Hardware Implementation of Spiking Neural Networks

Anatoly Syutkin

A Thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science (Electrical Engineering)
Concordia University
Montreal, Quebec, Canada

April 2021

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Anatoly Syutkin**

Entitled: **Hardware Implementation of Spiking Neural Networks**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Sebastien Le Beux

_____ Examiner
Dr. Sebastien Le Beux

_____ Examiner
Dr. Charalambos Poullis

_____ Supervisor
Dr. Glenn Cowan

Approved by _____
Dr. Y. R. Shayan, Chair
Department of Electrical and Computer Engineering

April 13, 2021

_____
Mourad Debbabi, Ph.D., P.Eng., Dean
Faculty of Engineering and Computer Science

Abstract

Hardware Implementation of Spiking Neural Networks

Anatoly Syutkin

The fields of Machine Learning and Artificial Intelligence have made great strides in the last decade due to the increasing computational power of Graphics Processing Units (GPUs). Neural networks make up for a very large portion of this research area, and come in great variety (e.g. feedforward, convolutional, etc.). Although they are inspired by the human brain, they have no biological plausibility aside from the high interconnectivity of nodes. Spiking Neural Networks (SNNs) are a step in the direction of greater biological plausibility with the use of inherently dynamic neurons.

As implied by the name, SNNs are composed of neurons that generate Boolean spikes when their accumulated input exceeds a threshold value. Thus, information is encoded in the timing of spiking events. Although they are computationally expensive to simulate with general-purpose computers, their dynamic behavior lends itself well to direct hardware implementations with very high parallelism and low power consumption.

This thesis proposes a scalable architecture for a hardware system that can be used to study the behavior of SNNs, as well as the trade-offs that result from the various design parameters. Using classic benchmark problems (i.e. MNIST classification and cart-pole stabilization), it was observed that SNNs are very robust against variations in neural parameters, but degrade quickly with mismatch in synaptic weights. An MNIST classification accuracy of 96.28% drops by $< 2\%$ for severe neural variations, but $> 5\%$ for small synaptic mismatches. Additionally, the performance is re-evaluated for several weight quantizations. Finally, the effects of router delays are observed.

Acknowledgements

I wish to express my gratitude:
To Concordia, FRQNT and NSERC for the research opportunities and financial support;
To my family and friends, who helped and encouraged me;
To Michael Segev, Michael Wright, and Jerrin Pathrose, for their collaboration;
And, of course, to my supervisor Dr. Glenn Cowan, for all of his guidance and advice.

Contents

List of Figures

List of Tables

List of Abbreviations

ANN Artificial Neural Network. 6, 7

CAM Content-Addressable Memory. 3

DAC Digital-to-Analog Converter. 78

FSM Finite-State Machine. 32

IC Integrated Circuit. 2

IF Integrate-and-Fire. 20

LIF Leaky Integrate-and-Fire. 12

RX Receiver. 31

SNN Spiking Neural Network. 11

TX Transmitter. 31

Chapter 1

Introduction

Neural Networks make up for a very large portion of Machine Learning (ML), and have a great variety of applications that include control systems, speech recognition, and image classification [1], [2], [3], [4]. As the name indicates, a neural network is a graph that consists of nodes (neurons) and edges (weights). These networks can differ greatly based on the type of application, and vary according to the number and type of neurons, and their connectivity (e.g. the number of layers). In simplest terms, neural networks can be thought of as function approximators [5], which are fitted by adjusting the values of the edges that interconnect the neurons.

Although neural networks are inspired from biology, their behavior and training have no biological plausibility. For example, typical neural networks are static, and only produce a constant output for a constant input. This, of course, has implications for the type of interconnections among the neurons (i.e. non-recurrent connections). The more advanced type of networks are the Spiking Neural Networks [6], which distinguish themselves with dynamic neurons and synapses (connections). As such, they behave with a certain biological plausibility, and combine the fields of computer science, neuroscience, and electronics. As implied by their name, spiking neurons indicate their activity by emitting voltage spike trains (events). Small currents at the input produce slow spike rates, and large currents - fast spike rates. In this work, the static and dynamic networks are referred to as Artificial and Spiking Neural Networks, respectively.

The specific architecture and type of neural network affect the choice of the training method, which is often based on the Backpropagation algorithm [7]. In the context of supervised learning, the output of an ANN is compared with the desired result, and the errors are propagated backwards through the layers of the network in order to adjust the weights (connections)

and biases of the neurons. Given the dynamic nature of SNNs, the backpropagation method cannot be applied in a straightforward fashion.

In practice, neural networks are trained and executed on graphics processing units (GPUs). However, in terms of power consumption, these may not always be the optimal solution due to the use of general-purpose computing architectures. This is especially true for SNNs that require the simulation of a set of differential equations.

This work proposes a scalable architecture for an integrated circuit (IC) that can be used to compute the behavior of an SNN. This architecture is evaluated with two benchmarks: the MNIST handwritten digit classification and the balancing of a cart-pole system. In this design, the basic computational units are the neuron and the synapse. The effects of parametric variation and the delays of the routing schemes are also observed.

1.1 SNN IC Architecture

The complete system presented in this work can be broken down into three fundamental components: a set of voltage signal encoders, an SNN integrated circuit, and a spike-train converter. Note that a single integrated circuit may have multiple cores, and it is the design of a single core that is presented in this thesis.

As mentioned above, a core consists of a set of neurons and synapses, and the strengths of the synapses (weights) are stored in memory modules. Although the neurons and synapses can be implemented in either digital or analog hardware, a scalable architecture requires the use of routers, which are digital. When a spike is produced by a neuron, it is encoded as the address of the neuron. This is known as *Address Event Representation* [8]. Hence, if a neuron produces multiple spikes, the address of that neuron is captured several times by a local router, and the activity of the neuron is encoded by the timing of those spikes. For this reason, it is crucial to minimize router delays, or at least keep them constant for all neurons. Event collisions occur when multiple neurons emit spikes simultaneously. These collisions may be dealt with differently, but in this work, an arbiter is used.

By their nature, SNNs are scalable and can be made arbitrarily large. For this reason, the architecture is designed in such a way that allows the tiling of multiple cores in a torus arrangement. Therefore, the router is responsible for broadcasting the events of local neurons to neighboring cores, and also to act as a transmission unit for non-adjacent ones. When the router captures external events, they may be delivered to the local neurons through the

use of content-addressable memory (CAM).

Given that the SNN core (or a multi-core IC) can only accept or produce address events, an encoder is required to convert a set of analog input signals into a set of spike trains. Constant inputs result in constant spike rates, and time-varying inputs result in spike trains with variable spike densities.

The neurons that are designated as network outputs emit spike trains that are converted to analog or digital signals. This is achieved by decoding the addresses that are transmitted from a core back into spike trains, and processing them with low-pass filters.

It should also be stated that a particular design has not been selected for the neural and synaptic circuits, and all the simulations make use of mathematical models with digital control signals. However, this leaves room for the exploration and comparison of various circuit designs for neurons and synapses.

1.2 Thesis Organization

Chapter 2 introduces the theory for both artificial and spiking neural networks, and highlights certain behaviors that can only be observed in the latter. Existing hardware architectures are also presented in this chapter.

Chapter 3 describes the architecture that is implemented in this work, and shows how multiple cores can be tiled to scale the size of an SNN.

Chapter 4 presents in detail the design of every module composing the SNN core.

Chapter 5 explains how an SNN IC may interface other circuits and systems.

Chapter 6 summarizes the performance of trained SNNs on two benchmark problems: the MNIST classification and stabilization of a cart-pole (inverted-pendulum) system. The results of system-level and hardware-level simulations are presented.

Finally, Chapter 7 concludes the work and presents further objectives, as well as potential improvements.

1.3 Contributions

Overall, the main contribution of this work is the setting-up of a framework that allows for the study of various circuit designs and parameter trade-offs. This is done using system-level simulations in Python and hardware-level simulations in VHDL.

The system-level simulations, which employ mathematical models and do not take routing delays into consideration, have been used to analyze the effect of synaptic strength quantization in order to find the minimal number of bits that are required to represent one synaptic connection while preserving the accuracy of the SNN regarding MNIST classification of handwritten digits.

Process variation during physical manufacturing of an IC always leads to parametric variations of circuits. Hence, another set of system-level simulations was used to observe the effect of increasing parametric variations in neural model parameters (i.e. membrane resistance and capacitance) on the SNN performance. It was concluded that, in regard to the studied benchmark problems, the SNN is fairly robust against such variations.

Next, a similar set of simulations was used to study the importance of synaptic circuit variations, and it was noted that, due to the scaling complexity of the synapses, their matching was far more important to preserve the accuracy of a trained SNN. More precisely, the most important portion of the synaptic circuit is the Digital-to-Analog Converter (DAC), and the relative matching of its bits requires careful design considerations.

At the hardware-level simulations with VHDL, it was determined that the greatest difference in operation between mathematical and hardware models is caused by the routers that are responsible for delivering spikes to and from local and external neurons. These differences may be mitigated by increasing the clock rate that drives the SNN cores, but this is done at the expense of a greater power consumption. Also, one of the proposed solutions consists of a module that allows the bypassing of the routing scheme for neurons located on a single core. Although this modification reduces the number of clock cycles to deliver a spiking event, this solution is only applicable to SNNs whose output layers can fit within a single core.

In addition, the designed architecture delivers spikes (and synaptic currents) in a serialized manner. Thus, there is a loss of synchronization of neural spikes receiving equal and constant inputs, and spiking events that are meant to be distributed simultaneously slightly diverge in time. But, it was observed that, in the case of the MNIST benchmark, the classification accuracy of the SNN is not very sensitive to this phenomenon.

In the future, the proposed SNN framework is to be used to study the various methods for mitigating the issues that are described above. Potential solutions include mixed-signal implementations of neural and synaptic circuits, as well as the use of dynamic element matching for improving the performances of the DACs. Alternative routing architecture are to be investigated as well.

Chapter 2

Neural Network Background and Literature Review

The focus of this chapter is the theoretical operation of Spiking Neural Networks (SNNs), as well as a comparison with the Artificial Neural Networks (ANNs) currently employed in Machine Learning methodologies.

2.1 Artificial Neural Networks

An Artificial Neural Network (ANN) is a nonlinear mathematical tool inspired by its biological counterpart: the brain [9]. As indicated by the name, an ANN is a network of interconnected nodes, or neurons, which in turn can essentially be described as functions.

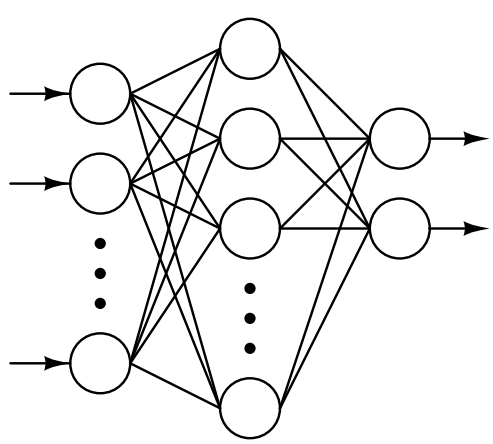


Figure 2.1: Neural Network

As shown in Figure 2.1, an ANN consists of three layers of neurons: an input layer, a hidden layer, and an output layer. In deep learning, the number of hidden layers is increased [10]. In the case of Artificial Neural Networks, the outputs of the input layer neurons are equal to the network inputs. Hence, the input layer is a misleading term as it is not composed of neurons. Neural networks are used in a large variety of tasks with image classification and control systems being the most common examples [2], [4]. Hence, the input vector can consist of an array of pixel intensities of an image, or the state vector of a system being controlled or stabilized.

