y_1}{\partial a_1^{(L)}}\frac{\partial a_1^{(L)}}{\partial w_{11}^{(L)}} & \frac{\partial \mathcal{L}}{\partial y_1}\frac{\partial y_1}{\partial a_1^{(L)}}\frac{\partial a_1^{(L)}}{\partial w_{12}^{(L)}} & \cdots & \frac{\partial \mathcal{L}}{\partial y_1}\frac{\partial y_1}{\partial a_1^{(L)}}\frac{\partial a_1^{(L)}}{\partial w_{1m}^{(L)}} \\ \frac{\partial \mathcal{L}}{\partial y_2}\frac{\partial y_2}{\partial a_2^{(L)}}\frac{\partial a_2^{(L)}}{\partial w_{21}^{(L)}} & \frac{\partial \mathcal{L}}{\partial y_2}\frac{\partial y_2}{\partial a_2^{(L)}}\frac{\partial a_2^{(L)}}{\partial w_{22}^{(L)}} & \cdots & \frac{\partial \mathcal{L}}{\partial y_2}\frac{\partial y_2}{\partial a_2^{(L)}}\frac{\partial a_2^{(L)}}{\partial w_{2m}^{(L)}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathcal{L}}{\partial y_p}\frac{\partial y_p}{\partial a_p^{(L)}}\frac{\partial a_p^{(L)}}{\partial w_{p1}^{(L)}} & \frac{\partial \mathcal{L}}{\partial y_p}\frac{\partial y_p}{\partial a_p^{(L)}}\frac{\partial a_p^{(L)}}{\partial w_{p2}^{(L)}} & \cdots & \frac{\partial \mathcal{L}}{\partial y_p}\frac{\partial y_p}{\partial a_p^{(L)}}\frac{\partial a_p^{(L)}}{\partial w_{pm}^{(L)}} \end{bmatrix} . \tag{A.8}$$

Noticed that

$$\nabla_{\boldsymbol{a}^{(L)}}\mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial y_1}\frac{\partial y_1}{\partial a_1^{(L)}} & \frac{\partial \mathcal{L}}{\partial y_2}\frac{\partial y_2}{\partial a_2^{(L)}} & \cdots & \frac{\partial \mathcal{L}}{\partial y_p}\frac{\partial y_p}{\partial a_p^{(L)}} \end{bmatrix} = \nabla_{\boldsymbol{y}}\mathcal{L} \odot f'\left(\boldsymbol{a}^{(L)}\right) , \tag{A.9}$$

and

$$\boldsymbol{a}^{(L)} = \boldsymbol{b}^{(L)} + \boldsymbol{W}^{(L)}\boldsymbol{h}^{(L-1)} . \tag{A.10}$$

From Equation A.10, we have:

$$h_i^{(L-1)} = \frac{\partial a_j^{(L)}}{\partial w_{ji}^{(L)}},\ j \in \{1, 2, \cdots p\},\ i \in \{1, 2, \cdots m\} . \tag{A.11}$$

From Equation A.8, A.9 and A.11, we can find that

$$\nabla_{\boldsymbol{W}^{(k)}}\mathcal{L} = \left(\nabla_{\boldsymbol{a}^{(L)}}\mathcal{L} \otimes \boldsymbol{h}^{(L-1)}\right)^{\mathsf{T}} . \tag{A.12}$$

Equation A.12 is exactly what Algorithm 2 does in Step 5 when $k = L$. For other layers, the processes are basically the same. From Equation A.11, we can also find that

$$w_{ji}^{(L)} = \frac{\partial a_j^{(L)}}{\partial h_i^{(L-1)}},\ j \in \{1, 2, \cdots p\},\ i \in \{1, 2, \cdots m\} . \tag{A.13}$$

With Equation A.13, we can also derive the expression in Step 6 in Algorithm 2.

B

Details of multi-core generation

B.1. Methods used in the generation process

Method: yield

The Scala keyword *yield* is used together with *for*-expression. It generates a new *List* from the existing *List* which is iterated over in the *for*-expression [90]. For example, in *MultiCore_Gen*, the core components are generated from a *List* of *CoreConfig*s, which is shown in Listing B.1.

