Article

Python-Based Circuit Design for Fundamental Building Blocks of Spiking Neural Network

Xing Qin *, Chaojie Li, Haitao He, Zejun Pan and Chenxiao Lai

Zhejiang Key Laboratory of Large-Scale Integrated Circuit Design, Hangzhou Dianzi University, Hangzhou 310013, China; 212040280@hdu.edu.cn (C.L.)
* Correspondence: qinx@hdu.edu.cn

Abstract: Spiking neural networks (SNNs) are considered a crucial research direction to address the “storage wall” and “power wall” challenges faced by traditional artificial intelligence computing. However, developing SNN chips based on CMOS (complementary metal oxide semiconductor) circuits remains a challenge. Although memristor process technology is the best alternative to synapses, it is still undergoing refinement. In this study, a novel approach is proposed that employs tools to automatically generate HDL (hardware description language) code for constructing neuron and memristor circuits after using Python to describe the neuron and memristor models. Based on this approach, HR (Hindmash–Rose), LIF (leaky integrate-and-fire), and IZ (Izhikevich) neuron circuits, as well as HP, EG (enhanced generalized), and TB (the behavioral threshold bipolar) memristor circuits are designed to construct the most basic connection of a SNN: the neuron–memristor–neuron circuit that satisfies the STDP (spike-timing-dependent-plasticity) learning rule. Through simulation experiments and FPGA (field programmable gate array) prototype verification, it is confirmed that the IZ and LIF circuits are suitable as neurons in SNNs, while the X variables of the EG memristor model serve as characteristic synaptic weights. The EG memristor circuits best satisfy the STDP learning rule and are suitable as synapses in SNNs. In comparison to previous works on hardware spiking neurons, the proposed method needed fewer area resources for creating spiking neurons models on FPGA. The proposed SNN basic components design method, and the resulting circuits, are beneficial for architectural exploration and hardware–software co-design of SNN chips.

Keywords: spiking neural network (SNN); Python; memristor; FPGA; neuron circuit; synaptic circuit; STDP

1. Introduction

In recent years, artificial intelligence (AI) has achieved commercial success in areas such as facial recognition and autonomous driving. However, existing AI processors face several challenges, such as transistors approaching physical limits, the Von Neumann system memory wall problem [1], and the power consumption wall of deep learning algorithms. In contrast, SNNs, inspired by the human brain, have more potential to overcome the limitations of traditional ANNs (artificial neural networks). As a highly efficient computing model, the human brain consumes only about 20 watts of power [2], while traditional ANNs require more power and resources to achieve similar performance. By simulating the structure and function of the brain, SNNs have the potential to significantly improve computing efficiency and reduce power consumption.

In addition to lower power consumption, SNNs have a broader range of applications than ANNs. Due to the physical nature of SNN neurons and the synapses being closer to biological neural systems, SNNs have certain advantages in processing unstructured data. For example, when processing continuous signals such as images and sounds, SNNs can better preserve the temporal information of these data, while ANNs need to discretize continuous signals into a series of static image or audio frames for processing [3]. Therefore,
considering the limitations of the existing artificial intelligence and the need for more efficient and scalable AI systems, the potential benefits of brain-inspired computing and the development and implementation of SNNs has become a current research focus.

So far, three different types of circuits have been used to implement SNNs: CMOS digital circuits such as TrueNorth [1], Loihi [4], and Darwin [5]; CMOS digital–analog mixed circuits such as Neurogrid [6] and ROLLS [7]; and memristor–CMOS hybrid circuits such as Tianjic [8]. Neurons and synapses are the major elements of these SNN chips, with neurons commonly employing the leaky integrate-and-fire (LIF) model [9]. Synapses in TrueNorth, Loihi, Darwin, Neurogrid, and ROLLS are generally realized through weight update circuits or weight memory. Most of them only support offline learning, with the exception of Loihi, which allows online learning of spike-time-dependent plasticity (STDP) rules [4,10]. Tianjic’s synapse is implemented using a crossbar-type amnestic blocker.

Biological neurons include synapses that allow signals to be transmitted between two neurons and weight the incoming signals. Learning is based on synaptic plasticity, namely the changing of weighting factors in the synapses [11,12]. Memristors are considered artificial synapses for neuromorphic circuits [13], as their conductance characterizes the synaptic weights and remains preserved after power-down. Furthermore, in terms of efficiency, memristors have the advantage of consuming less power and space for less cost [14,15]. Therefore, the appropriate neuron model, memristors, and connections with STDP learning rules, can be used to construct the most basic SNN models.

Designing a SNN chip is a highly challenging task that requires the continuous optimization of both hardware and software. This involves addressing multiple challenges, such as achieving effective interaction between the SNNs and traditional computers, performing computation and learning in real-time environments, and attaining high performance and scalability using limited resources and power consumption [4]. Currently, researchers are increasingly focusing on hardware circuit implementations based on SNNs. For example, Zhao and Wang et al. use memristors as connection devices between neurons to achieve low power consumption and small volume in neural network hardware implementations [16]. Bensimon and Greenberg et al. proposed an innovative reconfigurable digital spiking neuron (SN) model based on the classic LIF model, and the hardware implementation of the model significantly reduces area and power costs [17]. Due to the complexity of the SNN chip design, Abderrahmane and Lemaire et al. proposed a hardware design space exploration framework for SNN models [18]. The framework uses SystemC for high-level hardware modeling and obtains hardware parameters through design space search optimization before relying on manual design of RTL. Therefore, researching SNN implementation using a high-level language hardware design approach and memristor emulation hardware is extremely important. MSDSL is a Python-based synthesizable module generator that creates digital–analog mixed signal circuits by using Python [13], providing the possibility of developing SNN circuits through the ESL (electronic system level design methodology) approach. Python is simpler compared to SystemC and is more widely used in the field of artificial intelligence algorithms. The automatic generation of RTL designs is more efficient, and the developed circuits can run on FPGA (as in [19,20]), meeting the requirements of a low-cost, agile, and reconfigurable design, and enabling software and hardware co-simulation, facilitating the development of SNN computing architecture and algorithm.