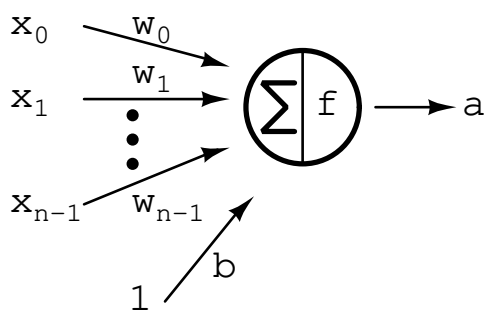


Figure 2.2: Artificial neuron

The basic functionality of a neuron is demonstrated in Figure 2.2. A set of values x_0, x_1, ... x_{n-1} is applied to the synapses at the input of the neuron. Each synapse, or weight, scales the input components, and the resulting products are summed-up. Thus, the weighted input is the dot-product of the input vector x and the weight vector w, as well as an additional biasing term. The resulting neuron output is the activation function applied to the weighted input:

$$a = f(\boldsymbol{w} \cdot \boldsymbol{x} + b) \tag{2.1}$$

The behaviour of biological neurons can be described with differential equations, and these neurons interact among each other by emitting voltage spikes [11]. The information that is being communicated is encoded in the relative timing of these outputs spikes [11]. Therefore, neurons that are very active output many spikes in a short period of time compared to those that are less active in a specified time frame.

Artificial neurons, on the other hand, are described with nonlinear activation functions [9], where a large static output symbolizes a neuron with a high spiking rate.

The simplest artificial neuron was proposed by McCulloch and Pitts, and makes use of a

step function to indicate whether a specific neuron is active or quiet [12]. However, the most commonly used activation functions are the sigmoid and the linear rectifier, as shown in Figures 2.3 and 2.4.

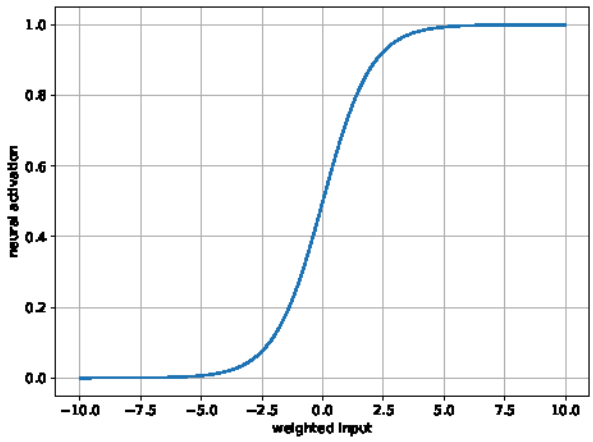


Figure 2.3: Sigmoid activation function

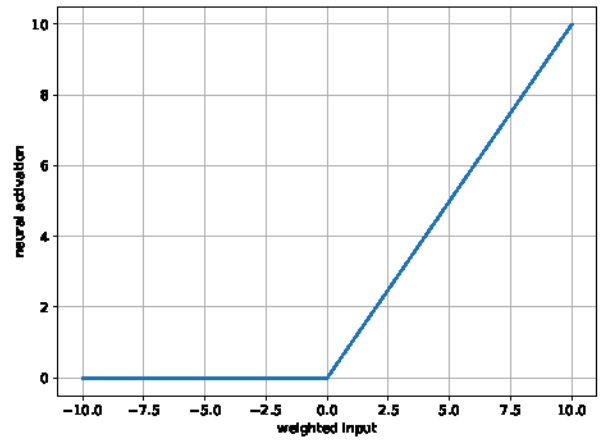


Figure 2.4: ReLU activation function

$$f_{SIG}(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

$$f_{ReLU}(x) = \max(0, x) \tag{2.3}$$

The nonlinearity of the activation function is crucial in the operation of ANNs. Deep learning has gained traction due to the large number of layers that a neural network can have [10]. If linear neurons are used, multiple layers can be expressed with a single equivalent linear layer.

Shown in Figure 2.5 is an ANN with a single input, a single output, and only two neurons in the hidden layer. The vectors $[\boldsymbol{w_0}]_{2\times1}$ and $[\boldsymbol{w_1}]_{1\times2}$ are the weights for the input-hidden and hidden-output interconnections. The biasing vectors $[\boldsymbol{b_0}]_{2\times1}$ and $[\boldsymbol{b_1}]_{1\times1}$ are applied to the hidden and output neurons. Note that in this particular example, the output neuron is linear, and its activation value is simply its weighted input.

The purpose of this example is to demonstrate how a simple neural-network can be used to fit a non-linear static equation. Figure 2.6 demonstrates the variability of the output curve based on the network parameters. In each sub-plot, the network input x_0 is swept five times for five different values of a specified parameter (e.g. $\boldsymbol{w_1}[0]$). For this reason, neural networks can be thought of as function approximators [5].

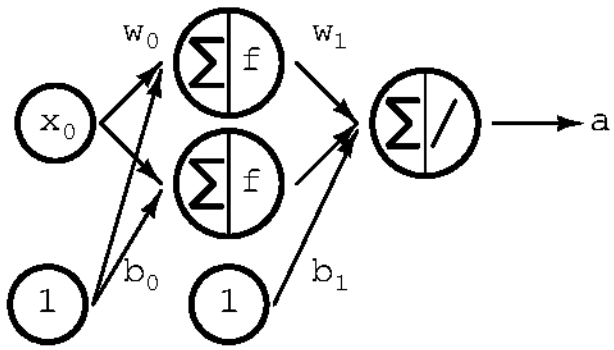
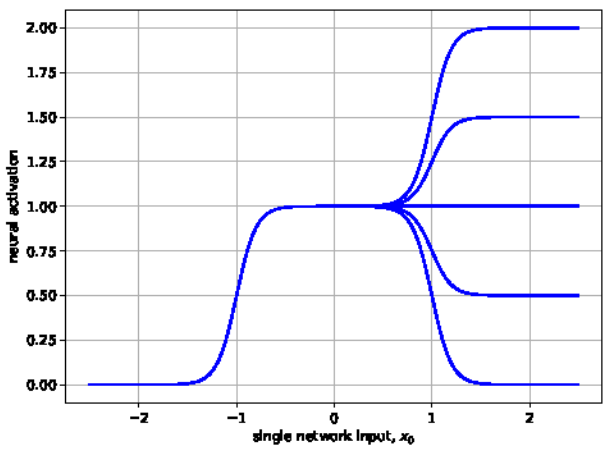
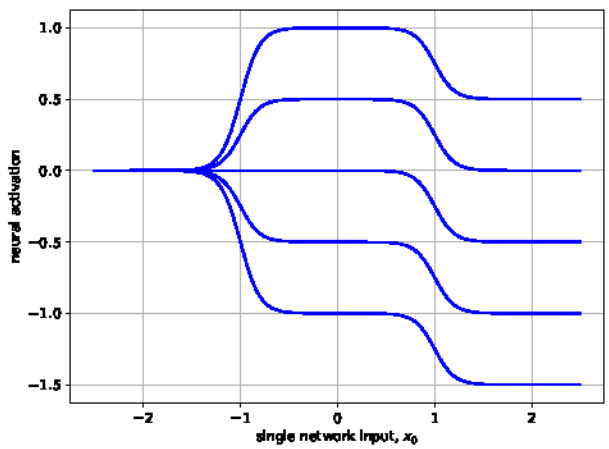


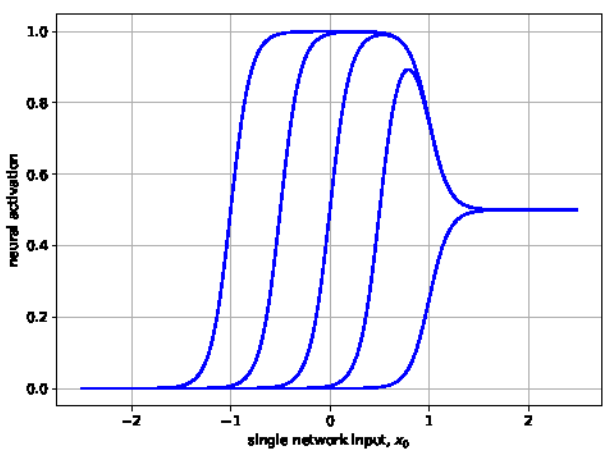
Figure 2.5: ANN layer



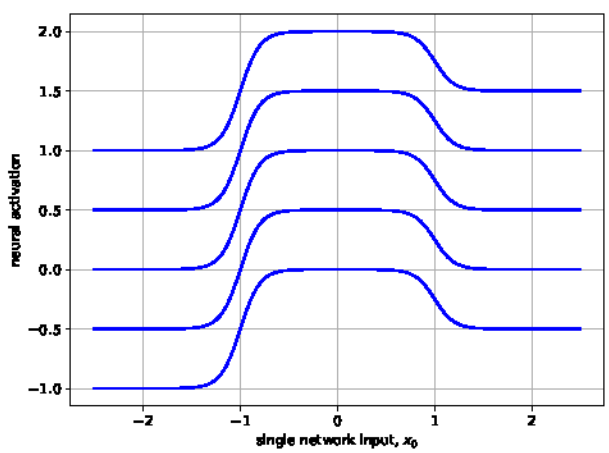
(a) $-1 \leq \boldsymbol{w}_1[0] \leq 1$



(b) $-1 \leq \boldsymbol{w}_1[1] \leq 1$



(c) $-10 \leq \boldsymbol{b}_0[1] \leq 10$



(d) $-1 \leq \boldsymbol{b}_1 \leq 1$

Figure 2.6: ANN parametric sweeping [13]

The output of a neural network with multiple input and multiple output neurons can be evaluated with feed-forward propagation:

1. Apply an input vector $\boldsymbol{a_1} = \boldsymbol{x}$ as input;
2. Compute weighted input of hidden layer: $\boldsymbol{z_1} = \boldsymbol{W_1}\boldsymbol{a_1} + \boldsymbol{b_1}$;
3. Compute activation values of hidden layer: $\boldsymbol{a_2} = f(\boldsymbol{z_1})$;
4. Compute weighted input of output layer: $\boldsymbol{z_2} = \boldsymbol{W_2}\boldsymbol{a_2} + \boldsymbol{b_2}$;
5. Compute activation values of output layer: $\boldsymbol{a_3} = f(\boldsymbol{z_3})$;

If the network has more than one hidden layer, the last two steps are repeated as necessary.

2.2 Spiking Neural Networks

Artificial neural networks are only loosely inspired by biology. Spiking neural networks (SNNs), on the other hand, make use of neurons and interconnecting synapses that are modelled with biologically plausible dynamic equations. Thus, SNNs are the third generation of neural networks [6]. Research in spiking neural networks is driven by the idea that biological plausibility may lead to greater computational power, efficiency and scalability [6], [14]. Although it is hard to define computational power, one may say that a specific network is more computationally powerful than another if it can compute the same function with a smaller number of neurons or parameters [6].

The network in Figure 2.7 below is a modification of Figure 2.1 where the SNN is shown to have recurrent neurons, synapses interconnecting neurons of the same layer, and synapses going into previous layers [15].

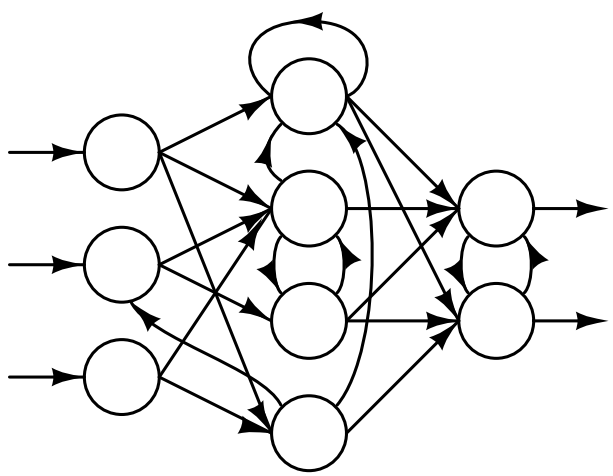


Figure 2.7: Spiking neural network

2.2.1 Neural Modeling

As mentioned previously, spiking neurons are governed by dynamic equations: differential equations that dictate the behaviour of a neuron until a spiking event, and the resetting of the neuron following an output spike. Just as there exists a variety of activation functions for ANNs, there also exist numerous neuron models. Among these models, there is always a trade-off between biological plausibility and computational cost during simulations [16], [17].

