Preliminaries of Implementing an SNN on an FPGA

MINOR PROJECT REPORT

Submitted to:
Department of Electronics and Communication Engineering

Submitted by:

Name – Vedika Jain
Roll Number – 21BEC018

Under the Supervision of:
Prof. Sajad Ahmad Loan

Faculty of Engineering and Technology
Jamia Millia Islamia, New Delhi

Certificate

This is to certify that the project report entitled ‘Preliminaries of Implementing an SNN on an FPGA’ submitted by Vedika Jain (21BEC018) to the Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jamia Millia Islamia, New Delhi – 110025 in partial fulfilment for the award of the degree of Bachelor of Technology in Electronics and Communication Engineering is a bonafide record of project work carried out by her under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institution or University for the award of any degree.

Prof. Sajad Ahmad Loan
Supervisor
D/o E&C Engg.
FET, JMI

Countersignature of HoD with seal

New Delhi

13 December 2024

Declaration

I hereby declare that the minor project report entitled ‘Preliminaries of Implementing an SNN on an FPGA’ submitted to the Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jamia Millia Islamia, New Delhi – 110025 in partial fulfilment for the award of the degree of Bachelor of Technology in Electronics and Communication Engineering is a bonafide record of project work carried out by me. The contents of this report, in full or in parts, have not been submitted to any other Institution or University for the award of any degree or diploma.

Vedika Jain

21BEC018

New Delhi

13 December 2024

List of Contents

Acknowledgements

It’s by the Almighty’s grace that this world goes round. I bow my head before the Lord for helping me work on this project. I will forever be indebted to my family, my parents and my sister, for looking out for me every step along the way. Without their unwavering support, this project report wouldn’t have come to existence.

I extend my gratitude to my supervisor, Prof. Sajad Ahmad Loan, for his patience and guidance throughout the project. The report stands in its present form thanks to his probing counter questions and keen insights.

Introduction

Traditional computing faces challenges in tasks such as pattern recognition, sensory processing, and real-time interaction, which are effortlessly performed by the human brain. AI is best run on hardware which is similar to the abstraction. Neuromorphic computing, inspired by the brain's efficiency and parallelism, aims to overcome these limitations. Neuromorphic computing seeks to mimic the structure and function of the human brain using electronic circuits. Present systems make use of the second generation ANNs.

Third-generation ANNs are Spiking Neural Networks (SNNs) that use biologically plausible spiking neurons as the basic computational units. The neuron model used depends on the application. Neural information in the spiking neuron is transmitted and processed by precisely timed spike trains. Efficacy of SNNs depends on the spike train encoding used. With SNNs, there exists a trade off between accuracy and simplicity. Compared with the first- and second-generation ANN models, SNNs can describe the real biological nervous system more accurately, so as to achieve efficient information processing.

These networks are well-suited for FPGA implementation due to their parallel nature. This project focuses on the design of an SNN for temporal processing applications. It uses Izhikevich neurons, exploring its high-level architecture, synaptic connectivity, and a simulation of its behaviour with STDP learning rules. The goal is to provide a clear pathway toward FPGA realisation in the future, ensuring scalability and performance optimisation.

High Level Design

Neuron Model: Izhikevich Neuron

The Izhikevich model is a powerful, simplified spiking neuron model known for its biological realism and computational efficiency. The model includes two main equations governing the dynamics of the membrane potential *v* and the recovery variable *u*.

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I$$

$$\frac{du}{dt} = a(bv - u)$$

where *I* is the input current, and *a*, *b*, *c*, and *d* are constants governing the behavior of the neuron.

Whenever *v* exceeds a threshold (set to 30mV), the neuron spikes and is reset. Upon firing, *v* is reset to a value *c*, and *u* is updated with a value *d*.

Synaptic Connectivity: Feedforward Network

The SNN is based on a simple feedforward architecture where each neuron in a layer connects to every neuron in the subsequent layer. There is no recurrent feedback between layers as of now, but they can be added later.

Dataflow: Clock-Driven

The dataflow model chosen for this SNN is time-stepped simulation. This method ensures that each timestep is handled sequentially, with spikes communicated between neurons. Each timestep represents an iteration where neurons compute their membrane potential, check if they exceed the firing threshold, and propagate spikes to the next layer.

