Design and Implementation of FPGA-based Hardware Accelerator for Bayesian Confidence Propagation Neural Network

Smart System/Faculty of Technology
Master's thesis

Author(s):
Deyu Wang

Supervisor(s):
Prof. Zhuo Zou
Prof. Juha Plosila
Dr. Hashem Haghbayan

4.11.2022
Turku

Master's thesis

Subject: Information and Communication Technology
Author(s): Deyu Wang
Title: Design and Implementation of FPGA-based Hardware Accelerator for Bayesian Confidence Propagation Neural Network
Supervisor(s): Prof. Zhuo Zou Prof. Juha Plosila Dr. Hashem Haghbayan

Number of pages: 65 pages
Date: 4.11.2022

The Bayesian confidence propagation neural network (BCPNN) has been widely used for neural computation and machine learning domains. However, the current implementations of BCPNN are not computationally efficient enough, especially in the update of synaptic state variables. This thesis proposes a hardware accelerator for the training and inference process of BCPNN. In the hardware design, several techniques are employed, including a hybrid update mechanism, customized LUT-based design for exponential operations, and optimized design an optimized design that maximizes parallelism. The proposed hardware accelerator is implemented on an FPGA device. The results show that the computing speed of the accelerator can improve the CPU counterpart by two orders of magnitude. In addition, the computational modules of the accelerator can be reused to reduce hardware overheads while achieving comparable computing performance. The accelerator's potential to facilitate the efficient implementation for large-scale BCPNN neural networks opens up the possibility to realize higher-level cognitive phenomena, such as associative memory and working memory.

Key words: Bayesian Confidence Propagation Neural Network (BCPNN), FPGA, Training, Inference.

Contents

Figures

Tables

1 Introduction

In recent years, artificial neural networks (ANNs) have made swift and remarkable progress in various practical applications, in speech recognition[1], image classification[2], and natural language processing[3]. While artificial neural netwroks have gained great popularity and achieved remarkable success in the data-driven computing paradigm, they also have some limitations. Firstly, most existing artificial neural networks rely on supervised learning, which necessitates vast amounts of labeled training data, in contrast to the unsupervised and reward-modulated learning mechanisms employed by biological brains. Secondly, the predominant learning algorithm employed by artificial neural networks, error backpropagation, demands high precision and lacks robustness and biological plausibility. Thirdly, mainstream artificial neural network models do not incorporate human cognition or functions that inspire artificial intelligence.

As a contrast, the human brain is a very complex system consisting of approximately 90 billion neurons [4]. It is structurally composed of trillions of interconnected synapses. Information is passed between neurons through electrical impulses called spikes. The effect of spiking sent by a presynaptic neuron to a receiving neuron depends on the strength of the synapse connecting the two neurons. Synaptic strength and connection patterns between neurons play an important role in the information processing capacity of the nervous system. Inspired by the human brain, the spiking neural network (SNN) is proposed [5]. In contrast to classical artificial neural networks that utilize non-spiking units, spiking neural networks employ the same event-based communication mechanism utilized by the human brain, where neurons communicate via spikes.

As computational neuroscience has progressed, extensive neuromorphic models have been developed to simulate particular brain functions. Neuromorphic computing is considered a promising approach to achieve artificial general intelligence (AGI) due to its brain-like features. Among the brain-like cognitive models constructed, the spiking neural network is regarded as the neural network model that closely mimics the

brain's mechanism. Compared with the artificial neural network (ANN), SNN possess distinct benefits in various aspects such as response latency and power consumption, and is more biologically interpretable. A variety of computational models have been developed, taking into account biological experimental evidence and constraints across various scales. [6]–[8]. These models usually adopt learning rules that mimic behaviours of biological neurons, i.e. spike-timing-dependent plasticity (STDP) and other Hebbian-based learning rules [9]. SNNs are considered a crucial foundation for realizing more intricate cognitive functions of the brain, including memory, concepts, and structured knowledge.

In a recent study, a working memory (WM) theory was implemented in an SNN model, showcasing how WM emerges through the close collaboration of short-term memory (STM) and long-term memory (LTM) [10]. The model employed is a scalable spiking neural network known as the Bayesian Confidence Propagation Neural Network (BCPNN) [11], which which can realize Bayesian statistics using either spiking or non-spiking neural networks. The simulations of BCPNN (with 29 million spiking units and 295 billion plastic connections) were conducted in a massively parallel manner on supercomputers. [12]. Nevertheless, the further expansion of BCPNN is frequently limited by computational capabilities. To put it in perspective, the human brain has over 86 billion neurons and 100 trillion synapses [13], necessitating significant financial resources and running costs to simulate such a vast scale of the brain on supercomputers. Although GPU is a good alternative platform for SNN training [14], [15], it can not handle spike communication and processing well in real-time. Besides, GPU suffers from high power consumption, which also limits its application in SNN. The development of ASIC chips [16]–[18] can offer superior performance and energy efficiency, but the disadvantages are high fabrication costs and restricted flexibility. A compromised solution is an FPGA-based design [19]–[21], which provides reasonable cost, low power consumption, and reconfigurability for the acceleration of neuromorphic computing.

The thesis is divided into the following sections. First, Chapter 2 introduces some basics, including the spiking neural network (SNN), the Bayesian confidence propaga-

tion neural network (BCPNN), and the current applications and implementations of BCPNN. Then chapter 3 presents the architecture of the FPGA-based hardware accelerator. The optimization methods employed in the design are also described. After that, Chapter 4 reports the experimental results of the FPGA-based hardware accelerator. Chapter 5 evaluates the hardware design and discusses the resource-saving scheme through module reuse. Finally, Chapter 6 concludes the thesis.

2 Fundamentals

In this chapter a broad overview of spiking neural networks is given, as well as the Bayesian confidence propagation neural network and its current applications and implementations.

2.1 Spiking Neural Network

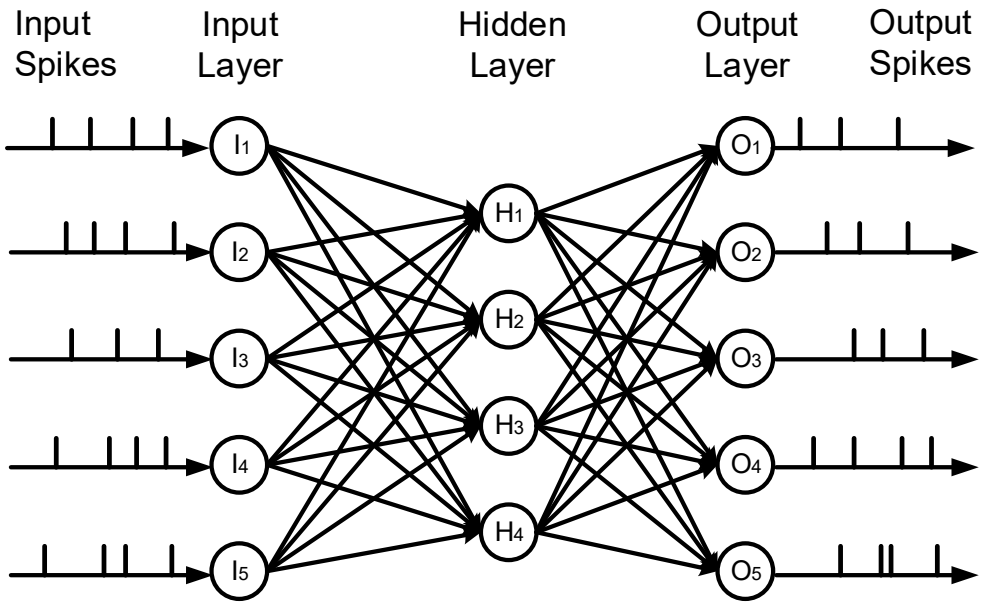


Figure 2.1: The structure of a typical spiking neural network.

Spiking neural networks are inspired by information processing in biology and are considered to be the third-generation neural networks [22]. A spiking neural network is composed of spiking neurons and interconnecting synapses that are modeled by adjustable scalar weights.

Fig. 2.1 demonstrates a typical case of the spiking neural network. The spiking neural network is composed of an input layer, several hidden layers and an output layer. In SNNs, the input data is binary spiking message, which is different from ANNs. Besides, the neurons and synapses of SNNs are different from that of the traditional ANNs.

2.1.1 Network Topology

The topology of spiking neural networks is usually classified into three types, the feedforward, recurrent, and hybrid networks [23].

The feedforward neural network is the network where the connections between neurons do not form a cycle. The information moves in only one direction, from the input layer, through the hidden layer and to the output layer. There is no cycle in such a network topology. A typical case of the feedforward network is visualized in Fig. 2.2.

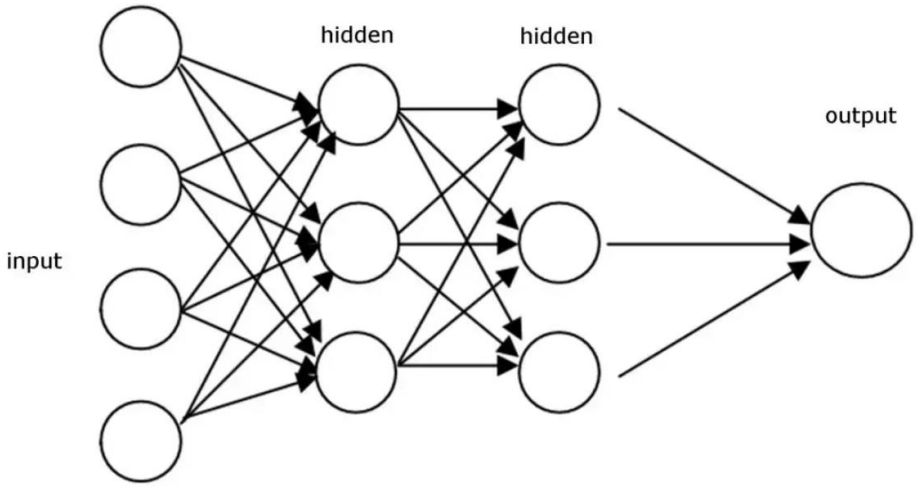


Figure 2.2: The network topology of a typical feedforward neural network.

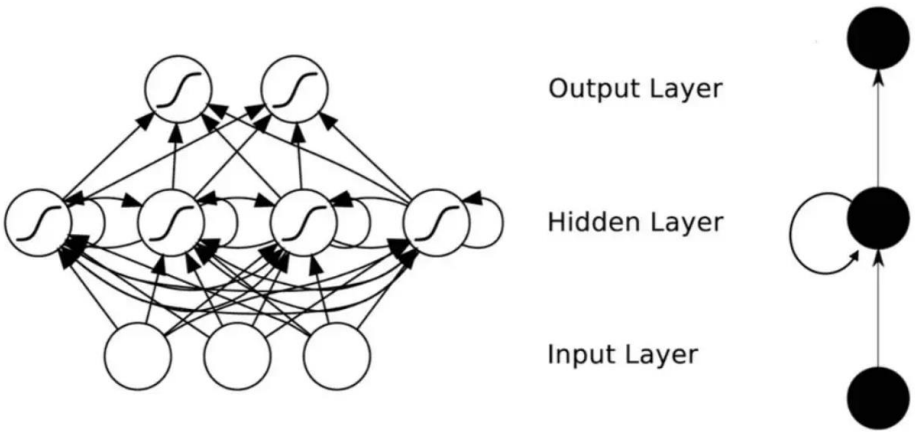


Figure 2.3: The network topology of a typical recurrent neural network.

The recurrent neural network (RNN) is a class of network in which connections between neurons can create a loop, allowing the output of some neurons to affect subsequent inputs of the same neuron. This allows the neural network to exhibit temporal dynamic behavior. A typical case of the recurrent network is visualized in Fig. 2.3.

2.1.2 Neuron Model

Neurons are the basic processing units of the biological brain. The neurons communicate with each other by sending and receiving action potentials [24].

In 1952, Hodgkin and Huxley experimented with giant axons in squid and developed a four-dimensional (4D) detailed neuron model that could reproduce electrophysiological measurements [25]. However, this neuron model has relatively high computational complexity, which increases the computational cost. Hence, simpler phenomenological spiking neuron models are used to model large-scale SNNs, neural coding, and memory. The Leaky Integrate-and-Fire (LIF) model [26] and the Spike Response Model (SRM) [27] are two popular low computational cost 1D spiking neural models, but they are less biologically plausible than the Hodgkin and Huxley models. Izhikevich's 2D model [28] provides a good trade-off between biological plausibility and computational efficiency. Although it can generate a variety of spiking dynamics, many of these properties, such as chaos and bistability, are not yet used in current learning algorithms.

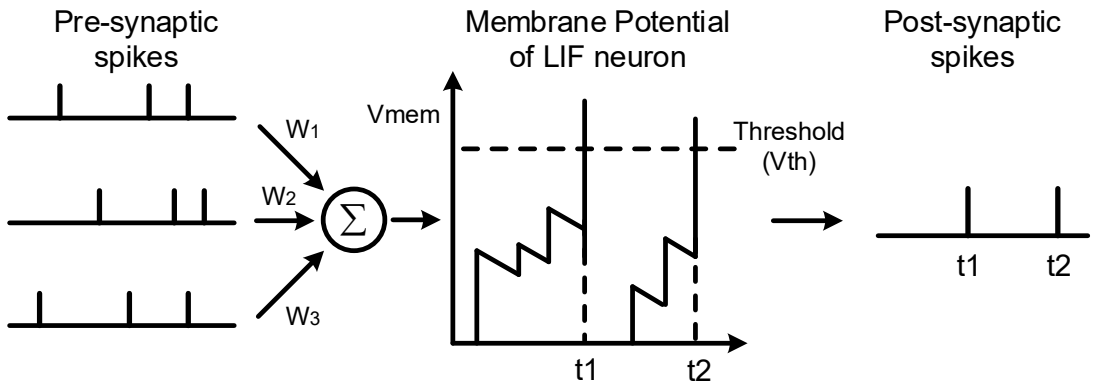


Figure 2.4: The leaky-integrate and fire (LIF) neuron model.

Here we take the most commonly used LIF model as an example, as shown in Fig. 2.4. The neurons integrate the input spiking signals, and once the membrane potential of neurons reaches a certain threshold, the neurons send out a spike.

2.1.3 Synaptic Plasticity

In the nervous system, a synapse is a structure that allows a neuron (or nerve cell) to transmit electrical or chemical signals to another neuron. Fig. 2.5 describes the structure of a synapse. Dendrites are projections of neurons (nerve cells) that receive signals (information) from other neurons. The transmission of information from one neuron to another takes place through chemical signals and electrical impulses, known as electrochemical signals. Information transmission is usually received by chemical signals at the dendrites, then travels to the cell body (cell body), continues along the neuron's axon as an electrical impulse, and finally passes to the next neuron at the synapse, which is the place where two nerves Elements exchange information through chemical signals. At a synapse, the end of one neuron meets the beginning of another—the dendrites.