The work at hand consists of three main contributions:

- We propose a method for designing hardware SNNs, which uses Python to describe the neuron and memristor models and then automatically generates HDL code to build neuron and memristor circuits using MSDSL. The most basic SNN model was constructed based on this method and imported into the FPGA for prototype verification.
- We demonstrate with experimental comparisons that the IZ and LIF neuron circuits are suitable for use as neurons in SNNs, and that the EG memristor model is more suitable as synapses in SNNs, as it better satisfies the STDP learning rule.
• We evaluated the resources and power consumption required for the generated neural model to be implemented on FPGA and compared it with other related studies. We found that the proposed method required fewer resources to implement the SNN model on FPGA.

The structure of the rest of the article is as follows: Section 2 introduces the electronic system level design methodology and various types of basic neuron models [21–23], memristor models [19,24,25], and STDP learning models. Section 3 presents simulation demonstrations and data analysis, followed by the validation of hardware implementation through FPGA. Finally, the article is summarized.

2. Materials and Methods
2.1. Electronic System Level Design Methodology

The core idea of ESL design is to use high-level languages such as C/C++ or Python to directly describe algorithms, and to automatically generate hardware designs using tools for fast system modeling and simulation analysis, thereby improving system software and hardware design efficiency. Python, as a high-level programming language widely used in fields such as scientific computing, artificial intelligence, and data analysis, has been increasingly favored in recent years. This article proposes an ESL method based on Python, which can automatically generate HDL code for components of SNNs.

Figure 1 illustrates two neurons connected by a synapse [13]. The pre-synaptic neuron sends a pre-synaptic spike through one of its axons into the synaptic junction. The synapse is characterized by a “synaptic strength” (or weight $w$), which determines the efficacy of the pre-synaptic spike in contributing to the cumulative action at the post-synaptic neuron. The synaptic weight $w$ is considered to be non-volatile and analog in nature, but it changes over time as a function of the spiking activity of the pre- and post-synaptic neurons. The non-volatile storage nature of a memristor allows it to store data, while the ability to constantly modify its state enables it to compute. When the two are combined, the memristor allows for the fusion of computation and storage in the same device. A memristor is similar to an artificial synapse in that it has synaptic plasticity for learning and storing information [16], represented by changes in weighting factors in the synapse. Therefore, the memristor can be an important component of neuromorphic circuits [26].

![Figure 1. Structure of biological neural network.](image)

STDP shows that the intensity of the effect between neurons will be adjusted according to the pulse duration of the pre-synaptic and post-synaptic neurons. The fast update of STDP synaptic weights has a significant impact on the learning ability of neural networks and can improve the accuracy of the classic Hebb learning rules [27]. Therefore, the neuron circuit, the memristor circuit, and the connection satisfying the STDP learning rules are the most essential components of the SNN circuit.

MSDSL is a Python-based synthesizable module generator [28]. Its working principle involves translating the syntax rules of a high-level domain-specific language (DSL) into low-level DSL code. By using Python to write high-level DSL code that describes the characteristics and behavior of the hardware design, MSDSL provides a general compiler
framework that parses the high-level DSL code into an abstract syntax tree (AST) and then transforms and optimizes the AST into low-level DSL code, which is the hardware description language SystemVerilog. Compared with other tools, such as myHDL [29], HDLConverter [30], and PyVerilog [31], MSDSL has a stronger advantage in mixed-signal modeling, especially in continuous-time and mixed-signal circuit modeling, while myHDL, HDLConverter, PyVerilog, and other tools are more suitable for pure digital circuit design and code generation. Since SNN uses “spikes” as the main signal, the advantage of MSDSL is more obvious, because neuron and memristor models are usually written in the form of differential equations.

As shown in Figure 2, we generate neuron and memristor models in Python and use MSDSL to automatically generate the corresponding HDL code for the neurons and memristors. Neuron circuits, memristor circuits, and the simple SNN circuit and its testbench are then created, with the commercial HDL simulation tool applied for simulation and parameter optimization. Finally, those circuits are validated by using a FPGA prototype.

If the model description contains a division or nonlinear function, MSDSL will build an isometric lookup table. Therefore, the better the accuracy of the model, the larger the memory area required for the lookup table. Therefore, we propose a strategy for generating unequally spaced lookup tables: divide the continuous nonlinear function into different regions, use denser sampling intervals to generate lookup tables for regions with fast changes, use sparse sampling intervals to generate lookup tables for regions with slow changes, and combine the outputs of multiple lookup tables into one output to reduce the area of the lookup table without sacrificing accuracy.