2.2.1.1 Leaky Integrate-and-Fire Model

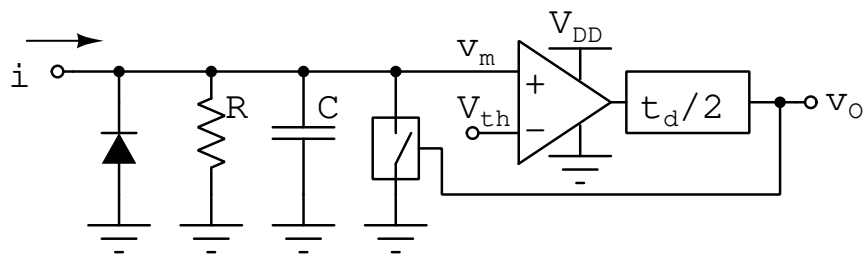


Figure 2.8: LIF neuron model

The simplest spiking neuron model is the leaky integrate-and-fire (LIF) neuron [18]:

$$\frac{dv_m}{dt} = \frac{i(t)}{C} - \frac{v_m(t)}{RC} \tag{2.4}$$

$$\text{If } v_m \geq V_{th} \quad \rightarrow \quad v_o = V_{DD}, \quad v_m = 0\ V \tag{2.5}$$

As seen in Figure 2.8, the LIF neuron has two parameters: membrane resistance R and membrane capacitance C. The input to the neuron is applied as a current signal, which is accumulated in the membrane capacitance. When the membrane voltage v_m become greater than the threshold voltage V_{th}, the operational amplifier outputs a high voltage V_{DD}. After a certain delay $t_d/2$, the ideal reset switch is enabled and the membrane capacitance is discharged to ground, leading the amplifier to output $v_O = 0V$. After another delay of $t_d/2$, the switch is turned OFF and the neuron can resume the integration of the input current. Therefore, the duration of the neuron spike lasts a total of t_d.

An ideal diode is added to the schematic to indicate that the membrane voltage cannot be brought to a negative voltage in case of a negative input current.

The functionality of the LIF neuron model is demonstrated in the following Figure 2.9, where the input current is plotted in orange and the membrane voltage is plotted in blue. The first segment of the simulation is a ramp input current. As this input is steadily increasing, the membrane voltage is rising in response until the first spiking event occurs around $t = 2s$ before the neuron is reset. With the current continuing to increase, the time duration between consecutive neuron spikes becomes shorter. Note that the duration of each spike is equal to the time-step of the simulation.

At time $t = 5s$, a negative input current is applied that quickly discharges the membrane capacitance to ground. In the third segment of the simulation, a constant current is applied.

As a result, the neuron begins emitting spikes at a constant rate. Once the input current is set to $i = 0mA$, the membrane simply discharges through the capacitor.

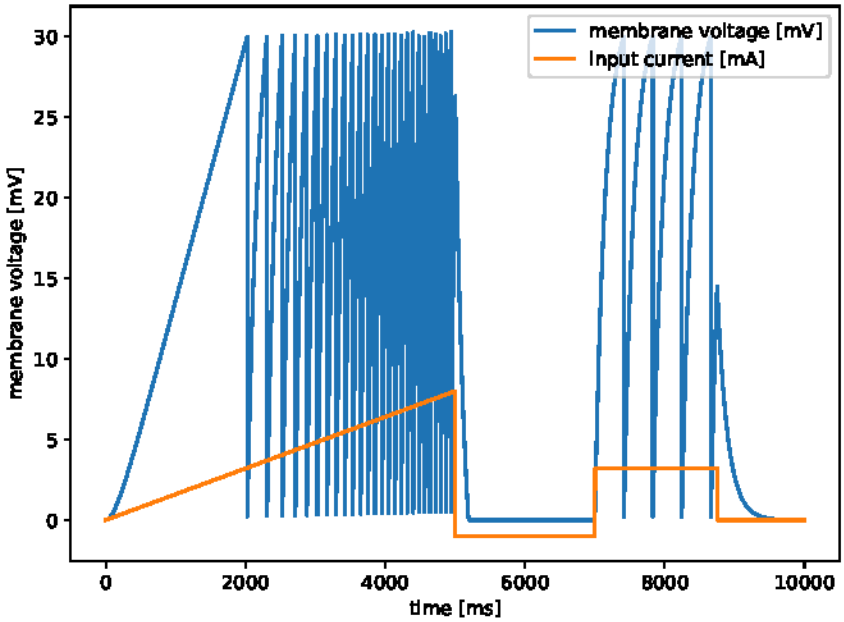


Figure 2.9: Neuron spiking activity

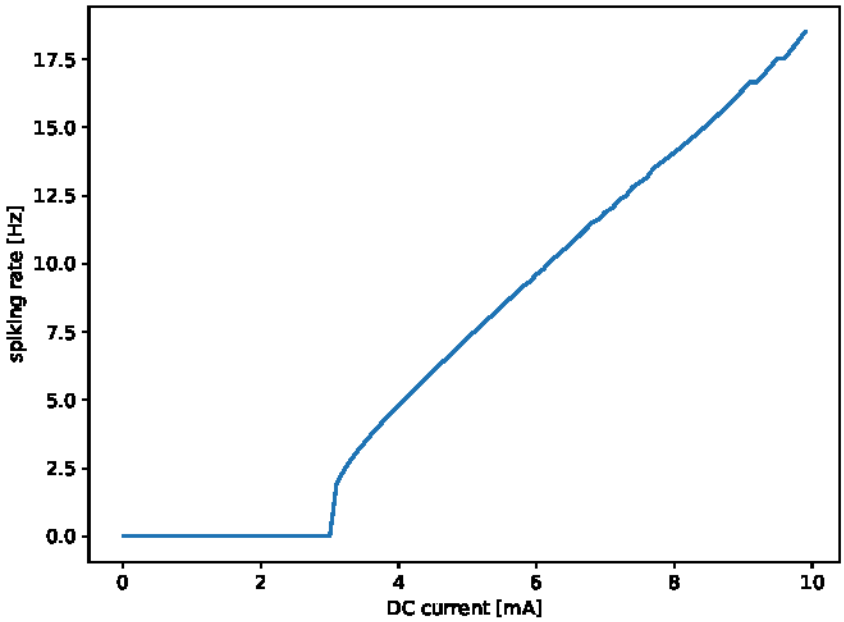


Figure 2.10: Spike frequency tuning curve

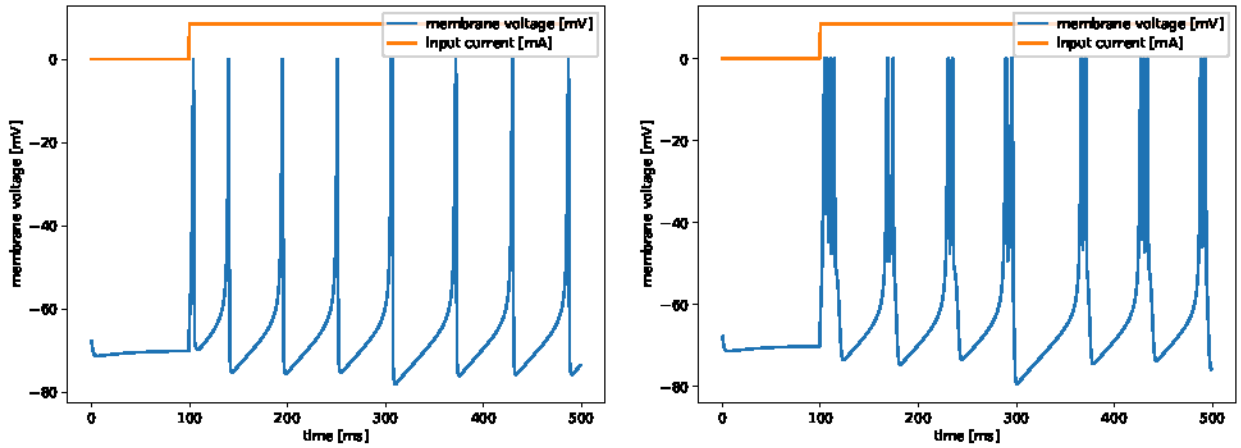
The Figure 2.10 shows the spike frequency tuning curve of the neuron with a specific set of membrane parameters R and C. In this graph, each value of the input DC current has a corresponding spiking frequency. Note the similarity with the $ReLU$ activation function shown in Figure 2.4. The discontinuity observed at $I_{DC} = 3\ mA$ is caused by the leakage resistance of the neuron.

2.2.1.2 Izhikevich Model

The LIF model illustrates the basic requirements of the neuron model. The Izhikevich model aims to model the behaviour of a neuron with greater biological plausibility [16]:

$$\dot{v} = 0.04v^2 + 5v + 140 - u + I$$
$$\dot{u} = a(bv - u)$$
$$\text{If } v \geq 30mV, \text{ then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases}$$

where v is the membrane potential, u is the membrane recovery variable, and I is the sum of all current inputs of a given neuron. The parameters a, b, c and d are used to configure the model. The Izhikevich model is also known as the Simple Model of spiking neurons. In part, this is due to the wide variety of biological neuron models that can be simulated with this system of equations, such as the regular spiking neuron, the chattering neuron, the resonator neuron, etc. [16].



(a) Regular spiking (b) Chattering

Figure 2.11: Izhikevich neuron model [16]

The plots in Figure 2.11 show the membrane voltages with greater biological plausibility, as well as the variation in behaviour due to changes in the parameters a, b, c, d. The plot on the left shows a regular spiking neuron biased with a constant current input. The plot on the right shows a chattering neuron that outputs bursts of spikes.

2.2.2 Synaptic Modeling

In a spiking neural network, the neurons are interconnected with synapses. The pre-synaptic neuron sends a spike through a synapse, and the post-synaptic neuron accumulates the current output from the latter.

When a spike is transmitted, the synaptic current spikes initially, and then decays to zero [19]. Hence, it can be modelled as a 1st-order low-pass filter with adjustable gain G_m and time-constant τ [19]. The impulse (or spike) response is shown below:

$$i_{SYN}(t) = G_m e^{-(t-t_0)/\tau} u(t - t_0) \tag{2.6}$$

where $u(t)$ is the unit-step function. Figure 2.12 shows a synaptic current waveform as a response to a set of voltage spikes.