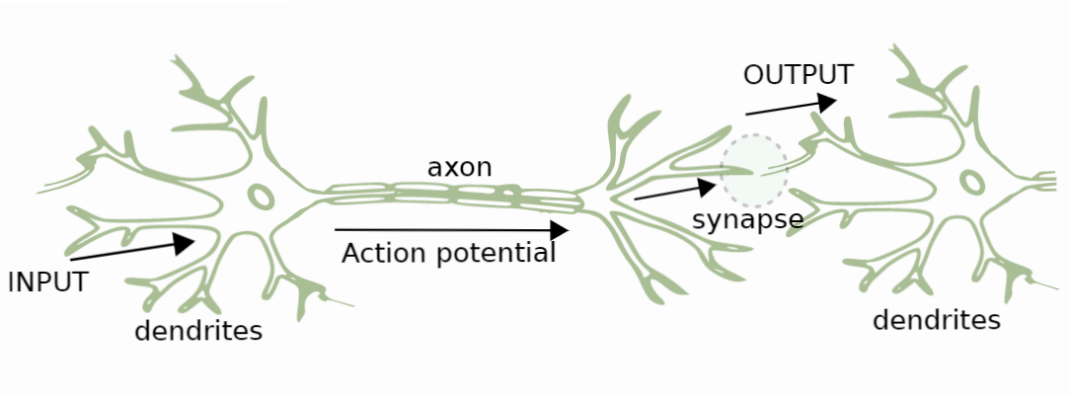


Figure 2.5: The structure of a biological synapse [29].

The synaptic learning of spiking neural networks mimics the biological synaptic plasticity. The learning process of SNNs is quite different from the global back-propagation algorithm adopted in ANNs. The learning of SNNs mainly adopts Hebbian-based learning rules, which means in short, that neurons that fire together, wire together. The spike-timing-dependent plasticity (STDP) learning rule is a widely-used Hebbian

synaptic learning rule whose weight update depends on the relative timing of pre- and post-synaptic spikes [9], as shown in Fig. 2.6. Unlike the global back-propagation algorithm in ANNs, the STDP learning rule is localized and unsupervised and does not require labeled data.

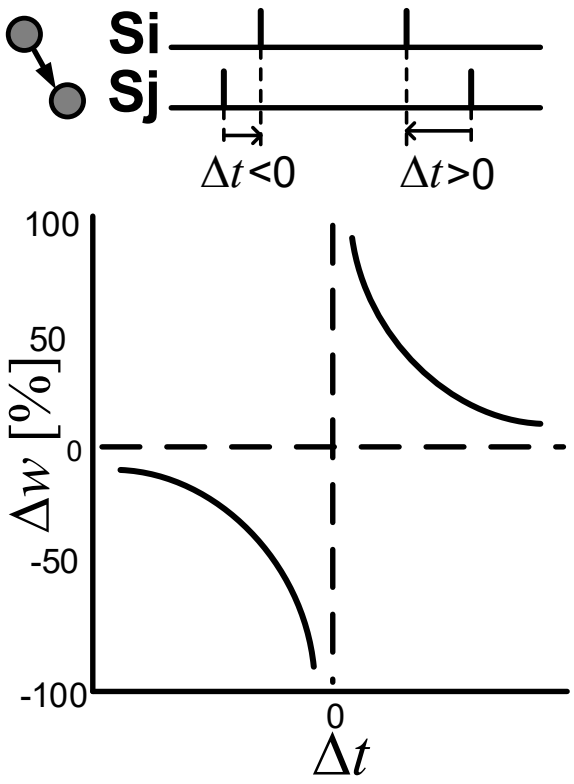


Figure 2.6: The spike-timing-dependent plasticity (STDP) learning rule.

2.2 Bayesian Confidence Propagation Neural Network (BCPNN)

The BCPNN, a neural network, initially drew inspiration from Bayesian inference principles [11], [30]. It evolved into a design patterned after the mammalian cortex's modular structure, utilizing hypercolumn units (HCUs) and minicolumn units (MCUs). Later, the BCPNN was incorporated into SNNs, enabling the mapping of neural and synaptic mechanisms in the human cortex [31]. Compared to other SNN models, BCPNN's hierarchical and coarse-grained architecture is a compact and practical option for large-scale neural network deployment [32].

2.2.1 Network Topology

The hypercolumn, also known as the cortical column, is a collection of neurons in the cerebral cortex of the biological brain that can be penetrated successively by a probe perpendicular to the cortical surface, with nearly identical receptive fields. Each hypercolumn comprises approximately one hundred minicolumns, with neurons within the same minicolumn possessing similar attributes. Figure 2.7 depicts the configuration of the mammalian cortex.

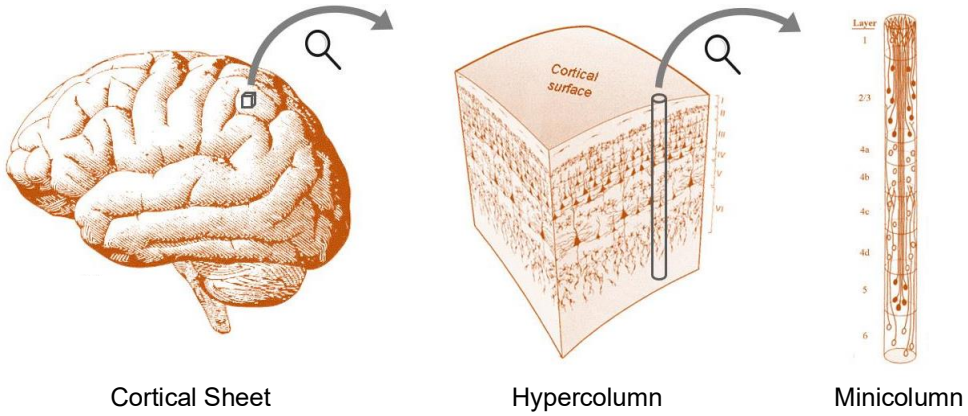


Figure 2.7: The organization of the mammalian cortex [33].

The BCPNN follows a modular design pattern using HCUs and MCUs, which are based on the generalization of the mammalian cortex's structure initially proposed by Hubel and Wiesel [34]. In models of the mammalian cortex, an HCU module is roughly 500 μm in diameter and comprises about 100 MCUs, each with a diameter of 50 µm. Each MCU consists of roughly 100 neurons, primarily excitatory pyramidal cells and one or two local inhibitory double bouquet cells [35]. Lateral inhibition is governed by inhibitory basket cells, which regulate activity within an HCU. In abstract models, this is achieved through softmax, which standardizes the total HCU activity (i.e., the sum of the corresponding MCU activities) to 1. The human cortex is estimated to contain approximately two million HCUs. The modular, hierarchical architecture of the BCPNN is an efficient and concise approach to building large-scale neural networks.

The BCPNN network is composed of H HCUs and M MCUs arranged in an $H \times M$ pattern. Typically, H is much larger than M, with no maximum limit on the number of HCUs. However, biological evidence limits the number of MCUs to approximately 100. Thus, in extensive networks, the number of HCUs is generally higher. In small networks, every MCU can connect entirely to its local HCU and other HCUs, as depicted in Fig.2.8. However, such full connectivity is not feasible in large networks due to the substantial computational and storage costs. Instead, sparse, patchy connectivity influenced by the cortex's structure is implemented, which reduces the number of connections while maintaining proper functionality [36].

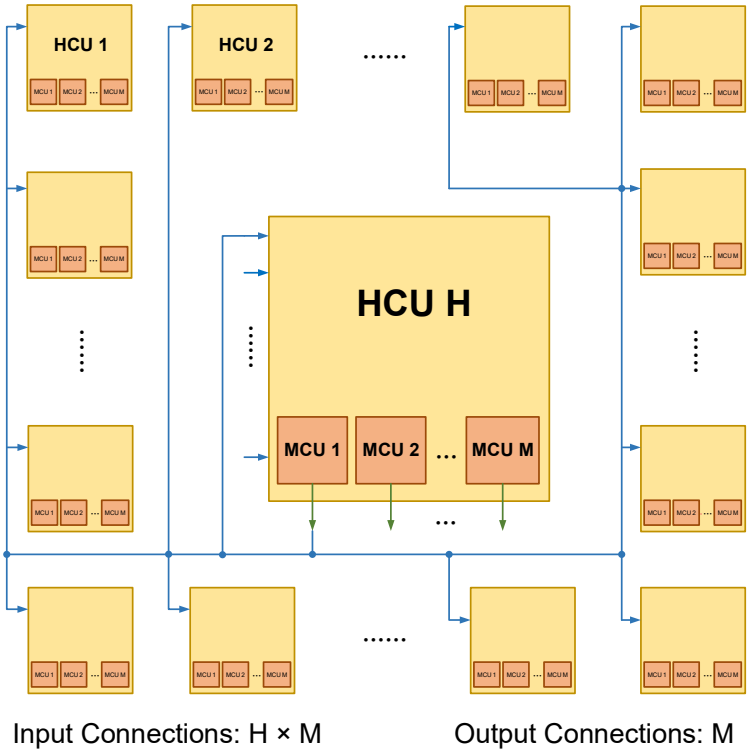


Figure 2.8: The structure of the bayesian confidence propagation neural network (BCPNN).

The HCU structure, illustrated in Fig.2.9, consists of four elements: 1) the presynaptic vector that stores presynaptic traces, including Z_i, E_i, and P_i; 2) the postsynaptic vector that stores postsynaptic traces, such as Z_j, E_j, P_j, and the bias β_j; 3) the synaptic matrix that stores synaptic traces like E_{ij}, P_{ij}, and the weight w_{ij}; and 4) a specific number of MCUs that integrate incoming spiking activities and fire in a soft winner-take-all manner.

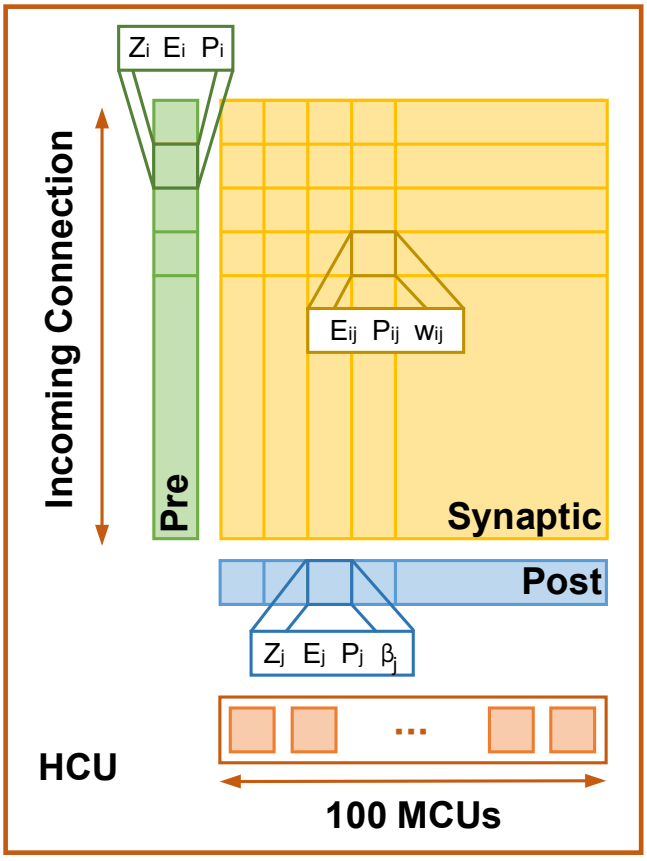


Figure 2.9: The structure of the hypercolumn unit (HCU).

At a higher level, HCUs function as independent network modules that exchange spikes with one another. The size of an HCU is determined by the number of incoming connections and MCUs it has. Due to biological limitations, the maximum number of incoming connections and MCUs is restricted to 10,000 and 100, respectively. Therefore, in an HCU of maximum size, a synaptic matrix with a dimension of 10000×100 is necessary, representing one million plastic synapses.

2.2.2 BCPNN Learning Rule

The BCPNN learning rule differs from the commonly used Spike Timing Dependent Plasticity (STDP) learning rule, as it is derived from Bayes' rule, and it assumes independence between neural activities. This probabilistic inference approach results in a unique neural activation function achieved through a transformation to log-space [11], [30], [37]. Unlike other Hebbian learning rules, in which synaptic updates are driven by co-activation between the pre- and post-synaptic neural units, the BCPNN

learning rule generates positive weights when neural activity is positively correlated, zero weights when they are uncorrelated, and negative weights when they are anti-correlated. Moreover, the BCPNN learning rule accounts for the prior activation of each neural unit, as observed experimentally [31], by including an intrinsic bias. The estimation of network unit activation and co-activation is performed using a cascade of three exponential running averages, which are illustrated in Fig.2.10, and the dynamics of the three types of traces are shown in Fig. 2.11.

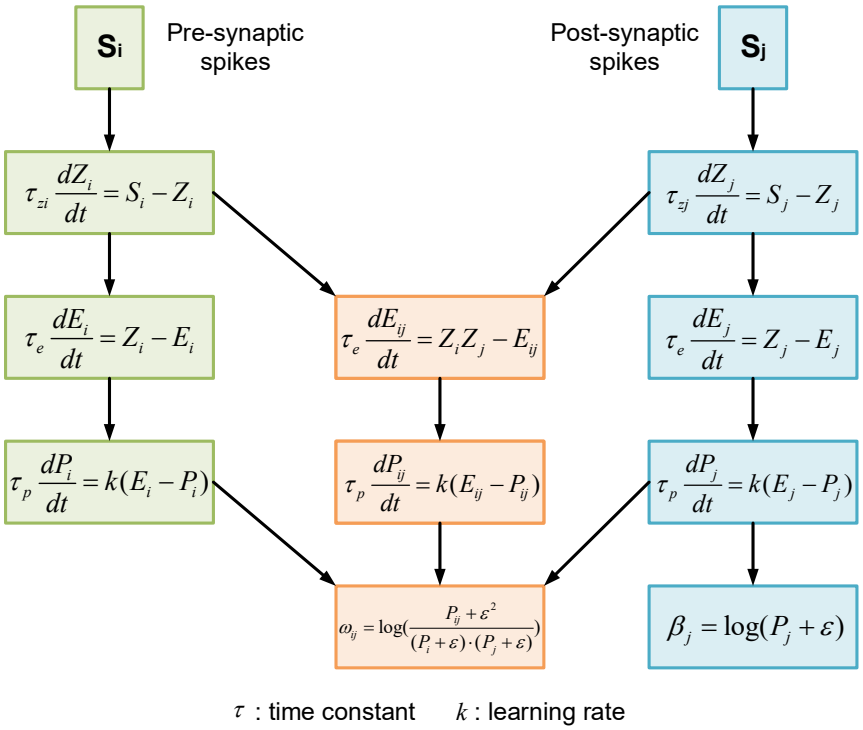


Figure 2.10: The BCPNN learning rule.