This strategy has been encapsulated in the function “make_nolinear_func”, which encompasses five parameters: object, function, configuration table, input signal, and output signal. The function generates synthesizable HDL circuits automatically. The segmented lookup table method occupies fewer resources and provides higher precision in rapidly changing regions while maintaining a reasonable precision in slowly changing regions.

2.2. Neuron Circuit Modeling

The “leak” term is introduced in the IF (integrate-and-fire) model to reflect the ion diffusion that occurs when the cell reaches a certain equilibrium. The LIF (leaky integrate-
and-fire) neuron [22] model can be represented by a parallel RC circuit with a threshold reset mechanism driven by an input current source \( I(t) \). Due to its simplicity and low computational cost, the LIF model and its variants are widely used in SNN models. Its mathematical model is expressed as:

\[
\tau_m \frac{dV}{dt} = E_l - V + R_m I_e
\]  

Equation (1) [22] describes the rule of the membrane potential \( V \) of a neuron over time. The \( \tau_m \) is the membrane time constant, which represents the ability of the membrane capacitor to store charge. \( E_l \) is the resting potential of the neuron, which is the membrane potential when there is no external stimulus. \( V \) represents the membrane potential of the neuron, and \( R_m \) is the membrane resistance, which represents the resistance of the membrane to current. \( I_e \) is the input current, which represents the external stimulus received by the neuron. The leakage current corresponds to the “leak” in LIF and represents the loss of membrane potential during the propagation process along the axon, which corresponds to the membrane resistance \( R_m \) in hardware. The leakage period represents the time required for complete charge release and corresponds to the membrane time constant \( \tau_m \). The modeling is performed using Python (as shown in Listing 1); the HDL circuit model is generated using MSDSL.

Listing 1. LIF neuron model in Python.

```python
1 v = m.add_analog_output('v', init = -2)
2 flag = m.add_digital_state('flag', init = 1, width = 1, signed = False)
3 m.set_this_cycle(flag, v < v_threshold)
4 m.set_next_cycle(v, (v + dt*(1/Tm)*((El - v) + R*I_e))*flag - 2*(-flag), clk = m.clk, rst = m.rst)
```

2.3. Memristor Circuits Modeling

The enhanced generalized memristor model (EG) [25] is proposed on the basis of approximating the experimental properties of OxRAM memristor devices based on HfO2, and the two main parts of the model are the mathematical functions presented in Equations (2) and (3) [25]:

\[
i(t) = \begin{cases} 
a_1 x(t) \sinh(b v(t)) & v(t) \geq 0 \\
a_2 x(t) \sinh(b v(t)) & v(t) < 0 
\end{cases}
\]

\[
\frac{dx}{dt} = \eta g(v(t))f(x(t))
\]  

Equation (2) is based on the hyperbolic sine curve to explain the \( I - V \) relationship of the memristor model, where the parameters \( a_1, a_2, \) and \( b \) are used to fit different device structures of EG memristors. Parameter \( b \) controls the strength of the threshold function between the conductance and the amplitude of the input voltage. The state variable \( x(t) \) represents the value of the memristor resistance change, ranging from 0 to 1. In Equation (3), the change in the state variable \( x(t) \) depends on the functions \( g(v(t)) \) and \( f(x(t)) \). The function \( g(v(t)) \) is a threshold function that must exceed a certain threshold voltage for the state variable \( x(t) \) to change. The function \( f(x(t)) \) divides the state variable motion into two different regions, where the state variable is at constant speed before it reaches a certain point and then decays exponentially, ensuring the state variable remains within the range of 0 to 1. The parameter \( \eta \) is used to indicate the direction of the state variable motion relative to the voltage polarity. When \( \eta = 1 \), positive voltage increases the value of the state variable, whereas when \( \eta = -1 \), positive voltage decreases the state variable, giving the EG memristor a polarity characteristic similar to that of biological neurons. For more mathematical details, refer to [25].

MSDSL is used to generate the HDL circuit model after describing the EG memristor model (as shown in Listing 2) using Python. The memristor model’s circuit can be possibly abstracted, as illustrated in Figure 3. The memristor is a two-port device, so its two inputs,
$V_1$ and $V_2$, represent the potential at its two ends. $I_D$ is the current flowing through the inductor, $G$ is the memristor’s conductor, and $P_{SET}$ can be used to configure the memristor’s type and parameters.

