Design of Highly-Accurate and Hardware-Efficient Spiking Neural Networks

Chengcheng Tang, *Student Member, IEEE,* and Jie Han, *Senior Member, IEEE,*

***Abstract*—Spiking neural networks (SNNs) have emerged as a promising alternative to conventional artificial neural networks (ANNs) for energy efficient design. The rate encoded computation in SNNs utilizes a number of spikes in a time window to encode information. In a similar but different scheme, stochastic computing (SC) encodes binary numbers into and operates on random binary bit streams. In this article, we first propose a hardware-efficient design of stochastic SNNs that attains a high accuracy. As a network becomes more complex or the number of neurons increases, memory usage tends to grow exponentially. Inspired by the notion of binarized neural networks (BNNs), we further propose the design of a weight-binarized SNN (WB-SNN) to reduce the stringent requirement in memory usage in SNNs. Both designs take advantage of a priority encoder to transform the spikes between layers of neurons into index-based signals. In this way, it mitigates the issue of requiring significant hardware resources for a relatively low information density. Additionally, a WB-SNN based convolutional neural network (CNN) is designed for the recognition task of larger datasets. An implementation on field programmable gate arrays (FPGAs) for the Modified National Institute of Standards and Technology (MNIST) image recognition dataset shows that the stochastic SNN design achieves a higher accuracy with smaller hardware compared to other SNNs. Validated by using a multi-layer-perceptron and a CNN on the MNIST and CIFAR-10 datasets, respectively, the WB-SNN achieves a significant saving in memory with only a limited accuracy loss compared with its SNN and BNN counterparts.**

***Index Terms*—Spiking neural networks (SNNs), priority encoder, binarized weights, field programmable gate arrays (FPGAs), stochastic computing, convolutional neural networks (CNNs).**

I. Introduction

INSPIRED by neuroscience, a simplified model of the brain consisting of neurons and synapses proves effective in tackling difficult computing tasks and machine learning problems [1]. This simplified computational model as an attempt to exploit the structure of the human brain provides a basis for the so-called artificial neural networks (ANNs). In an ANN, the inputs to each neuron carry signals encoding the information of external data in a form of feature values, such as the pixels in an image [2]. The weights on different inputs mimic excitatory or inhibitory activations from other neurons. In each neuron, a weighted sum of the inputs is processed through an activation function and is produced as the output of the neuron. This output, also known as activation [3], acts as the inputs of other neurons.

Received 6 September 2024; revised 23 November 2024 and 27 February 2025; accepted 24 April 2025. The review of this article was arranged by Associate Editor Yang Yi. (Corresponding author: Jie Han.) The authors are with the Department of Electrical and Computer Engineering, the University of Alberta, Edmonton, AB T6G 1H9, Canada.

Digital Object Identifier 10.1109/TCASAI.2025.3569509.

Spiking neural networks (SNNs) process information or data transmitted from neurons to neurons through synapses. It has a significant potential to save energy and hardware usage compared with conventional ANNs [4]. SNNs, as its name implies, utilize spikes other than binary numbers to produce interpretable computation results. The encoding schemes, such as firing rate (or frequency) [5]- [6] or the firing latency (or order), enable the operation of SNNs to be event-driven, that is, neurons only have to update their states upon the arrival of spikes, thus saving energy. Another merit of spike-based computation is its potential to decompose costly arithmetic operations into repetitive simple tasks. One example is the neuromorphic chip, TrueNorth, which contains one million neurons working in parallel and each neuron uses a random number generator, an integrator and a threshold unit to instantiate the augmented Integrate-and-Fire (IF) model [7] without using multipliers. In addition, with the rate coding scheme, the commonly used IF model in SNNs is equivalent to the rectified linear unit (ReLU) function in ANNs [9]. This equivalence makes converting ANNs into SNNs very straightforward with nearly no accuracy loss when a proper weight and threshold balancing strategy is adopted [8]- [9].

Although SNNs are usually energy-efficient, they are not hardware friendly because of the large number of neurons and connecting synapses in the network. When implemented on field programmable gate arrays (FPGAs), a significant challenge arises from the limited availability of block-random-access-memories (BRAMs). This limitation constrains the number of neurons and synapses that can be implemented, because large-scale SNNs require extensive memory to store substantial synaptic weight matrices [10]. Multi-Layer Perceptron (MLP) models feature fully connected nodes, but they can pose scalability issues as the image resolution increases. In a fully connected (FC) layer, each neuron is connected via a synapse with every neuron in the previous layer. When the number of neurons in consecutive layers is approximately N, the number of synapses grows quadratically with respect to the number of neurons per layer, i.e., $O(N^2)$, which makes it costly in implementing large neural networks. Simulating and implementing SNNs usually require more computational resources than ANNs [11]. In addition, SNNs perform better on specifically designed neuromorphic hardware such as IBM's TrueNorth and Intel's Loihi, which are not as widely available as general computing platforms such as central processing units (CPUs) and graphics processing units (GPUs) [1]. These have become the major limitations in many SNN designs, prompting researchers to seek improvements.

Stochastic Computing (SC) operates on randomly gener-

ated binary bit sequences, wherein the probability of a '1' is employed to encode a number [12]. Utilizing the same input interpretation as in SC, we obtain what are known as stochastic SNNs. In this paper, a highly accurate stochastic SNN is designed with two unique features: a shared random number generator (RNG) for all input neurons, and reduced connections to subsequent neurons through a priority encoder (PE), thereby significantly saving hardware. To resolve the problem that several spikes may arrive at the same clock cycle in a hidden layer, a First-in, First-out (FIFO) and a Priority Resolving Circuit (PRC) are utilized so that no spike is discarded in the computation.

Fully-connected neural networks (FCNNs) consisting of an input layer, two hidden layers and an output layer are used for verification. The input layer converts the input data into spike trains using the same encoding method as in SC, but using only one RNG, so reducing the inefficiencies of traditional methods that require an RNG for each input. The output layer converts data back into the binary format and the classification result is determined by the neuron with the largest membrane voltage value. A binary tree (BT) method is employed to perform the comparison in this layer. Due to the inherent hardware limitations in FPGAs, the network architectures that can be implemented tend to be constrained in scale. This restricts the complexity of tasks that SNNs can perform on FPGAs, thus affecting applications in more complex scenarios.

As another efficient model, binarized neural networks (BNNs) employ binary values ($+1$ and -1) for both weights and activations [14], so the key arithmetic operation, multiply-and-accumulate can be implemented by a 1-bit XNOR-count operation [15]. Such simplifications enable considerable memory savings without incurring a significant loss in accuracy [15]. Inspired by the notions of BNNs, we further propose a weight binarized SNN (WB-SNN) by introducing binarized weights ($+1$ and -1) into SNNs. It differs from the so-called binary weight SNN (BW-SNN) [16], where the weights are -1, 0, and $+1$. In this scheme, a sign extension of the weights is required, thus incurring additional hardware costs. However, the weights in the proposed WB-SNNs are truly binary in $+1$ and -1 that can respectively be stored as 1 and 0 in the memory. Then, the binary weights can be accumulated by using counters during the IF process. In [17], a weight-threshold conversion method is utilized during the conversion of high-precision trained convolutional neural networks (CNNs) into SNNs with binary weights. In the WB-SNN model, however, binary weights are directly used and obtained in the training process. In [18], a method is presented for training SNNs with binary weights using Bayesian learning. However, it is focused on the software implementation of training without reaching a high accuracy. Instead, the efficiency of the WB-SNN is validated and enhanced by innovative hardware design, as discussed previously and implemented in FPGAs as a proof of concept.

Unlike MLPs, for which the input image is typically flattened into a one-dimensional vector, leading to a loss of spatial hierarchy, CNNs are designed to preserve and exploit the spatial hierarchy. This makes CNNs more effective for recognition tasks, especially in large datasets, where spatial relationships are important. In this work, a spiking CNN model is specifically designed in hardware for implementing the WB-SNN. It is shown that the WB-SNN is scalable and capable of working with large networks and datasets.

Some preliminary results have been reported in [19] and [20]. In [19], each result in the final layer was compared with others to determine the digit being recognized. However, the use of additional comparators resulted in unnecessary hardware overhead. In this paper, a binary tree method is implemented in hardware to improve the comparison process in the stochastic SNN design. In [20], only FCNNs are considered for the design of WB-SNNs, whereas in this paper, CNNs are designed to show the hardware efficiency of WB-SNNs in more complex networks and larger datasets. To the best of our knowledge, this is the first design of a WB-SNN-based CNN with a detailed hardware architecture.

A. Major Contributions

1) **Contribution 1:** A design framework for stochastic SNNs that utilizes only one RNG in the input layer. The number of connections to each neuron in the subsequent layer is reduced from N to $\log_2 N$ by using a PE. A novel binary tree method compares values in the output layer and obtains the recognition result with significantly enhanced hardware efficiency.
2) **Contribution 2:** Substantially reduced memory usage in the WB-SNNs due to the binary weights. Additionally, the XNOR-count operation in BNNs is replaced by PEs shared among all neurons in one layer for an even higher hardware efficiency. Different from the previous design of a simplistic WB-SNN based MLP, a WB-SNN-based CNN is specifically designed and tailored for handling recognition tasks with larger datasets.
3) **Contribution 3:** Implemented on FPGAs and verified using the Modified National Institute of Standards and Technology (MNIST) and CIFAR-10 datasets, the proposed designs show significant hardware savings over some state-of-the-art designs.