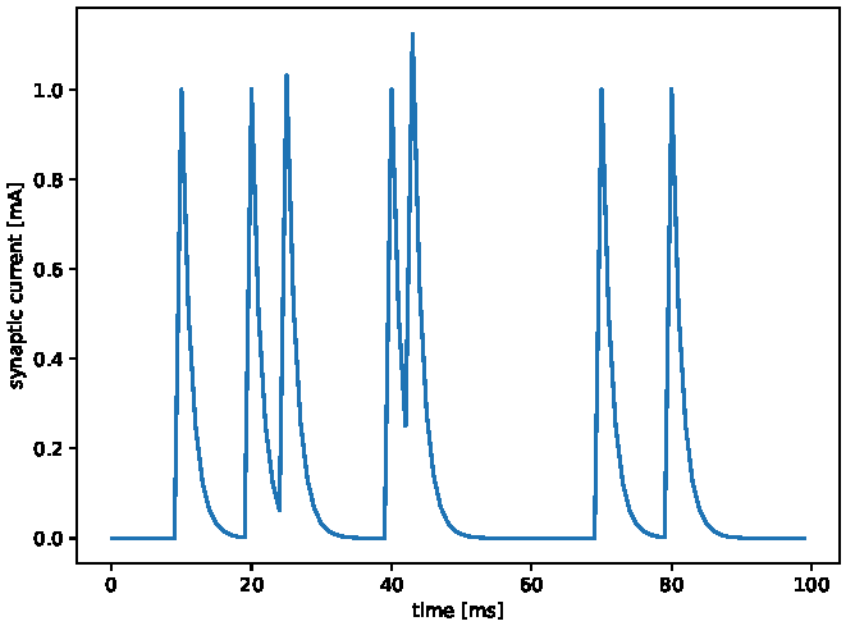


Figure 2.12: Synaptic current

2.2.3 Network Simulation

The structure of the spiking neural network simulator is represented in Figure 2.13. The column labeled as `NEURONS` represents the state variables of each neuron of the SNN. The synapse matrix stores the value of each synaptic strength, as well as the state of each synapse (i.e. the current flowing through). In this matrix, the synapse labeled as S_{ij} is excited by the neuron n_i and the supplies current to neuron n_j. With this topology, the synapses on the main diagonal S_{00}, $S_{11}, \ldots$ have strengths of zero, unless recurrent connections are desired. At every simulation step, the currents of each column are summed up to form a row vector, and its transpose is added to the column of source vectors. The latter are used as a substitute for the biases of the ANN layers.

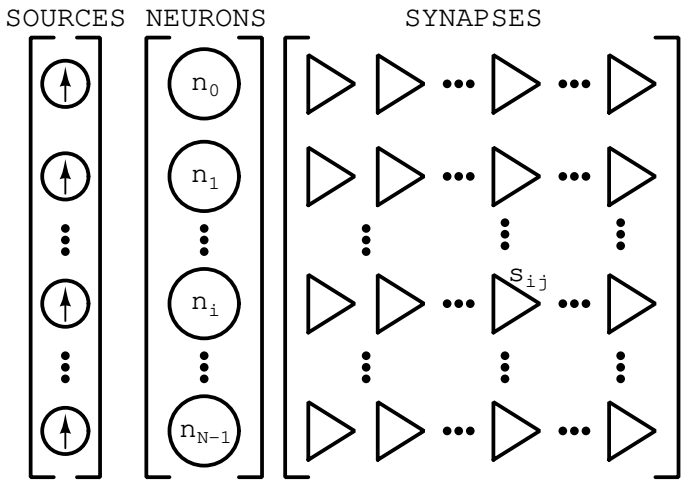


Figure 2.13: Structure of SNN simulator

The implemented simulator is configured by specifying the number of neurons N, the time-step of the simulation t_s, the simulated time t_f, and the type of neural model (LIF or Izhikevich). The neurons in the column are configured with the corresponding model parameters, as well as their biasing currents. Similarly, the matrix of synapses is configured with a global time-constant for decay, and a synaptic strength for each synapse. The synapse is said to be excitatory if it emits a positive current, and is called inhibitory if it emits a negative current [20].

The pseudo-code below summarizes the operation of the simulator. The complete program

can be found in Appendix A.

```
configure simulator parameters; initialize neurons and synapses;
for each time step do
    collect indices of spiked neurons; reset those neurons;
    update neurons' states based on model;
    apply spikes to synapses; update synapses' states based on exponential decay;
    update raster plot;
end
```

Algorithm 1: High-level simulator operation

The SNN shown in Figure 2.14 is composed of four neurons. The neurons n_0 and n_1 are supplied with constant current sources (or biases). The current source of n_1 is smaller in order to make n_1 spike at a lower rate. As indicated by the synaptic current labels, an excitatory synapse is used to connect n_0 to n_2 and n_3, and an inhibitory synapse for the connection of n_1 and n_3. The two excitatory synapses are equal, and the inhibitory synapse is slightly weaker in magnitude than the excitatory ones.

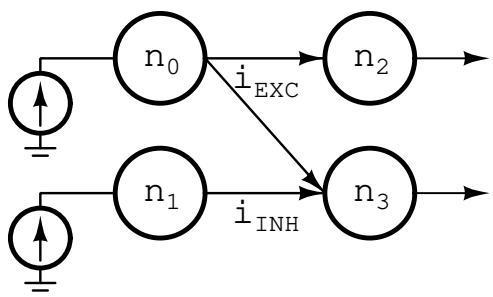


Figure 2.14: Simple SNN

The operation of this network is plotted in Figure 2.15. The excitatory synapse drives the neuron n_2 with sufficient strength to make it spike at the same rate as neuron n_0. In comparison, neuron n_3 is inhibited by the activity of neuron n_1, and has a smaller spiking frequency. Below, the first two graphs show the excitatory and inhibitory currents. And the two graphs at the bottom show the membrane voltages of the neurons n_2 and n_3, respectively. (Note that both excitatory currents are equal).

The spiking activity of large SNNs can be displayed with spike raster plots, such as in Figure 2.16. The simulated network is composed of a thousand neurons, and is fully-connected with random synapses. Each dot on this graph indicates a spike: the values of the x and y coordinates correspond to the time at which a spike occurred and the index of the source neuron, respectively.

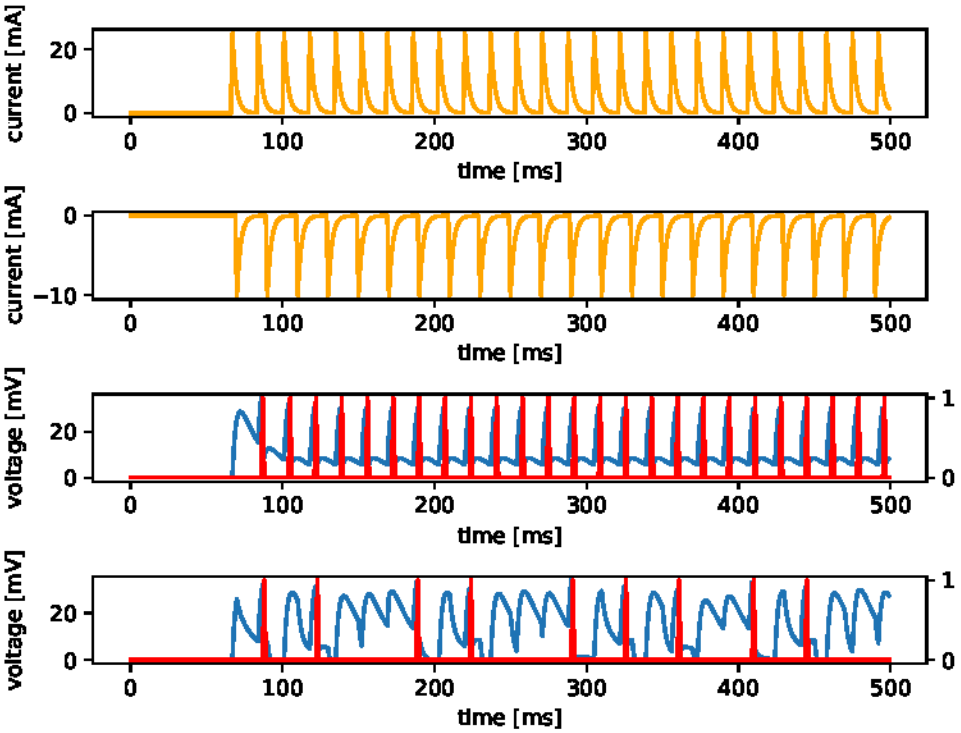


Figure 2.15: Simple SNN operation

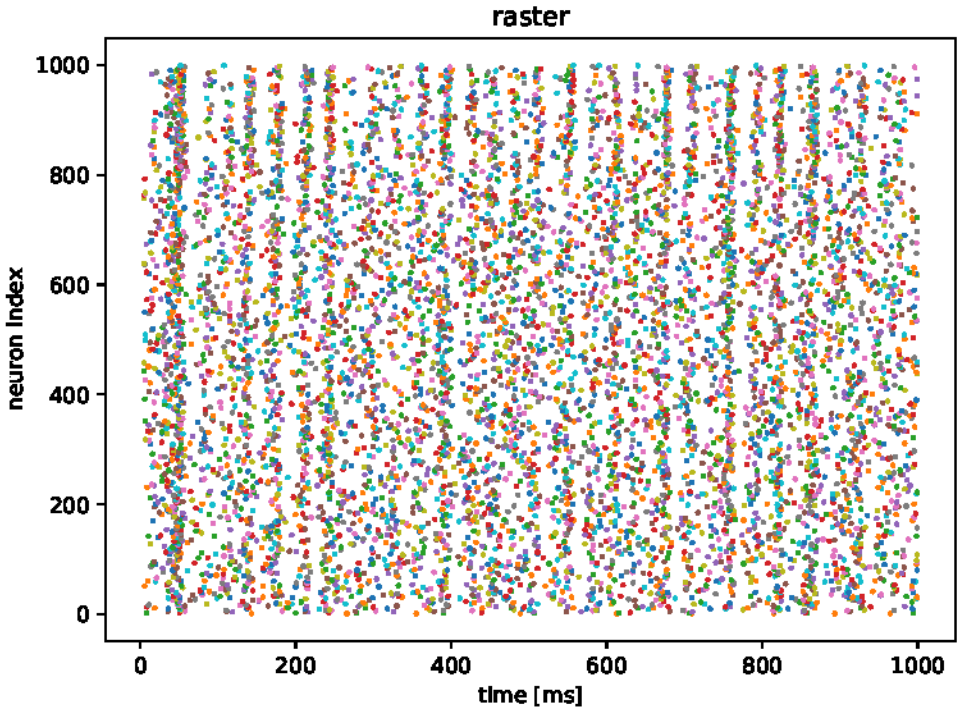


Figure 2.16: Spike raster plot

2.2.4 Spike-Timing Dependent Plasticity

Spike-Timing Dependent Plasticity (STDP) is a type of learning behavior that is observed in biology [21], and is often used as a training algorithm for SNNs [22]. This is a form of unsupervised learning, and the basic premise is to strengthen the connection between two neurons with a positive spike-train correlation, or to weaken it when the spiking correlation is negative [22]. If a post-synaptic neuron outputs a spike shortly after receiving a spike from a pre-synaptic neuron, their connection is increased. And, if a post-synaptic neuron spikes before an incoming pre-synaptic spike, their connection is decreased. When the time delay between pre- and post-synaptic spikes is sufficiently large, no changes are made to the synapse. A possible interpretation is that, if a neuron receives inputs from multiple pre-synaptic neurons, STDP prioritizes the connection with the neuron that is the last to spike prior to the post-synaptic event. Also, in contrast with the episodic training of ANNs with backpropagation, STDP-based learning happens continuously.

Figure 2.17 shows one of the possible ways for implementing the STDP rule, where the vertical axis gives the increment or decrement in synaptic strength for a specified time delay between pre- and post- synaptic events.

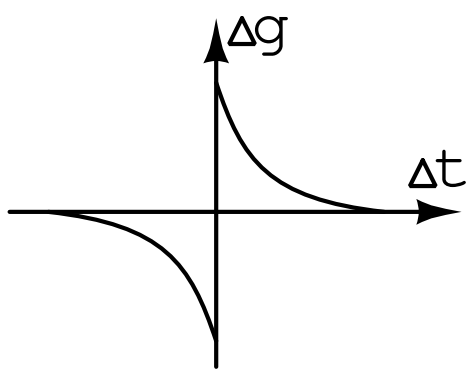


Figure 2.17: STDP rule

2.2.5 SNN Behaviors

This subsection provides examples of behaviors that can only occur with SNNs due to their dependence on time.

