EE-705

VLSI Design Lab

FPGA Accelerator for Spiking Neural Networks

Anubhav Bhatla, 200070008

Hemant Hajare, 20d070037

Navneet, 200070048

May 6, 2024

Contents

Acknowledgement

We would like to express our gratitude to Prof. Sachin B. Patkar and the TAs of the EE705 course for providing us with the opportunity to work on such an insightful project.

Anubhav Bhatla
Hemant Hajare
Navneet

1 Proposal

The goal is to develop a hardware accelerator for an SNN for high-performance inference. The model used inside the neuron will be Integrate-and-Fire (IF). The inference capabilities of the accelerator shall be evaluated using the MNIST dataset, where each image is a 28x28 matrix of pixels.

2 Introduction

Convolutional Neural Networks (CNNs), commonly used in ANNs, require significant computational resources for each neuron, leading to inefficient resource utilisation. Spiking Neural Networks (SNNs), an emerging event-based neural network type, offer a solution by exchanging information via binary spikes, minimising resource usage. SNNs treat time as an additional dimension, making them suitable for processing time series data.
Although SNNs have been primarily implemented on CPUs and GPUs, the demand for specific ASIC processors has risen due to the inefficiency of traditional computing architectures in supporting the sparse features of SNNs. Field Programmable Gate Arrays (FPGAs) provide a promising solution for implementing accelerators at the edge due to their programmability.

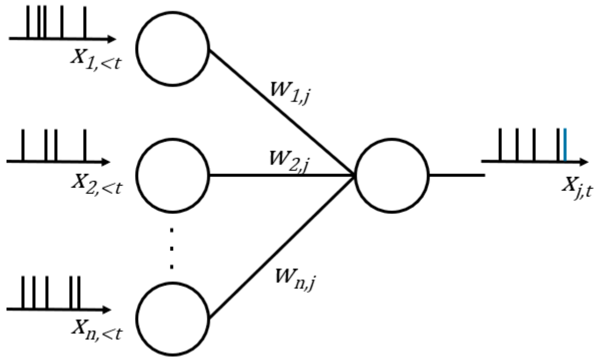


Figure 1: Spiking Neural Network

3 Design

3.1 Neuron

The neuron block is designed to simulate the working of an actual brain neuron using the Integrate-and-Fire neuron model with a refractory period. The potential of a neuron is defined by the input spike coming in from each of its connections multiplied by the weight of the connection. Once this potential value reaches a threshold, and the neuron is not in its refractory period, an output spike is generated. An overview of the neuron block is given in Fig. 2

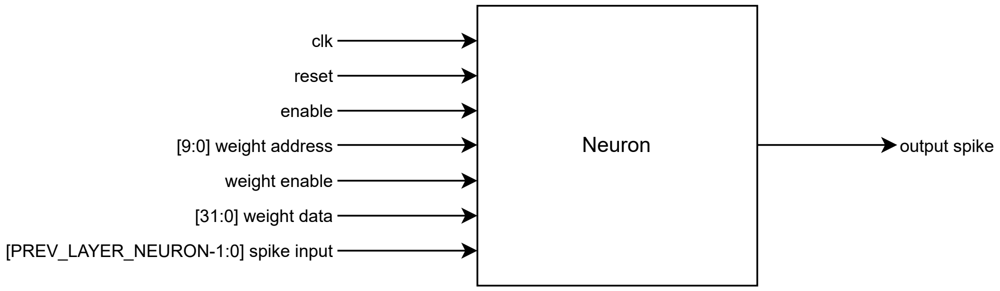


Figure 2: Overview of the neuron block

3.2 Input Layer

The input layer comprises 784 neurons, each corresponding to a pixel in the MNIST image. The MNIST image is encoded as a spike train so that it can be correctly interpreted by our neuron logic. Each of these neurons receives a spike train as their input and sends out an output spike train.

3.3 Hidden Layer

The hidden layer comprises 100 neurons which are used to perform classification of the image. The number of these neurons can be varied. Increasing the number of neurons helps improve accuracy but requires more memory bits (more weights stored in ROM) while decreasing it will impact classification accuracy.