II. Preliminaries

A. Spiking Neural Networks (SNNs)

As illustrated in Fig. 1, SNNs imitate the bioelectric activities observed in biological systems. In an SNN, a neuron receives several channels of input spike trains from other neurons in the former layer through its synapses [12]. Each incoming spike will change the internal membrane potential of the neuron by an amount determined by the amplitude of the synapse weight. When this potential exceeds a predefined threshold V_{th}, an output spike is generated and the potential value is reset. The structure of SNNs is composed of several layers of such neurons that rely on these spikes to communicate and transmit information.

Among various neuron models, each of which uses a different function to describe the dynamics of neuron potential,

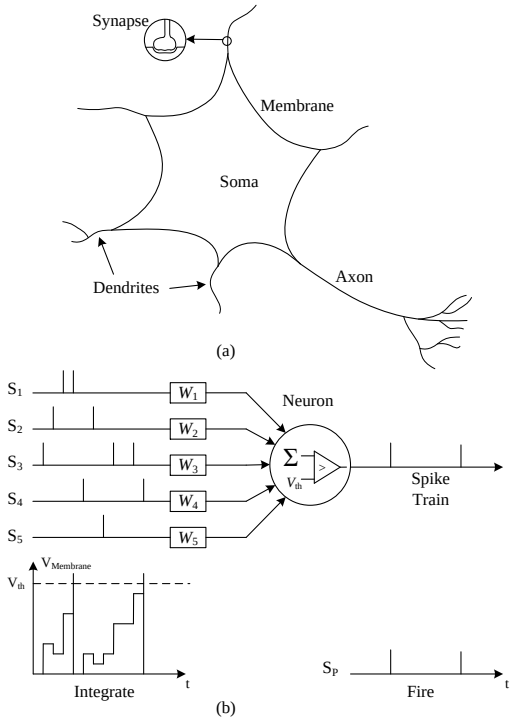

Fig. 1. Analogue between (a) a biological neuron and (b) a neuron in stochastic SNNs (with the spike train generation mechanism in the Integrate-and-Fire process).

the Integrate-and-Fire (IF) model [4] is the simplest and most commonly used one. Its behavior can be expressed as

$$v_i = \int \sum_j W_{i,j} \cdot S_j(t) dt, \tag{1}$$

where v_i is the internal membrane potential of neuron i in one layer and $W_{i,j}$ is the weight for the synaptic connection between neuron i in the current layer and neuron j in the previous layer. $S_j(t)$ is an impulse function given by a spike train of '0's and '1's. This equation essentially states that the membrane potential v_i is the integral over (continuous) time of the weighted sum of incoming spikes. In its discrete form, this relationship is described by [4]:

$$V_i\left(t+1\right) = V_i\left(t\right) + \sum_j W_{ij} \cdot S_j\left(t\right), \tag{2}$$

where $V_i\left(t+1\right)$ and $V_i\left(t\right)$ are the membrane potentials of neuron i at the next time step $(t+1)$ and the current time step (t), respectively. $\sum_j W_{ij} \cdot S_j\left(t\right)$ is the sum of the products of the synaptic weight W_{ij} between neuron j in the previous layer and neuron i, and the spike output S_j from neuron j at time t.

Although there are graded spikes (with either single- or multi-level magnitudes) in certain neuromorphic hardware like Loihi-2, SNNs typically utilize binary spikes to mimic the way biological neurons operate [13]. Thus, in this work, the computation in SNNs is rather simple because the neuron is activated by event-based spike trains so that the multiplication is transformed into integration. Therefore, no multiplier is needed in the hardware implementation. The variation in the neuron potential can be realized by an integrator that accumulates the synaptic weights on an event-triggered basis. It can also be viewed that SNNs use binarization in the neuron activation while the weight values are in the floating-point format. Moving one step forward toward binarizing both the weights and activations of neurons will lead to (fully) binarized SNNs. The computation is anticipated to be even simpler.

B. Binarized Neural Networks (BNNs)

The idea of BNNs was proposed in [15] as an effort to replace arithmetic operations with bit-wise operations. Before that, many efforts were made to reduce the precision of weights and it has been shown that the number of neurons is more important than the bit-width of weights [22]. In a BNN, the weights and activations are constrained to binary values, such as -1 and $+1$. Therefore, a single bit can be used to quantize both synaptic weights and neuron activations. For example: the bit value 1 represents $+1$ and 0 indicates -1. Thus, the multiplication is equivalent to an XNOR operation, as shown in Table I [14].

TABLE I
EQUIVALENCE OF MULTIPLICATION AND XNOR OPERATION IN BNNS

Multiplication		Results	XNOR		Results
-1	-1	1	0	0	1
-1	1	-1	0	1	0
1	-1	-1	1	0	0
1	1	1	1	1	1

In order to convert the real values in ANNs into this binary representation, several functions have been utilized, including the deterministic binarization function and stochastic binarization function [15]. The first one is known as $Sign(x)$ for the binarized weight or activation, x_b, given by,

$$x_b = Sign(x) = \begin{cases} -1 & x < 0, \\ +1 & otherwise, \end{cases} \tag{3}$$

where x is a real-valued variable as either a weight or an activation. In stochastic binarization, the variables get -1 or $+1$ depending on a probability determined by the hard sigmoid function $\sigma(x)$,

$$\sigma(x) = min(1, max(0, \frac{x+1}{2})). \tag{4}$$

The stochastic binarization needs to generate random bits to simulate probabilistic events, which would incur additional hardware. Therefore, we use the deterministic function to implement the binarization. However, the stochastic binarization function is used in the training process to approximate the derivative of binarization during back propagation.

III. SYNERGY BETWEEN SNNS AND BNNS

Integrating the concept of binarization in BNNs into SNNs can potentially reduce the significant memory demands by limiting all weights and neuron activations to single bits. Considering that there is no negative spike in biological neurons, we choose to encode these spikes as 0 and 1. Additionally,

weights are encoded as $+1$ and -1 to represent excitatory and inhibitory synapses, respectively. However, the neuron activations are encoded into $0/1$ to keep the event-driven feature of SNNs. Both encodings can be instantiated by just one bit in hardware, although they implement different meanings, as will be shown in the following sections. Under such encoding schemes, some of the commonly used techniques in ANNs can find their analogies in SNNs.

1) Batch Normalization + Binarization vs. IF neuron model: Batch normalization is a technique used in neural networks to standardize the inputs to a layer for each mini-batch, which is a subset of the training dataset used to train a model. It helps reduce the internal covariate shift and accelerate the training process [27], so it stabilizes the learning process and is considered essential in the training and inference of ANNs. The internal covariate shift here refers to a phenomenon in deep learning, in which the distribution of each layer's inputs changes during training; it may result in some issues, such as a slow convergence rate. For the IF neuron model, which deals with membrane potentials and spikes rather than continuous activation functions, the approach would focus on normalizing the membrane potentials across a network or batch of neurons. For a minimal batch of input activation values V_i, the basic equation for batch normalization is:

$$y_i = \frac{V_i - \bar{V}}{\sqrt{\sigma^2 + \epsilon}} \cdot \gamma + \beta, \tag{5}$$

where $\bar{V}$ and σ^2 are the mean and variance of the inputs, ϵ is a small constant value to avoid division by zero, γ and β are re-scaling and bias parameters that are learned during the training process, and y_i is the batch normalized input activation value. Once the training is completed, all these values stay constant in the inference process.

As per the encoding schemes considered for the proposed designs, all neuron activations are binarized to 0 or 1, i.e.,

$$y_i^b = \begin{cases} 1, & (y_i > 0) \\ 0, & (y_i \leq 0) \end{cases}. \tag{6}$$

Considering that all the variables in (5) take constant values during inference, we can merge the batch normalization and binarization into one step [27], such that the neuron activation is determined by

$$y_i^b = \begin{cases} 1, & (V_i > V_{th}) \\ 0, & (V_i \leq V_{th}) \end{cases}, \tag{7}$$

where V_{th} is also a constant. This process is similar to the IF neuron model in SNNs: whenever the membrane potential reaches the threshold V_{th}, an output spike is generated. When incorporating batch normalization in SNNs, the threshold V_{th} may be adapted based on the normalized membrane potential [6]. Therefore, it is determined by the following equations:

$$V_{th} = \sigma_{bat} \cdot \gamma + \beta, \tag{8}$$

$$\sigma_{bat} = \frac{\bar{V}}{\sqrt{\sigma^2 + \epsilon}} \tag{9}$$

where σ_{bat} represents the mini-batch standard deviation of the membrane potential.

2) $0/1$ Encoded Spikes vs. Dropout: Dropout is an effective approach to prevent a neural network from over-fitting [28]. It is done by randomly choosing a fraction of the neuron nodes and removing them from the network. The dropped nodes, along with their weights, are temporarily excluded during both forward and backward propagation. This random selection ensures that all neuron nodes are eventually considered and updated after a sufficient number of training iterations. Dropout helps prevent overfitting and ensures that the network does not become overly reliant on any specific node, thereby making the network more robust. Under the proposed scheme of using 0/1 encoded spikes, there is always a portion of neurons that generate no spike, i.e., remain inactive. These inactive neurons vary case by case for different inputs, so they can be considered as arbitrarily picked, similar to the working mechanism of dropout. Another benefit of this similarity is the sparsity that is brought into the regime as the inactive neurons will take no part in computation, resulting in a reduced computation cost.

Thus, the use of 0/1 encoded spikes aligns with the concept of dropout as applied in computational practice. While dropout improves the robustness of a neural network by making it less sensitive to specific weights of neurons, the inherent sparsity and randomness of spiking in SNNs can help the network avoid overfitting to noise in the training data, therefore enhancing the network's robustness and efficiency. On the other hand, the neuron activations are encoded to -1 or 1 in BNNs. Hence, all neurons are considered active and must be taken into account, which results in a more costly computation flow.