Initially, the pre- and post-synaptic Z-traces are activated by incoming spikes:

$$\frac{dZ_i}{dt} = \frac{S_i - Z_i}{\tau_{zi}} \qquad \frac{dZ_j}{dt} = \frac{S_j - Z_j}{\tau_{zj}} \tag{2.1}$$

Here, the symbol i refers to pre-synaptic variables, j refers to post-synaptic variables, and S represents incoming and generated spiking activity. The Z-traces, which are

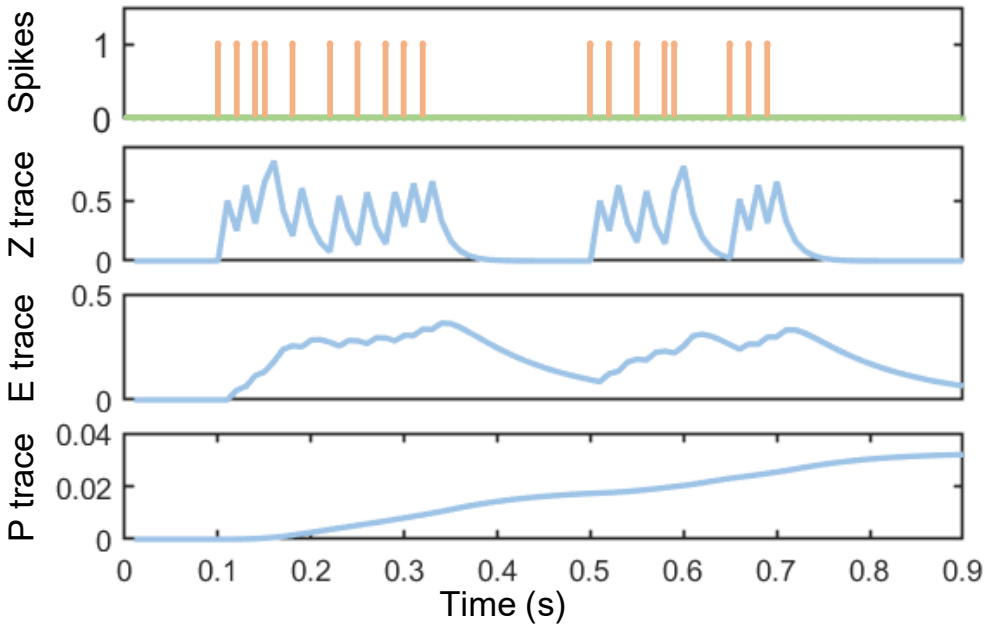


Figure 2.11: The dynamics of three kinds of traces in the BCPNN learning rule.

driven by the spiking activity, then influence the E-traces and P-traces through a similar dynamic process, but with distinct time constants:

$$\frac{dE_i}{dt} = \frac{Z_i - E_i}{\tau_e} \qquad \frac{dE_j}{dt} = \frac{Z_j - E_j}{\tau_e} \qquad \frac{dE_{ij}}{dt} = \frac{Z_i Z_j - E_{ij}}{\tau_e} \tag{2.2}$$

$$\frac{dP_i}{dt} = \frac{(E_i - P_i)}{\tau_p}\kappa \qquad \frac{dP_j}{dt} = \frac{(E_j - P_j)}{\tau_p}\kappa \qquad \frac{dP_{ij}}{dt} = \frac{(E_{ij} - P_{ij})}{\tau_p}\kappa \tag{2.3}$$

The learning process is modulated by the learning rate κ in the dynamics of P-traces. Ultimately, network unit biases and weights are updated using the P-traces, as shown in equation (2.4), with an additional parameter ε that accounts for the minimum spiking activity assumed for the pre- and postsynaptic units:

$$\beta_j = \log(P_j + \varepsilon) \qquad W_{ij} = \log\left(\frac{P_{ij} + \varepsilon^2}{(P_i + \varepsilon) \cdot (P_j + \varepsilon)}\right) \tag{2.4}$$

2.3 Application and Implementation of BCPNN

The BCPNN model has been utilized in various neural computation applications, including modeling synaptic plasticity, such as long-term potentiation (LTP) and long-term depression (LTD). One prominent application of BCPNN is in the creation of scalable self-organizing associative memory [38]. Recent advances in cortical associative memory models have focused on using BCPNN as a framework for rapid cortical synaptic plasticity in synaptic working memory [10], [39]. In these models, positive BCPNN weights serve as excitatory connections between pyramidal cells, while negative weights inhibit distant pyramidal cells through disynaptic connections, such as those mediated by double bouquet cells. Although these spiking neural network (SNN) models are much smaller than their biological counterparts, typically composed of up to a thousand MCUs partitioned into around thirty HCUs, they represent a promising direction in the pursuit of artificial general intelligence (AGI) by emulating various aspects of human cognitive function within a system based on brain-like BCPNN.

Furthermore, the BCPNN model has been applied in the field of machine learning, particularly in building cortex-inspired neural networks for pattern recognition tasks. A recent breakthrough in this area involves the integration of a novel structural plasticity algorithm inspired by the brain, which constructs a hidden layer in an unsupervised manner using the original synaptic trace variables of BCPNN [40], [41]. These models have achieved competitive classification performance on benchmark datasets like MNIST and Fashion-MNIST, attaining test set accuracies of 98.6% and 88.9% respectively [41]. The unsupervised nature of the structural plasticity algorithm permits efficient use of unlabeled training examples, enabling semi-supervised learning that yields promising results even when only 10-1000 labeled training samples are available [42]. These advances in simulating cognitive function with BCPNN-based neural networks offer hope for the development of artificial general intelligence.

The BCPNN model has been deployed in a range of software and hardware systems, including supercomputer clusters and GPUs. Custom hardware implementations utilizing 3D integration of DRAM for synaptic weights have been developed [43]–[46].

Due to the ability to execute the BCPNN learning rule with low precision [47], cortical memory models based on BCPNN have demonstrated robustness and resilience to both external and internal noise, as well as imprecision in weights and unit biases. Therefore, it has potential as a highly scalable, modular, and hardware-friendly neuromorphic architecture suitable for compact and low-power digital or mixed-signal design.

The most demanding computational tasks in the BCPNN implementation are updating the synaptic and MCU internal state variables during training and inference processes. To enhance computation efficiency while retaining programming flexibility, this thesis presents a hardware accelerator for BCPNN, implemented on FPGA.

3 Architecture and Design

This chapter first introduces the two computationally intensive parts of BCPNN, including the training process where the synaptic state variables are updated, and the inference process where the internal state variables of minicolumn units (MCUs) are updated. Especially, a lazy-update method is employed in the update of synaptic state variables. To accelerate these two computationally intensive parts, an FPGA-based hardware accelerator is architected and designed. The hardware accelerator can work in two modes, the training mode and the inference mode. In the hardware design, several optimization methods are employed, including a hybrid update mechanism, a LUT-based block memory for exponential operations, and the optimization technique by maximizing parallelism.

3.1 The training process and inference process of BCPNN

3.1.1 The training process

In the spike-based BCPNN, the strength of the connections between MCUs is modulated based on the simultaneous activation of pre- and post-synaptic neural units. A correlated pre- and post-synaptic activity gives a positive weight, while an anti-correlation gives a negative weight. The probability of pre-, post-synaptic and synaptic activities are tracked with eight local synaptic state variables called traces. These traces are updated in a cascaded manner. The training process can be seen in Figure 2.10, where S_i is the presynaptic spike train, S_j is the postsynaptic spike train, Z_i, Z_j, E_i, E_j, P_i, P_j, E_{ij}, P_{ij} are synaptic traces, τ_{z_i}, τ_{z_j}, τ_e, τ_p are time constants, and k is the learning rate.

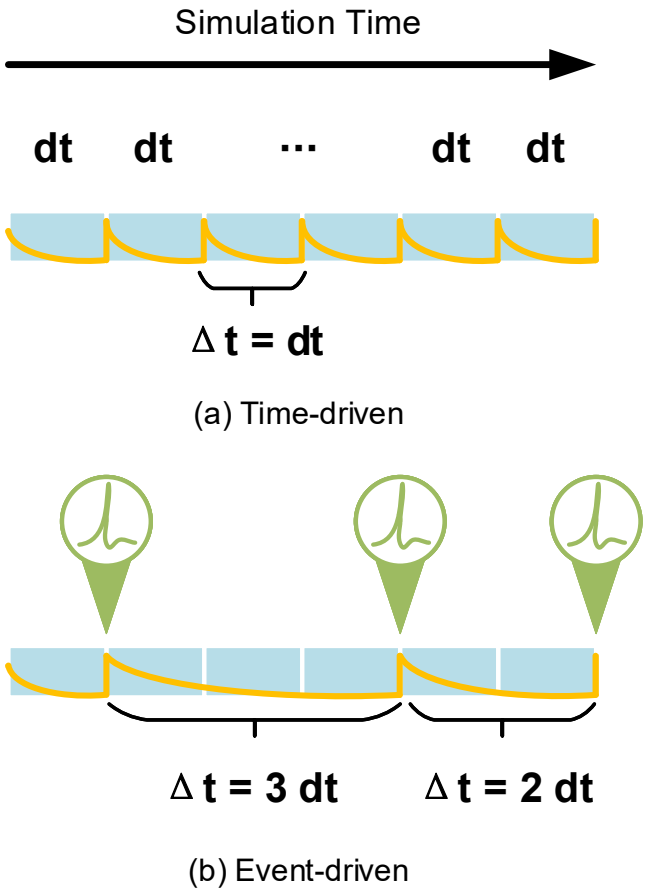


Figure 3.1: The hybrid update mechanism.

As depicted in Figure 2.10, the pre- and post-synaptic spikes are first filtered into Z, E, and P traces with specific time constants. Three P traces (P_i, P_j, and P_{ij}) are utilized to calculate the post-synaptic bias β_j and weight w_{ij}.

In the simulation of spike-based BCPNN, the time-driven simulation with a fixed step size is a typical approach. A time-driven simulation employing the explicit Euler method with a relatively long time step of dt = 1 ms is presented in [44]. However, this approach demands more computational resources and frequent memory access compared with the event-driven manner. As shown in Figure 3.1, in a time-driven manner, the state variable is updated at each time step, and the update interval Δt is always equal to the simulation time step dt. While in an event-driven manner, the state variable is only updated when an event (a spike) happens, and the update interval Δt can be the multiples of time step dt.

To alleviate the computation intensity of the training process, a method called the lazy-update method, which combines both time-driven and event-driven approaches, was proposed in [47]. With this method, the post-synaptic state variables Z_j, E_j, P_j, and β_j are updated every time step dt, while the other synaptic state variables are only updated when a pre- or post-synaptic event occurs.

For the presynaptic traces Z_i, E_i and P_i, the lazy-update solutions are as follows:

$$Z_i(t) = Z_i(t^{last}) \cdot e^{-\frac{\Delta t}{\tau_{zi}}} + S_i(t) \tag{3.1}$$

$$E_i(t) = E_i(t^{last}) \cdot e^{-\frac{\Delta t}{\tau_e}} + Z_i(t^{last}) a_i (e^{-\frac{\Delta t}{\tau_{zi}}} - e^{-\frac{\Delta t}{\tau_e}}) \tag{3.2}$$

$$\begin{aligned} P_i(t) &= P_i(t^{last}) \cdot e^{-\frac{\Delta t}{\tau_p^*}} + a_i b_i (e^{-\frac{\Delta t}{\tau_{zi}}} - e^{-\frac{\Delta t}{\tau_p^*}}) Z_i(t^{last}) \\ &+ (E_i(t^{last}) - a_i Z_i(t^{last})) c (e^{-\frac{\Delta t}{\tau_e}} - e^{-\frac{\Delta t}{\tau_p^*}}) \end{aligned} \tag{3.3}$$

Here, the symbol t^{last} represents the time when each state variable was last updated, while $\Delta t = t - t^{last}$ is the time elapsed since the last update until the current time t. The coefficients a_i, b_i, and c are used in the update equations. To obtain the update formulas for the postsynaptic traces Z_j, E_j, and P_j, the indices i in the presynaptic update formulas (3.1, 3.2, 3.3) are replaced by j.

For the synaptic traces E_{ij} and P_{ij}, the lazy-update solutions are as follows, where a_{ij}, b_{ij}, and c are constants:

$$\begin{aligned} E_{ij}(t) &= E_{ij}(t^{last}) \cdot e^{-\frac{\Delta t}{\tau_e}} \\ &+ Z_i(t^{last}) Z_j(t^{last}) a_{ij} (e^{-\frac{\Delta t}{\tau_{z_{ij}}}} - e^{-\frac{\Delta t}{\tau_e}}) \end{aligned} \tag{3.4}$$

$$\begin{aligned} P_{ij}(t) &= P_{ij}(t^{last}) \cdot e^{-\frac{\Delta t}{\tau_p^*}} \\ &+ a_{ij} b_{ij} (e^{-\frac{\Delta t}{\tau_{z_{ij}}}} - e^{-\frac{\Delta t}{\tau_p^*}}) Z_i(t^{last}) Z_j(t^{last}) \\ &+ (E_{ij}(t^{last}) - a_{ij} Z_i(t^{last}) Z_j(t^{last})) c (e^{-\frac{\Delta t}{\tau_e}} - e^{-\frac{\Delta t}{\tau_p^*}}) \end{aligned} \tag{3.5}$$

3.1.2 The inference process

With the bias β_j and weight w_{ij} obtained in the above training process, the activation o_j and firing probability r_j of each MCU can be calculated in the following inference process [47]. Firstly, the pre-synaptic spike S_i leads to a synaptic current $s_{syn,j}$:

$$\tau_{zi}\frac{ds_{syn,j}(t)}{dt} = \sum_i w_{ij}(t)S_i(t) - s_{syn,j}(t) \tag{3.6}$$

Then the synaptic current $s_{syn,j}$ is accumulated with the bias β_j and an external input I_j to obtain the support value s_j:

$$s_j(t) = \beta_j(t) + s_{syn,j}(t) + I_j(t) \tag{3.7}$$

The support value s_j is low-pass filtered to get the membrane potential m_j:

$$\tau_m\frac{dm_j(t)}{dt} = s_j(t) - m_j(t) \tag{3.8}$$

Then the activation o_j of each MCU is calculated as:

$$o_j = \begin{cases} \frac{e^{\gamma_m m_j}}{\sum_{k=1}^{M} e^{\gamma_m m_k}}, if \sum_{k=1}^{m} e^{\gamma_m m_k} > 1 \\ \\ e^{\gamma_m m_j}, otherwise \end{cases} \tag{3.9}$$

The activation o_j is then transformed to the firing rate r_j:

$$r_j(t) = o_j(t) \cdot r_{\text{max},HCU} \tag{3.10}$$

Here, τ_{zi}, γ_m, $r_{\text{max},HCU}$ are all constants.

3.2 Architecture for the Hardware Accelerator

To accelerate the computationally intensive parts of both the training and the inference process of BCPNN, a hardware architecture is proposed for efficient processing.

As shown in Figure 3.2, the design of the training process consists of five modules: the update module for presynaptic traces, the update module for synaptic traces, the update module for postsynaptic traces, the computation part for the weight, and the computation part for the bias. These trace update modules are used to update eight trace variables, Z_i, Z_j, E_i, E_j, P_i, P_j, E_{ij}, and P_{ij}, to track pre-synaptic, post-synaptic, and synaptic activities at different time scales. The weight and the bias computation modules calculate the weight w_{ij} and the bias β_j based on the three updated P traces (P_i, P_j, P_{ij}) from the three trace update modules. As for the inference process, it is implemented with an inference computing core, which is responsible for the computation of the activation and firing rate of MCUs.

In order to maximize the parallelism, these update modules all require two adders and two multipliers respectively, which will be explained in Section 3.4.3. Besides, a block memory is employed to implement efficient exponential operations due to the lazy-update method, whose detailed design will be presented in Section 3.4.2. As for the weight and the bias computation module, a logarithmic IP core is employed for the logarithmic calculation. As for the inference core, three adders and two multipliers are needed to maximize parallelism. In addition, a block memory is used to store the weights, and an exponential IP core is employed for general exponential operations, which is different from the exponential operation in the training process. It should be noted that the operations in the weight and bias modules are both in the form of floating-point, thereby a fixed2float core and a float2fixed core are needed as well. It is the same for the inference module.