3.4 Max-comparer

The max comparer block takes the spike count of each of the hidden layer neurons and finds the neuron with the highest spike count. The prediction made by this neuron has the highest probability of being correct, therefore we shall use this assignment as our classification output. An overview of the max-comparer block is given in Fig. 3

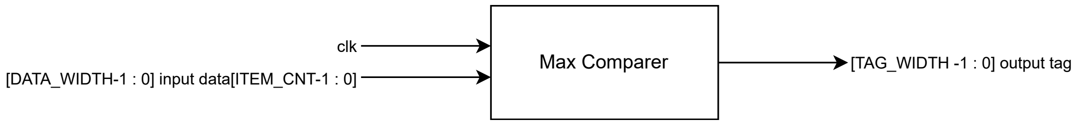

clk
[DATA_WIDTH-1 : 0] input data[ITEM_CNT-1 : 0]
Max Comparer
[TAG_WIDTH -1 : 0] output tag

Figure 3: Overview of the max-comparer block

3.5 IPs

We use the BRAM IP provided by Vivado to instantiate the different read-only memories required in our implementation. We require three different ROM instances, the first one to store the input image in a spike-encoded format, the second one to store all the weights for the synapse connections between input-layer neurons and hidden-layer neurons. The third ROM is used to define the classification assignment for each of the hidden-layer neurons. Table 1 describes the properties of the various BRAM instances used.

BRAM instance	**Width (bits)**	**Depth (entries)**
Input Memory	784	200
Weight Memory	3200	784
Assignment Memory	400	2

Table 1: Overview of the BRAM instances used in our design

3.6 SNN

The complete block diagram of our design is given in Fig. 4. The input memory stores an MNIST image in an encoded format which is then read by each of the input neurons. These neurons then propagate their output spikes to each of the hidden layer neurons. The weights of each of these synapses are provided by the weight memory. The output of these hidden layer neurons is sent to the max-comparer block which finds the neuron with the highest spike count. The assignment corresponding to the neuron is found using the assignment memory, which finally gives us our final output.