IV. A Design Framework for Stochastic SNNs

Leveraging the similarities between the encoding schemes of SNNs and SC, we propose an efficient hardware design for stochastic SNNs. To illustrate it, fully-connected neural networks are considered as they are versatile and widely used in many applications. The key components and the detailed design of each layer are discussed as follows.

A. Hardware Design of a PE with a PRC

As a basic and key unit used in the proposed SNN designs, a PE encodes the index of an activated bit with the highest priority among all inputs. In this way, the index of an active neuron (i.e., a neuron with an output spike) is identified in a group of neurons. The schematic and truth table of an 8-input priority encoder are shown in Fig. 2. Reserved for the case when there is no input spike, the least significant input D_0 is optional with no impact on the output. Thus, it is not instantiated in hardware. One useful feature of a PE is its ability to construct a large PE using smaller ones. Fig. 2(c) shows how a 16-input PE is constructed using two 4-input PEs. This feature provides considerable flexibility to make it fit into neuron layers of different sizes [20].

To locate all the input channels with logic-high signals, a PE works with a PRC [25], so the found indexes are sent to the output one by one. A PRC is designed to work with the PE, as shown in Fig. 3. Its main function is to avoid repeated

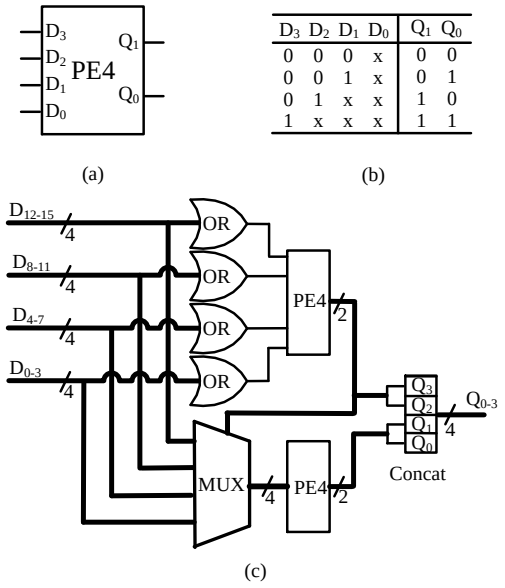

Fig. 2. (a) An 4 to 2 priority encoder (PE4). (b) The truth table. (c) A 16 to 4 priority encoder (PE16) constructed by two PE4s.

encoding of the same activated input bit. When the PE finishes encoding the bit with the highest priority, the PRC clears this bit so that the PE can move on to encode the bit with the next highest priority.

An 8-bit PRC design is shown in Fig. 3, where $load$, Q, and E are the inputs. Q is the output from the PE, indicating the index of the highest priority bit. E is the signal that captures spike pattern at one time step. The output D is sent to the PE aligned with this PRC. AS shown in Fig. 3, when the input signal $load$ is '1', indicating that this PRC is active, S_1 becomes '1'. The other selection signal, S_0, for each multiplexer in the PRC, is the output of the demultiplexer that processes the PE's output Q. Specifically, S_0 is set to '1' for the index corresponding to the current highest priority. Thus, if $S_0 = 0$, the output of this multiplexer will retain its previous value. Otherwise, the output of this multiplexer will be set to '0', indicating that the PRC clears the bit with the highest priority. In this manner, the '1' corresponding to the current highest priority is cleared. Additionally, if $load$ is '0', then D will be equal to the input E, indicating that this PRC is inactive under this circumstance. The truth table of these multiplexers is provided in Table II. For example, consider a spike pattern E represented as "11000001", where E_7, E_6, and E_0 are '1'. When the $load$ signal is '1', indicating that the PRC is active, the signal Q from the PE outputs the index of the highest priority bit. In this scenario, $Q = 7$. Consequently, the S_0 for the topmost multiplexer is set to '1', while it remains '0' for the others. With $S_1S_0 = 11$, this multiplexer for E_7 performs the selection operation, resulting in the output D_7 being '0'. In this manner, the current priority bit is cleared, and the signal D, in which now the priority bit is D_6, is sent to the PE to determine the index of this priority bit. This process continues until all the '1's in the spike pattern E has been processed.

Since the output of a PE is in the binary format, the bit width of the signal is reduced to $\log_2 N$ from an N-channel input. When the PE is placed between two layers of neurons in a network, its output can directly be utilized to find the memory location and thus, the signal width is also reduced from N to $\log_2 N$. The benefits include an increased throughput, and reduced signal width, circuit size and power consumption. Although it is important to consider the potential increase in the critical path length, this increase is not significant in this design because the PE is not very large.

TABLE II
TRUTH TABLE FOR THE MULTIPLEXERS IN THE PRC

S_1S_0	11	10	01	00
Output Value	0	D	E	E

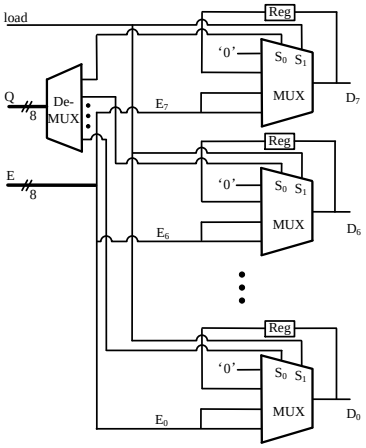

Fig. 3. An 8-bit PRC design.

B. Design of the Input and Hidden Layers

1) The Input Layer: The input layer takes external data, such as image pixel values, and then converts them into spike trains. In a spike train, each bit is generated by comparing the numbers from an RNG and an input channel, in a similar way to how a stochastic sequence is generated in SC. The sequence produced follows a Poisson distribution, a basic model of neuronal firing [23]. However, this model is not optimal for hardware implementation, as it requires one RNG for each input. One way to address this problem is to use the incremental accumulation of the input signal to map uniformly distributed random numbers to a cumulative distributed function (CDF) [24]. The input data, x_i, are placed on an axis where the interval lengths, corresponding to the data values, are accumulated into cumulative value F_i. To clarify, F_i is equal to x_i added to F_{i-1}, as shown as the inputs in Fig. 4 (a). For example, if $x_1 = 1$, $x_2 = 2$ and $x_3 = 3$, then $F_1 = 1$, $F_2 = 3$ and $F_3 = 6$. Then a random number would always fall into one of these intervals, namely $(0, 1]$, $(1, 3]$, $(3, 6]$. The probability that the random number falls into an interval is proportional to the value in the input data with the same index. If a random number R falls within the interval F_i, neuron i with the same index as F_i will generate a spike, namely '1',

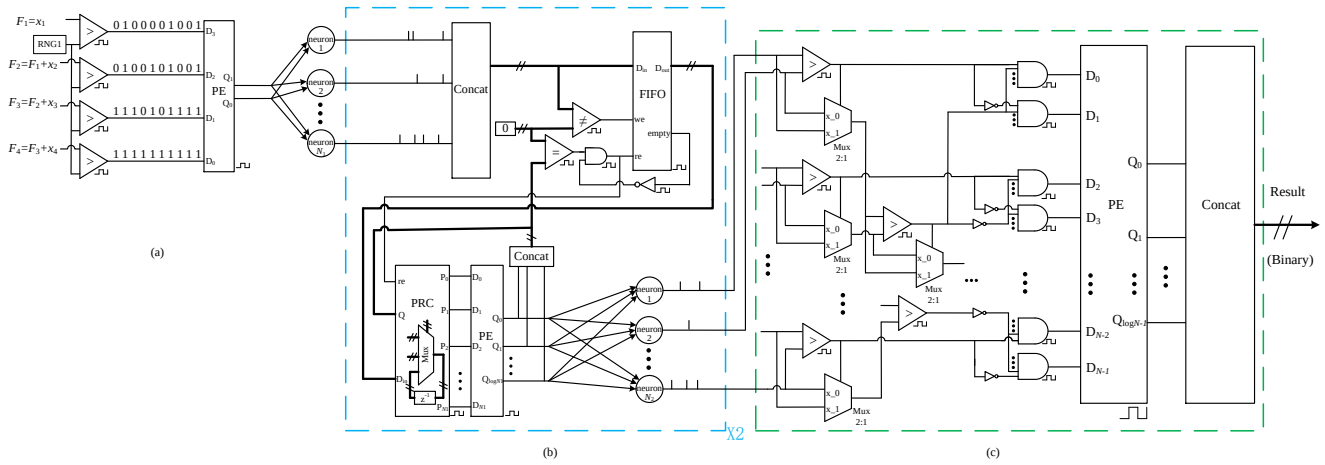

Fig. 4. The design of the stochastic SNNs. (a) The input layer that uses one RNG and a PE. (b) Two hidden layers (The second hidden layer is hidden and noted as $\times 2$). (c) The newly designed output layer.

for the input spike train of this neuron. Therefore, the input layer of the proposed SNN is implemented by comparing the CDF of input signals with a common RNG, as shown in Fig. 4(a). The resulting spike trains are forwarded to a PE and then the output is sent to the neurons in the subsequent layer. In this way, the output binary signal Q from the PE gives the index of the section in the CDF a random number falls into. Since the random number is uniformly distributed between one and its maximum value, Q follows the same distribution as the input, i.e.,

$$P(Q = \text{index}(x_i)) = \frac{x_i}{\sum_k x_k}, \tag{10}$$

where the variable k ranges over all the input elements that contribute to the CDF. This structure has three unique features. Firstly, the RNG is shared by every input channel. Inevitably, the required bit-width of the shared RNG is larger than that of the single RNGs in conventional spike train generation. For example, if there are q-channels of inputs and each of them is p-bit wide, the total sum across all channels can reach up to $q \times (2^p - 1)$, so we need a $p + \log(q)$ bit wide signal to avoid potential overflow. Nevertheless, hardware savings are still achieved compared with the use of multiple smaller RNGs because the overhead only increases logarithmically with the number of input channels. Secondly, by properly assigning the range of the random numbers (i.e., by setting it to be less than the maximum interval value, $F_{\max}$), it will always be located within one section of the CDF, which means that spikes are generated in every clock cycle of the digital system. In the input layer, as shown in Fig. 4(a), the sequence generated by the comparator for F_4 (leading to the input of the PE with the least priority) is a sequence of all "1"s. Hence, at least one spike is passed through the PE in every clock cycle. Lastly, by utilizing a PE, the number of connections to each neuron in the subsequent layer is reduced from N to $\log_2 N$ by converting the number of connections into the binary representation, leading to significant hardware savings. Since the outputs of the PE can be used as address signals by the succeeding neurons to find the corresponding weight during computation, it is convenient to utilize distributed memory on FPGAs to store weights for the neuron.