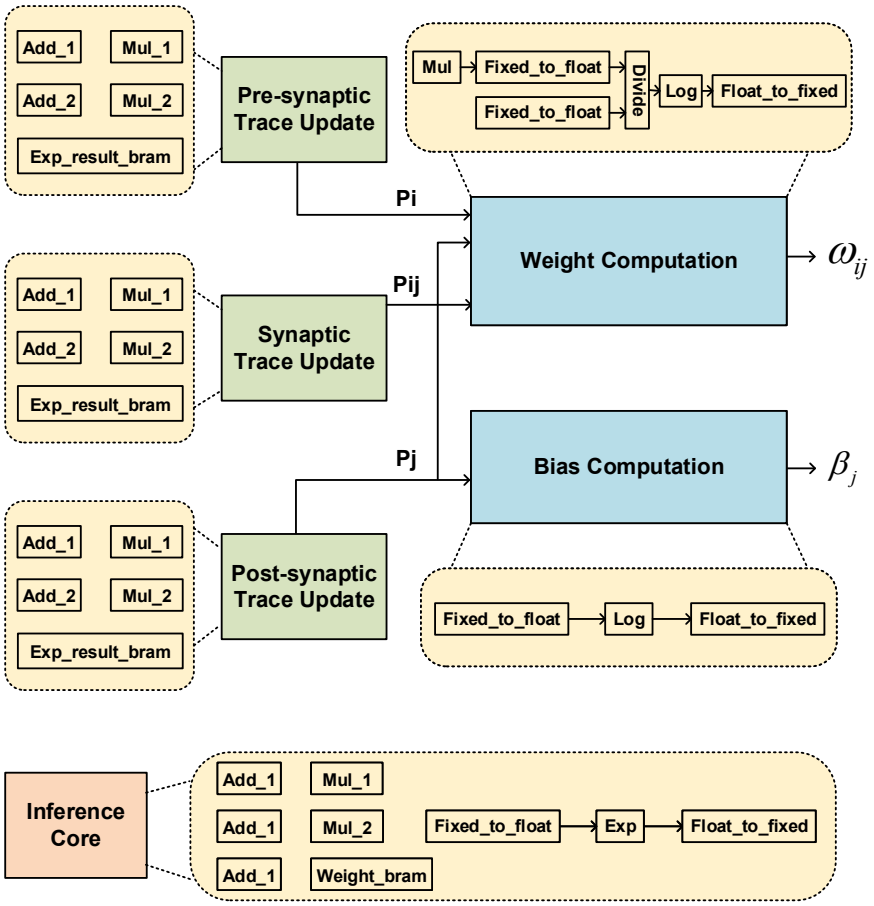


Figure 3.2: The architecture of the hardware accelerator.

3.3 The update mode and the inference mode

The proposed hardware accelerator can work in two modes, the training mode and the inference mode. The training mode is used to update eight local synaptic traces, the weight, and the bias. During the inference mode, the activation and firing probability of each MCU are calculated and updated. As illustrated in Figure 3.3 (a), the training mode consists of three parts: the update of the pre-synaptic vector (Z_i, E_i, P_i), the update of the synaptic matrix (E_{ij}, P_{ij}, w_{ij}), and the update of the post-synaptic vector (Z_j, E_j, P_j, β_j). Especially, the update of the synaptic matrix can be further divided into the row update and the column update, because both the pre- and post- spikes can trigger the update of the synaptic matrix.

As shown in Figure 3.3 (b), the hardware design is based on the lazy-update method mentioned in Section 3.1.1. The pre-synaptic vector, the specific row and column of the weight matrix are all updated in an event-driven manner, which are triggered by pre-synaptic spikes and post-synaptic spikes. Besides, the update of the post-synaptic vector and the inference process are both in a time-driven manner, which are triggered at every simulation time step.

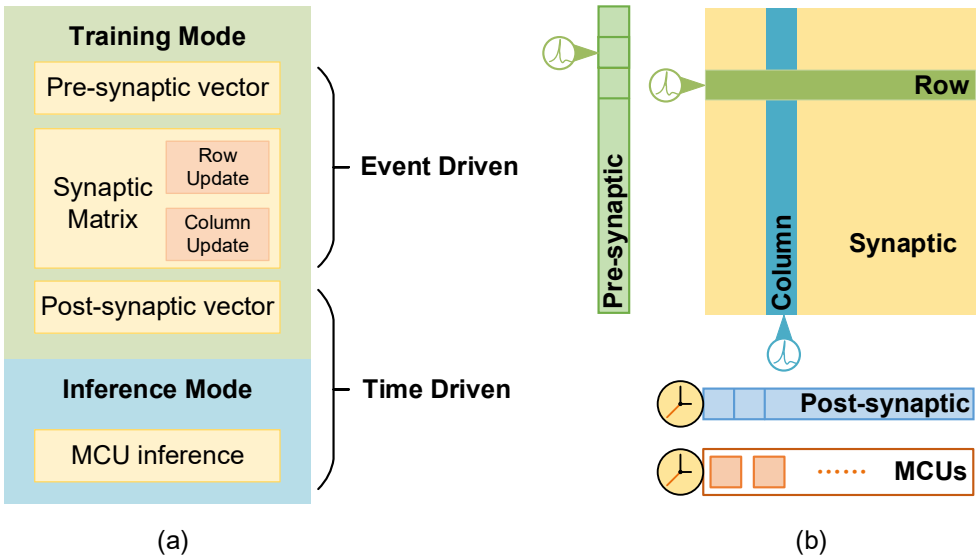


Figure 3.3: The training mode and the inference mode.

3.4 Optimization Methods

3.4.1 Hybrid Update Mechanism

This hardware accelerator adopts a hybrid update mechanism, where the event-driven manner is used for the update of presynaptic vector and the synaptic weight matrix, and the time-driven manner is used for the update of postsynaptic vector and the MCU state variables.

In the training process, the input presynaptic spike S_i and the output presynaptic spike S_j of the MCU are detected first. If there is a S_i, then the corresponding cell of the presynaptic vector will be updated. If not, the cell of the presynaptic vector will remain the same as the last timestep. In this case, time will be saved compared with the original

time-driven manner. Similarly, if there is a S_i, the corresponding row of the synaptic weight matrix will be updated. Otherwise, this update step will be skipped. As for the column update of the weight matrix, if the corresponding MCU cell sends out a spike, the corresponding column of the synaptic weight matrix will be updated, and vice versa.

In the inference process, the postsynaptic vector and the MCU state variables are updated in a time-driven manner. Therefore, their values will change in each timestep.

3.4.2 The Exponential Operation for the Training Process

During the training mode, the synaptic state variables are updated using exponential operations. More specifically, this refers to the computation of the attenuation factor of traces, which is defined as follows:

$$f(\Delta t) = e^{-\frac{\Delta t}{\tau_{zi}}} \tag{3.11}$$

Equation (3.11) uses the variable Δt which represents the duration between the current time t and the last time the variable was updated, denoted as t^{last}, as shown below:

$$\Delta t = t - t^{last} \tag{3.12}$$

Furthermore, hardware design complexity is increased by other exponential operations involving different time constants, such as τ_{z_j}, $\tau_{z_{ij}}$, τ_{z_e}, and τ_p^*.

Considering the computational accuracy and resource cost, implementing exponential operations in FPGA is a challenging task. Several methods for implementation include piecewise linear approximation, CORDIC algorithm, higher-order polynomial approximation, and table lookup. The piecewise linear approximation method is simple to implement, but its accuracy level is low. The CORDIC algorithm is commonly used in FPGA, but it has a limited input range and requires longer latency, which is not ideal

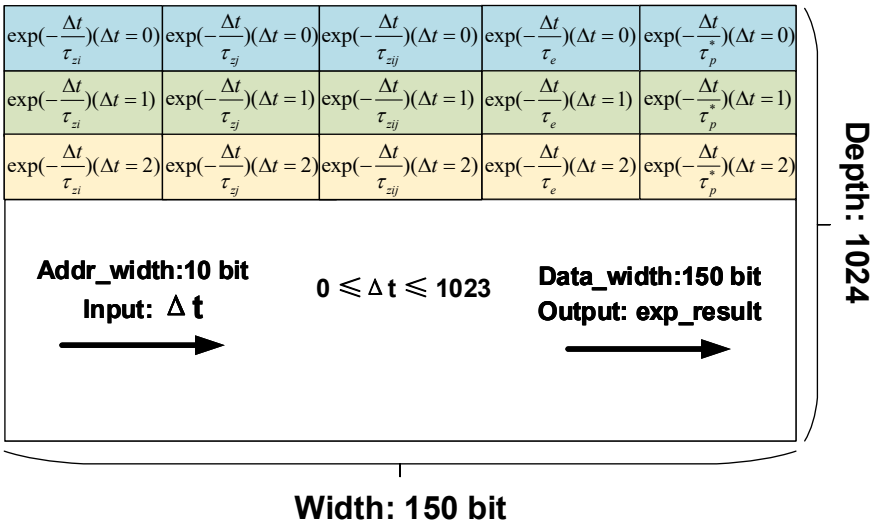


Figure 3.4: The structure of the block memory for exponential operations.

in this scenario. While the higher-order polynomial approximation method offers good accuracy, it requires too many hardware resources.

In this design, a modified table lookup method is utilized due to the lazy-update mechanism described earlier. One advantage of this method is that the output of Equation (3.11) is bounded between 0 and 1, and the exponential function decreases monotonically. Therefore, when the time interval Δt becomes sufficiently large (e.g., 1023), the exponential result can be approximated as zero, leading to good accuracy with a limited depth of block memory. Additionally, this method allows for obtaining the exponential results with five different time constants simultaneously for the same input variable Δt, which significantly accelerates the calculation process.

Figure 3.4 illustrates the structure of the block memory used for exponential operations in the training process. Increasing the width and depth of the block memory can provide a wider calculation range and higher accuracy.

The modified table lookup structure is particularly well-suited to the training process design, as it enhances the calculation speed and accuracy, while substantially reducing the consumption of hardware resources. Besides, it should be noted that the exponential operations in the training process and the inference process are implemented with different methods. In the training process, the exponential operation can be implemen-

ted with a modified table lookup structure due to the lazy-update mechanism. While in the inference process, an exponential IP core is employed due to the need for general exponential calculations.

3.4.3 The Optimal Design by Maximizing Parallelism

To optimize the hardware accelerator's computing performance, we have implemented an optimal design that maximizes parallelism in the calculation process. This is accomplished by breaking down the formula calculation steps and analyzing their inherent data dependencies. Independent calculation can be allocated to separate computation units for better parallelism while operations that have temporal relevance may need to be executed sequentially.

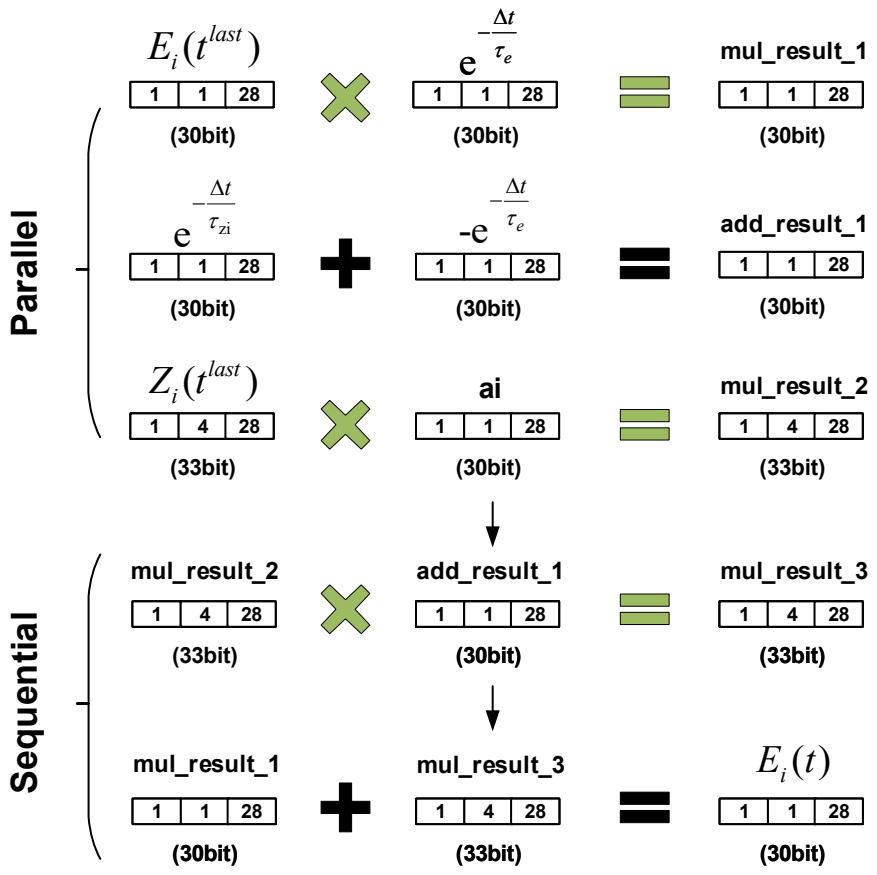


Figure 3.5: The computation process of E trace in the presynaptic module.

To optimize the computing performance of the hardware accelerator, we implemented a design that maximizes parallelism in the calculation process. This was accomplished

by breaking down the computation steps of equations and analyzing the inherent data dependency. For instance, let us consider the calculation of the E trace in the presynaptic module, as defined in Equation (3.2). Equation (3.2) can be decomposed into five steps consisting of two additions and three multiplications, as illustrated in Figure 3.5. Given the data dependency of the operands, the first three steps can be executed independently and in parallel. The fourth and fifth steps, however, must be performed sequentially since they are interdependent. As a result, we employ two multipliers to allow for the simultaneous execution of the two multiplication operations. This same design principle is applied to all calculation steps. In total, the three trace update modules require two adders and two multipliers each, while the inference module needs three adders and two multipliers. Thus, we achieved the optimal design by maximizing parallelism to achieve the best computing performance.

4 Implementation of the FPGA-based Hardware Accelerator

This section reports the experimental results of the FPGA-based implementation of the hardware accelerator, including the hierarchy of the design, the behavior simulation results for both the training mode and inference mode, and the overall implementation results.

4.1 Hierarchy of the Design

The hierarchy of the hardware accelerator is illustrated in Fig. 4.1. The design is composed of two main modules, the update module and the inference module.

The update module is used to calculate five parts in the training mode, including the pre-synaptic trace, the post-synaptic trace, the synaptic trace, the weight, and the bias. To realize the pre-, post-synaptic, and synaptic traces, several computing units are needed, including two adders, two multipliers, and a block memory ”exp_result_bram” for exponential operations. The bit width of the two input operands and the output of the adder are all 33 bits. As for the multiplier, the bit width of the two input operands of the multiplier is 33 bits, and the output bit width is also 33 bits after truncation. Especially, the computing blocks can be reused and reconfigured to reduce hardware overheads, which will be discussed in Section 5.

The inference module is responsible for the update of the internal state variables of MCUs. In this module, three adders and two multipliers are needed. Besides, a block memory ”mcu_inf_bram” is used to store the state variables of MCUs. The weight memory ”weight_bram” is used for weight storage. Another block memory ”exp_oj_bram” is used for the exponential calculation of the activation of MCUs.