Training Algorithm: STDP (Spike-Timing Dependent Plasticity)

The synaptic weights are updated via STDP (Spike-Timing Dependent Plasticity), where the strength of the connection depends on the relative timing of spikes between the pre- and post-synaptic neurons. The weight update rule for STDP is:

$$\Delta w = \eta \cdot \begin{cases} A_+ \cdot \exp\left(-\frac{\Delta t}{\tau_+}\right) & \text{if post spikes after pre (long-term potentiation)} \\ A_- \cdot \exp\left(\frac{\Delta t}{\tau_-}\right) & \text{if pre spikes after post (long-term depression)} \end{cases}$$

where Δt is the time difference between the pre- and post-synaptic spikes; η is the learning rate; τ_+ and τ_- are time windows for potentiation and depression respectively; and A_+ and A_- are constants governing the LTP and LTD respectively.

Base-2 Approximation

STDP equations are made from exponential functions. Accuracy and hardware cost are the main concerns simultaneously when a large-scale implementation is targeted. The exponential function can be converted to a base-2 function.

$$exp(x) \cong 2^{x+2^{-1}x-2^{-4}x} \cong 2^{1.4375x}$$

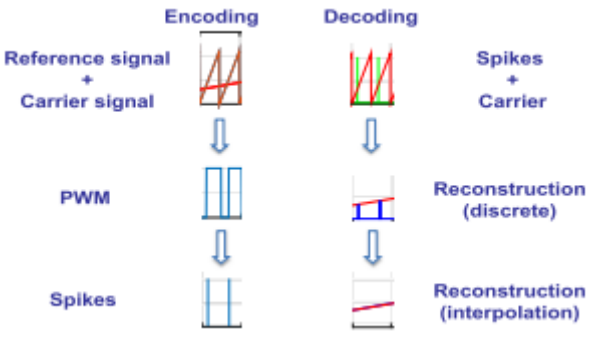
The approximated equations therefore become:

$$\Delta w = \begin{cases} \Delta w^+ = A^+.2^{-1.4375(\frac{\Delta t}{\tau^+})}, & \text{if } \Delta t \geq 0 \\ \Delta w^- = -A^-.2^{1.4375(\frac{\Delta t}{\tau^-})}, & \text{if } \Delta t \leq 0 \end{cases}$$

Spike Encoding Scheme: PWM based Encoding

To encode input stimuli into spikes, PWM encoding is used. Pulse-width modulation (PWM) is a method of encoding the intensity of the input signal into the width of the pulses. This encoding allows the input data to be represented as a series of spikes, which can be processed by the SNN.

To decode the signal, the spike timings are used to recover the original values. After the discrete values are recovered, the full signal can be reconstructed using interpolation methods, ensuring an accurate representation of the original signal from the encoded spike train.



Graphical representation of the PWM-based algorithm for spike encoding and decoding of analog signal

Python Simulations

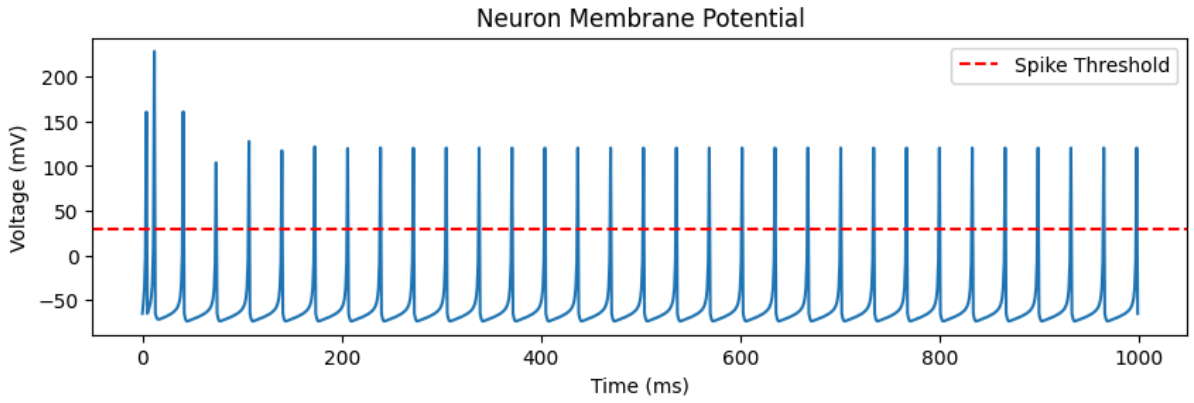
Izhikevich neuron

Code:

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters for the Izhikevich Neuron Model
a = 0.02
b = 0.2
c = -65
d = 8
Vth = 30  # Spike threshold
Vreset = -65
I = 15  # Input current
dt = 1.0  # time step (ms)
T = 1000  # total simulation time (ms)
time = np.arange(0, T, dt)

# Initializing variables for the neuron
v = Vreset * np.ones_like(time)  # Membrane potential
u = b * v  # Recovery variable
spikes = []  # Store spike times

# Simulate the Izhikevich neuron
for t in range(1, len(time)):
    if v[t-1] >= Vth:
        v[t] = c  # Reset voltage
        u[t] = u[t-1] + d
        spikes.append(t)
    else:
        dv = 0.04 * v[t-1]**2 + 5 * v[t-1] + 140 - u[t-1] + I
        du = a * (b * v[t-1] - u[t-1])
        v[t] = v[t-1] + dv * dt
        u[t] = u[t-1] + du * dt

# Plot the neuron's membrane potential and spikes
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(time, v)
plt.title('Neuron Membrane Potential')
plt.axhline(Vth, color='r', linestyle='--', label="Spike Threshold")
plt.xlabel('Time (ms)')
plt.ylabel('Voltage (mV)')
```

```
plt.legend()
```

Result:



Base-2 STDP

Code:

```
# STDP Parameters
A_plus = 0.005
A_minus = -0.005
tau_plus = 20.0
tau_minus = 20.0

#potentiation spikes
pre_time_p = 5
post_time_p = 15

#depression spikes
pre_time_d = 17
post_time_d = 8

# Synaptic weight (initial)
w_i_p = 0.5
w_i_d = 0.5

dt_p = post_time_p - pre_time_p
w_f_p = w_i_p + A_plus * 2 ** (1.4375 * (-dt_p / tau_plus))  # Potentiation

dt_d = post_time_d - pre_time_d
w_f_d = w_i_d + A_minus * 2 ** (1.4375 * (dt_d / tau_minus))  # Depression

print('initial weight =', w_i_p)
```

```
print('post potentiation weight =', w_f_p)
print('post depression weight =', w_f_d)
```

Result:

```
initial weight = 0.5
post potentiation weight = 0.5030381183999512
post depression weight = 0.4968066889346772
```

PWM Based Encoding and Decoding

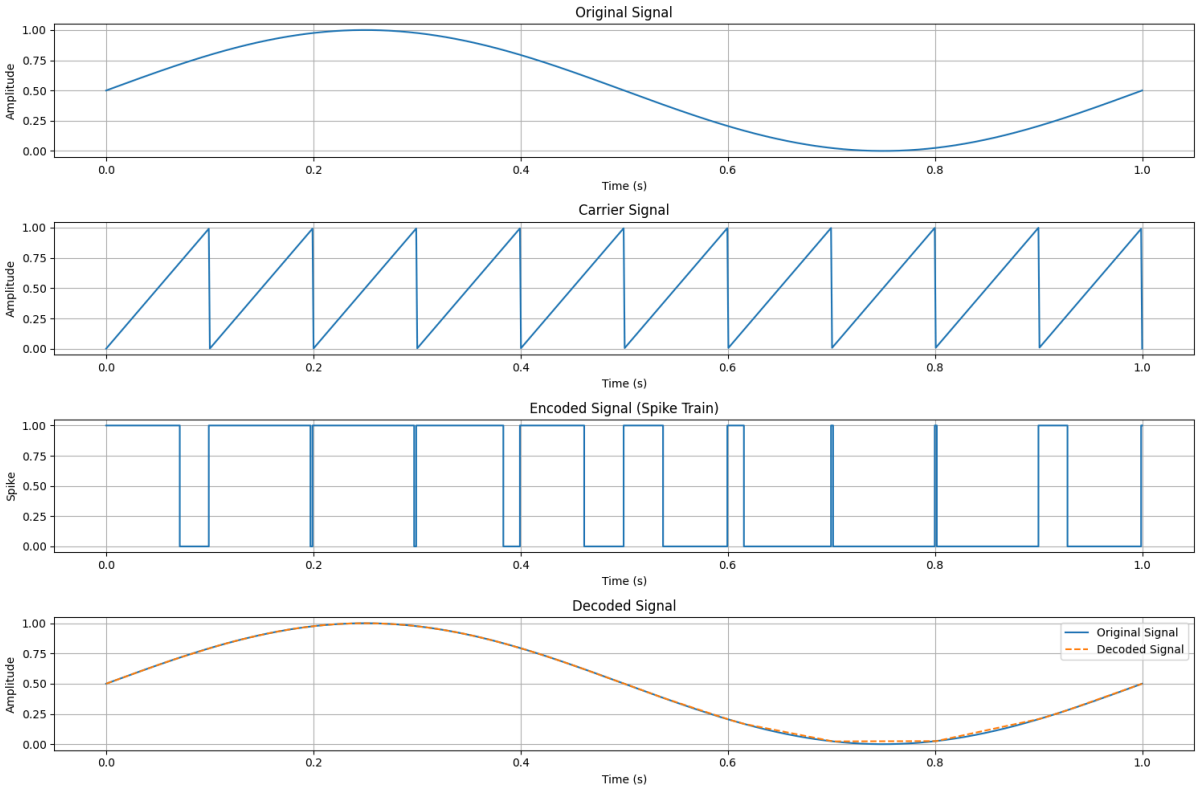
Code:

```
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Define the reference signal (e.g., a sine wave)
def reference_signal(t):
    return 0.5 * np.sin(2 * np.pi * 1 * t) + 0.5  # Sine wave in
[0, 1]

# Step 2: Generate the carrier signal (sawtooth wave)
def carrier_signal(t, frequency):
    return (t * frequency) % 1

# Step 3: PWM encoding
def pwm_encode(reference, carrier):
    spikes = reference > carrier
    return spikes

# Step 4: PWM decoding
def pwm_decode(spikes, carrier):
    decoded_points = carrier[spikes]
    return decoded_points

# Step 5: Reconstruct the signal from decoded points
def reconstruct_signal(decoded_points, t):
    interpolated_signal = np.interp(t, t[decoded_points],
reference_signal(t)[decoded_points])
    return interpolated_signal
```

```
# Parameters
sampling_rate = 1000  # Hz
duration = 1  # second
time = np.linspace(0, duration, sampling_rate * duration)
carrier_freq = 10  # Hz

# Signals
ref_signal = reference_signal(time)
carrier = carrier_signal(time, carrier_freq)

# Encoding
spike_train = pwm_encode(ref_signal, carrier)

# Decoding
decoded_points = np.where(spike_train)[0]
decoded_signal = reconstruct_signal(decoded_points, time)

# Plotting
plt.figure(figsize=(15, 10))

# Original Signal
plt.subplot(4, 1, 1)
plt.plot(time, ref_signal, label="Original Signal")
plt.title("Original Signal")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid()

# Carrier Signal
plt.subplot(4, 1, 2)
plt.plot(time, carrier, label="Carrier Signal")
plt.title("Carrier Signal")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.grid()