2) The Hidden Layers: A hidden layer receives spike trains from the previous layer and propagates these spikes to the next layer according to some predefined propagation rules. Unlike the input layer, only one spike needs to be processed in every clock cycle. Once the spike train departs from the input layer, the spikes on different channels might arrive at the same time. Therefore, a structure is designed for the hidden layers in stochastic SNNs to ensure that no spike is discarded, as shown in Fig. 4(b). The spike trains from the previous layer are first concatenated into a data bus and then sent to a FIFO. Whenever a spike appears on the data bus, the FIFO intakes the data on the bus and stores it in its queue. The output of the FIFO is connected to a PRC followed by a PE. Then the encoded results from the PE are sent back to the FIFO and PRC to release the next available data or clear the bit that has been encoded. Note that although two or more spikes can arrive at the same time, the spike trains are still sparse. The FIFO only accepts a signal that does not contain all zeros; when all the spikes in the data bus have been processed, a new read enable (re) signal is generated to release the next data. The PRC circuit is used to avoid repeated encoding of the same bit by collaborating with the PE. When the PE has encoded the bit with the highest priority, the PRC clears it for the PE to encode the next highest priority bit. By using this process, all spikes in the train are sequentially processed and propagated to the neurons in the next layer.

C. Design of the Output Layer

The function of the output layer is to produce computation results for an external system. As such, the output layer needs to convert the data back into the binary format. Thus, unlike the neurons in other layers, the neurons in the output layer integrate the incoming spike trains but no longer need to propagate them. When the stochastic SNNs are used for classification or identification applications, what matters the most is the comparison of values in the output layer. The

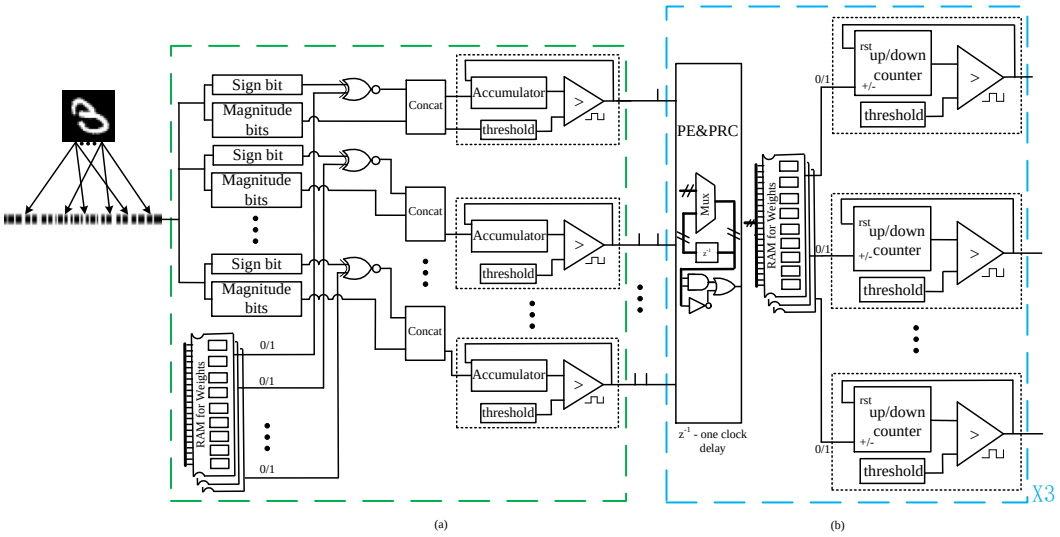

Fig. 5. The design of input and fully-connected layers in WB-SNN. (a) The input layer design, (b) Two hidden layers and the output layer (noted as $\times 3$ to save space).

neuron with the highest membrane voltage must be identified, as it gives the result of the classification. Hence, the output layer is designed for the data conversion, as shown in Fig. 4(c).

In this work, we implemented an approach, known as the binary tree method, for comparisons in the final layer. This has been incorporated into the hardware design, as depicted in Fig. 4(c). In this new design, pairs of membrane voltage values from different neurons are compared in the output layer. Each comparison is conducted using a comparator, followed by a 2-to-1 multiplexer (MUX) that selects the higher value from each pair. This selection process continues iteratively, comparing the higher values from the successive pairs, until only two values remain for the final comparison. The comparison results for each pair of neurons are then connected to an AND gate, determining if neuron i has the highest membrane voltage value. The outputs of N AND gates are subsequently processed by a PE, which converts the index of the neuron that has the highest membrane voltage into the binary format. This binary output provides the final identification result. However, in our previous design [19], we employed a relatively simple method for comparison. Before the PE executes its task, the neuron i is compared to all the other neurons in column i. The $N-1$ comparison results are connected to an AND gate to indicate if neuron i has the largest membrane voltage value. The number of comparison in the previous design is $O(N^2)$ because each neuron i is compared with all the other neurons in the output layer. However, with the BT method, the number of comparisons in the output layer is decreased from $O(N^2)$ to $O(N)$. This reduction will undoubtedly result in substantial hardware savings.

V. A Design Framework for WB-SNNs

To further improve the hardware efficiency of SNNs, the design principles of BNNs are incorporated into the so-called WB-SNNs. This section describes how different types of neuron layers are designed and connected in the WB-SNN. They can be used to build various types of neural networks as reusable units.

A. Input and Fully-connected (FC) layers

FC layers can be found in various types of ANN instantiations including the MLP. While the WB-SNN uses spikes as a medium in all its inner layers, it is not the case for the input layer. For instance, the system receives real pixel values in images as inputs. Nonetheless, this does not cause any further alteration to the design except that the counter is replaced with an accumulator to support the accumulation of real values in this layer. As can be seen in Fig. 5 (a), the pixel values are sent to the input layer one by one. Although causing extra latency, this operation makes this design hardware efficient. The sign bit of each value is XNORed with the binary weight stored in the memory. The output of the XNOR gate is then concatenated with the magnitude bits and sent to an accumulator for integration. The accumulation is then compared with the predetermined threshold for firing a spike. After this IF process, the neurons in the next layer will receive the generated spike train.

The design of FC layers in the WB-SNN is illustrated in Fig. 5 (b). Since the neuron output is either 0 or 1, all neurons from the previous layer are first connected to a PE&PRC block. This block will transfer the active or inactive status of each neuron into a binary number that indicates the neuron for which the input is 1. This binary number is then sent to the random access memory (RAM) where the weight is stored. The RAMs have multiple layers, each of which corresponds to one output neuron. Note that all RAMs share the same input address signal because each output neuron is connected with all input neurons. The only difference lies in the weight signal which is stored separately in the RAM. This feature also facilitates the parallel processing of the neurons. The output from the

RAM is then sent to the processing unit for integration and spike generation.

One may wonder how -1 is stored and realized in the WB-SNN. It is achieved by using an up/down counter, as shown in Fig. 5. When the input is 1, the counter counts up; when the input is 0, the counter counts down. In that way, we only need one bit to store the weight and at the same time, to achieve excitatory or inhibitory synapse behavior. Compared with conventional FC layers, only neurons with active outputs are selected and processed because of the use of PE&PRC. This feature does not only save energy but also reduces the inference latency.

B. The WB-SNN design for CNNs

The potential hardware savings in WB-SNNs suggest that they are suitable for implementations in more intricate architectures designed for classifying large datasets. CNNs have been chosen to fulfill this task. Because of their weight-sharing characteristics, CNNs can help reduce the memory requirements used for storing parameters. Typically, a CNN is composed of convolutional layers, pooling layers, and FC layers. While the design of FC layers is discussed in the previous subsection, we focus on the design details of the convolutional and pooling layers.

1) Design of the Convolutional Layer: As the core component in CNNs, the convolution layer makes it possible to significantly reduce the parameter size compared with the use of FC layers. At the same time, it allows for local features, such as edges and corners, to be detected [29]. In the convolutional layer, the kernel or filter is intricately designed to facilitate channel-wise mapping, that is, to process each channel of the input data independently, thus allowing the filter to effectively capture and emphasize the unique features across different channels. Unlike node- or pixel-wise mapping that focuses on individual pixels, channel-wise mapping processes all channels, e.g., for different color layers in an image, which enables the convolutional layer to extract more nuanced and complex features from the input data. Hence, the number of parameters is determined by three main factors: the size of each filter (i.e., its height and width), the number of filters used in the layer, and the number of input channels, as shown in Fig. 6. Following this design philosophy, the working mechanism of the convolutional layer in this WB-SNN is determined in a similar way, i.e., the data fetch and processing are conducted channel-wise.