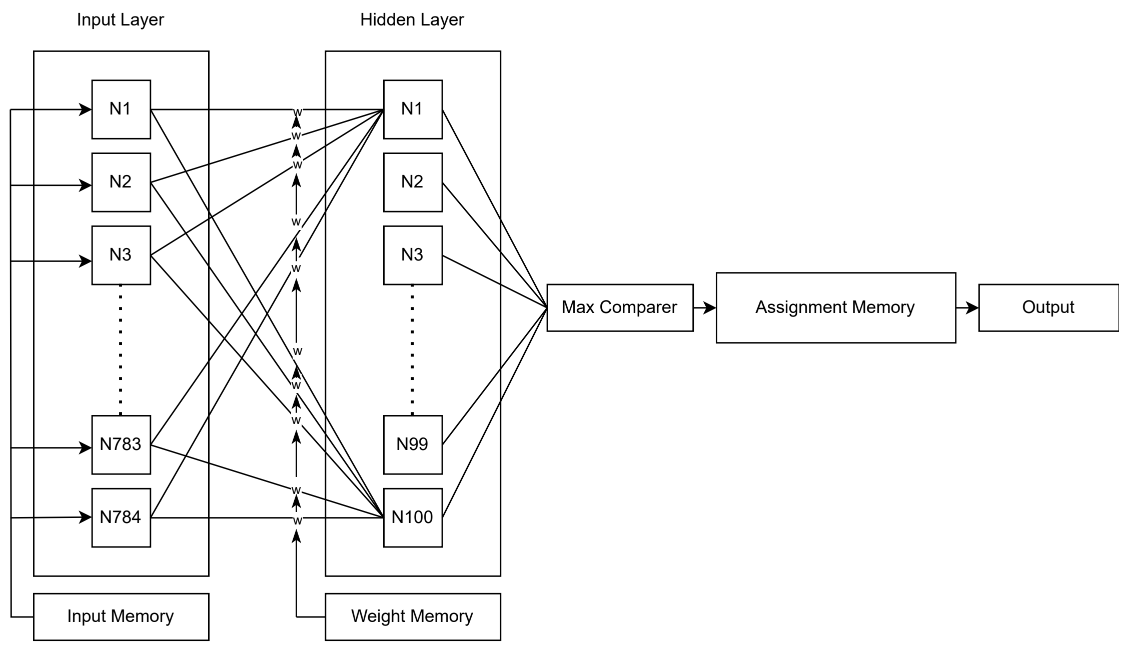


Figure 4: Overview of the entire SNN flow

4 Testbench

The testbench is quite straightforward, with the `clk`, `rst`, `en` signals being sent to the DUT and the `result` being read from it. However, it is important to note that the DUT must be simulated for a long enough time so that the `timestep` reaches its maximum value (200), i.e., the entire input spike train has been read successfully and the classification output is steady.

```
`timescale 1ns / 1ns

module snn_sim();

    logic clk, rst, en;
    always begin clk <= 1'b1; #5; clk <= 1'b0; #5; end
    initial begin rst = 1'b1; #18; rst = 1'b0; end
    assign en = 1'b1;

    logic [3:0] result;
    snn dut(clk, rst, en, result);

endmodule
```

5 Simulation

5.1 Input Format

The MNIST image is provided as a set of input spikes to each input neuron. Each input neuron receives values for 200 time steps, and these spikes represent the data at that pixel in the MNIST image. To generate this encoded image, we use the scripts provided by https://github.com/oshears/fpga_snn_models. We simulate our setup for different MNIST labels and note the classification accuracy for the same.

5.2 Simulation output

The input image has a label of 5. Fig. 5 shows the output waveform for this test case. We can observe that after `timestep` reaches its final value (`ca`), and stays stable suggesting all the timesteps have been processed the classification result is 5.

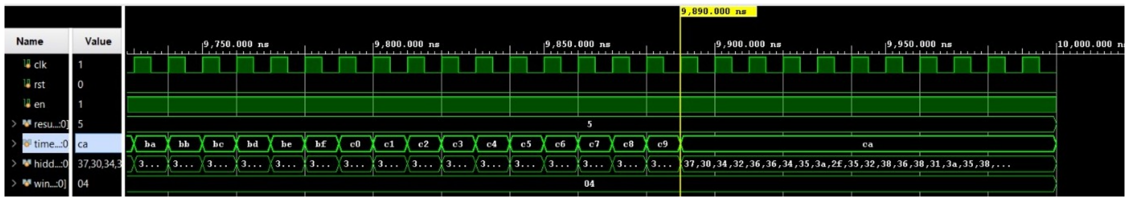
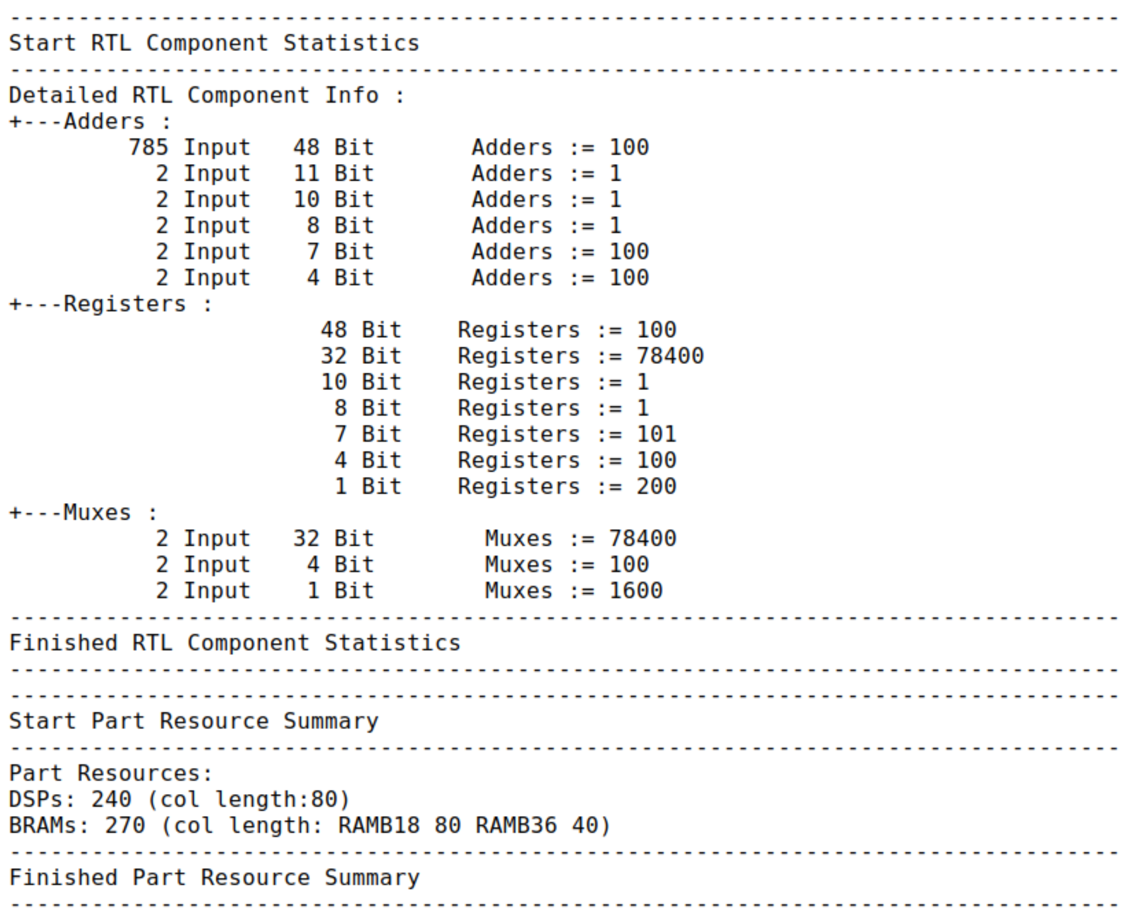