# Encoded Signal
plt.subplot(4, 1, 3)
plt.plot(time, spike_train, label="Spike Train",
drawstyle='steps-pre')
plt.title("Encoded Signal (Spike Train)")
plt.xlabel("Time (s)")
```

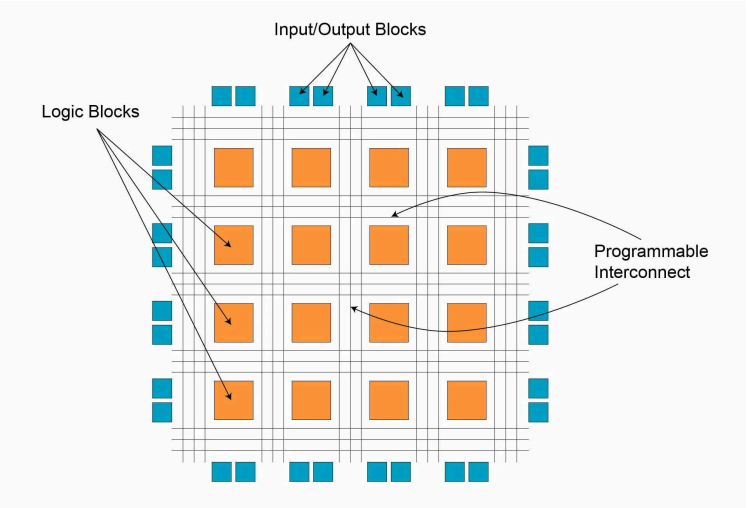
```
plt.ylabel("Spike")
plt.grid()

# Decoded Signal
plt.subplot(4, 1, 4)
plt.plot(time, ref_signal, label="Original Signal")
plt.plot(time, decoded_signal, label="Decoded Signal",
linestyle='dashed')
plt.title("Decoded Signal")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.legend()
plt.grid()