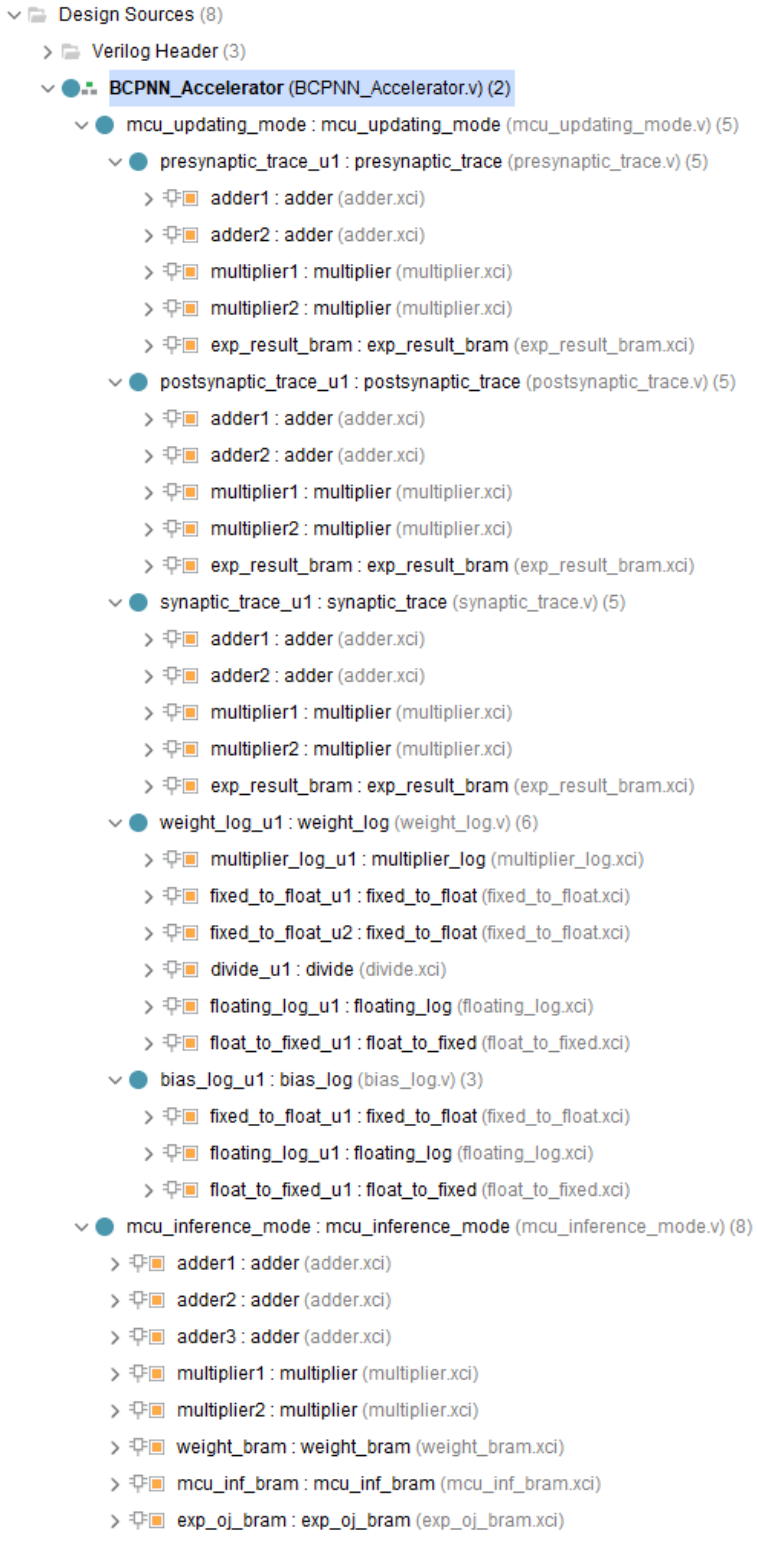


Figure 4.1: The hierarchy of the design.

In the top-level file ”BCPNN_Accelerator.v”, it calls two submodules, the ”mcu_update_mode.v” and the ”mcu_inference_mode.v”, as shown in Fig. 4.2.

```
mcu_updating_mode mcu_updating_mode (
    .clk(clk_200M),
    .rst_n(sys_rst_n),
    .syn_init(syn_init),
    .update_en(update_en),
    .spike_i_value(spike_i_value),
    .spike_j_value(spike_j_value),
    .s_axis_tvalid(s_axis_tvalid),
    .curr_time(curr_time),
    .output_tvalid_1(output_tvalid_1),
    .output_tvalid_2(output_tvalid_2)
    );

mcu_inference_mode mcu_inference_mode (
    .clk(clk_200M),
    .rst_n(sys_rst_n),
    .update_en(update_en),
    .const_init(const_init),
    .mcu_id(mcu_id),
    .spike_connections(spike_connections),
    .s_axis_tvalid(s_axis_tvalid),
    .s_axis_tready(s_axis_tready)
    );
```

Figure 4.2: The instantiation part of the top-level module.

In the submodule ”mcu_update_mode.v”, it calls five modules, including the presynaptic trace update, the postsynaptic trace update, the synaptic trace update, the weight update, and the bias update module, as shown in Fig. 4.3 and 4.4.

The definition of parameters including the bit width of the variables is defined in the header file, as shown in Fig. 4.5.

```
presynaptic_trace presynaptic_trace_ul(
    .clk(clk),
    .rst_n(rst_n),
    .update_en(update_en),
    .syn_init(syn_init),
    .spike_i_value(spike_i_value),
    .state_variable_inf(state_variable_inf),
    .s_axis_tvalid(s_axis_tvalid),
    .s_axis_tready_i(s_axis_tready),
    .curr_time(curr_time),
    .Zi(Zi),
    .Ei(Ei),
    .Pi(Pi),
    .Pi_log(Pi_log)
);

postsynaptic_trace postsynaptic_trace_ul(
    .clk(clk),
    .rst_n(rst_n),
    .update_en(update_en),
    .syn_init(syn_init),
    .spike_j_value(spike_j_value),
    .state_variable_inf(state_variable_inf),
    .s_axis_tvalid(s_axis_tvalid),
    .s_axis_tready_j(s_axis_tready),
    .curr_time(curr_time),
    .Zj(Zj),
    .Ej(Ej),
    .Pj(Pj),
    .Pj_log(Pj_log)
);
```

Figure 4.3: The instantiation part of the update mode (part 1).

```
synaptic_trace synaptic_trace_u1(
    .clk(clk),
    .rst_n(rst_n),
    .update_en(update_en),
    .syn_init(syn_init),
    .state_variable_inf(state_variable_inf),
    .s_axis_tvalid(s_axis_tvalid),
    .s_axis_tready_ij(s_axis_tready),
    .log_tvalid_ij(log_tvalid_ij),
    .curr_time(curr_time),
    .write_back_flag(write_back_flag),
    .Eij(Eij),
    .Pij(Pij),
    .t_last_sp(t_last_sp),
    .Pij_log(Pij_log)
);

weight_log weight_log_u1(
    .Pi_log(Pi_log),
    .Pj_log(Pj_log),
    .Pij_log(Pij_log),
    .input_tvalid_1(log_tvalid_ij),
    .input_tvalid_2(log_tvalid_ij),
    .output_tvalid(output_tvalid_1),
    .wij(wij)
);

bias_log bias_log_u1(
    .Pj_log(Pj_log),
    .input_tvalid(log_tvalid_ij),
    .output_tvalid(output_tvalid_2),
    .bias_j(bias_j)
);
```

Figure 4.4: The instantiation part of the update mode (part 2).

```
`define ADD_PORTA_MUX_WIDTH     4// The select signal for input port A of the Adder
`define ADD_PORTB_MUX_WIDTH     4// The select signal for input port B of the Adder
`define ADD_PORTA_WIDTH        33// Bit width of the input port A of the adder
`define ADD_PORTB_WIDTH        33//
`define ADD_PORTS_WIDTH        33//

`define MUL_PORTA_MUX_WIDTH     3//The select signal for input port A of the MUL
`define MUL_PORTB_MUX_WIDTH     3//The select signal for input port B of the MUL
`define MUL_PORTA_MUX_WIDTH_IJ  4//
`define MUL_PORTB_MUX_WIDTH_IJ  4//
`define MUL_PORTA_WIDTH        33//Bit width of the input port A of the MUL
`define MUL_PORTB_WIDTH        33//
`define MUL_PORTP_WIDTH        33//


`define Z_TRACE_WIDTH          33 //Zi,Zj trace; 1,4,28. rangge[0,10]
`define E_TRACE_WIDTH          30 //E_trace; 1,1,28  rangge[0,1]
`define P_TRACE_WIDTH          30 //P_trace; 1,1,28
`define P_LOG_WIDTH            33 //bias; 1,4,28
`define BIAS_WIDTH             33 //bias; 1,4,28
`define WEIGHT_WIDTH           33 //weight; 1,4,28
`define PARA1_WIDTH            30 //width:1,1,28
`define PARA2_WIDTH            33 //width:1,4,28

`define EXP_RESULT_WIDTH       30//exp: 1,1,28
```

Figure 4.5: The parameter definition in the header file.

4.2 Computing Blocks of the Design

In the implementation of the design, a variety of computing blocks are involved, including the adder, the multiplier, the logarithmic module, and some format conversion modules.

4.2.1 Adder

The adder is responsible for the addition operations in the computing process. The input width of the two operands A and B are both 33 bits, as shown in Fig. 4.6. The output width of the output S is 33 bits. The adder modules are implemented with the DSP48 resources in the FPGA.

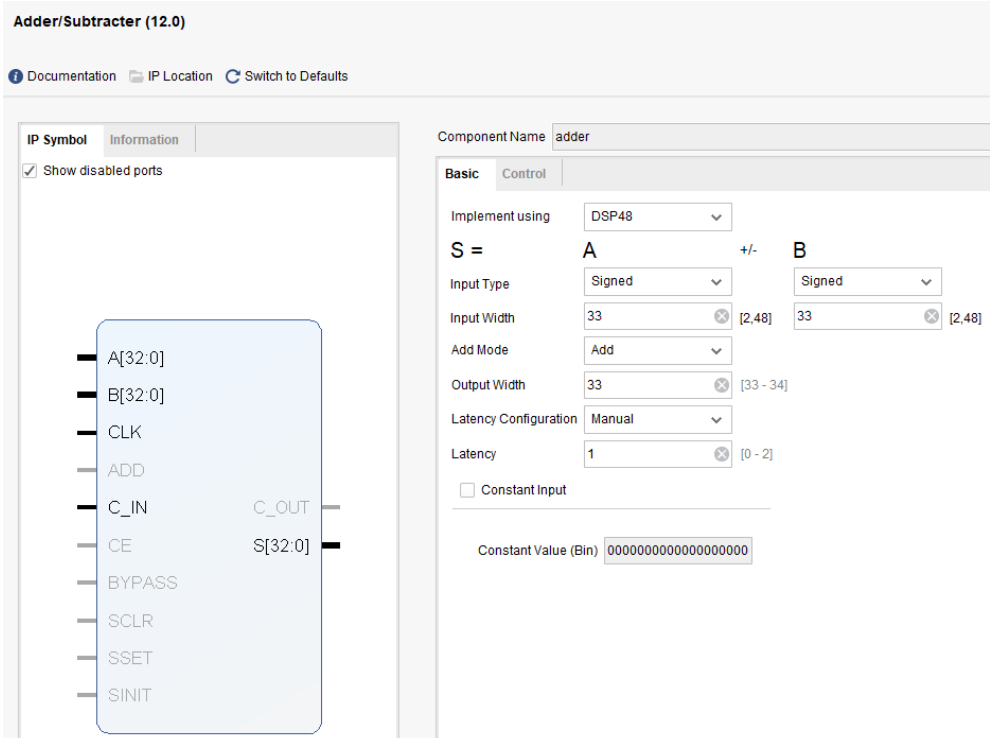


Figure 4.6: The adder module.

4.2.2 Multiplier

The diagram of the multiplier is illustrated in Fig. 4.7. The input width of the two input operands A and B are both 33 bits. It should be noted that the width of the calculation result of the multiplier is truncated. As shown in Fig. 4.8, 33 bits (from the 28 th to the 60 th bit) are selected from the original 66-bit calculation result.

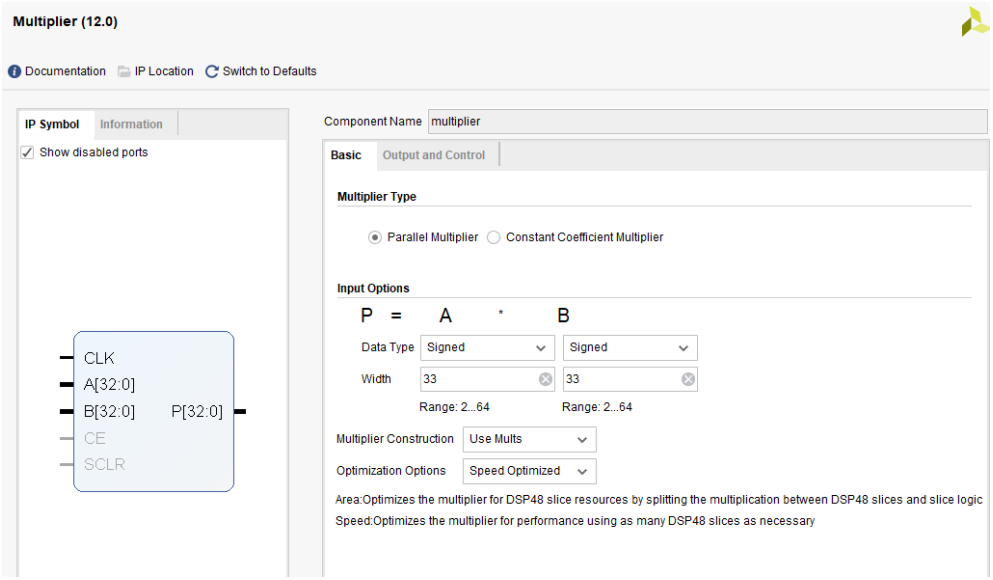


Figure 4.7: The multiplier module.

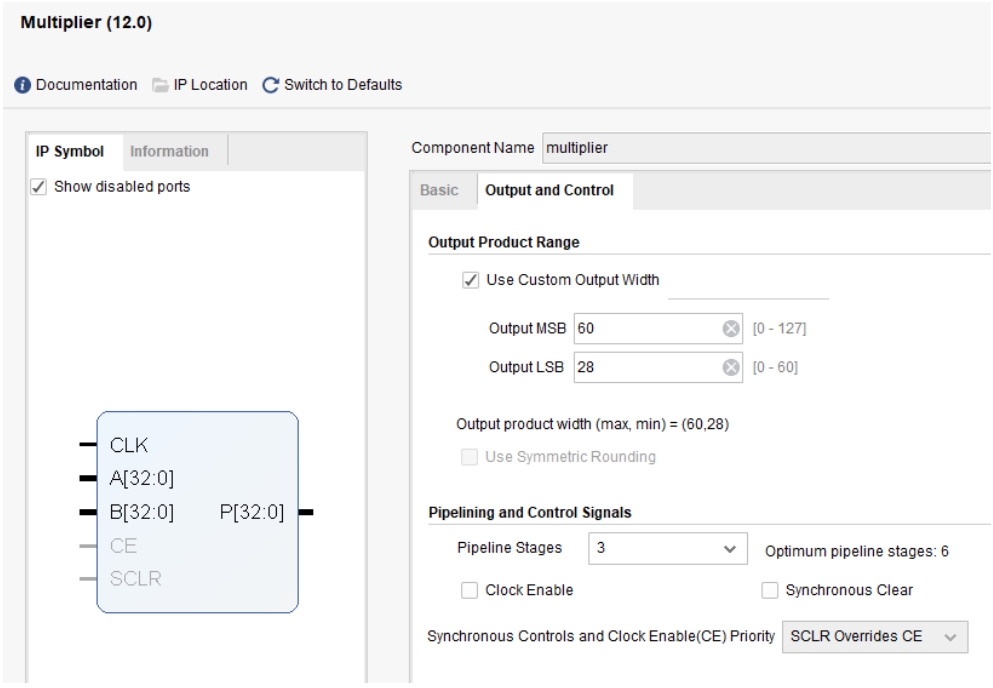


Figure 4.8: The output bit width truncation of the multiplier.