Listing B.1: Generation of cores.

```
1    val Cores = for (elements <- coreList) yield new TinyODIN_Gen(elements)
```

In Listing B.1, *coreList* is a *List* of *CoreConfig*s which is passed as a parameter to *MultiCore_Gen*. The *for* -expression iterates each element (*CoreConfig*) in *CoreList* and each element is passed as a parameter to the module *TinyODIN_Gen*, which will generate a core with the parameters specified in the corresponding *CoreConfig*. The generated cores are collected as a new *List*, which is *Cores*.

Method: count

count is a straightforward method that counts the number of elements in a *List* that match the provided condition. For example, in Listing B.2, *externalAERCount* represents the total number of *ArbiterConfig*s with *externalAERin* = *true*.

Listing B.2: Number of elements with *externalAERin* = *true* in *arbiterConfigs*.

```
1  def externalAERCount = arbiterConfigS.count(_.externalAERin == true)
```

Method: filter

The filter method creates a new list containing the element that satisfies the condition provided [90]. For example, in Listing B.3, *inputList* is a new *List* that contains the *CoreConfig*s from *coreList* that have an empty *inputCore*.

Listing B.3: Filter for the *CoreConfig*s with empty *inputCore*

```
1  def inputList = coreList.filter(_.inputCore == Nil)
```

Method: collect

The *collect* method applies a partial function to every element in a *List* and collects the result of this partial function in a new *List* [90]. For example, in Listing B.4, *inputWidthList* is a *List* containing the *AERWidth* of elements in *inputList*. The partial function *case i => i.AERWidth* returns *AERWidth* of *i* where *i* here indicates any element in *inputList*.

Listing B.4: Collection of the *AERWidth*s in *inputList*

```
1  def inputWidthList = inputList.collect{ case i => i.AERWidth }
```

B.2. Multi-core generation example

As an example, a multi-core architecture in Figure B.1 can be generated with the code shown in Listing B.5.