**Listing 2.** EG memristor model in Python.

```
# setting various critical conditions in eg memristor model
m.set_this_cycle(flag1,v < 0)
m.set_this_cycle(flag2,v < -Vn)
m.set_this_cycle(flag3,v > Vp)
m.set_this_cycle(flag4,x < Xp)
m.set_this_cycle(flag5,x > 1-Xn)

# setting constants in eg memristor model
ean = exp(an*(Xn - 1))
eap = exp(ap*Xp)
ev1 = m.set_from_sync_func('ev1', f1, v, clk = m.clk, rst = m.rst)
ev2 = m.set_from_sync_func('ev2', f2, v, clk = m.clk, rst = m.rst)
ev3 = m.set_from_sync_func('ev3', f3, v, clk = m.clk, rst = m.rst)
ev4 = m.set_from_sync_func('ev4', f4, v, clk = m.clk, rst = m.rst)
ev5 = m.set_from_sync_func('ev5', f5, x, clk = m.clk, rst = m.rst)
ev6 = m.set_from_sync_func('ev6', f6, x, clk = m.clk, rst = m.rst)

# use the lookup table function to complete the modeling of exponential function
func1 = lambda v:np.exp(v)
ev1 = m.set_from_sync_func('ev1', f1, v, clk = m.clk, rst = m.rst)
ev2 = m.set_from_sync_func('ev2', f2, v, clk = m.clk, rst = m.rst)
ev3 = m.set_from_sync_func('ev3', f3, v, clk = m.clk, rst = m.rst)
ev4 = m.set_from_sync_func('ev4', f4, v, clk = m.clk, rst = m.rst)
ev5 = m.set_from_sync_func('ev5', f5, x, clk = m.clk, rst = m.rst)
ev6 = m.set_from_sync_func('ev6', f6, x, clk = m.clk, rst = m.rst)

# modeling of Equation (3)
func3 = lambda v:exp(0.05*v)
ev3 = m.set_from_sync_func('ev3', f3, v, clk = m.clk, rst = m.rst)
ev4 = m.set_from_sync_func('ev4', f4, v, clk = m.clk, rst = m.rst)
ev5 = m.set_from_sync_func('ev5', f5, x, clk = m.clk, rst = m.rst)
ev6 = m.set_from_sync_func('ev6', f6, x, clk = m.clk, rst = m.rst)

# modeling of Equation (4)
func4 = lambda v:1/np.exp(v)
ev2 = m.set_from_sync_func('ev2', f2, v, clk = m.clk, rst = m.rst)
ev3 = m.set_from_sync_func('ev3', f3, v, clk = m.clk, rst = m.rst)
ev4 = m.set_from_sync_func('ev4', f4, v, clk = m.clk, rst = m.rst)
ev5 = m.set_from_sync_func('ev5', f5, x, clk = m.clk, rst = m.rst)
ev6 = m.set_from_sync_func('ev6', f6, x, clk = m.clk, rst = m.rst)

# modeling of Equation (3)
func5 = lambda x:1/np.exp(0.05*x)
ev2 = m.set_from_sync_func('ev2', f2, v, clk = m.clk, rst = m.rst)
ev3 = m.set_from_sync_func('ev3', f3, v, clk = m.clk, rst = m.rst)
ev4 = m.set_from_sync_func('ev4', f4, v, clk = m.clk, rst = m.rst)
ev5 = m.set_from_sync_func('ev5', f5, x, clk = m.clk, rst = m.rst)
ev6 = m.set_from_sync_func('ev6', f6, x, clk = m.clk, rst = m.rst)

# modeling of Equation (4)
func6 = lambda x:1/np.exp(0.05*x)
ev2 = m.set_from_sync_func('ev2', f2, v, clk = m.clk, rst = m.rst)
ev3 = m.set_from_sync_func('ev3', f3, v, clk = m.clk, rst = m.rst)
ev4 = m.set_from_sync_func('ev4', f4, v, clk = m.clk, rst = m.rst)
ev5 = m.set_from_sync_func('ev5', f5, x, clk = m.clk, rst = m.rst)
ev6 = m.set_from_sync_func('ev6', f6, x, clk = m.clk, rst = m.rst)
```

![Figure 3. Abstraction of memristor circuit.](image-url)

2.4. STDP (Spike-Timing-Dependent Plasticity)

The STDP (spike-timing-dependent plasticity) is a learning rule based on the timing of pre- and post-synaptic spikes to modify the synaptic weights [32], which extends the Hebb learning rule. Studies have shown that the relative timing of pre- and post-synaptic spikes plays a crucial role in determining the direction and magnitude of synaptic weight changes between neurons. The STDP learning mechanism has been found in the neural systems of many biological species and has good biological interpretability, making it a common biological plausibility rule.

The change in synaptic weight, $\Delta w$, is a function of the relative timing difference, $\Delta t$ (where $\Delta t = T_{post} - T_{pre}$), between the pre- and post-synaptic spikes. $T_{pre}$ and $T_{post}$ are the times of the pre- and post-synaptic spikes, respectively. When $\Delta t > 0$, the synapse exhibits long-term potentiation (LTP), whereas when $\Delta t < 0$, the synapse exhibits long-term depression (LTD).

$$f(\Delta w) = \begin{cases} A^+e^{-\frac{\Delta t}{\tau}}, & |\Delta t| > 0 \\ A^-e^{\frac{\Delta t}{\tau}}, & |\Delta t| < 0 \end{cases}$$ (4)
In Equation (4) [32], $A^+$ and $A^-$, respectively, indicate the minimum and maximum modifications of the synaptic weight with respect to the initial state during potentiation and depression functions. The $\tau^+$ and $\tau^-$ are the time windows that define the synaptic weight update rate for long-term depression (LTD) and long-term potentiation (LTP) cases, respectively.

We have built a neuron circuit model and a memristor circuit model and connected them to form a basic SNN structure consisting of a neuron–memristor–neuron configuration (as shown in Figure 4). We then assessed the performance of the STDP learning rules within this structure.