Fig. 6 shows the convolution process for computing a pixel value (marked in blue) in the output feature map using a 3 by 3 kernel. Pixel values are specifically encoded as 0s and 1s. They are represented and managed within a CNN as multi-dimensional arrays, where each dimension corresponds to a channel encoding various feature values. These feature values are then processed through the network's layers. In other words, the converted pixel values at the same location in every channel are acquired and transmitted in parallel under the control of a convolution address generator, which will be introduced in the following paragraph. In the meantime, the weights belonging to one kernel are organized into and stored as a vector at the same address in the RAM.

A convolution address generator is designed to produce all necessary addresses and control signals. The working mechanism is presented in Algorithm 1. The w_addr_row and the w_addr_col indicate the row and column of the write address, respectively. r_addr_row and r_addr_col respectively denote the row and column of the read address. r and c are the row and column, respectively, of the kernel element. K equals to the kernel size and N is the size of the feature map. ker_addr refers to the kernel address. Moreover, $padding$ represents the padding signal, a flag when the kernel reaches the boundary of the feature map. In this address generator, two nested loops are used to generate the write address (w_addr) and read address (r_addr). The outer loop traverses all pixel positions in the output feature maps in a sequential manner. The inner loop uses the outer loop position (indicated by the row and column numbers) as the center and traverses all its neighboring positions defined by the kernel's dimensions to perform the convolution. Each channel of pixels in the input feature map contributes to calculating the neighboring pixels in the output feature map by convolving with the kernel at various positions of the feature map matrix. To facilitate this, a kernel address (ker_addr) is generated. Multiplexers use the kernel address to select the corresponding weight value from the kernel at the right position. Once the appropriate weight value is selected by the MUX, it is applied to the corresponding pixel value in the input feature map. This involves performing the multiplication of the selected weight value with the input pixel value.

The PRC&PE block takes the encoded pixel values from the input feature maps and indicates which input neuron gets a pixel value 1, i.e., the neurons that elicit spikes. This information is then forwarded to the RAM to find the corresponding kernel weight vector. Upon arrival, the kernel address signal is used to select the corresponding weight from the kernel weight vector. Subsequently, this weight is sent to the integration and compare models to compute the pixel values in the output feature maps. In the output feature maps, computing a single pixel value requires convolution with data from all channels of the input feature maps. In this process, the output from the PE is compared with 0. Only when spikes from all channels have been processed, is a move forward (mov_fwd) signal generated. This signal prompts the convolution process to advance to the next stage. It is also worth mentioning that a padding signal $padding$ is also produced by the convolution address generator. Padding is a technique used to adjust the size of the input feature map. Then, the convolution operation outputs a feature map that retains the same dimensions as the input feature map. Whenever it finds that the read address is outside the bounds of the feature map, an active high is triggered and forwarded to a multiplexer to select an all zero vector for the input of PRC&PE. Then, the output of the PRC&PE is also all zero to drive the mov_fwd signal to skip convolution at this padded position. Note that this design is equivalent to zero-padding in a convolution.

2) Design of the Pooling Layer: A pooling layer is commonly used in CNNs to reduce the size of feature maps and therefore the size of weights as well. In this layer, feature values in a small region are summarized and squeezed into one value. It also increases the robustness of the NN model as

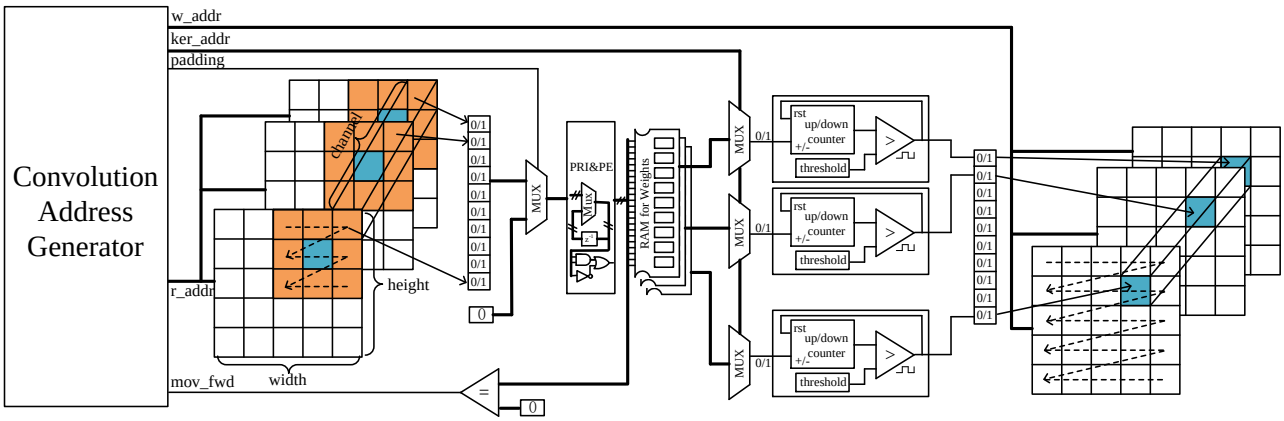

Fig. 6. The design of the convolution layer in the WB-SNNs.

variations in feature maps are filtered to some extent. There are usually two types of pooling in practice: average-pooling and max-pooling. In a spike-based system, max-pooling is preferred because spikes are represented in the binary form. Compared to other pooling methods, such as average-pooling that computes the average of activations in a pooling region, max-pooling is more effective and suitable for the binarized feature maps because average-pooling could result in non-binary outputs and cause additional operations. For example, if there is an odd number of '1's in a 2×2 pooling filter, the result could be 0.25 or 0.75, neither of which is binary.

Algorithm 1 Convolution Address Generator

Input: Write address row: w_addr_row; Write address column: w_addr_col; Kernel size: K;
Output: Write address: w_addr; Read address: r_addr; Kernel address: ker_addr; Padding flag: $padding$;
Begin
for all $w\_addr\_row = 0 : N-1$ **do**
  for all $w\_addr\_col = 0 : N-1$ **do**
    for all $r, c = K : 1$ **do**
      $r\_addr\_row = K - r + w\_addr\_row$;
      $r\_addr\_col = K - c + w\_addr\_col$;
    end for
  end for
end for
r_addr =concat(r_addr_row, r_addr_col);
w_addr =concat(w_addr_row, w_addr_col);
ker_addr =concat(r, c);
if $(0 \leq r\_addr\_row < N)||(0 \leq r\_addr\_col < N)$ **then**
  $padding = 1$;
end if

The designed structure of a 2×2 max-pooling layer is illustrated in Fig. 7. This design aims to reduce the dimensionality of the feature map while preserving significant features. The inputs to this layer include feature values from the previous layer as a feature map value ($feature\_in$), a validation signal ($valid\_in$), and an address signal ($addr\_in$). The output signals are the same, except that the size of the feature map is reduced and the values are filtered. As can be seen from Fig. 7, the key components include a FIFO block, counters, and various logic blocks that facilitate the processing of data streamed in a row-major order. The feature values are first ORed to determine the maximum between the current value and the previous value, delayed by one clock cycle (z^{-1}). The resulting maximum is then sent to the FIFO when we is '1', effectively performing a row-wise max-pooling operation. After row-wise pooling, the data is managed by a FIFO system, which handles the synchronization and temporary storage of data for column-wise pooling.

For row-wise pooling, the data processed by the OR gate is directed into the FIFO. Here, a counter which follows the AND gate, becomes responsive when $valid\_in$ is active and the least significant bit (LSB) of $addr\_in$ is set. The bit-width of the counter matches the row size of the feature map, and the MSB of the output controls the FIFO's write and read enable signals (we, re). The processing or storage of data from the upper or lower half of the feature map filter is triggered by the value of the MSB. The MSB of the counter output (driven by $valid\_in$ and $addr\_in$) plays a crucial role in determining when to switch between processing the upper half and the lower half of the input data. When the MSB is '0', the system processes or stores data from the upper half of the pooling filter; when it is '1', it switches to the lower half of the pooling filter. This differentiation is crucial for managing how data is pooled from different regions of the feature map.

The output of the FIFO is combined with the previous output through an OR gate, along with a one-clock cycle delay, to implement column-wise max-pooling. The pooling operation is synchronized with the activation of the FIFO's re signal, from which the $valid\_out$ signal also originates, ensuring that the generated output addresses ($addr\_out$) correspond to the newly pooled feature values. The design ensures that the entire feature map is processed effectively in two stages: first row-wise and then column-wise, utilizing the FIFO to temporally separate and then combine the data for maximum pooling operations.

3) Other Miscellaneous Designs — Using A Convolutional Layer As A Large FC Layer: The FC layer can be made

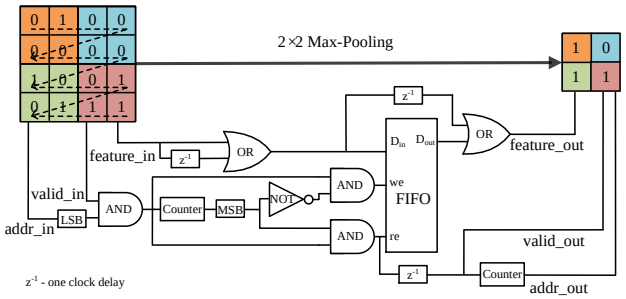

Fig. 7. Hardware design of a 2×2 max-pooling layer.

arbitrarily large in theory as a large PE can be constructed using smaller PEs in a cascaded way. However, it is not recommended to build a very large PE (with several thousands of inputs, for example) because it not only requires large hardware but also significantly increases the length of its critical path. On the other hand, the convolutional layer is organized in such a way that it uses one channel as an input unit. To be able to calculate one value in the output feature map, several channels of input data in the feature map are processed sequentially. If the multi-channel data are considered as a part of a single FC layer, a convolutional layer can be configured to perform the same functions as the FC layer. All we need to do is to partition data in the FC layer into several channels and reuse the control signals in the convolutional layer to perform computation between different channels.