Figure 5: Simulation output for test case

6 Synthesis

6.1 Resource Usage

Fig. 6 shows the overall resource usage of our design. We can observe that due to the use of such large ROMs, the BRAM resource usage is quite high at around 270 BRAMs.



```
---------------------------------------------------------------------------------
Start RTL Component Statistics
---------------------------------------------------------------------------------
Detailed RTL Component Info :
+---Adders :
	 785 Input   48 Bit       Adders := 100
	   2 Input   11 Bit       Adders := 1
	   2 Input   10 Bit       Adders := 1
	   2 Input    8 Bit       Adders := 1
	   2 Input    7 Bit       Adders := 100
	   2 Input    4 Bit       Adders := 100
+---Registers :
	                48 Bit    Registers := 100
	                32 Bit    Registers := 78400
	                10 Bit    Registers := 1
	                 8 Bit    Registers := 1
	                 7 Bit    Registers := 101
	                 4 Bit    Registers := 100
	                 1 Bit    Registers := 200
+---Muxes :
	   2 Input   32 Bit        Muxes := 78400
	   2 Input    4 Bit        Muxes := 100
	   2 Input    1 Bit        Muxes := 1600
---------------------------------------------------------------------------------
Finished RTL Component Statistics
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
Start Part Resource Summary
---------------------------------------------------------------------------------
Part Resources:
DSPs: 240 (col length:80)
BRAMs: 270 (col length: RAMB18 80 RAMB36 40)
---------------------------------------------------------------------------------
Finished Part Resource Summary
---------------------------------------------------------------------------------
```

Figure 6: Detailed resource summary for our design.

7 Challenges faced

- **Generating encoded input images:** The scripts for generating the input image binaries had a few bugs and outdated library functions. We also had to write additional scripts to generate a `.coe` file using a binary.
- **Resource usage:** Due to the use of large ROMs to store weights and the input image, the design has a very high resource usage. We were not able to compile the design on Quartus so we decided to used Vivado for compilation.
- **RTL:** The synthesis of the design was very time consuming. The total time taken for synthesis was well more than 12 hours. We had to stop the synthesis after we got an account of the resource usage.