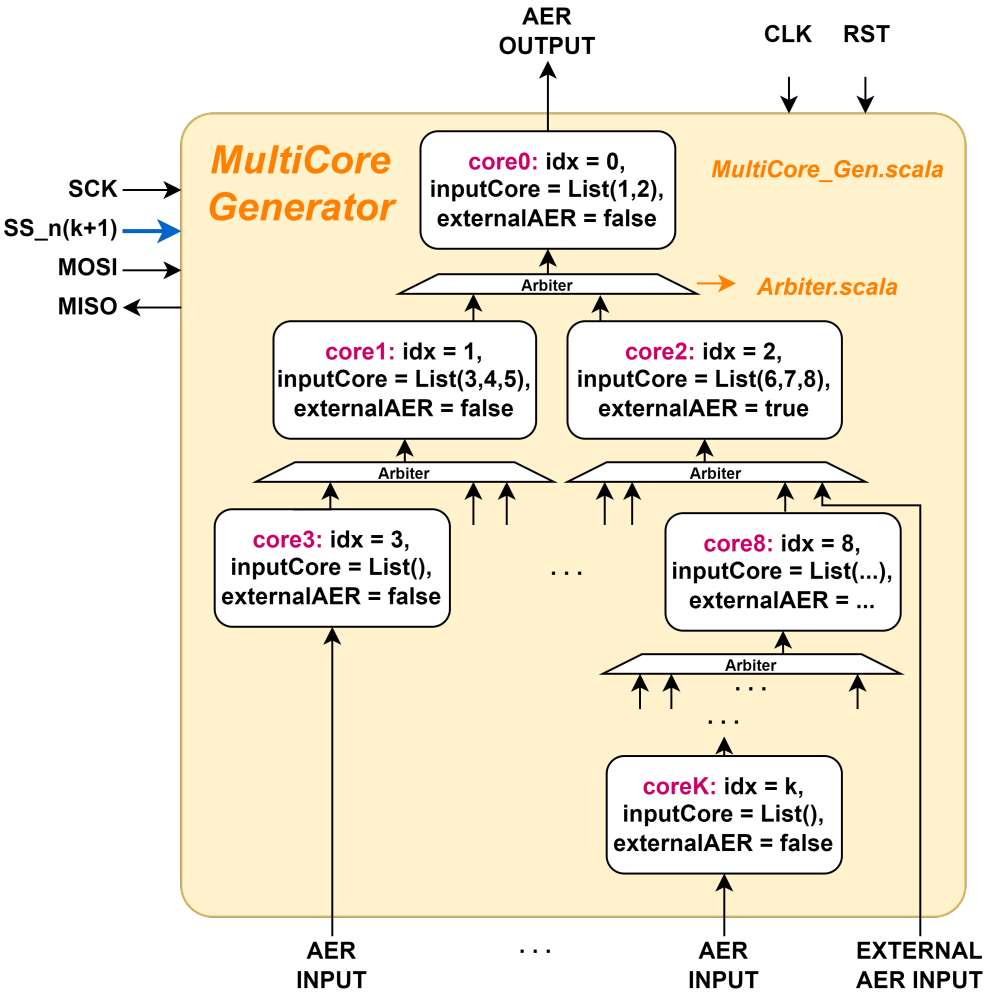

Figure B.1: Example of tree-structured cores.

Listing B.5: Code for defining a multi-core structure in SpinalHDL

```
// Core definitions:

// Core with idx = 0 is the core connected to the AER OUTPUT
val core0 = CoreConfig( ... , idx = 0, inputCore = List(1,2), externalAER = false, ... )
val core1 = CoreConfig( ... , idx = 1, inputCore = List(3,4,5), externalAER = false, ... )
val core2 = CoreConfig( ... , idx = 2, inputCore = List(6,7,8), externalAER = false, ... )
// Core with idx = 3, this core has an empty input list and is connected to the AER INPUT
val core3 = CoreConfig( ... , idx = 3, inputCore = List(), externalAER = false, ... )
//...
// Core with idx = 8, this core has externalAER = true and is connected to an external AER INPUT
val core8 = CoreConfig( ... , idx = 8, inputCore = List...(), externalAER = true, ... )
//...
val coreK = CoreConfig( ... , idx = k, inputCore = List(), externalAER = false, ... )

// Core generation:

Spinal.Config().generateVerilog(new MultiCore_Gen(List(core0, core1, ... , coreK)))
```

B.3. IO definitions and connections

IO interfaces of this architecture (i.e., the *MultiCore_Gen* module) are described in Listing B.6.

Listing B.6: IO definitions of *MultiCore_Gen*

```
  val io = new Bundle {
    // SPI bus
    val SCK = in Bool()
    val MOSI = in Bool()
    val MISO = out Bool()
    // SPI chip-select
    val SS_n = in Vec(Bool(), coreList.length - virtualCount)
    // Output
    val AEROut = out UInt (log2Up(coreList(0).nOut) bits)
    val AEROutAck = in Bool()
    val AEROutReq = out Bool()
    // AER inputs
    val AERInAck = out Vec(Bool(), inputList.length)
    val AERInReq = in Vec(Bool(), inputList.length)
    val AERIn = in Vec(UInt(inputWidthList.max bits), inputList.length)
    // External AER inputs
    val extAERInAck = out Vec(Bool(), externalAERCount)
    val extAERInReq = in Vec(Bool(), externalAERCount)
    val extAERIn = in Vec(UInt(extAERWidthList.max bits), externalAERCount)
  }
```

In Listing B.6, *Bool()* indicates a signal is a 1-bit logic and *UInt* refers to a bit vector operated as an

unsigned integer. Take *AERIn* as an example, it is a vector of *inputList.length* (the length of *inputList*) *UInt* channels, each channel has *inputWidthList.max* (the max value of *inputWidthList*) bits. In this way, the input and output interfaces of the multi-core architecture is parameterized and automatically generated from user configurations.

As *List* in Scala and *Vec* provided by SpinalHDL are iterable collections, we can access a specific element in these collections with a given index. This allows us to easily connect corresponding signals by iterating over the generated cores and arbiters, as well as elemets of *Vec*s in the IO interface. For example, the code in Listing B.7 connects the AER output requests from all arbiters to the AER input requests of the associated cores.