2.2.5.1 Nonlinear Amplification of Spike Rates

In Artificial Neural Networks, the activation value of a pre-synaptic neuron is multiplied by a weight. As such, each synapse of an ANNs can be thought of as a linear amplifier. And, if a neuron only has one input, scaling the activation rate of a pre-synaptic neuron by a factor α causes the weighted input of the post-synaptic neuron to also change by the same factor. Therefore, when the weighted input is positive, and a ReLU activation function is used (Figure 2.4), the activation of a post-synaptic neuron can be increased in a linear fashion by the weight.

In Spiking Neural Networks, the average spike rate cannot have this kind of effect. Figure 2.18 shows neuron n_0 with a constant spike rate driving neuron n_1 through a low-pass filtering synapse with a positive transconductance g_{SYN}.

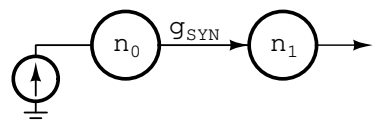


Figure 2.18: Neuron driven by a filtered impulse train

For the sake of simplicity, assume that the Integrate-and-Fire (IF) neuron model is used without a leaky parameter. Every spike from n_0 delivers an amount of charge q_0 into the membrane capacitance of n_1. Thus, if the transconductance of the synapse is strong enough to trigger a spike with one impulse ($q_0 = Q_{TH} = C_m V_{TH}$), then the spike rate r_1 of neuron n_1 will be equal to the rate r_0 of neuron n_0. If however, $0.5Q_{TH} \leq q_0 < Q_{TH}$, then two spikes from n_0 are required to trigger n_1. This behavior is summarized in Figure 2.19. Contrast this with Figure 2.10.

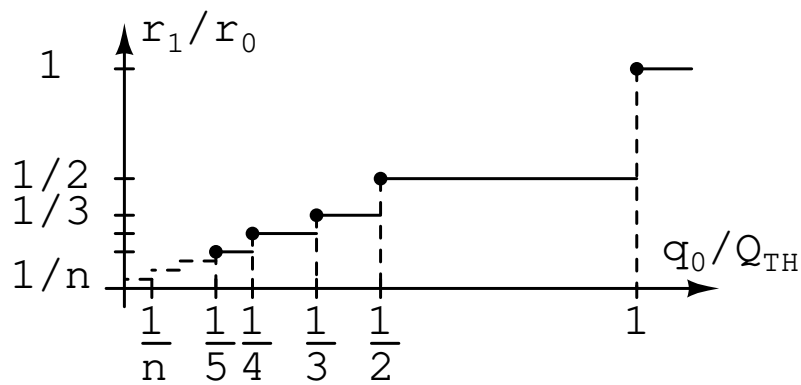
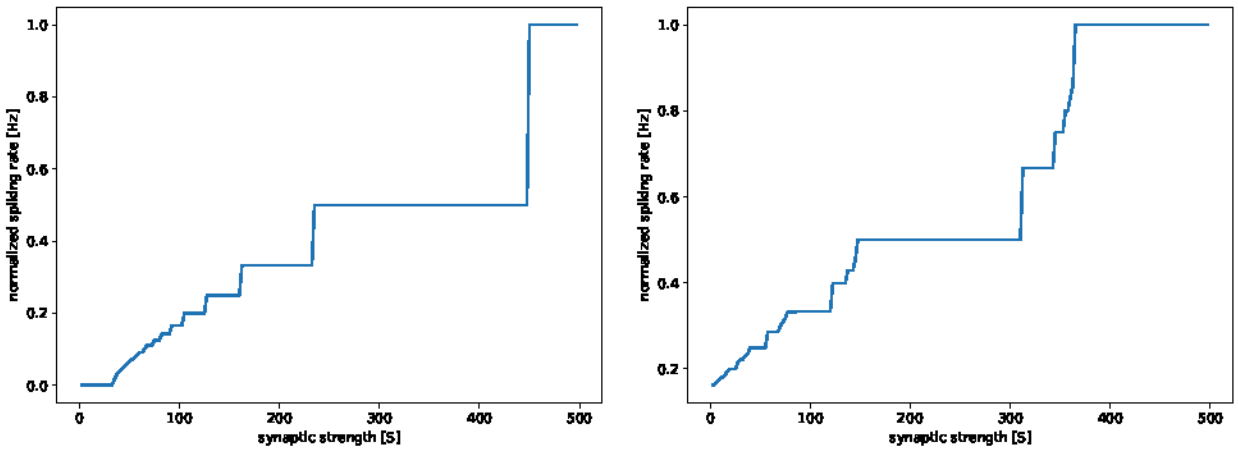


Figure 2.19: Spike rate as a function of synaptic strength

In short, due to the encoding of neural information in the timing of events, the spike frequency of a pre-synaptic neuron cannot be linearly increased in the post-synaptic neuron. In addition, it is even more difficult to estimate the spike rate of a neuron with multiple synaptic inputs. It is worth mentioning that this curve is somewhat linear for very small synaptic values. However, a driven neuron would require many spikes prior to outputting an event, which requires significantly longer simulation times, and the delays may make them unsuitable for control applications.

Lastly, as demonstrated in Figures 2.20a and 2.20b, adding a positive biasing current to the driven neuron n_1 does shift the parametric curve toward the left, albeit with some distortions. As a result, it is relatively difficult to estimate the spike frequency of a neuron with synaptic pulses and a constant current.



(a) Neuron n_1 without biasing current

(b) Neuron n_1 with biasing current

Figure 2.20: Spiking rate of neuron n_1 as a function of synaptic strength g_{SYN}

2.2.5.2 Spike Bursts

Typically, the spike rate of a post-synaptic neuron is at most the same as the rate of the driving neuron since the state of a receiving neuron is reset after every event. In other words, the spike frequency cannot be amplified by a synapse. However, there are exceptions to this statement. If an SNN is chosen to have low-pass filtering synapses, large decaying time-constants may cause a receiving neuron to output a burst of spikes in response to a single current pulse. This occurs when the amount of charge transmitted to the neuron during every time-step of the simulation is sufficiently large to trigger an event. This behavior is illustrated in Figure 2.21. If this effect is undesirable, it may be avoided by adding a

saturation point for the total input current of a neuron, or by using neurons with refractory periods [23].

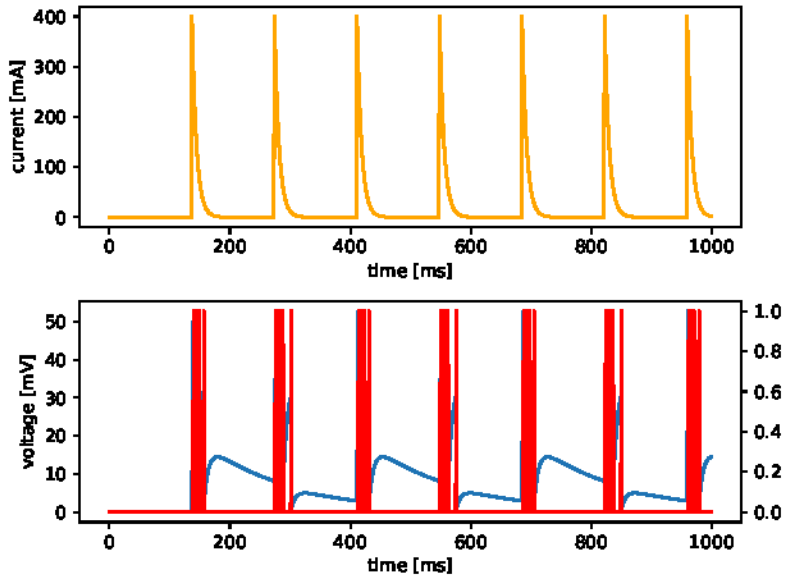


Figure 2.21: Spike bursts

2.2.5.3 Spike Synchronization

Spike synchronization is a phenomenon that makes SNNs drastically different from ANNs. Referring to Figure 2.22, the red and blue synapses represent excitatory and inhibitory connections, respectively. All connections are of equal magnitude, and the excitatory ones are sufficiently large to trigger a spike in the post-synaptic neurons. The current sources that power neurons n_0 and n_1 are turned on at time $t = t_0$, and the last current source is turned on at a time $t = t_1 > t_0$.

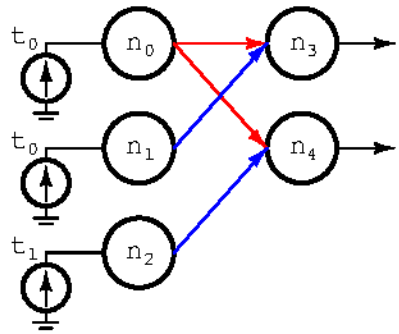


Figure 2.22: SNN with simulated synapse delay

On average, the total currents applied to the neurons n_3 and n_4 are both $I_{AVG} = 0\ mA$. But, because the neurons n_0 and n_1 are activated at the same time, the pulses that they transmit to n_3 are synchronous and cancel-out each other. As a result, the membrane voltage of this neuron remains grounded.

Given that the current source of n_2 is turned-on at a later time, there are delays in the spikes transmitted to neuron n_4. The positive pulse triggers a spike before the negative pulse can inhibit it. Hence, synchronization of spikes or lack thereof can cause a large difference in the spiking frequency of a receiving neuron.

This effect is caused by the fact that the membrane voltage of the LIF neuron is clipped at $V_m = 0V$. With alternative models, this may not be seen. Also, the importance of spike synchronization can be lessened if the magnitudes of the synapses are capped to prevent single spike triggering.

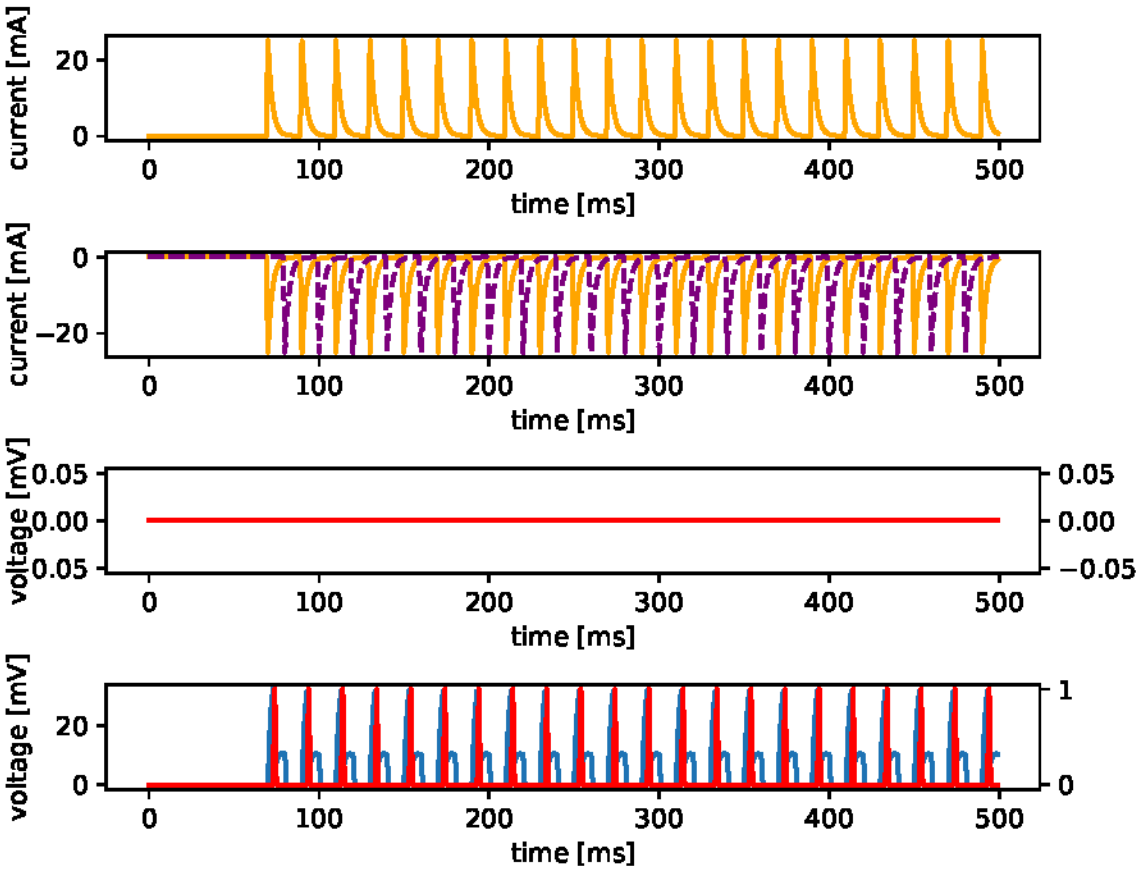


Figure 2.23: Effect of de-synchronized spikes

2.3 Literature Review

The objective of this section is to give a brief overview of some existing SNN processors.