4.2.3 Logarithmic Module

The logarithmic operations in the computing process are implemented with the logarithmic module, as shown in Fig. 4.9. The computation of the logarithmic module is

in the form of floating point. Therefore, a fixed2float module and a float2fixed module are also needed, which will be introduced in the next section.

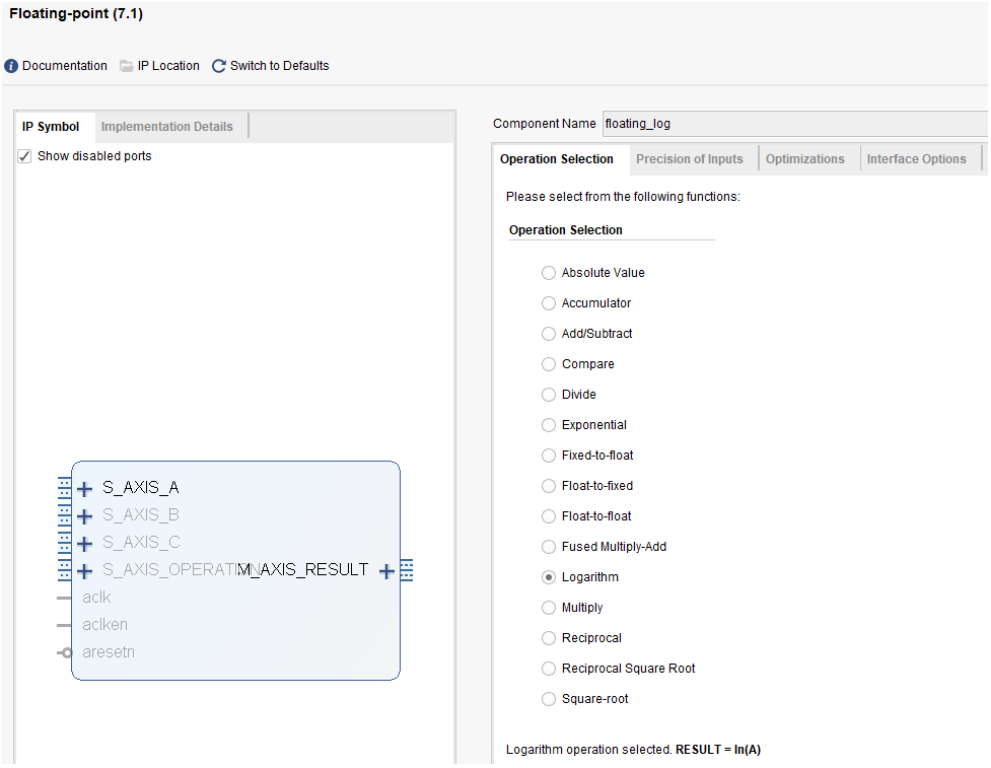


Figure 4.9: The logarithmic module.

4.2.4 Format Conversion Module

As mentioned above, the addition and multiplication are in fixed-point format, but logarithmic operations are in floating-point format. Therefore, the fixed2float module and the float2fixed module are required for the format conversion, as shown in fig. 4.10 and Fig. 4.11.

The fixed-point format is shown in Fig. 4.12. There is one bit for the sign, four bits for the integer part, and also 28 bits for the decimal part. The floating-point format is illustrated in Fig. 4.13. The total bit width of single-precision floating-point numbers is 32 bits. Among them, the sign bit is 1 bit, the exponent width is 8 bits, and the fraction width is 24 bits.

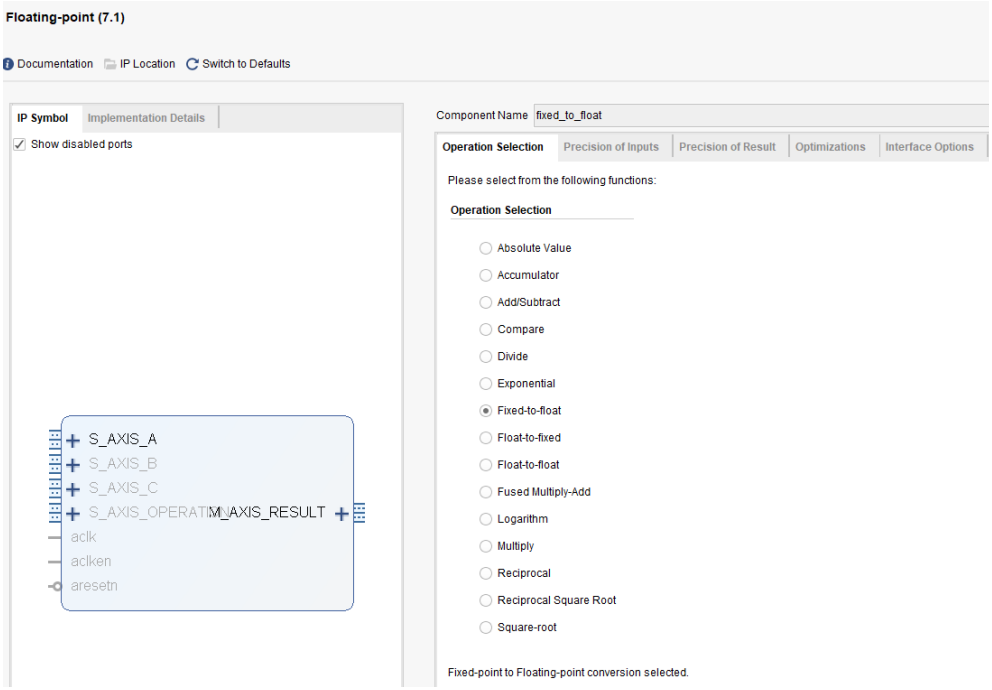


Figure 4.10: The fixed2float module.

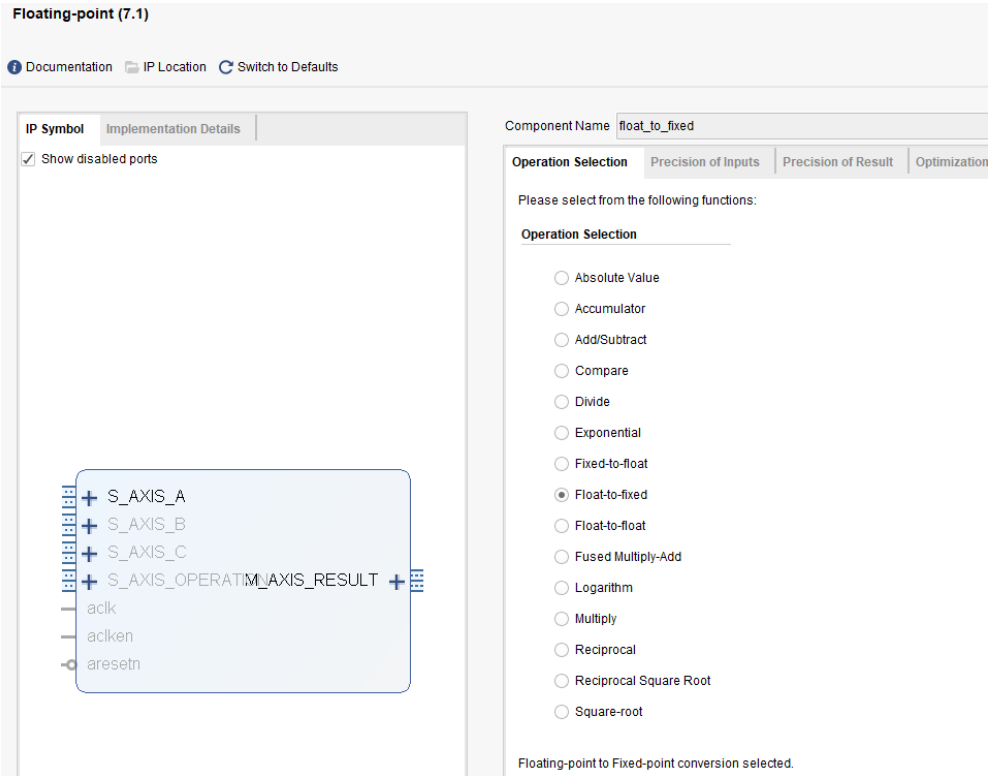


Figure 4.11: The float2fixed module.

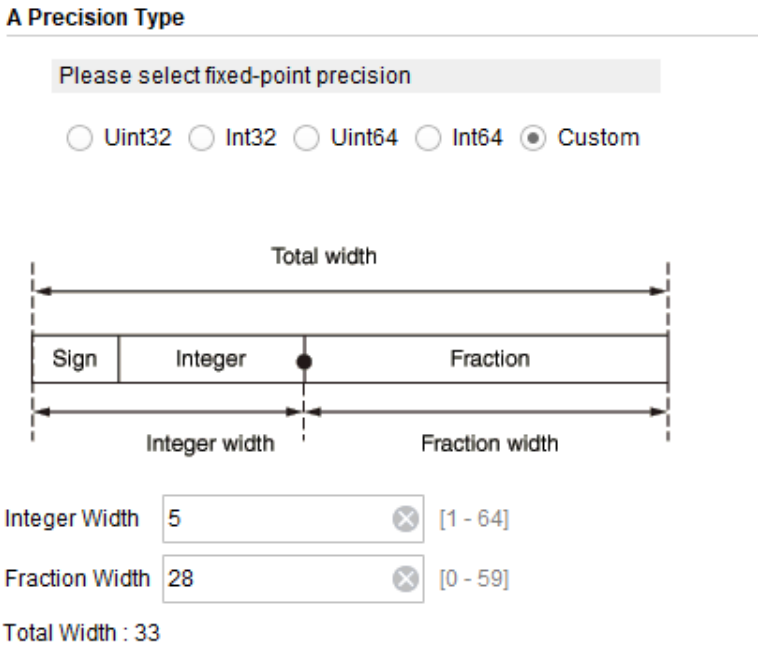


Figure 4.12: The format of the fixed-point number.

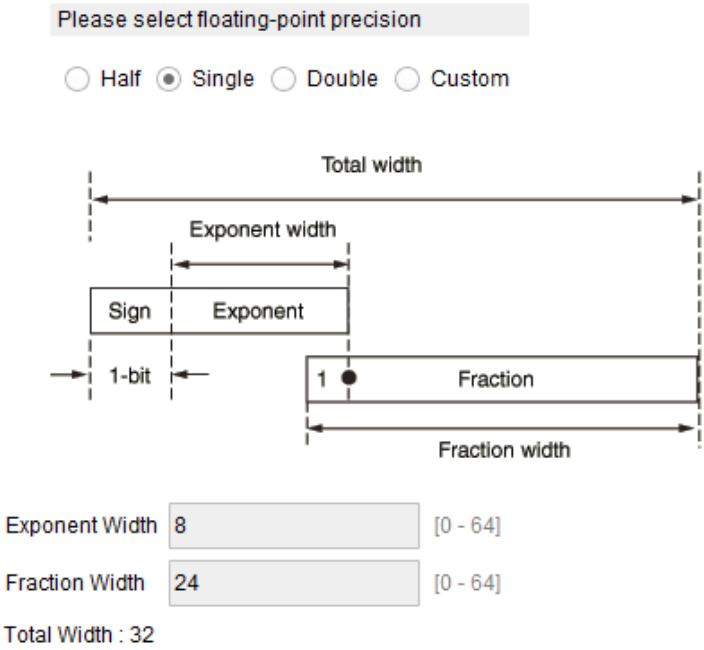


Figure 4.13: The format of the floating-point number.

4.3 Behavior Simulation

To verify the functionality of the proposed hardware accelerator, the behavior simulation is performed.

The signal control part of the testbench file is shown in Fig. 4.14. The ”clk” is the clock signal. The ”rst_n” is the reset signal, which is valid when the value equals 0. The ”update_en” is the signal that controls the switch between the training stage and the inference stage. When the value of ”update_en” is 1, the accelerator works in the training mode. Vice versa, when ”update_en” is 0, the accelerator works in the inference mode.

```
initial
    begin
        clk = 1;                        // clock signal
        rst_n = 0;                      // reset signal
        update_en = 1;                  // training stage enable
        syn_init = 1;
        s_axis_tvalid = 1;
        spike_i_value = 1;
        spike_j_value = 1;
        curr_time = 0;                  // recoding the current ti
        time_lms = 0;
        const_init = 0;
        mcu_id = 4'b0000;
        spike_connections = 10'b1111111111;
        s_axis_tvalid = 1;
        #10;
        rst_n = 1;
        #20;
        syn_init = 0;
        #300;
        update_en = 0;                  // inference stage enable
        const_init = 1;
        #20;
        const_init = 0;
    end
```

Figure 4.14: The signal control part in the testbench.

During the simulation, each timestep can be adjusted and reconfigured in the simulation time control part of the testbench, as shown in Fig.4.15.

```
always @(posedge clk)
begin
    if (syn_init) begin
        time_1ms <= 18'd0;
    end
    else if (time_1ms == 18'd199999) begin // increase 1 every cyc
        time_1ms <= 18'd0;
    end
    else begin
        time_1ms <= time_1ms + 1'b1;
    end
end

always @(posedge clk)
begin
    if (syn_init) begin
        curr_time <= 40'd1;
    end
    else if (time_1ms == 18'd199999) begin // used to record 1 ms
        curr_time <= curr_time + 1'b1;
    end
    else begin
        curr_time <= curr_time;
    end
end
```

Figure 4.15: The simulation time control part in the testbench.

Fig. 4.16 and Fig. 4.17 present the behavior simulation results. As mentioned before, the accelerator can work in the training mode and the inference mode. The switching of the training mode and the inference mode is controlled by the ”update_en” signal.