![Figure 4. Basic SNN structure of neuron–memristor–neuron.](image)

### 3. Results

#### 3.1. Simulation Results of Neuron HDL Models

An HDL model of a LIF neuron was simulated, and the resulting membrane potential waveform is shown in Figure 5. The LIF neuron was also modeled using Brain2 [33], and the result is shown in Figure 6. The parameters included in the HDL model of the LIF neuron, such as the membrane time constant ($\tau_m$), resting potential ($E_r$), membrane resistance ($R_m$), and input current ($I_i$), were consistent with those used in the Brain2 model, which were $\tau_m = 20$, $I_i = 0.06$, $E_r = -2$, and $R_m = 100$. The membrane potential waveforms generated by the HDL model of the LIF neuron and the Brain2 model of the LIF neuron showed high consistency, with both having a voltage threshold of 2 mV and a reset voltage of −2 mV.

Using the same method, HDL models were designed for Hindmash–Rose (HR) model [21] and Izhikevich (IZ) model [23]. The membrane potential waveform obtained by simulating the HDL model of HR neuron is shown in Figure 7. The HR neuron was also modeled using Brain2, and the result is shown in Figure 8.

![Figure 5. Simulation result of HDL model of LIF neuron.](image)
The HDL model of HR neuron includes parameters such as the recovery rate of membrane voltage $a$, sensitivity of membrane voltage recovery $b$, ratio of membrane capacitance to membrane resistance $c$, reset value of membrane voltage $d$, reset potential $Urst$, external input current $I$, and density of potassium ion channels $s$, which are consistent with the parameters used in the Brain2 model, i.e., $a = 1.0$, $b = 3.0$, $c = 1.0$, $d = 5.0$, $I = 1$, $s = 4$, and $Urst = −1.6$. The membrane potential waveforms generated by the HDL model of HR neuron and the Brain2 model of HR neuron exhibit high consistency, with a reset voltage of $−1.6 \text{ mV}$. This indicates that the HDL models are accurate and suitable for use in designing and simulating SNN circuits.

The HDL model of the Izhikevich (IZ) neuron was simulated, and the resulting membrane potential waveform is shown in Figure 9. The IZ neuron was also modeled using Brain2, and the result is shown in Figure 10. The HDL model of the IZ neuron includes parameters such as the recovery variable time constant $a$, recovery variable sensitivity $b$, membrane potential reset constant $c$, membrane potential reset amplitude $d$, external input current $I$, and the membrane potential threshold voltage $p$. These parameters are consistent with the ones used in the Brain2 model, with values of $a = 0.02$, $b = 0.25$, $c = −65$, $d = 0.05$, $I = 15$, and $p = 30$. 

Figure 6. LIF neuron model in Brain2.

Figure 7. Simulation result of HDL model of HR neuron.

Figure 8. HR neuron model in Brain2.
0 50 100 150
Time (ms)
200 250
-60
-40
-20
0
 20
 40Voltage (mV)
Figure 9. Simulation result of HDL model of IZ TC (thalamo-cortical) neuron.

0 10 20 30 40 50
Time (ms)
-60
-40
-20
0
 20
 40Voltage (mV)
Figure 10. IZ TC (thalamo-cortical) neuron model in Brain2.

The membrane potential waveforms generated by the HDL model of the IZ neuron and the Brain2 model of the IZ neuron exhibit a high degree of consistency, with voltage thresholds of 30 mV and reset voltages of −65 mV. By adjusting the parameters of the IZ neuron model, the model is able to replicate several firing patterns exhibited by biological neurons, such as regular spiking, intrinsically bursting, chattering, and fast spiking, as shown in Figure 11.

Figure 11. Four firing patterns of IZ neurons: (a) RS (regular spiking); (b) IB (intrinsically bursting); (c) CH (chattering); (d) FS (fast spiking).

In addition, it should be noted that all the IZ neuron models mentioned subsequently default to the thalamo-cortical firing pattern. If other firing patterns are used in later experiments, they will be specified separately. This versatility in the IZ neuron model
allows researchers to simulate various types of neurons, making it suitable for designing and simulating SNN circuits that closely mimic the behavior of biological neurons.

Taking the IZ neuron as an example, it is observed that there is a non-linear relationship between the amplitude of the input current and the frequency of the output voltage pulse signal. As seen in Figure 12, the current amplitude of the input current pulse is stepped, with values of 0 mA, 5 mA, 10 mA, 15 mA, 20 mA, and 30 mA, respectively. The frequency of the pulse signal is 0 Hz, 100 Hz, 152 Hz, 200 Hz, 227 Hz, and 294 Hz, respectively.

![Figure 12. Simulation results of the IZ neuron circuit.](image)

After passing through the IZ neuron circuit, its output spike potential remains unchanged at 30 mV once it is output, and its frequency gradually increases. This non-linear relationship may be the basis for SNN to realize complex computation.

### 3.2. Simulation Results of Memristor HDL Models

Figure 13a depicts the application voltage and current response curve of the EG memristor circuit. The sine voltage input applied to the circuit has an amplitude of 0.46 V and a frequency of 100 Hz. The $V-I$ characteristics obtained from the EG model show a symmetrical waveform which is consistent with experimental data of the GST (Ge2Sb2Te5) memristor device [34]. This indicates that the EG memristor circuit model accurately represents the behavior of the real memristor device.

![Figure 13. The $V-I$ characteristic curve obtained by HDL models of memristor: (a) EG; (b) HP; (c) TB.](image)

Similarly, circuits were designed for the HP memristor model [24] and TB memristor model [19] based on the same approach. The $V-I$ curves obtained from the simulations of these models are shown in Figure 13b,c, respectively. Comparing these two results, it is evident that the unique synaptic plasticity of the memristor is ideally reflected in both the HP and TB memristor models.