plt.tight_layout()
plt.show()
```

Result:



FPGA Architecture

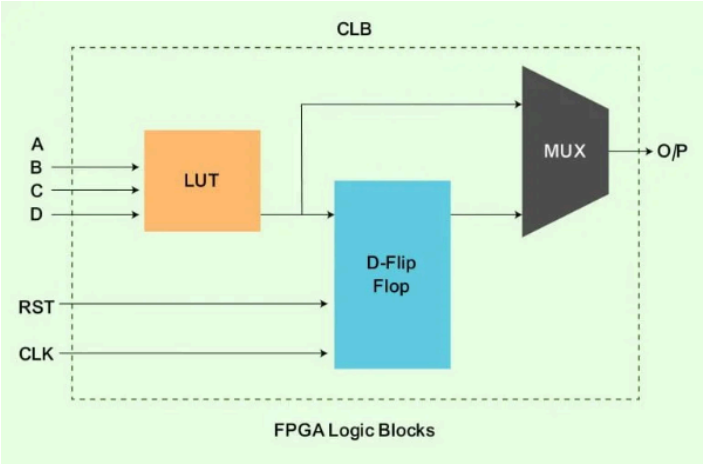
A basic FPGA architecture consists of thousands of fundamental elements called configurable logic blocks (CLBs) surrounded by a system of programmable interconnects, called a fabric, that routes signals between CLBs. Input/output (I/O) blocks interface between the FPGA and external devices.



Blueprint of a bare bones FPGA

CLB

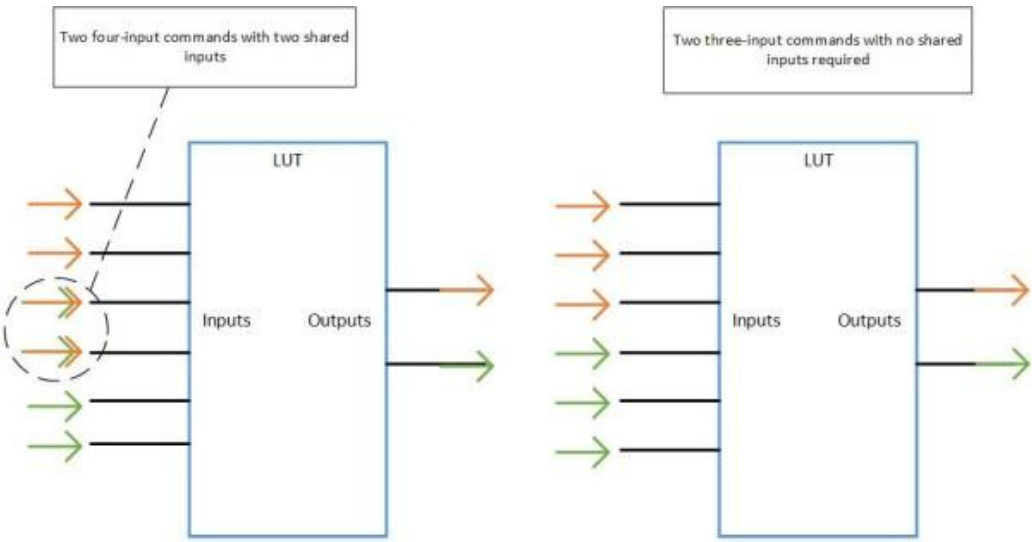
A CLB gives the FPGA its ability to accept different hardware configurations. CLBs can be programmed to perform almost any logic function. The individual CLB contains a number of discrete logic components including look-up tables (LUTs) and flip-flops. Current-generation FPGAs include more complex CLBs capable of multiple operations with a single block; CLBs can combine for more complex operations such as multipliers, registers, counters and even digital signal processing (DSP) functions.



A simple CLB

LUT

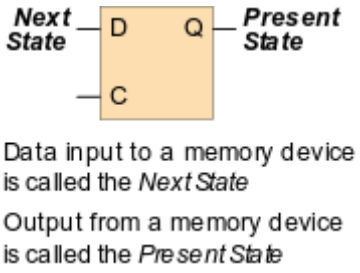
A lookup table (LUT) determines what the output is for any given input(s). In the context of combinational logic, it is the truth table and defines how combinatorial logic behaves. A truth table is a predefined list of outputs for each combination of inputs. The LUT holds a custom truth table that is loaded when the chip is powered up.



LUTs with four to six input bits are widely used

Flip-Flops

A flip-flop is a circuit that has two stable states and can be used to store state information. Flip-flops are binary shift registers that synchronize logic and save logical states between clock cycles within an FPGA circuit. A flip-flop stores a single bit of data.



A D flip-flop

Programmable Routing

In FPGAs, routing is made up of wire segments of variable lengths that are joined by electrically programmable switches. To complete a user-defined design unit, programmable routing connects logic blocks and input/output blocks. It consists of multiplexers, pass transistors and tri-state buffers. Pass transistors and multiplexers are used in a logic cluster to connect the logic elements.

Input/Output (IO) Blocks

They are the components through which data transfers into and out of the FPGA. Input and output on the chip goes through component groups called IO banks, which consist of individual IO blocks. The IO blocks themselves are configurable in a number of ways depending on the type of data the user is either expecting to receive or transmit.

Transceivers

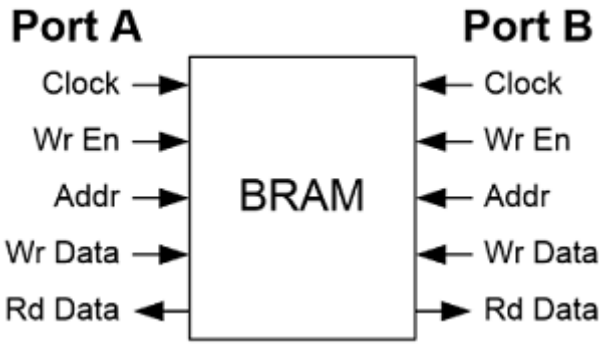
They are made to transmit and receive serial data (individual bits) to and from the FPGA at extremely high rates. Having a dedicated component available for this allows for easy implementation of high speed data transfer by the user without consuming the logic resources of the FPGA.

Digital Signal Processing (DSP) Slice

This is one of the specialized components in an FPGA. It is designed to carry out digital signal processing functions, such as filtering or multiplying, much more efficiently than if the same functions were implemented using many CLBs.

Block Random Access Memory (BRAM)

The dedicated memory on the chip itself is referred to as block RAM or BRAM. While each block individually is of a set size, these blocks can be subdivided or cascaded to make smaller or larger sizes of BRAM available. They also are capable of a variety of operational settings and can support special functionality such as error correction.



Dual Port BRAM

Hardware Blocks

Neuron Processing Unit (NPU)

The (NPU) is responsible for simulating individual neurons' behaviors based on the spiking neuron model. It computes membrane potential (V) over time, fires a spike when the membrane potential exceeds the threshold, and handles reset conditions (e.g., refractory periods). FPGA resources like DSP slices or LUTs can be used for calculations like additions, multiplications, and exponentials. The NPU also requires a Finite State Machine (FSM) to handle neuron states (e.g., resting, active, spiking).

Synaptic Weight Memory

This block stores the synaptic weights for connections between neurons. It stores and retrieves weights for synaptic connections, and updates weights dynamically based on learning rules. Block RAM (BRAM) in FPGA is used to store weights for fast access. The memory can be organised for simultaneous weight access by neurons.