2.3.1 DYNAPs

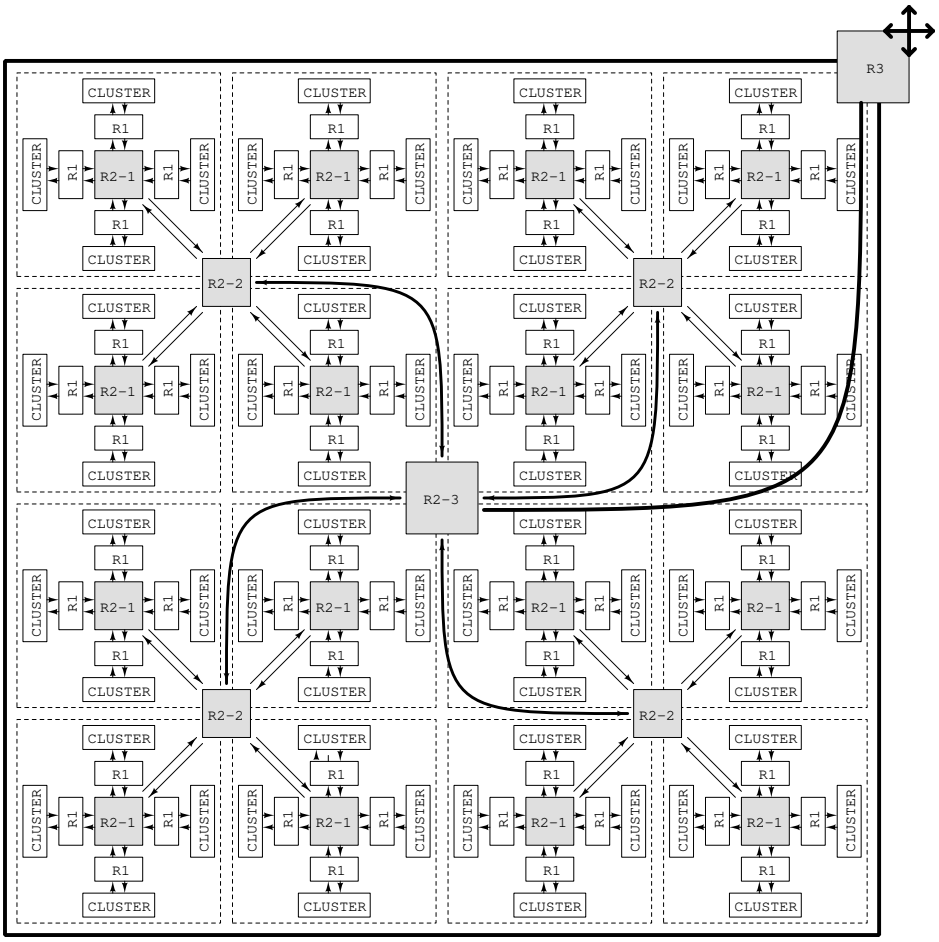


Figure 2.24: DYNAPs architecture [24]

Shown in Figure 2.24 is the architecture of the Dynamic Neuromorphic Asynchronous Processors (DYNAPs) IC that is composed of 64 neuron clusters divided over 4 cores [24]. All clusters are interconnected through a routing tree with bi-directional connections among the nodes of upper and lower levels. Each cluster interfaces the tree through R1 routers. The next router R2 is made up of three levels, with each level connecting a set of four nodes (e.g. router R2-3 connects to four routers R2-2, and so on). Finally, at the root of the tree, the router R3 is used to interconnect multiple ICs.

The DYNAPs comprise 1k mixed-signal neurons and 64k synapses. And, as indicated by the name, all circuits are asynchronous. In [25], DYNAPs is used to implement an ultra-low power ($< 722\ \mu W$) ECG anomaly monitoring system, which turns-on a micro-processor when one is detected. Although the quantization of the synapses is not described, a recurrent SNN architecture is chosen for the reservoir-computing paradigm. In other words, the prediction error of the SNN is minimized by adjusting the gains of the low-pass filters that monitor the output neurons.

2.3.2 SpiNNaker

The SpiNNaker is a very-large and highly-parallel computing system for the simulation of SNNs [26]. The complete assembly is made-up of 1200 Printed Circuit Boards (PCBs) with each having 48 nodes. Every node consists of a custom IC and a separate SDRAM. As can be seen in Figure 2.25, each node comprises 18 cores, where each core is an ARM9 processor. Thus, there are over a million cores in the entire computer [26], which are all connected in a torus arrangement. This is done through the ports shown at the top of Figure 2.25 using Address Event Representation.

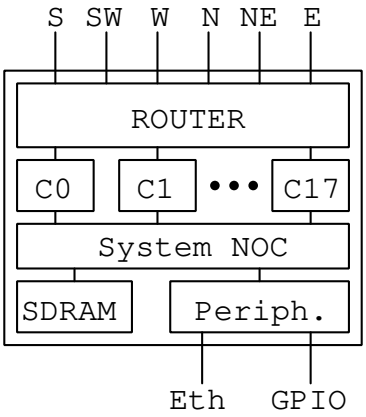


Figure 2.25: SpiNNaker node [27]

There are approximately 1000 neurons in a single core [26]. However, given that they are simulated on ARM9 processors, these neurons need to be multiplexed. With clock rates of $f_{CLK} \approx 200\ MHz$, the average spiking rate of the neurons is $f_{SP} \approx 10\ Hz$, which is similar to the spiking rates observed in the biological brain [26]. Moreover, each neuron has a fan-in/out of about 1000 synapses, but the resolution of these is not specified. Due to its sheer size, the power consumption of SpiNNaker is at most 90 kW.

2.3.3 TrueNorth

TrueNorth is an integrated circuit developed by IBM with 1 million digital neurons and 256 million synapses that are divided over 4096 cores [28]. In Figure 2.26, only 4 cores are shown. The input buffers shown on the left of every core receive inputs from the routers. A global clock of $f_{CLK} = 1\ kHz$ triggers the distribution of these inputs through the grid of synapses, and the neurons at the bottom of each core integrate the currents. If spikes are generated, they are sent to the remainder of the network [28].

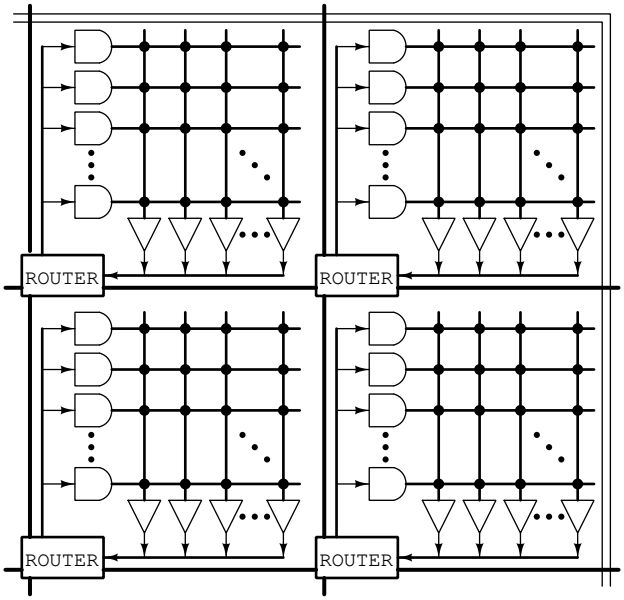


Figure 2.26: TrueNorth architecture (top-right corner) [28]

2.3.4 Loihi

Designed by Intel, Loihi is an IC composed of 128 neuromorphic cores, with 1024 digital neurons per core, and three x86 processors [29]. The entire system uses an asynchronous network to communicate, and could be scaled up to 16384 ICs with 4096 cores on each chip [29]. The synapses on Loihi may have resolutions of up to 9 bits, and can be updated using programmable online learning rules, such as STDP. A LASSO benchmark problem is used to compare the performance of Loihi with that of well-known solvers running on an Atom CPU. It is observed that Loihi scales far better for larger problems, with significantly smaller delay and energy consumption [29].

2.3.5 BrainScaleS

The BrainScaleS system is composed of 20 wafers, each comprising 384 analog neural network ICs [30]. Every IC has 512 analog neurons, which can connect to at most 220 synapses. The biologically plausible spiking rates of $\sim 40Hz$ are scaled-up by a factor of 10^4 due to the fast time constants of the SNN circuits [30]. A small network of 135 neurons is used to evaluate the classification accuracy of a subset of the MNIST dataset, where only 5 digits are categorized. An initially trained network is first converted to 4-bit synaptic weights. Finally, the spiking rates measured in hardware are used to update the synaptic weights in order to compensate for process variations. An initial ANN classifier with a 97% accuracy is mapped to an SNN whose accuracy falls to 72% due to process variations. Through the iterative hardware training loop, this performance is brought back up to 95% [30].

Chapter 3

Proposed Architecture

The objective of this chapter is to present the general architecture for the hardware implementation of spiking neural networks. Also shown is the scalability of these networks through the tiling of multiple cores in a toroidal arrangement, which is inspired by SpiNNaker [26].

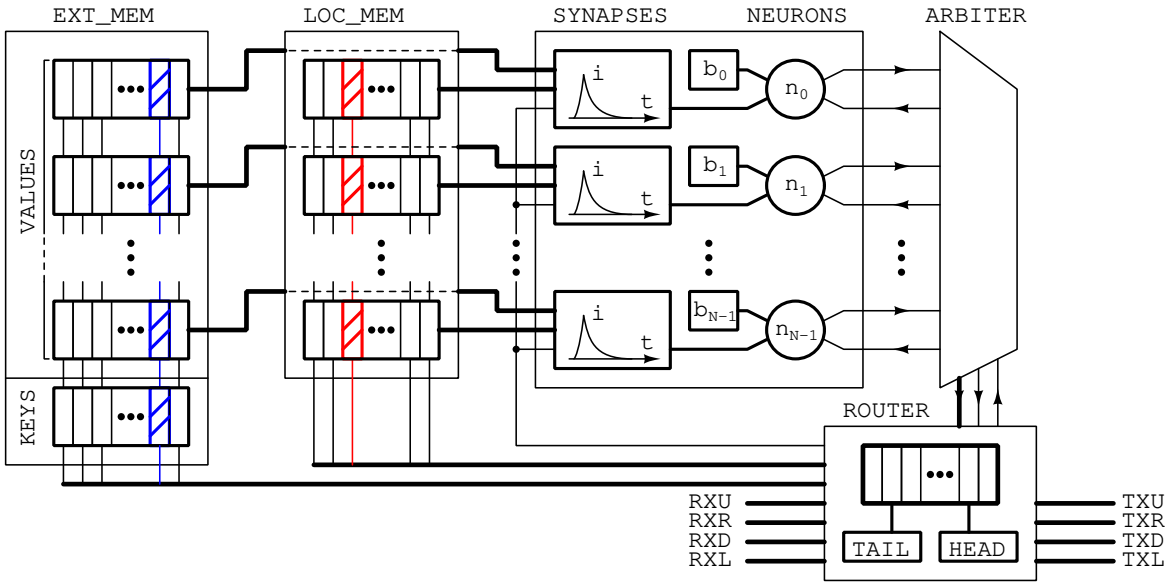


Figure 3.1: Architecture block-diagram

As a whole, the architecture proposed in this work serves as a platform for making design decisions. Among the numerous design variables, some of these decisions include the type of neural and synaptic models, the number of neurons per core, the resolution (number of bits) allocated for synaptic strengths and biasing neural currents, etc. Due to their nature, spiking neural networks can become very large and an all-to-all connectivity among neurons

is impractical. In fact, due to the finite memory available for network parameters and routing entries, the number of destinations that a neuron can have is also finite. Therefore, another very important design variable is the maximum number of synapses that a neuron can interface.