VI. Hardware Implementation and Performance Evaluation

To verify these two designs, we implemented them on FPGAs as configurable units. Two widely used NN models, the MLP and CNN, were tested and its performance on inference accuracy and hardware utilization were evaluated on the MNIST [30] and CIFAR-10 [31] datasets, respectively. The training and inference processes with two datasets have been conducted using Pytorch for the two datasets. The FPGA platform adopted in the experiments is the Xilinx Virtex7 XC7VX485T board. The results were compared with the state-of-the-art designs from the literature.

A. The MLP Model

The MLP is one type of classical feedforward ANNs. It usually consists of several fully connected layers and is widely benchmarked for small image classification datasets such as the MNIST. For hardware implementation, MLP models are considered for both the stochastic SNN and WB-SNN to demonstrate their capabilities in recognizing the MNIST dataset. In our experiments, these models were first trained on a GPU platform and then implemented on the FPGA. Different from the preliminary results in [19], hardware implementations of the stochastic SNNs with the new output layer design were conducted for several MLP networks, with sizes of 784-N-N-10, i.e., one input layer with 784 nodes, two hidden layers with N nodes each and one output layer with 10 nodes. Four different networks with a hidden layer of 63, 127, 255 and 511 nodes, were evaluated. Note that the size of hidden layers we chose is always equal to 2^M-1 with a chosen integer M. That is to reserve an empty space in the PE for the all-zero input. This is one significant feature in both the stochastic SNN and the WB-SNN.

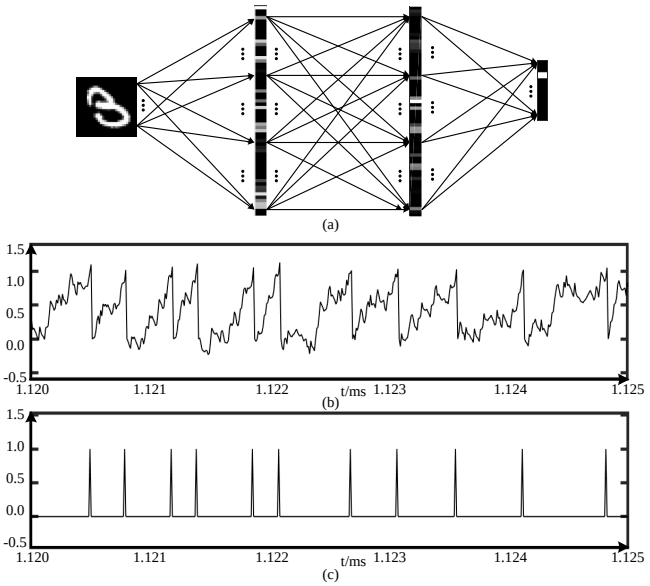

Fig. 8. Simulation process for the recognition of an image of digit 3. (a) A snapshot of the inference process (neuron/pixel intensity as shown in the grey stripes). (b) The membrane potential of a neuron, and (c) the generated spike stream. (All the values in this figure were captured during the real-time inference simulation. However, due to space limitations, only a portion of them is presented here for illustration.)

The inference on 10,000 images in the MNIST dataset was performed to record the accuracy and testing process. Fig. 8(a) shows a snapshot of the inference process for the recognition of the digit 3. The grey value represents the intensity of a neuron or a pixel, with a brighter level standing for a stronger signal intensity. The membrane potential in a neuron and the generated spike stream in time are presented in Fig. 8(b) and (c), respectively. They also showcase the IF process in a neuron. However, a neuron is not always active during inference. It remains silent for a certain amount of time and does not elicit spikes. That is why we employ PEs to obtain the indexes of '1's and get rid of all the '0's. It can be seen in Fig. 8(a) that the number in the input image has been correctly identified, i.e., the 3rd neuron in the output layer is the brightest.

The hardware costs of the MLP models with selected sizes and their inference accuracy are summarized in Table III. Compared to the previous design in [19], as indicated in Table III, the required number of look-up-tables (LUTs) decreases from 16,555 to 14,692, 24,089 to 22,226, and 39,015 to 37,148 when the hidden layer has 63, 127, and 255 nodes, respectively. This reduction in the required number of LUTs translates to an improvement in hardware efficiency: about 11.25% for a hidden layer of 63 nodes, approximately 7.73% for a size of 127, and around 4.79% for a size of 255. At the same time, the usage of Flip-Flops (FFs) and BRAMs remains

the same. Most importantly, the inference accuracy stays high. It is worth mentioning that the recognition rate in terms of processing time per each processed image of the stochastic SNN and the WB-SNN on FPGAs, are 100 μs and 5 μs, respectively. We have previously conducted a comparison with prior art in stochastic SNNs in our prior work, as documented in [19]. It shows that the design achieves a higher accuracy than the state-of-the-art implemented on FPGAs with a faster recognition rate. In this paper, with the implementation of the binary tree method in the output layer, the overall hardware cost of the design has been reduced. Moreover, a comparison of this design with some previous ones in stochastic SNNs is presented in Table IV. The results indicate that the stochastic SNN with the BT method achieves a higher accuracy and, mostly, a faster recognition rate in terms of processing time per image. While the processing time per image for this stochastic SNN design does not surpass that reported in [34] or the WB-SNN design on MLP [20], it achieves higher accuracy. Additionally, the precision of the weights is typically greater than 8 bits in a fixed-point format as can be seen in Table IV, resulting in substantial memory usage. The weight precision of the stochastic SNN using the BT method is the highest in this comparison because we aim for a higher recognition accuracy. Although the hardware utilization of stochastic SNNs does not show a clear advantage over the other designs, this design achieves a higher recognition accuracy and a fast processing rate in terms of time per image, hence a good design trade-off.

TABLE III
HARDWARE UTILIZATION AND INFERENCE ACCURACY OF DIFFERENTLY SIZED MLP MODELS

Structure Size	LUTs	FFs	BRAMs	Inference Accuracy	
784-63-63-10	16555 (5.45%)	20347 (3.36%)	101.5 (9.85%)	97.37%	[19]
	14692 (4.84%)				With BT method
784-127-127-10	24089 (7.93%)	24982 (4.11%)	199.5 (19.37%)	97.88%	[19]
	22226 (7.31%)				With BT method
784-255-255-10	39015 (12.85%)	34217 (5.64%)	395.5 (38.4%)	98.24%	[19]
	37148 (12.22%)				With BT method
784-1023-1023-10	24784 (8.16%)	14603 (0.81%)	56.5 (5.49%)	97.97%	WB-SNN [20]

LUT: look-up-table; FF: flip-flop;
BRAMs: block-random-access-memories (18 Kbits each)

We have included one of the WB-SNN results alongside the MLP model in the stochastic SNNs using the BT method in Table III to facilitate a comparison between stochastic SNNs and WB-SNNs. It is worth noting that the hardware utilization is higher than the WB-SNN design for the MLP model, as indicated in Table III, because 32-bit long fixed-point data format was used for the weights in the stochastic SNNs. In the meantime, the inference accuracy of the WB-SNN is lower than its full-precision counterpart. As shown in the last row of Table III, the accuracy of WB-SNN with 1023-node hidden layers is higher than the 127-node full-precision model while requiring less hardware. Therefore, there is a trade-off between the hardware utilization and inference accuracy.

TABLE IV
HARDWARE UTILIZATION AND RECOGNITION ACCURACY COMPARED TO THREE OTHER SNN DESIGNS

	Han et al., [32]	Gupta et al., [33]	Liang et al., [34]	Stochastic SNN with BT method
Platform	Xilinx ZC706	Xilinx XC6VLX240T	Xilinx Virtex7 VC7VX485T	Xilinx Virtex7 XC7VX485T
Structure	784×1024 ×1024×10	784 × 16	784 × 512 × 10	784×255 ×255×10
Weight Precision	16-bit fixed-point	24-bit fixed-point	8-bit fixed-point	32-bit fixed-point
LUT	5381	56230	16324	37148
FF	7309	23238	11612	34217
BRAMs	40.5+ external DDR	16	-	395.5
Accuracy	97.06%	-	96%	98.24%
Time/Image	6210 μs	500 μs	2.8 μs	100 μs

B. Training the WB-SNN

In order to attain a high accuracy while keeping the weights binarized, some common training tactics in BNNs are applied in the training of the WB-SNN CNN models. For example, an L-2 norm regularization $R_2(w) = (1 - |w|)^2$ is added to the total loss function so that the weight value, w, that diverges from 1 or -1 will be penalized [35]. The augmented loss function, $J(\mathbf{W}, \mathbf{b})$, is constructed from the original loss function, $L(\mathbf{W}, \mathbf{b})$, as

$$J(\mathbf{W}, \mathbf{b}) = L(\mathbf{W}, \mathbf{b}) + \lambda \sum_i R_2(\mathbf{W_i}), \qquad (11)$$

where $\mathbf{W}$ and $\mathbf{b}$ represent weights and bias terms, respectively, $\mathbf{W_i}$ contains the weights at the ith layer, and λ is a parameter that is used to adjust the ratio of regularization applied to the total loss. It is selected to be 0.001 in our experiments. Other training settings are summarized in Table V. We used a spike-based learning approach that incorporated the temporal and binary characteristics of spiking neural activities into the training pipeline. Moreover, the Adam optimizer is implemented as it is commonly employed for model training and widely favored for deep training in PyTorch, especially when dealing with complex models. Furthermore, it tends to outperform other optimization methods and the algorithm is available as a built-in package [21]. Lastly, the activations and the weights of neurons are trained to be 0, 1 and -1, 1, respectively. The hard sigmoid function in (4) (Section II.B) is used to provide a smooth transition during training,

before another binarization method (e.g. the sign function) is applied. The sign function is used to binarize weights during the forward propagation, while activations are thresholded to 0 or 1 when they are smaller or larger than 0. In the backward propagation, gradients are allowed to pass only for inputs in the range [0, 1] to circumvent the non-differentiable nature when the derivative does not exist. For SNNs, the derivative does not exist at the spikes, and it is zero everywhere else.