These results demonstrate the effectiveness of the proposed method for designing and simulating memristor circuits. The consistency between the simulated $V-I$ curves and experimental data highlight the potential for using these circuit models in the development
of SNNs, allowing researchers to explore the integration of memristor devices with neural networks for applications in neuromorphic computing and artificial intelligence.

The synaptic plasticity, or synaptic connection strength, is correlated with the change in memristor resistance. The HP memristor model does not involve the concept of threshold voltage, so it causes a change in resistance no matter how small the input voltage is, even in extreme cases where the resistance $R$ is not between $Ron$ and $Roff$. The EG and TB memristor models compensate for this limitation in the HP memristor model by incorporating threshold voltage constraints. The $V − I$ curves of the EG model are closer to the performance of real memristor devices, making it more suitable for accurately modeling memristive behavior.

The conductance of the memristor is usually related to the weight of the synapse. Unlike the HP and TB memristor models, the conductance of the EG memristor model cannot be obtained directly. However, experimental studies have shown that the theoretically calculated conductance is positively correlated with the value of the $X$-state variable of the EG model, except when it is near 500 ms (because the voltage tends to zero when it is near 500 ms, causing the theoretically calculated conductance to change abruptly) (as shown in Figure 14). Therefore, the $X$-state variable of the EG model can be used to approximate the conductance of the memristor, providing a useful parameter for understanding and controlling the synaptic strength in SNNs built with the EG memristor model.

![Figure 14](https://example.com/image14.png)

**Figure 14.** The conductivity and $X$ variable of the EG model.

### 3.3. Simulation Results of STDP

In this basic SNN model, two LIF neurons with different initial membrane potential values are connected using an EG memristor, creating a phase difference between the pre-synaptic and post-synaptic neurons in their initial state. The voltage of the memristor is the difference in membrane potential between the two neurons, and the output current of the memristor serves as a weighted input to the post-synaptic neuron’s input.

Figure 15a demonstrates the simulation results, with $V_{pre}$ representing the membrane potential of the pre-synaptic neuron, $V_{post}$ representing the potential of the post-synaptic neuron, and $V (V = V_{post} − V_{pre})$ representing the voltage across the EG memristor. $X$ represents the state variable in the EG memristor, which is linearly related to conductance, meaning $X$ represents the conductance change in the memristor.

From Figure 15a, it is observed that when the spike of the pre-synaptic neuron arrives at its peak earlier than the spike of the post-synaptic neuron (i.e., when $\Delta T(\Delta T = T_{post} − T_{pre}) > 0$), the value of the state variable $X$ increases, consistent with the LTP mechanism. As the time difference between the pre-synaptic and post-synaptic neurons reaching their peaks decreases, the value of $X$ increases ($|\Delta X|$ becomes larger). Conversely, when $\Delta T < 0$, the value of $X$ decreases, consistent with the LTD mechanism. Additionally, as the time difference decreases, the value of $X$ decreases ($|\Delta X|$ becomes larger).

By recording all of the time differences $\Delta T$ between the pre-synaptic and post-synaptic neurons reaching their peaks in Figure 15a, and the corresponding change in the state variable $\Delta X$, we obtained a dataset and plotted the corresponding scatter plot. By using MATLAB’s curve fitting toolbox to fit these data, we used the function relationship of $\Delta w(\Delta t)$ in Equation (4) as a reference to obtain the function relationship between $\Delta X / X$
and \( \Delta T \), as shown in Figure 15b. When performing curve fitting, we usually obtain an
\( R \)-squared \( (R^2) \) value, which is a statistical measure of the degree to which the fitted curve
fits the actual data. The range of \( R^2 \) values is between 0 and 1, with values closer to 1
indicating better fit results and values closer to 0 indicating poorer fit results. The \( R^2 \) value
obtained from our curve fitting was 0.9311, indicating a good fit. This indicates that the
created LIF-EG-LIF, as the most basic SNN model, is consistent with the STDP learning rule.
This validates the model’s ability to implement STDP learning mechanisms in a simplified
neural network setting.

![Figure 15. EG-LIF SNN: (a) simulation results; (b) the curve of the change rate \((\frac{\Delta X}{\Delta t})\) of state variable X with \(\Delta t\).](image)

In this experiment, IZ neurons are used instead of LIF neurons, and the simulation
results in Figure 16a support the previous conclusions. When \( \Delta T > 0 \), the value of X
increases, which is in line with the LTP mechanism; when \( \Delta T < 0 \), the value of X decreases,
which is in line with the LTD mechanism. Moreover, as the time difference decreases, the
degree of increase or decrease in X also increases.

![Figure 16. EG-IZ SNN: (a) simulation results; (b) the curve of the change rate \((\frac{\Delta X}{\Delta t})\) of state variable X with \(\Delta t\).](image)

By recording the time difference \( \Delta T \) and the corresponding changes in the state
variable \( \Delta X \) in Figure 16a, we plotted the scatter plot and fitted these data using the
function relationship of \( \Delta w(\Delta t) \) in Equation (4) as a reference. The resulting \( \Delta X / X \) and \( \Delta T \)
function relationship is shown in Figure 16b, with an \( R^2 \) value of 0.9589 indicating a good fit.
This indicates that the IZ-EG-IZ model, as the most basic SNN model, conforms to the STDP
learning rule. This further validates the versatility of the STDP learning rule in different
neuron models and confirms that the IZ-EG-IZ model can also effectively implement STDP learning mechanisms in a simplified neural network setting.