The simulation wave is generated with the testbench file above. As marked with a yellow line in Fig. 4.16, when the ”update_en” is set to 1, the accelerator works in the

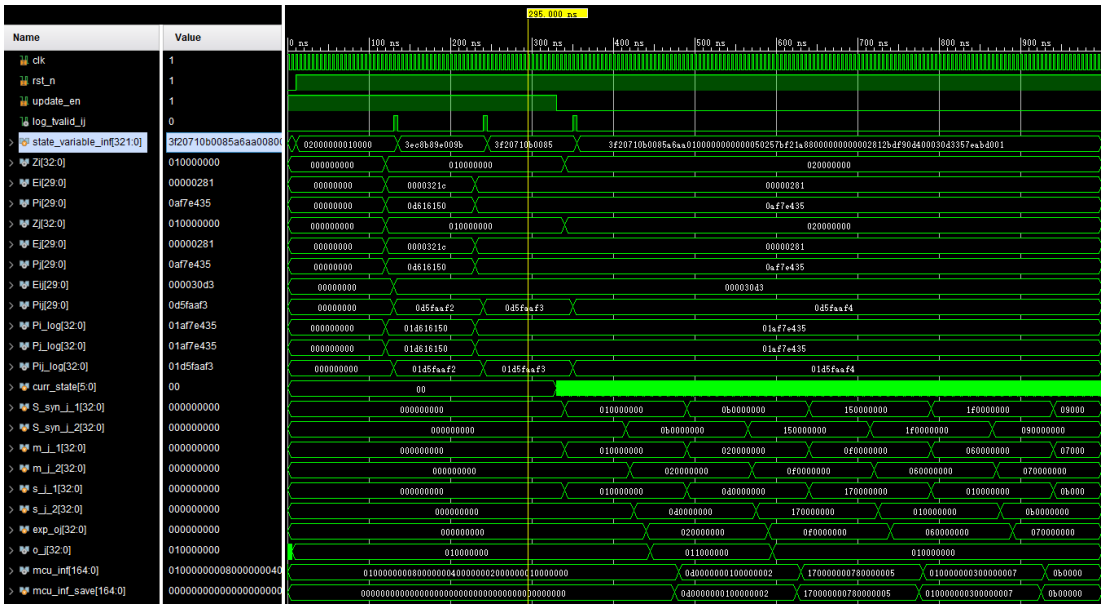


Figure 4.16: The behavior simulation of the training mode.

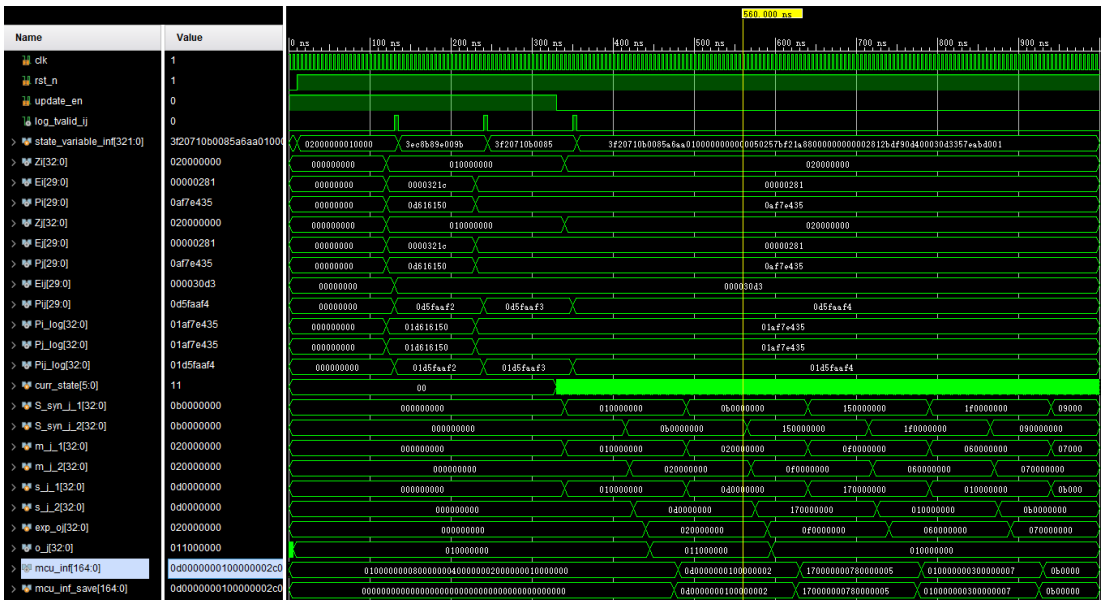


Figure 4.17: The behavior simulation of the inference mode.

training mode. In this mode, the synaptic traces Z_i, E_i, P_i, Z_j, E_j, P_j, E_{ij} and P_{ij} are updated.

When the ”update_en” is set to 0, the accelerator works in the inference mode. In this mode, the internal state variables of the MCUs are calculated and updated, as marked

with the yellow line in Fig. 4.1. The results of the behavior simulation are compared with the ideal reference model and the simulation results are as expected. The error analysis of the results will be explained in Section 5.1.

4.4 Implementation Results

The Xilinx Artix-7 XC7A100T FPGA is used to implement the RTL design, as illustrated in Figure 4.18. The implementation results are summarized in Table 4.1. The total on-chip power consumption is 265 mW. The timing constraints are satisfied with a worst negative slack of 0.219 ns. The timing and power summary can be found in Figure 4.19 and Figure 4.20, respectively.

To carry out a weight update in training mode, 22 cycles are needed, while an inference process of a spiking unit takes 36 cycles. The accelerator has been designed and implemented on an FPGA to speed up these processes while maintaining programming flexibility. At a clock frequency of 200 MHz, the accelerator can update each synaptic weight in 110 ns and complete an MCU's inference process in 180 ns. Assuming a time step of 1 ms, up to 9090 synaptic connections can be updated simultaneously.

Total On-Chip Power (mW)	Worst Negative Slack (ns)	Cycles for Weight Update	Cycles for Inference
265	0.219	22	36

Table 4.1: The overall synthesis and implementation results.

The basic programmable unit of an FPGA is the configurable logic block (CLB). The CLB consists of the lookup table (LUT), the multiplexer (MUX), a carry chain, and registers. LUTs and MUXs can realize combinational logic functions, and registers (which can be configured as flip-flops or latches) can realize sequential logic functions.

In the Xilinx FPGA devices, a CLB consists of multiple (usually two or four) identical slices and additional logic. The slice can be further divided into two types, the SLICEL

Figure 4.18: Implementat the accelerator on the Xilinx Artix-7 XC7A100T FPGA.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.219 ns	Worst Hold Slack (WHS):	0.073 ns	Worst Pulse Width Slack (WPWS):	1.728 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	6392	Total Number of Endpoints:	6392	Total Number of Endpoints:	2835

All user specified timing constraints are met.

Figure 4.19: The timing summary.

for logic and the SLICEM for memory. SLICEL and SLICEM both contain four 6-input lookup tables (LUT6), three multiplexers, a carry chain and eight flip-flops. The schematic of the device after implementation can be seen in Fig. 4.21 (a). Fig. 4.21 (b) demonstrates the routing of the hardware resources on the FPGA device.

To take a close look at the utilization of the hardware resources on FPGA, Fig. 4.22 demonstrates part of the used resources after implementation. The one marked with a red box on the right is a SLICEL unit. In this slice, four LUTs and a carry chain

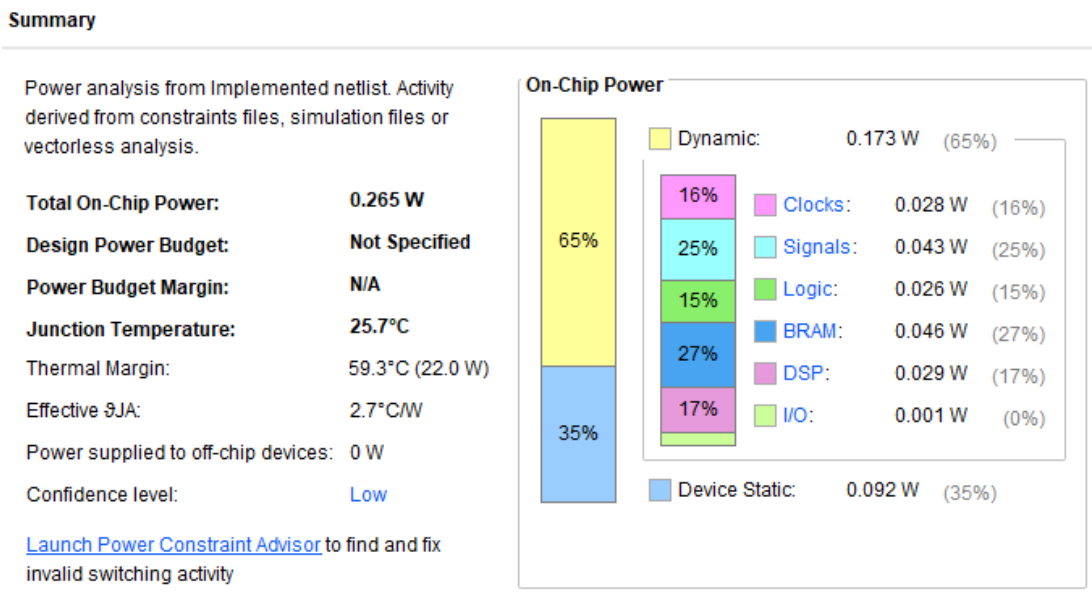


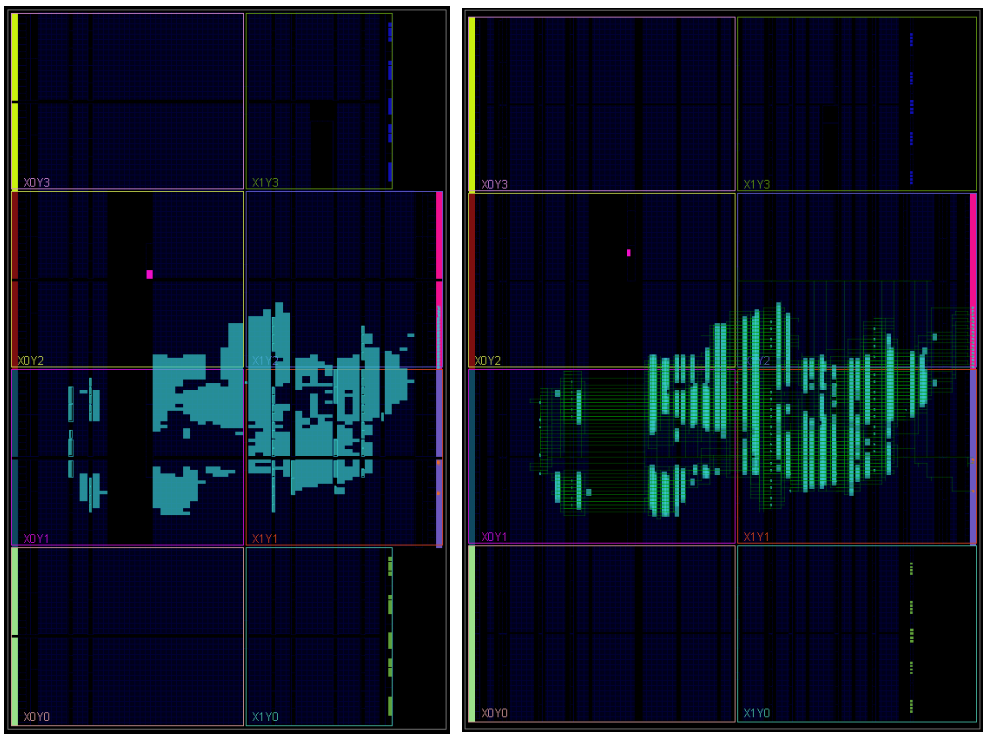
Figure 4.20: The power summary.

are utilized. The one marked on the left is a DSP, which is utilized for addition and multiplication operations.

Table 4.2 presents the utilization of FPGA hardware resources. The results indicate that DSPs consume the majority of the hardware resources, with a maximum utilization rate of 17.50 %. In Section 5.3, we will elaborate on how we can minimize the consumption of DSP slices through module reuse.

Resource	Utilization	Available	Utilization (%)
LUT	4024	63400	6.35
FF	2778	126800	2.19
BRAM	11.50	135	8.52
DSP	42	240	17.50
IO	21	285	7.37

Table 4.2: Hardware resource utilization.



(a) (b)

Figure 4.21: (a) The schematic of the FPGA device. (b) Routing resources on the FPGA device.

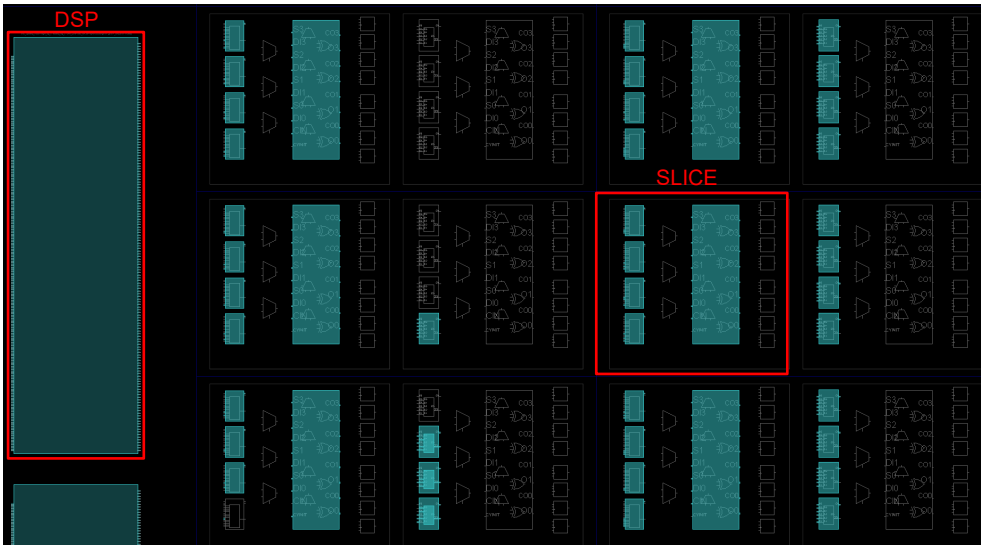


Figure 4.22: The schematic of the hardware resources on the FPGA device.

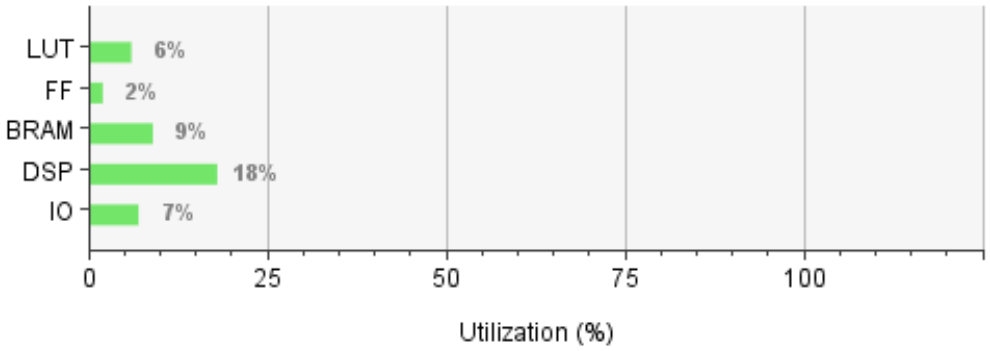


Figure 4.23: The utilization of the hardware resources on the FPGA device.

5 Evaluation and Discussion

5.1 Performance Benchmark and Comparison

Since there are no hardware accelerators for BCPNN currently available, we compared the performance of the FPGA accelerator with that of the CPU through experiments. Table 5.1 displays the time taken by the FPGA and CPU to perform a synaptic weight update and an inference process of the MCU. This comparison was conducted to provide a rough idea of the accelerator's performance.