TABLE V
TRAINING SETTINGS OF THE WB-SNN MODEL

Number of epochs	50
Batch size	100
Batch normalization momentum	0.1
Learning rate	0.001
Loss function	CrossEntropyLoss
Optimizer	Adam

C. The Convolutional Neural Network (CNN) Model

For the MLP model, it has been shown that the WB-SNN design does not only achieve higher hardware efficiency but also maintains good inference accuracy [20]. To expand the applicability of WB-SNNs, we consider CNNs, another type of ANN instantiations that is able to harvest good accuracy on large datasets with a reduced parameter size. Using the proposed CNN design on the same FPGA, a performance comparison with prior art was carried out for the CIFAR-10 dataset with three channels of 8-bit RGB data for 32×32 pixels in an input image. This CNN model is trained by using PyTorch.

The structure of the CNN model is inspired from the VGG model in [41]. We use three convolution modules, each has two sequential convolution layers with a kernel size 3×3 and stride 1, followed by one 2×2 max-pooling layer. Zero padding is employed to keep the image size the same in one module. The channels we choose for the convolution layers in these modules are 63, 127 and 255, respectively. Following the convolution module, there are three fully connected layers that contain 504, 504 and 10 nodes, respectively. 504 is chosen to reuse the PE64 module in the fully connected layer (504=8×(64−1)). Let xCy denote the convolution layer, where x is the channel size and y is the stride. The kernel size is omitted because a 3×3 kernel is used for convolution, unless otherwise noted. The max-pooling layer is denoted by MPx, where x is the kernel size. The fully connected layer is denoted by FCx, where x is the node size. The input data has three channels with the size of 32×32 pixels. Therefore, the structure of this CNN is 3×32×32-63C1-63C1-MP2-127C1-127C1-MP2-255C1-255C1-MP2-FC504-FC504-FC10.

The convolution process involves taking a filter (or kernel) and passing it over the feature map of an image, and transforming the feature map based on the filter's parameters. The following formula is used to calculate the values Y in a subsequent layer's feature map, with the input image denoted as X and the kernel as W_b. The indexes of output channels, rows and columns of the resulting matrices are marked as n, i and j, respectively. The indexes of input channels, rows and columns of the resulting matrices are marked as d, x and y, respectively.

$$Y[n,i,j] = \sum_{d=0}^{D-1}\sum_{y=0}^{K-1}\sum_{x=0}^{K-1} W_b[n,d,2-x,2-y]\cdot X[d,i+x,j+y], \tag{12}$$

where W_b is the binarized weight, and D, K refer to the input layer channel size and the kernel size, respectively. As per Eq. (12), once the filter is positioned over the selected pixels in one channel, each weight value from the kernel is multiplied with the corresponding pixel value from the feature map. In the end, all the products will be summed up for the channel, and this summation will be placed in the corresponding position within the output feature map.

This spiking CNN model is trained on the GPUs first and then implemented on an FPGA platform. The CIFAR-10 dataset has been selected for the evaluation to verify that the designed WB-SNN framework can work with a larger dataset as well. By using the VGG model and the parameters specified in Table V, a training accuracy of 88.68% is achieved. After applying the weights for each synapse obtained from the training process into inference, the resulting accuracy is 79.85%. Fig. 9 shows the overall structure of this CNN model and the general inference process. As can be seen in Fig. 9, this network comprises six convolutional layers and three fully connected layers, with pooling layers interspersed between every two convolutional layers. The input of the inference in each channel is a true grey-level image. Following the convolution and comparison processes, the activations of the neurons are binarized (to 0 or 1), as well as the trained weights (to -1 and 1). In Fig. 9, how the feature map looks like during the inference is illustrated for a test image, where a white pixel denotes the neuron that elicits a spike while a black pixel indicates that it does not. We can also see that considerable numbers of neurons remain silent during the inference, which helps to save energy and reduce inference time.

The BRAM utilization with this more complicated structure is 134 units (with 18 Kbits in each unit), which is almost a quarter of the usage in the stochastic SNNs of a MLP model with 255 neurons in the hidden layer. As a requisite for more comprehensive assessment, the performance of this design is compared with some related work in the literature for a better evaluation. We selected both SNN and BNN designs, as well as one general CNN model. The results are shown in Table VI. It can be seen that the proposed design is more efficient in terms of hardware usage, that is, it uses the least amount of hardware resources to reach a comparable accuracy on the CIFAR-10 dataset. For example, the WB-SNN design only uses 21% of LUTs, 7% of FFs and 61% of BRAMs required by the model in [39] with an accuracy drop of only 0.74%. It can be seen that the hardware utilization of the WB-SNN is also better than the other BNN counterparts. This design only requires about $1/5$ of the LUTs compared to other designs, while the BRAM utilization averages at 85% of what is typically used in others. Hardware efficiency improves even more significantly compared to other SNN counterparts. The WB-SNN design

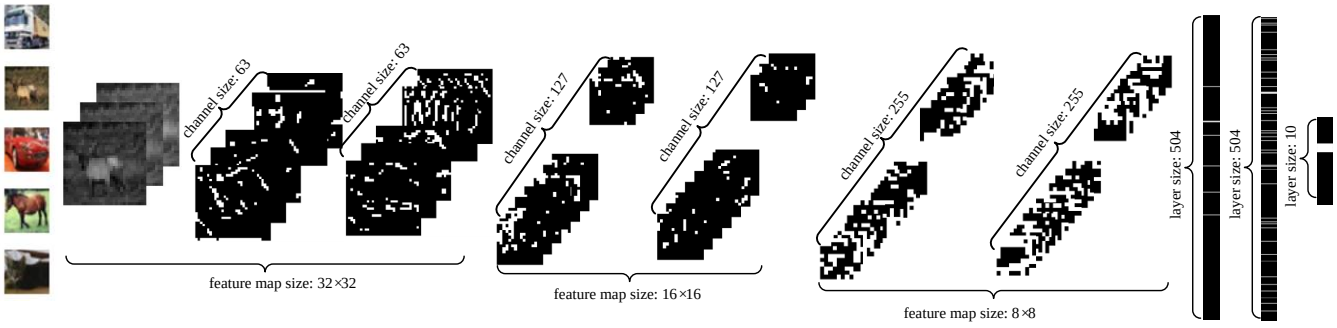

Fig. 9. Architecture of the WB-SNN for CNN implementation. (All the grey level values in this figure were captured during the real-time inference simulation. However, due to space limitations, only a portion of them is presented here for illustration.)

TABLE VI
COMPARISON OF THE WB-SNN WITH OTHER SNN AND BNN DESIGNS ON CIFAR-10 DATASET

Design Source	Framework	Platform	Structure	Bit Width	LUTs	FFs	BRAMs	Accuracy
[42]	General CNN	Zynq 7Z020	AlexNet	16-bit	36,278	10,357	75	74.2%
[43]	SNN	Virtex UltraScale+ XCVU13P	AlexNet	6-bit	48,000	50,000	–	80.6%
[38]	SNN	Virtex UltraScale+ VCU118	28C1-96C1-256C2-384C1-384C2-256C1-FC2048-FC2048-FC10	8-bit	386,000	–	969	81.8%
[39]	BNN	Spartan XC7S50	64C1-64C1-MP2-128C1-128C1-MP2-256C1-256C1-MP2-FC512-FC512-FC10	1-bit	53,200	106,400	280	80.59%
[40]	BNN	Zynq 7000 SoC ZC706	64C1-64C1-MP2-128C1-128C1-MP2-256C1-256C1-MP2-FC512-FC512-FC10	1-bit	46,253	–	186	80.1%
this work	WB-SNN	Virtex-7 XC7VX485T	63C1-63C1-MP2-127C1-127C1-MP2-255C1-255C1-MP2-FC504-FC504-FC10	1-bit	10,966	6,862	134	79.85%

merely uses 2.84% of LUTs and 13.83% of BRAMs required by the model in [38]. The only drawback is that the classification accuracy is slightly lower. However, this limitation is not a result of the hardware design itself. It can be effectively addressed and potentially improved through the application of specific training techniques. Overall, the proposed WB-SNN shows an encouraging advancement on hardware efficiency that is beneficial and essential for implementations on resource constrained devices.

VII. CONCLUSION

SNNs closely emulate a biological system and typically operate with stochastic sequences and real-valued weights. This approach can lead to increased latency and demand extensive memory usage. To address these limitations, we propose stochastic SNNs using PEs for hardware efficiency. The stochastic SNNs achieve high recognition accuracy, although the memory requirements remain high for the real-valued weights. To mitigate this issue, we further incorporate binarized weights into the design of SNNs, leading to the so-called WB-SNNs. This article presents a design framework for WB-SNNs that exploits features from both SNNs and BNNs. The weights in a WB-SNN are binarized and the PE is employed as a fundamental unit to build the constituent layers in the neural network. The WB-SNNs are capable to achieve a high hardware efficiency with minimal memory requirements. Applications of the CNN models for the CIFAR-10 datasets show that the WB-SNN substantially reduces the BRAM usage by up to 86.17%, without significantly compromising the recognition accuracy.