In this experiment, a basic SNN model consisting of two LIF neurons and a TB memristor is used, similar to the previous experiments. The results in Figure 17a,b indicate that when the time difference $\Delta T > 0$, the value of $X$ increases, consistent with LTP. When the time difference $\Delta T < 0$, the value of $X$ decreases, consistent with LTD, as in the previous two cases.

![Figure 17. TB-LIF SNN: (a) LTP; (b) LTD.](image)

However, there are differences in how the changes in $X$ relate to the time difference $\Delta T$. When $\Delta T > 0$, the degree to which $X$ increases with decreasing time difference becomes smaller ($|\Delta X|$ becomes smaller). Similarly, when $\Delta T < 0$, the degree to which $X$ decreases with decreasing time difference becomes smaller ($|\Delta X|$ becomes smaller). The resulting $\Delta X / X$ and $\Delta T$ function relationship does not correspond to the $\Delta w(\Delta t)$ function relationship in the STDP learning rule.

This implies that using an EG memristor as the synaptic connection between neurons is more suitable for constructing SNN models than using a TB memristor. The EG memristor appears to more closely model the expected behavior of the STDP learning rule, making it a better choice for implementing this learning mechanism in SNNs.

3.4. FPGA Experiments

During the FPGA experiments, the hardware SNN model was first verified by simulation using Vivado to ensure the correctness of the code. Then, the code was synthesized using the Vivado Design Suite synthesizer and implemented on the Xilinx xc7a200tfg484-2 FPGA. In order to convert the digital signals into analog signals and display them, we also used a DAC (digital-to-analog converter) of the AD9764 model, with a maximum rate of 125 MHz and 14-bit precision. The digital signals in the FPGA were connected to the DAC, which converted them into analog signals and outputted them to the oscilloscope for display. By connecting the oscilloscope, signal waveforms could be monitored in real time, which provided a more intuitive observation of signal characteristics and changes. This is very helpful for circuit design debugging and optimization.

The results from the hardware implementation of the basic SNN model, consisting of two IZ neuron modules and EG memristor modules on an FPGA development board, further demonstrate the effectiveness of the STDP learning rule in SNNs. The experiment shows that the initial uncoordinated activities of the two IZ neurons eventually synchronize due to the online learning of synapses based on the STDP principles. This synchronization is evident in Figure 18c, where the activities of the post-synaptic neurons converge with those of the pre-synaptic neurons after a certain period.
Figure 18. The results of the SNN model displayed in the oscilloscope: (a) when the SNN model just started running; (b) weight was continuously changing; (c) the IZ neurons began to converge; (d) weight did not change upon convergence.

Moreover, the state variable $X$, which is related to the conductance in the memristor, changes with the phase difference between the neurons at the beginning (Figure 18b) but stabilizes once the neurons’ activities have synchronized (Figure 18d). This result indicates that the STDP learning rule, implemented using IZ neurons and EG memristor modules, allows the SNN to adapt its synaptic weights and achieve synchronization between the pre-synaptic and post-synaptic neurons.

These findings support the effectiveness of using IZ neurons and EG memristors for building SNN models with STDP learning rules, both in simulations and in hardware implementations. The hardware model’s success in demonstrating the STDP learning rule shows promise for future development and deployment of SNNs in real-world applications.

The observations from Tables 1 and 2 provide valuable insights into the resource requirements and power consumption of different neuron models and their FPGA implementations.

### Table 1. Comparison of basic SNN models constructed by two different neurons.

<table>
<thead>
<tr>
<th>FPGA Resource Utilization</th>
<th>Slice LUT’s</th>
<th>Slice Registers</th>
<th>On-Chip Power</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>IZ_EG_IZ</td>
<td>541</td>
<td>107</td>
<td>186 mW</td>
<td>Xilinx xc7a200tfbg484-2</td>
</tr>
<tr>
<td>LIF_EG_LIF</td>
<td>448</td>
<td>86</td>
<td>196 mW</td>
<td>Xilinx xc7a200tfbg484-2</td>
</tr>
</tbody>
</table>

### Table 2. Comparison between proposed method and previously published works.

<table>
<thead>
<tr>
<th>FPGA Resource Utilization</th>
<th>Slice LUT’s</th>
<th>Slice Registers</th>
<th>Max Speed (MHz)</th>
<th>DSPs</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSDSL</td>
<td>175</td>
<td>103</td>
<td>100</td>
<td>0</td>
<td>Xilinx xc7a200tfbg484-2</td>
</tr>
<tr>
<td>IZHCOR6 [35]</td>
<td>410</td>
<td>229</td>
<td>183.4</td>
<td>0</td>
<td>Spartan-6 XC6SLX75</td>
</tr>
<tr>
<td>Soleimani et al. [36]</td>
<td>617</td>
<td>493</td>
<td>241.9</td>
<td>0</td>
<td>Virtex-II Pro XC2VP30</td>
</tr>
<tr>
<td>Grassia et al. [37]</td>
<td>1048</td>
<td>646</td>
<td>105</td>
<td>22</td>
<td>Virtex-5 XC5VLX50</td>
</tr>
</tbody>
</table>
Table 1 shows that constructing a basic SNN model with LIF neurons requires fewer area resources compared to using IZ neurons. The power consumption, however, remains nearly the same for both models. This indicates that in situations where area resources are a crucial factor, LIF neurons may be a more efficient choice. However, the trade-offs between model complexity and functionality should also be taken into account when selecting the appropriate neuron model for a specific application.