The Artix-7 FPGA chip was used in the experiment, which required 22 cycles for a weight update and 36 cycles for an inference process. Running at a clock frequency of 200 MHz, the FPGA can perform a weight update in 110 ns and an inference process in 180 ns.

The performance evaluation of the CPU was carried out on a MacBook Pro equipped with a 2.6 GHz dual-core Intel Core i5 processor that can reach a Turbo Boost speed of up to 3.2 GHz. It comes with 3 MB shared L3 cache and has 8 GB of onboard memory with a clock speed of 1600 MHz. The operating system used was Mac OS 10.15, and the code was compiled using Xcode. The weight update and inference process were implemented in C++ using Xcode, taking on average 25,800 ns and 10,600 ns, respectively, to complete each operation.

The hardware accelerator based on FPGA greatly enhances the computation speed when compared to the CPU test, with the processing time decreased by two magnitudes.

The precision of calculations is evaluated by recording and comparing the specific values of weights, biases, and activations, which is presented in Table 5.2. It is important to mention that the recorded activation of each MCU needs to be normalized based on the total number of MCUs.

Platform	IDE	Working Frequency (Hz)	Weight Update Time (ns)	Inference Time (ns)
CPU	Xcode	2.6 G	25800	10600
FPGA	Vivado	200 M	110	180

Table 5.1: Comparison of calculation speed.

Although the FPGA processes computations in fixed-point format, conversion from floating-point to fixed-point can introduce errors. Moreover, truncation of multiplication results can also cause errors during the FPGA calculation process. Nevertheless, the comparison of calculation accuracy between the FPGA experiment and the CPU test presented in Table 5.2 demonstrates that the weight, bias, and activation calculations in the FPGA experiment match those in the CPU test up to the sixth decimal place. This suggests that the hardware accelerator achieves computational precision similar to that of the CPU.

Mode	Format of operation	Result of weight	Result of bias	Result of activation
CPU	Floating point	-0.60796681	0.60773897	7.38905610
FPGA	Fixed point	-0.60796648	0.60773867	7.38905621

Table 5.2: Comparison of calculation accuracy.

In conclusion, the experiment illustrates that the FPGA-driven hardware accelerator delivers considerably better computing speed while preserving a high degree of accuracy in calculations.

5.2 Resource Saving through Module Reuse

Based on the architecture depicted in Figure 3.2, the adders and multipliers used in the training mode can be customized and reused to minimize hardware resource requirements, especially the DSPs used. The adders and multipliers utilized in the inference

mode can be optimized with those in the training process after FPGA synthesis and implementation. Therefore, the reuse of adders and multipliers in the inference process is not addressed in this study.

No.	Reused	Number of adders	Number of multipliers	Total cycle	DSP used
1	None	6	6	22	42
2	Partially	4	4	22	32
3	Fully	4	4	25	28
4	Fully	2	2	30	18

Table 5.3: Reuse the adders and multipliers to reduce hardware overheads.

Table 5.3 presents the performance and hardware overheads of different levels of adder and multiplier reuse. In the first scenario, where no computational modules are reused, the basic design consists of 2 adders and 2 multipliers used in three trace update modules, which results in a total of 6 adders and 6 multipliers. This configuration requires 42 DSP slices and takes 22 cycles to complete a weight update process.

In the second scenario, if a partial reuse of adders and multipliers is employed, a configuration with 4 adders and 4 multipliers can achieve the same weight update time of 22 cycles while reducing the DSP slice usage by 23.8%. It should be emphasized that only the multipliers in the three trace modules are being reused, except for the multiplier in the weight computation module.

Furthermore, in order to further minimize the hardware resource overheads, it is possible to fully reuse the computational modules, including the multiplier in the weight computation module. In the third case, the implementation requires 28 DSP slices and takes 25 cycles with 4 adders and 4 multipliers when fully reused. Although the weight update time increases by 13.6%, the DSP usage decreases by 33.3%. Finally, in the fourth scenario, only 18 DSPs are used, but this comes at the cost of increased update time.

The accelerator design can employ module reuse to decrease hardware resource usage while maintaining the same computational performance. Depending on the performance requirements and hardware resource constraints, a balance between performance and hardware overheads can be considered. If there is room for slight performance degradation, hardware overheads can be decreased, and if not, then they can be increased.

6 Conclusion

Over the last decade, ANNs have achieved substantial advancements in practical applications such as speech recognition, image classification, and natural language processing. Despite their success and popularity in the data-driven computing paradigm, these networks have some limitations.

In contrast to conventional ANNs that employ non-spiking units, spiking neural networks utilize an event-based communication mechanism similar to that of the human brain, where neurons communicate via spikes.

The BCPNN is a unique type of spiking neural network that is based on Bayesian inference principles and has an architecture inspired by the modularity of the mammalian cortex, HCUs and MCUs. It has been implemented within the framework of SNNs, allowing for mapping to neural and synaptic processes in the human cortex. In comparison to other SNN models, BCPNN has a modular, coarse-grained, and hierarchical architecture that makes it a practical and compact solution for the implementation of large-scale neural networks.

The main contribution of this thesis is the proposal of a hardware architecture that accelerates the training and inference process of BCPNN. To achieve this, various techniques have been utilized such as a hybrid update mechanism, customized LUT-based design for exponential operations, and optimization by maximizing parallelism. The proposed hardware accelerator is implemented on the Xilinx Artix-7 XC7A100T FPGA. Experimental results demonstrate that the computing speed of the FPGA-based hardware accelerator is significantly improved by two orders of magnitude when compared to the CPU. Additionally, by reusing computational modules, the accelerator can reduce hardware resource overheads while maintaining comparable computing performance.

The proposed accelerator presents a promising solution for efficiently implementing training and inference processes for large-scale BCPNN neural networks. This ad-

vancement may enable the implementation of higher-level cognitive phenomena such as synaptic working memory based on BCPNN in an efficient manner.

7 Acknowledgments

In this acknowledgement, I would like to thank my family, teachers and classmates who have accompanied me for these three years.

First of all, I would like to thank my supervisors, Prof. Juha Plosila, Dr. Hashem Haghbayan in UTU and Prof. Zhuo Zou in FDU. With your supports, I finished this thesis.

Thanks to Xiao-hui Liu and Tian-yi Wang. When I met with difficulties in study, I benefited a lot from your advice.

Thanks to my family for your support and understanding.

Finally, I would like to express my sincere thanks to all the teachers who reviewed this thesis and participated in the graduation examination. Thank you!

Bibliography

[1] G. Hinton, L. Deng, D. Yu et al. 'Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups'.
In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97.

[2] D. Ciregan, U. Meier and J. Schmidhuber.
'Multi-column deep neural networks for image classification'.
In: *2012 IEEE conference on computer vision and pattern recognition*.
IEEE. 2012, pp. 3642–3649.

[3] W. Yin, K. Kann, M. Yu and H. Schütze.
'Comparative study of CNN and RNN for natural language processing'.
In: *arXiv preprint arXiv:1702.01923* (2017).

[4] F. A. Azevedo, L. R. Carvalho, L. T. Grinberg et al.
'Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain'.
In: *Journal of Comparative Neurology* 513.5 (2009), pp. 532–541.

[5] W. Maass.
'Networks of spiking neurons: the third generation of neural network models'.
In: *Neural networks* 10.9 (1997), pp. 1659–1671.

[6] S. A. Aamir, Y. Stradmann, P. Müller et al. 'An accelerated lif neuronal network array for a large-scale mixed-signal neuromorphic architecture'.
In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 65.12 (2018), pp. 4299–4312.

[7] N. Qiao, H. Mostafa, F. Corradi et al. 'A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses'.
In: *Frontiers in neuroscience* 9 (2015), p. 141.

[8] G. Urgese, F. Barchi, E. Macii and A. Acquaviva.
‘Optimizing network traffic for spiking neural network simulations on densely interconnected many-core neuromorphic platforms’. In: *IEEE Transactions on Emerging Topics in Computing* 6.3 (2016), pp. 317–329.

[9] N. Caporale, Y. Dan et al.
‘Spike timing-dependent plasticity: a Hebbian learning rule’.
In: *Annual review of neuroscience* 31.1 (2008), pp. 25–46.

[10] F. Fiebig, P. Herman and A. Lansner.
‘An indexing theory for working memory based on fast hebbian plasticity’.
In: *eneuro* 7.2 (2020).

[11] A. Lansner and A. Holst.
‘A higher order Bayesian neural network with spiking units’.
In: *International Journal of Neural Systems* 7.02 (1996), pp. 115–128.

[12] S. Benjaminsson and A. Lansner.
Extreme scaling of brain simulation on JUGENE. 2011.

[13] S. Herculano-Houzel. ‘Scaling of brain metabolism with a fixed energy budget per neuron: implications for neuronal activity, plasticity and evolution’.
In: *PloS one* 6.3 (2011), e17514.

[14] J. M. Nageswaran, N. Dutt, J. L. Krichmar, A. Nicolau and A. V. Veidenbaum.
‘A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors’.
In: *Neural networks* 22.5-6 (2009), pp. 791–800.

[15] A. K. Fidjeland and M. P. Shanahan.
‘Accelerated simulation of spiking neural networks using GPUs’.
In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*.
IEEE. 2010, pp. 1–8.

[16] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza et al. ‘A million spiking-neuron integrated circuit with a scalable communication network and interface’.
In: *Science* 345.6197 (2014), pp. 668–673.

[17] M. Davies, N. Srinivasa, T.-H. Lin et al.
‘Loihi: A neuromorphic manycore processor with on-chip learning’.
In: *Ieee Micro* 38.1 (2018), pp. 82–99.

[18] J. Pei, L. Deng, S. Song et al.
‘Towards artificial general intelligence with hybrid Tianjic chip architecture’.
In: *Nature* 572.7767 (2019), pp. 106–111.

[19] J. Han, Z. Li, W. Zheng and Y. Zhang.
‘Hardware implementation of spiking neural networks on FPGA’.
In: *Tsinghua Science and Technology* 25.4 (2020), pp. 479–486.

[20] J. Zhang, H. Wu, J. Wei, S. Wei and H. Chen. ‘An asynchronous reconfigurable snn accelerator with event-driven time step update’.
In: *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE. 2019, pp. 213–216.

[21] L. Liu, D. Wang, Y. Wang et al. ‘A FPGA-based Hardware Accelerator for Bayesian Confidence Propagation Neural Network’.
In: *2020 IEEE Nordic Circuits and Systems Conference (NorCAS)*. IEEE. 2020, pp. 1–6.

[22] S. Ghosh-Dastidar and H. Adeli. ‘Spiking neural networks’.
In: *International journal of neural systems* 19.04 (2009), pp. 295–308.

[23] A. Taherkhani, A. Belatreche, Y. Li, G. Cosma, L. P. Maguire and T. M. McGinnity.
‘A review of learning in biologically plausible spiking neural networks’.
In: *Neural Networks* 122 (2020), pp. 253–272.

[24] W. Gerstner and W. M. Kistler.
Spiking neuron models: Single neurons, populations, plasticity.
Cambridge university press, 2002.

[25] 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), p. 500.

[26] C. Koch and I. Segev. *Methods in neuronal modeling: from ions to networks*. MIT press, 1998.

[27] W. Gerstner, W. M. Kistler, R. Naud and L. Paninski. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

[28] E. M. Izhikevich. 'Simple model of spiking neurons'. In: *IEEE Transactions on neural networks* 14.6 (2003), pp. 1569–1572.

[29] J. L. Lobo, J. Del Ser, A. Bifet and N. Kasabov. 'Spiking neural networks and online learning: An overview and perspectives'. In: *Neural Networks* 121 (2020), pp. 88–100.

[30] A. Lansner and Ö. Ekeberg. 'A one-layer feedback artificial neural network with a Bayesian learning rule'. In: *International journal of neural systems* 1.01 (1989), pp. 77–87.

[31] P. J. Tully, M. H. Hennig and A. Lansner. 'Synaptic and nonsynaptic plasticity approximating probabilistic inference'. In: *Frontiers in synaptic neuroscience* 6 (2014), p. 8.

[32] D. Wang, J. Xu, D. Stathis et al. 'Mapping the BCPNN Learning Rule to a Memristor Model'. In: *Frontiers in Neuroscience* 15 (2021), p. 750458.

[33] M. Thiboust. *Insights from the brain, the road towards Machine Intelligence*. 2020.

[34] D. H. Hubel and T. N. Wiesel. 'Ferrier lecture-Functional architecture of macaque monkey visual cortex'. In: *Proceedings of the Royal Society of London. Series B. Biological Sciences* 198.1130 (1977), pp. 1–59.

[35] J. DeFelipe, I. Ballesteros-Yáñez, M. C. Inda and A. Muñoz. 'Double-bouquet cells in the monkey and human cerebral cortex with special reference to areas 17 and 18'. In: *Progress in brain research* 154 (2006), pp. 15–32.

[36] C. Meli and A. Lansner. ‘A modular attractor associative memory with patchy connectivity and weight pruning’.
In: *Network: Computation in Neural Systems* 24.4 (2013), pp. 129–150.

[37] A. Sandberg, A. Lansner, K. M. Petersson and Ekeberg.
‘A Bayesian attractor network with incremental learning’.
In: *Network: Computation in neural systems* 13.2 (2002), pp. 179–194.

[38] C. Johansson and A. Lansner. ‘Towards cortex sized artificial neural systems’.
In: *Neural Networks* 20.1 (2007), pp. 48–61.

[39] F. Fiebig and A. Lansner.
‘A spiking working memory model based on Hebbian short-term potentiation’.
In: *Journal of Neuroscience* 37.1 (2017), pp. 83–96.

[40] N. B. Ravichandran, A. Lansner and P. Herman. ‘Learning representations in Bayesian Confidence Propagation neural networks’.
In: *2020 International Joint Conference on Neural Networks (IJCNN)*.
IEEE. 2020, pp. 1–7.

[41] N. B. Ravichandran, A. Lansner and P. Herman. ‘Brain-like approaches to unsupervised learning of hidden representations-a comparative study’.
In: *International Conference on Artificial Neural Networks*. Springer. 2021, pp. 162–173.

[42] N. B. Ravichandran, A. Lansner and P. Herman. ‘Semi-supervised learning with bayesian confidence propagation neural network’.
In: *arXiv preprint arXiv:2106.15546* (2021).

[43] N. Farahini, A. Hemani, A. Lansner, F. Clermidy and C. Svensson.
‘A scalable custom simulation machine for the Bayesian confidence propagation neural network model of the brain’. In: *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2014, pp. 578–585.

[44] A. Lansner, A. Hemani and N. Farahini. ‘Spiking brain models: computation, memory and communication constraints for custom hardware implementation’.

In: *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2014, pp. 556–562.

[45] D. Stathis, C. Sudarshan, Y. Yang et al.
‘eBrainII: a 3 kW realtime custom 3D DRAM integrated ASIC implementation of a biologically plausible model of a human scale cortex’.
In: *Journal of Signal Processing Systems* 92.11 (2020), pp. 1323–1343.

[46] Y. Yang, D. Stathis, R. Jordão, A. Hemani and A. Lansner.
‘Optimizing BCPNN Learning Rule for Memory Access’.
In: *Frontiers in Neuroscience* 14 (2020), p. 878.

[47] B. Vogginger, R. Schüffny, A. Lansner, L. Cederström, J. Partzsch and S. Höppner.
‘Reducing the computational footprint for real-time BCPNN learning’.
In: *Frontiers in neuroscience* 9 (2015), p. 2.