Event Router (Spike Communication Block)

The Event Router is responsible for transmitting spike events from one neuron to another, managing the connectivity structure of the network. It routes spike events to connected neurons and handles full or sparse connectivity. A crossbar switch or multiplexer can be used for spike communication, depending on the network's connectivity.

Input Interface

This block handles input stimuli (e.g., sensory data or digitized signals) and converts them into spike events. It converts continuous signals into spike events and handles time-encoded or rate-encoded representations. It often uses GPIOs and ADCs for external inputs and BRAM to store pre-processed input data.

Output Interface

It outputs the spike activity or results of network computation. It sends spike data to external systems (e.g., visualization tools or actuators) and decodes the network output for control tasks. GPIOs, LEDs, or UART are usually used for serial output.

Spike Scheduler

The Spike Scheduler controls the scheduling of neuron updates. It enforces synchronous or event-driven updates for neurons, and manages timing to ensure deterministic operation. It uses clock dividers or FSMs for scheduling.

Learning Unit (Optional)

This block implements plasticity rules such as STDP or Hebbian learning. Its main task is to update synaptic weights based on spiking activity. Arithmetic units are utilised for weight updates, and memory for storing and retrieving weights.

Global Controller

The Global Controller manages the network's initialization, operation, and reset conditions. It initialises neurons and synaptic weights, and controls simulation steps (e.g., start, stop, reset). An FSM can control the entire system, ensuring correct sequencing.

Clocking and Timing Blocks

These blocks synchronize all operations across the FPGA. They generate clock signals for neuron updates and timing of spike events. FPGA's internal MMCM (Mixed-Mode Clock Manager) or PLL (Phase-Locked Loop) is generally used for timing synchronization.

Debugging and Monitoring Interfaces

These blocks allow for testing and debugging during design and deployment. They monitor neuron states, synaptic weights, and spike events. Data for analysis is obtained via tools like JTAG (Joint Test Action Group) or UART.

Future Work

Given the evidence of the viability of the design decisions taken, a neural network would be designed for the desired application, which is temporal processing. The network would be simple enough to be implemented on an FPGA, but it should perform the task with reasonable accuracy. Resource requirements for the model would be computed, and an FPGA would be chosen.

From the study of the architecture of the FPGA, the model would be adapted for hardware implementation using Verilog. The design would be checked for performance, and be optimised accordingly. If our resources allow, the model may be made more complex.

This process would involve frequent iterations through the hardware model, and possibly, the software model as well, as both go hand in hand.

References

[1] Pham, Quoc Trung, et al. "A review of SNN implementation on FPGA." 2021 international conference on multimedia analysis and pattern recognition (MAPR). IEEE, 2021.

[2] Wang, Kuanchuan, et al. "Comparison and Selection of Spike Encoding Algorithms for SNN on FPGA." IEEE Transactions on Biomedical Circuits and Systems 17.1 (2023): 129-141.

[3] Arriandiaga, Ander, et al. "Pulsewidth modulation-based algorithm for spike phase encoding and decoding of time-dependent analog data." IEEE Transactions on Neural Networks and Learning Systems 31.10 (2019): 3920-3931.

[4] Gomar, Shaghayegh, and Majid Ahmadi. "Digital realization of PSTDP and TSTDP learning." 2018 International Joint Conference on Neural Networks (IJCNN). IEEE, 2018.

[5] Izhikevich, Eugene M. "Simple model of spiking neurons." IEEE Transactions on neural networks 14.6 (2003): 1569-1572.

[6] Smith, S. T., and A. R. Hall. "FPGA-Based Hardware for Spiking Neural Networks." Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002.

[7] Farooq, Umer, et al. "FPGA architectures: An overview." Tree-Based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization (2012): 7-48.