The entire system is developed in VHDL, and in order to allow for experimentation, every module is implemented with generic parameters. (In VHDL, generic parameters allow the user to easily reconfigure entities. For example, 4-bit and 8-bit memory registers can be instantiated from the same source code if a generic width parameter is used in its definition.) In other words, the developed framework allows the user to easily reconfigure the values of the design variables mentioned above depending on the requirements of a given test bench. As an example, if a small SNN is being tested, an IC core with only a few neurons can be instantiated. This is useful for keeping the required computing resources at a minimum. Moreover, this greatly simplifies the exploration of the design space at both high and low levels of implementation.

One of the most important set of phenomena that can be studied with the use of this platform are the parametric variations across all the components used in this core, with an emphasis on the analog circuits. If analog neurons and synapses are used, it is expected for parametric variation to have a negative impact on the performance of trained SNNs. In addition, due to the need to convert synaptic values from digital memory into analog currents, the Digital-to-Analog converters will also suffer a certain degradation in accuracy.

Another important issue that needs to be studied are the routing delays, which are a function of the clock rates. It is desirable to optimize the routing scheme in order to keep the clock rate at a minimum and reduce overall power consumption.

3.1 Core Operation

In contrast with the multiplexed simulation of neurons on the SpiNNaker system, the design proposed in this work consists of N separate hardware neurons that operate in parallel [26]. As stated in the introduction, a fully-connected spiking network has two synaptic connections between every pair of neurons (for a bi-directional transmission of currents). Hence, the number of synapses scales with a square complexity $O(N^2)$ where N is the number of neurons. This implies that the area of a hardware network implementation would increase geometrically, and that most of the IC area is allocated for the storage of synaptic strengths. One possible solution for mitigating this problem is to assign only one synaptic

circuit per neuron.

As shown in Figure 3.1, every SNN core is composed of a column of N synapses and N neurons. The latter are directly connected to an arbitration module, which in turn, interfaces a router that is used for handling local as well as external events. A memory array is used to store a column of biasing currents labelled as b_0, b_1, $\cdots$, b_{N-1} that are seen to the left of each neuron. A local memory grid stores the values of the synapses that interconnect the local neurons. And, an external memory provides a similar function by storing the synaptic strengths whose source (pre-synaptic) neurons are located on external cores.

Each SNN core operates in the following fashion. When a neuron's accumulated charge triggers a spike, it sends a positive-edge to one of the arbiter's inputs. The arbiter's main function is to encode the index of the firing neuron into a source address. This address is then captured by the router and stored inside an event buffer, which behaves as a queue. Next, the router retrieves the event address from the other end of this queue, and uses it to access a column of synaptic values from the local memory grid, as highlighted in red in Figure 3.1. After waiting for one clock cycle that allows the retrieval of the memory contents, the router sends a positive-edge that causes current pulses to be injected into each synaptic circuit. The amplitudes of these pulses correspond to synaptic strengths. The neurons continue accumulating the currents, and the procedure repeats. The type of communication where an event is encoded by the address of a spiking neuron is also known as *Address Event Representation* [8].

As can be seen, the synapses can only be injected with one pre-synaptic spike at a time. In the case of simultaneous neuron events, each spike is delivered one at a time. And, the greater the speed of the clock driving each core, the more the delivery of synaptic pulses appears to be simultaneous. However, it is important to mention that the total current of a given synapses can consist of overlapping pulses, and one pulse does not need to decay before another one can be applied.

The operation of this system differs greatly from the operation of the TrueNorth computer, whose every core has a square interconnection grid and multiple events are delivered together [28]. Furthermore, the clock of the TrueNorth is used to synchronize "frames" of asynchronous operation, whereas the proposed design requires a fully synchronous approach.

Although the number of synaptic circuits has been reduced from N^2 to N, the issue of $O(N^2)$ scaling complexity cannot be fully solved due to the fact that there's still a need for the storage of all synaptic strengths in memory. But, the overall reduction in area can still be significant considering that a synaptic circuit is much larger than a register of an SRAM.

Given that the scalability of SNNs is a critical requirement, the proposed architecture is designed such that multiple cores can be tiled together. The proposed router design is capable of communicating with external cores through four receiving (RX) ports and four transmitting (TX) ports. Therefore, each core can have four neighbors.

When an event is received from an external core, one of four RX routing tables is used to determine whether the event is to be delivered to the local neurons, to be transmitted to other neighboring cores, or both. If the set of destinations does include the local neurons, the address of the event is used as a key for content-addressable memory (CAM), which provides a set of synaptic values (highlighted in blue in Figure 3.1) that are then applied to the column of synaptic circuits. In the case of SpiNNaker, two additional ports are used for diagonal North-East and South-West directions [26].

3.2 Scaling

Suppose that a given SNN is made up of $M \times N$ neurons, where M is the number of cores and N is the number of neurons per core. (For the sake of simplicity, assume that $n = \log_2 N$ is an integer, and $m = \log_2 M$ can be expressed as a product of two integers.) Thus, the number of entries that every router must be able to receive is $(M - 1) \times N$. The simplest solution is to tile all of the M cores in a matrix arrangement. The cores located in the middle of the grid would have RX routing tables with an equal number of entries. However, the tiles on the right side of this matrix would have a large table on its left RX port and an empty table on its right port. The same applies for every other core. In order to maintain an even distribution of routing entries, a toroidal arrangement is used to interconnect every circuit. Figure 3.2 presents the overall topology of the system, and Figure 3.3 demonstrates that a torus can be build by connecting the ports at the top with those at the bottom, and the ports on the left with those on the right.

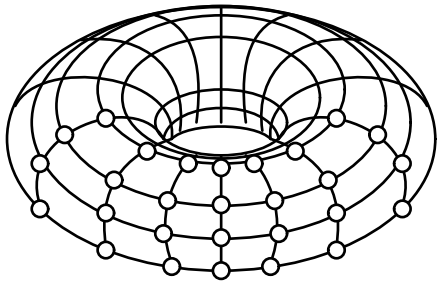


Figure 3.2: Torus topology [26]

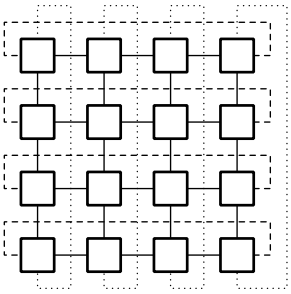


Figure 3.3: Torus core arrangement

3.3 Local Router Bridge

An improvement to the proposed architecture is presented in Figure 3.4. It consists of adding a module that allows for the address events of local neurons to be redistributed without the use of the router. This bridge disconnects the arbiter module from the local router, and instead re-routes it directly to the local memory grid that stores the local synaptic values. A finite-state machine (FSM) is used to generate the spike acknowledge signals for the neurons, as well as the pulses that trigger the synapses. No spikes are dropped in the case of collisions. However, due to the faster handling of these spikes, the number of collisions decreases.

The main disadvantage of this method is the inability to broadcast local address events to external cores. Thus, the neurons that comprise the output layer are to be monitored through external pins of the integrated circuit. For this reason, the router bridge can only be enabled on SNN cores whose local neurons do not have any external destinations. Therefore, if a network spans multiple cores, the local router can only be bypassed on the core that contains the output layer, and the remaining cores need to have a fully operational router. Events arriving from external cores are still handled by the RX routers.

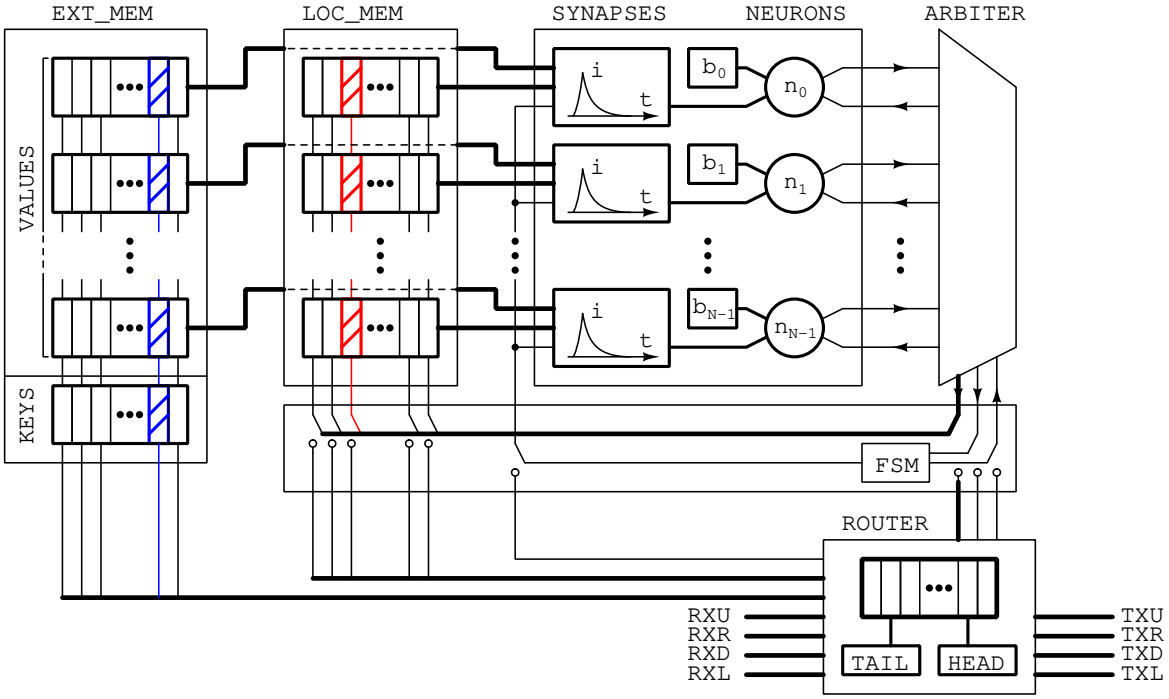


Figure 3.4: Architecture with local router bridge

3.4 Overview and Memory Programming

Figure 3.5 presents the complete detailed architecture of the integrated circuit. The left portion of the diagram shows the combined memory interface consisting of the signals `IC_PROG_WR`, `IC_PROG_ADDR` and `IC_PROG_REG`. The total length of the address bus `IC_PROG_ADDR` is expressed in terms of generic parameters as

$$\texttt{ALEN\_ICS + 2ALEN\_NRNS + 3} \tag{3.1}$$

where `ALEN_ICS` is the address length for the cores. As an example, if `ALEN_ICS = 4`, the memory for external synapses and the routing tables are scaled for a full torus of 16 cores. Similarly, the number of neurons in each core is specified by the address length `ALEN_NRNS`.