In conclusion, the stochastic SNN is best suited for applications requiring high accuracy, whereas the WB-SNN framework presents a significant advancement in efficient hardware design of neural networks. The latter is potentially useful in various applications for which hardware efficiency and accuracy are of paramount importance. Future work will address the scalability issue, common in neural network applications, and focus on optimizing hardware designs for implementation on application-specific integrated circuits (ASICs) and advanced neuromorphic hardware platforms.

REFERENCES

[1] M. Davies *et al*, "Loihi: a neuromorphic manycore processor with on-chip learning," *IEEE Micro,* vol. 38, no. 1, pp. 82–99, Jan. /Feb. 2018.
[2] Braspenning PJ, Thuijsman F, Weijters A., "Artificial neural networks: an introduction to ANN theory and practice," *Springer Verlag,* Heidelberg, 1995.

[3] Rasamoelina AD, Adjailia F, Sinčák P, "A review of activation function for artificial neural network," in *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI),* pp. 281-286, 2020.

[4] M. Bouvier, A. Valentian, T. Mesquida, F. Rummens, M. Reyboz, E. Vianello, and E. Beigne, "Spiking neural networks hardware implementations and challenges: a survey," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 15, no. 2, pp. 1-35, 2019.

[5] H. Tang, H. Kim, H. Kim and J. Park, "Spike counts based low complexity SNN architecture with binary synapse," *IEEE Trans. Biomed. Circuits Syst.,* vol. 13, no. 6, pp. 1664-1677, Dec. 2019.

[6] P. U. Diehl and M. Cook, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Front. Comput. Neurosci.,* vol. 9, pp. 99, Aug. 2015.

[7] F. Akopyan et al., "TrueNorth: design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.,* vol. 34, no. 10, pp. 1537-1557, Oct. 2015.

[8] Y. Cao, Y. Chen and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *International Journal of Computer Vision,* vol. 113, no. 1, pp. 54-66, 2015.

[9] P. U. Diehl, D. Neil, J. Binas, M. Cook, S. Liu and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *Proc. 2015 Int. Joint Conf. on Neural Networks (IJCNN),* Killarney, Ireland, pp. 1-8, 2015.

[10] Pham QT, Nguyen TQ, Hoang PC, Dang QH, Nguyen DM, Nguyen HH, "A review of SNN implementation on FPGA," in *2021 international conference on multimedia analysis and pattern recognition (MAPR),* pp. 1-6, 2021.

[11] Michael Pfeiffer and Thomas Pfeil "Deep Learning With Spiking Neurons: Opportunities and Challenges," *Front. Neurosci.*, Vol. 12: 774, 2018

[12] Yidong Liu, Siting Liu, Yanzhi Wang, Fabrizio Lombardi and Jie Han, "A Survey of Stochastic Computing Neural Networks for Machine Learning Applications," *IEEE Transactions on Neural Networks and Learning Systems,* vol. 32, no. 7, pp. 2809 - 2824, 2021.

[13] Orchard G, Frady EP, Rubin DB, Sanborn S, Shrestha SB, Sommer FT, Davies M., "Efficient neuromorphic signal processing with loihi 2," in *2021 IEEE Workshop on Signal Processing Systems (SiPS),* pp. 254-259, Oct 19, 2021.

[14] T. Simons and D.-J. Lee, "A review of binarized neural networks," *Electronics*, vol. 8, no. 6: 661, 2019.

[15] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1," 2016, [online] Available: https://arxiv.org/abs/1602.02830.

[16] P. -Y. Chuang, P. -Y. Tan, C. -W. Wu and J. -M. Lu, "A 90nm 103.14 TOPS/W Binary-Weight Spiking Neural Network CMOS ASIC for Real-Time Object Classification," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1-6, 2020.

[17] Yixuan Wang, et al. "Deep spiking neural networks with binary weights for object recognition," *IEEE Transactions on Cognitive and Developmental Systems,* 13.3, 514-523, 2020.

[18] Hyeryung Jang, Nicolas Skatchkovsky, and Osvaldo Simeone. "BiSNN: training spiking neural networks with binary weights via bayesian learning," in *2021 IEEE Data Science and Learning Workshop (DSLW),* pp. 1-6, 2021.

[19] C. Tang and J. Han, "Design and Implementation of a Highly Accurate Stochastic Spiking Neural Network," in *Proc. 2021 IEEE Workshop on Signal Processing Systems (SiPS)*, Coimbra, Portugal, pp. 1-6, 2021.

[20] Chengcheng Tang and Jie Han, "Hardware Efficient Weight-Binarized Spiking Neural Networks," in *Design, Automation and Test in Europe Conference (DATE 2023)*, Antwerp, Belgium, April 17-19, 2023.

[21] Kingma DP., "Adam: A method for stochastic optimization," *arXiv [preprint],* arXiv:1412.6980, 2014.

[22] H. Qin et al., "Binary neural networks: A survey," *Pattern Recognition*, vol. 105, pp. 1-14, 2020.

[23] P. U. Diehl, M. Cook, M. Tatsuno, and S. Song, "Unsupervised learning of digit recognition using spike-timing-dependent plasticity," *Front. Comput. Neurosci.*, vol. 9: 99, 2015.

[24] M. Alawad, H. Yoon and G. Tourassi, "Energy efficient stochastic-based deep spiking neural networks for sparse datasets," in *Proc. 2017 IEEE Int. Conf. on Big Data (Big Data),* Boston, USA, pp. 311-318, 2017.

[25] X. Nguyen, H. Nguyen and C. Pham, "A Scalable High-Performance Priority Encoder Using 1D-Array to 2D-Array Conversion," *IEEE Trans. Circuits Syst. II, Exp. Briefs,* vol. 64, no. 9, pp. 1102–1106, Sep. 2017.

[26] M. -L. Wei, M. Yayla, S. -Y. Ho, J. -J. Chen, C. -L. Yang and H. Amrouch, "Binarized SNNs: Efficient and Error-Resilient Spiking Neural Networks through Binarization," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD),* Munich, Germany, pp. 1-9, 2021, doi: 10.1109/ICCAD51958.2021.9643463.

[27] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. 32nd Intern. Conf. Int. Conf. on Machine Learning,* vol. 37, pp. 448-456, 2015.

[28] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.,* vol. 15, no. 1, pp. 1929-1958, 2014.

[29] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proc. of the IEEE*, vol.86, no. 11, pp. 2278-2324, 1998.

[30] Y. LeCun, C. Cortes, and C. Burges, "MNIST Handwritten Digit Database," AT & T Labs, vol. 2, 2010, http://yann.lecun.com/exdb/mnist

[31] Alex Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Technical Report TR-2009, University of Toronto, Toronto, 2009.

[32] J. Han, Z. Li, W. Zheng, and Y. Zhang, "Hardware implementation of spiking neural networks on FPGA," *Tsinghua Sci. Technol.,* vol. 25, no. 4, pp. 479–486, Aug. 2020.

[33] S. Gupta, A. Vyas and G. Trivedi, "FPGA implementation of simplified spiking neural network," in *Proc. 2020 27th IEEE Int. Conf. on Electron., Circuits and Sys. (ICECS),* Glasgow, UK, pp. 1-4, 2020.

[34] M. Liang, J. Zhang and H. Chen, "A 1.13μJ/classification spiking neural network accelerator with a single-spike neuron model and sparse weights," in *Proc. 2021 IEEE Int. Symp. on Circuits and Sys. (ISCAS),* Daegu, South Korea, pp. 1-5, 2021.

[35] W. Tang, G. Hua, L. Wang, "How to Train a Compact Binary Neural Network with High Accuracy?" in *Proc. the Thirty-First AAAI Conf. on Artificial Intelligence*, San Francisco, CA, USA, pp. 2625-2631, Feb. 2017.

[36] Y. Liu, Y. Chen, W. Ye and Y. Gui, "FPGA-NHAP: A general FPGA-based neuromorphic hardware acceleration platform with high speed and low power," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 69, no. 6, pp. 2553-2566, June 2022.

[37] D. Ma et al., "Darwin: A neuromorphic hardware co-processor based on spiking neural networks," *J. Syst. Archit.*, vol. 77, pp. 43-51, 2017.

[38] M. T. L. Aung, C. Qu, L. Yang, T. Luo, R. S. M. Goh and W. -F. Wong, "DeepFire: acceleration of convolutional spiking neural network on modern field programmable gate arrays," in *Proc. 2021 31st Int. Conf. on Field-Programmable Logic & Appl. (FPL)*, pp. 28-32, 2021.

[39] M. Ghasemzadeh, M. Samragh and F. Koushanfar, "ReBNet: residual binarized neural network," in *Proc. Annu. IEEE Symp. Field-Program. Cust. Comput. Mach.*, pp. 57-64, 2018.

[40] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, K. Vissers, "Finn: a framework for fast, scalable binarized neural network inference," in *Proc. ACM/SIGDA Int. Symp. on Field-Programm. Gate Arrays*, pp. 1-10, 2017.

[41] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," arXiv:1409.1556, 2014.

[42] T. -H. Tsai and Y. -C. Ho, "A CNN Accelerator on FPGA using Binary Weight Networks," in *Proc.2020 IEEE Int. Conf. Consum. Electron. Taiwan*, pp. 1-2, 2020.

[43] D. Gerlinghoff, Z. Wang, X. Gu, R. S. M. Goh and T. Luo, "E3NE: An End-to-End Framework for Accelerating Spiking Neural Networks With Emerging Neural Encoding on FPGAs," *IEEE Trans Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 3207-3219, 1 Nov. 2022.