Listing B.7: IO definitions of *MultiCore_Gen*

```
1     for (i <- 0 until (Arbiters.length)) {
2       Arbiters(i).io.reqOut <> Cores(coreConnection(i)._1).io.AERInReq
3       }
```

C

Analysis on EMG Result

C.1. Possible reasons

The main possible reason for the mismatch between the baseline results and the verification result of the EMG classification is the negative affect of event binning in the training process. As mentioned in Section 4.1.1, recordings are binned with 500 time bins. Because of the property of EMG data, it is likely that multiple events are binned into the same bin and passed to the network defined in snnTorch at the same time in a tensor. In the training process, these events are handled at the same time before a neuron is updated. However, in our hardware, a neuron is updated per event. This will cause the different response shown in Figure **??**.

C.2. Possible solutions

To compensate for this mismatch, possible solutions are to:

- increase the number of bins as well as add temporal jitter to the events to make sure that the events are located in different bins;
- introduce a burst mode to the hardware to handle a group of events at the same time.

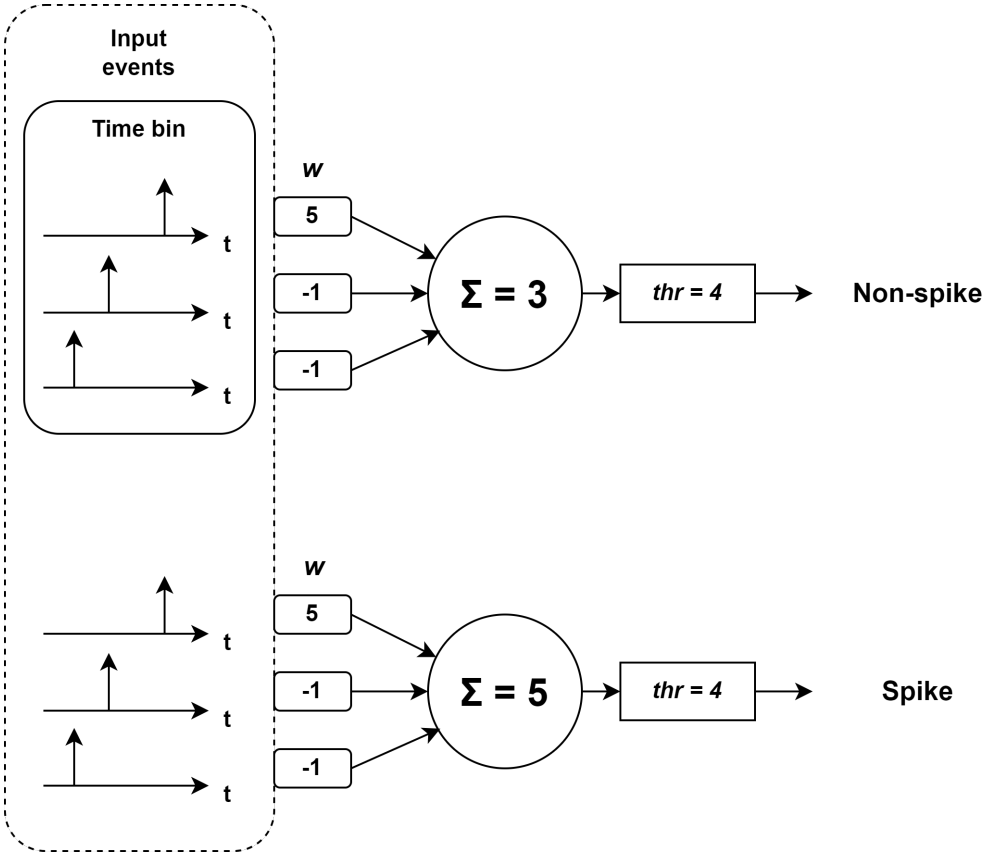

Figure C.1: Different response of binned input and separated inputs. Taking a neuron with an initial membrane potential of 0 and the input synapses shown in the figure as an example. The upper subfigure: The binned events are multiplied and accumulate with the weights and result in a membrane potential of 3 before it is passed to activation. The lower sub-figure: A single event might trigger the activation immediately as it is handled separately and results in a membrane potential of 5. This is the case in our hardware, as the scheduler only schedules a single event at a time. As spikes in DVS recordings are more uniformed distributed than in EMG recordings, and we only take positive spikes in DVS recordings and these spikes are separated to 4 sub-MLPs, it is much less likely that multiple adjacent spikes sent to a single sub-MLP are binned in the same bin. Therefore, it is safe to train the network for DVS classification in this binning manner.