In Table 2, a comparison of area resources required for modeling IZ neurons using MSDSL with previously published works is presented. Although different FPGA devices and synthesizers were used in these studies, making the comparison relative, it can be observed that the resources needed for modeling IZ neurons with MSDSL are generally lower than in previous works. However, there are no significant advantages in terms of speed. This suggests that MSDSL offers a more resource-efficient approach for implementing IZ neuron models on FPGA, which could be advantageous in the development of SNNs with limited hardware resources.

These findings contribute to our understanding of the trade-offs between different neuron models and their hardware implementations. They can help guide future research and the development of SNNs, considering the balance between area resources, power consumption, and the complexity of the neuron models.

4. Discussion

This paper proposes a hardware design method for SNN key components based on Python. The method is compared with software modeling and simulation and circuit simulation methods based on Python. On this basis, future research directions are discussed.

4.1. Compared with the SNN Software Simulator Based on Python

Python has been successfully applied to the second generation of neural network modeling, such as Tensorflow and PyTorch, and also applied to SNN networks, such as Brain2 [33] and BindsNET [38]. Brain2 writes code in Python, converts dynamic equations into efficient low-level code, and can stagger to complete SNN modeling and simulation on the CPU. BindsNET is built on the PyTorch deep neural network library, which can quickly build SNN and map it to CPU and GPU computing platform. The proposed method also uses Python to complete the modeling of complex dynamic equations such as neurons and memristors, however the generated HDL code of the corresponding circuit can finally be mapped to FPGA to simulate SNN networks.

4.2. Compared with Circuit Simulation

Compared to CMOS circuit simulation [39] and hardware simulation based on FPGA prototype [40], the Python-based ESL method proposed in this paper for designing SNN models and importing them into FPGA has lower cost, higher flexibility, and is more reconfigurable. Although CMOS circuit simulation can verify the effectiveness of the EG model in simulating synapses, it cannot meet the requirements of co-design of software and hardware due to its inability to run in real time. Hardware simulation based on FPGA prototype provides a hardware verification platform accelerated by an SNN coprocessor. However, the design of its basic components requires proficient digital circuit design experience, making it difficult to achieve fast prototype design and iteration. Therefore, the proposed method in this paper has higher practicality and feasibility, as well as providing effective technical support for the rapid design and simulation of SNN.

4.3. Future Direction

On the basis of this paper, future research will be carried out in the following areas:

1. Establish SNN coprocessor automatic generation, resource evaluation, and software and hardware co-simulation driven by SNN basic primitives (similar to the primitives of CNN model) to provide methods and platforms for the agile design of SNN chips;
2. Build a large SNN network with FPGA array, map the typical intelligent recognition algorithm to the large SNN network, and evaluate the calculation effect.

5. Conclusions

An ESL design approach for SNN circuits using Python is introduced in this paper. Based on this approach, we build HR, LIF, and IZ neurons, as well as the HDL circuits of HP, EG, and TB memristors. We also construct the most basic SNN network and carry out extensive simulation validation, prototype validation, and experimental analysis. Through experimental analysis, it has proved that LIF and IZ neuron hardware models are suitable for building SNN neurons, and the EG memristor hardware model is suitable for building synapses of SNN. Practice has shown that for the generation of a SNN hardware model based on high-level language—Python is both practical and beneficial.

SNN still has a lot of unexplored potential, and memristor technology is still in its early stages of development. We will continue our research in the future, trying to automatically generate RTL projects and testbench code, enhancing resource analysis and optimization capabilities, providing support for architectural exploration, and hardware-software co-simulation design of SNN chips.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/electronics12112351/s1, Code: Blocks of spiking neural network.

Author Contributions: Conceptualization, X.Q. and C.L. (Chaojie Li); methodology, X.Q.; software, C.L. (Chaojie Li) and H.H.; validation, X.Q., C.L. (Chaojie Li) and H.H.; formal analysis, Z.P.; investigation, X.Q. and C.L. (Chaojie Li); data curation, C.L. (Chaojie Li) and H.H.; writing—original draft preparation, X.Q., C.L. (Chaojie Li), H.H., Z.P. and C.L. (Chenxiao Lai); writing—review and editing, X.Q., C.L. (Chaojie Li), H.H. and C.L. (Chenxiao Lai); funding acquisition, X.Q. All authors have read and agreed to the published version of the manuscript.

Funding: Supported by the Ministry of Science and Technology of China (Grant No. 2016YFB1000401).

Data Availability Statement: The data presented in this study are available in Supplementary Material.

Acknowledgments: The authors acknowledge the MSDSL and the SVREAL, which was used in this research. We wish to thank all who assisted in conducting this work.

Conflicts of Interest: The authors declare no conflict of interest.

References


2. Meier, K. Special report: Can we copy the brain?—The brain as computer. *IEEE Spectr.* 2017, 54, 28–33. [CrossRef]


33. Stinemberg, M.; Brette, R.; Goodman, D.F.M. Brian 2, an intuitive and efficient neural simulator. eLife 2019, 8, e47314. [CrossRef]