In Table 3.1, the column labelled as `XYZ` is a concatenation of specific address bits from the bus `IC_PROG_ADDR`. The bits `X` and `Y` correspond to the 2 most significant bits of the address. The bit `Z` is taken at index `2ALEN_NRNS`.

Table 3.1: Address ranges for module memories

Branches	**XYZ**	**Address Range**
1	000	ALEN_NRNS - 1 : 0
2	001	2ALEN_NRNS - 1 : ALEN_NRNS
3	010	ALEN_ICS + ALEN_NRNS : 0
4	011	ALEN_ICS + 2ALEN_NRNS : ALEN_ICS + ALEN_NRNS + 1
-	1XX	ALEN_ICS + ALEN_NRNS : 0

In order to program the values of the constant current sources that bias the neurons, the first branch is accessed with `XYZ = 000`, and the programmer iterates through the address range of all neurons. When writing the values of the local synapses, the next branch is accessed with `XYZ = 001`, and due to having N^2 synapses, the right-most `2ALEN_NRNS` bits of the address bus are used to write the synaptic values. The least significant `ALEN_NRNS` bits are used to iterate through the X coordinates of the memory grid, and the most significant `ALEN_NRNS` bits are used for the Y coordinates. The procedures are similar for the content-addressable memories that store the external synaptic values, as well as the tables found in the router.

Prior to the manufacturing of the IC, the memory modules are to be synthesized with SRAM technology. Given that SRAM is volatile, the IC can be used for various applications.

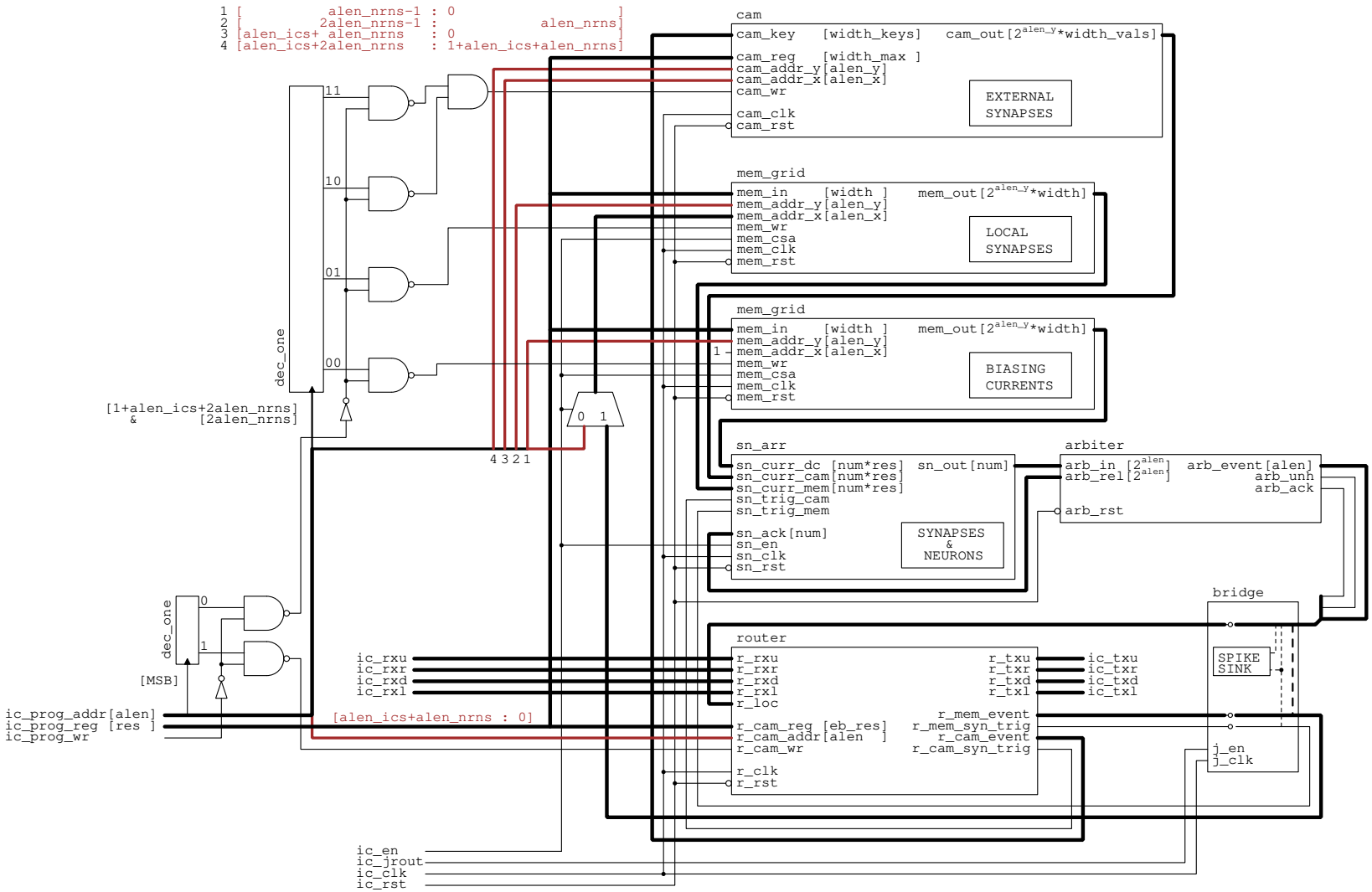


Figure 3.5: Detailed architecture diagram

Chapter 4

Design of an SNN Core

4.1 Memory

As described in the previous section, the architecture of the integrated circuit contains several memory modules. They are used to store the values of the biasing currents for the neurons, the synaptic strengths that interconnect local neurons, the synaptic strengths corresponding to external pre-synaptic neurons, as well as entries for routing tables. This section elaborates the implementation of the memory circuits.

4.1.1 Random-Access Memory

4.1.1.1 *D* Flip-Flop

The simplest memory circuit employed across the IC sub-systems is the positive-edge triggered *D* flip-flop with an asynchronous reset. It is implemented in VHDL using a behavioural architecture, with the *PROCESS* statement shown in the left column below. The declaration of the *ENTITY* has been omitted for the sake of clarity.

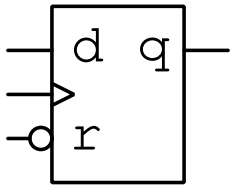


Figure 4.1: *D* flip-flop interface

```
18 ...
19 architecture dff_arch of dff is
20 begin
21   process(dff_rst, dff_clk)
22   begin
23     if dff_rst = '0'
24     then
25       dff_q <= '0';
26     elsif rising_edge(dff_clk)
27     then
28       dff_q <= dff_d;
29     end if;
30   end process;
31 end dff_arch;
```

4.1.1.2 Register

The memory register is implemented similarly to the D flip-flop with the exception of storing a vector of bits that is specified by the generic parameter `WIDTH`. Moreover, the register has control signals `MEM_W` and `MEM_R` for enabling the write and read operations, respectively.

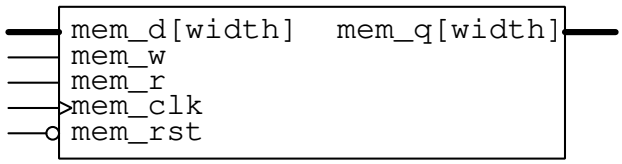


Figure 4.2: Register interface

4.1.1.3 Stack

The next level of complexity corresponds to a memory stack, or an array of registers, with random access. This stack is designed with a dual port that allows the write and read operations on different addresses simultaneously. Although the memory stack corresponds to the next level of complexity among memory circuits, it is not implemented by combining the previously defined memory registers. Instead, the VHDL keyword `ARRAY` is used. First, this is done in order to have optimal simulation run-times. Second, this approach simplifies the physical layout of the integrated circuit by providing the option to use SRAM modules that have higher density than D flip-flops.

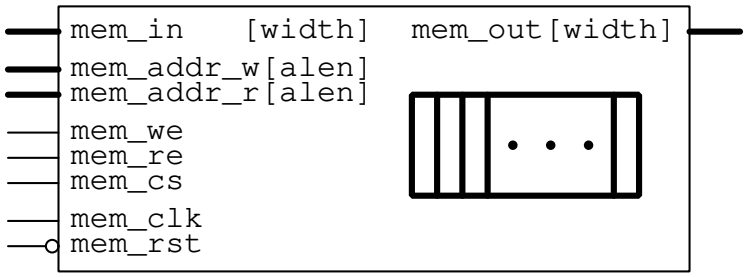


Figure 4.3: Stack interface

The simulation waveforms in Figure 4.4 demonstrate the usage of this memory array. After raising the reset `MEM_RST` and chip-select signals `MEM_CS` at a logical '1', the memory array is first enabled for a write operation. A list of values ranging from `0x8` to `0xF` is written into it by incrementing the write address `MEM_ADDR_W`. Finally, with the writing disabled and reading enabled, the contents of this array are iterated with the read array `MEM_ADDR_R`, and are displayed in the output of the memory stack vector `MEM_OUT`.

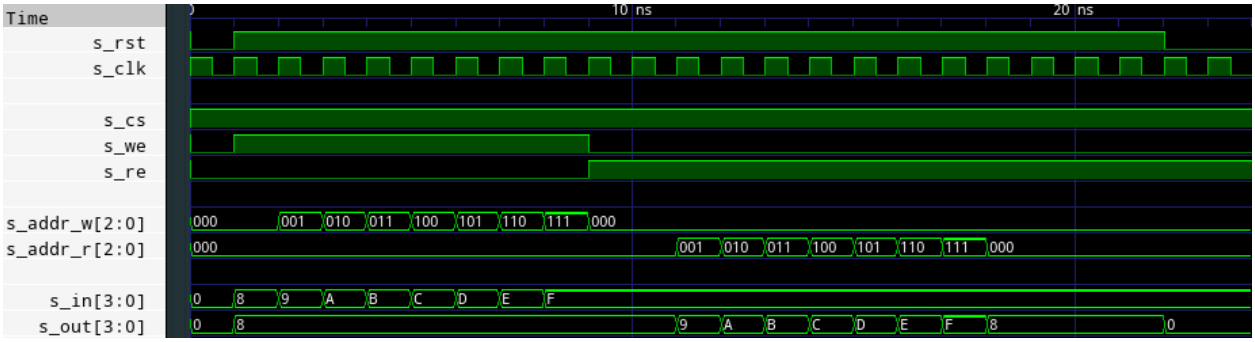


Figure 4.4: Stack operation

4.1.1.4 Grid

The memory grid is essentially an array of memory stacks, and as the name indicates, stores words of a specified width in a matrix arrangement. This memory circuit has two addresses: `MEM_ADDR_Y` for accessing a specific stack, and `MEM_ADDR_X` for accessing a specific word inside that stack. The lengths of each address are specified by the generic parameters `ALEN_Y` and `ALEN_X`. These indicate the number of stacks used in the grid, as well as the number of entries in each stack. The grid is programmed when `MEM_WR = 0`, and read when `MEM_WR = 1`.

The input to the grid consists of a single bus `MEM_IN` with a fixed parameter `WIDTH`. The output bus comprises multiple busses concatenated together, and has the length of

(2^{ALEN_Y}) WIDTH. For example, if ALEN_Y = 2 and WIDTH = 8, then the number of stacks in the grid is 4, and the width of the output bus is $4 \times 8 = 32$. As a consequence, during the read operation, only the horizontal address bus MEM_ADDR_X is of importance, and the vertical address MEM_ADDR_Y can be ignored.

The concatenation is only done for the read operation. In order to use a common programming bus IC_PROG_REG for all memories, the write operation is performed on the basis of individual registers.

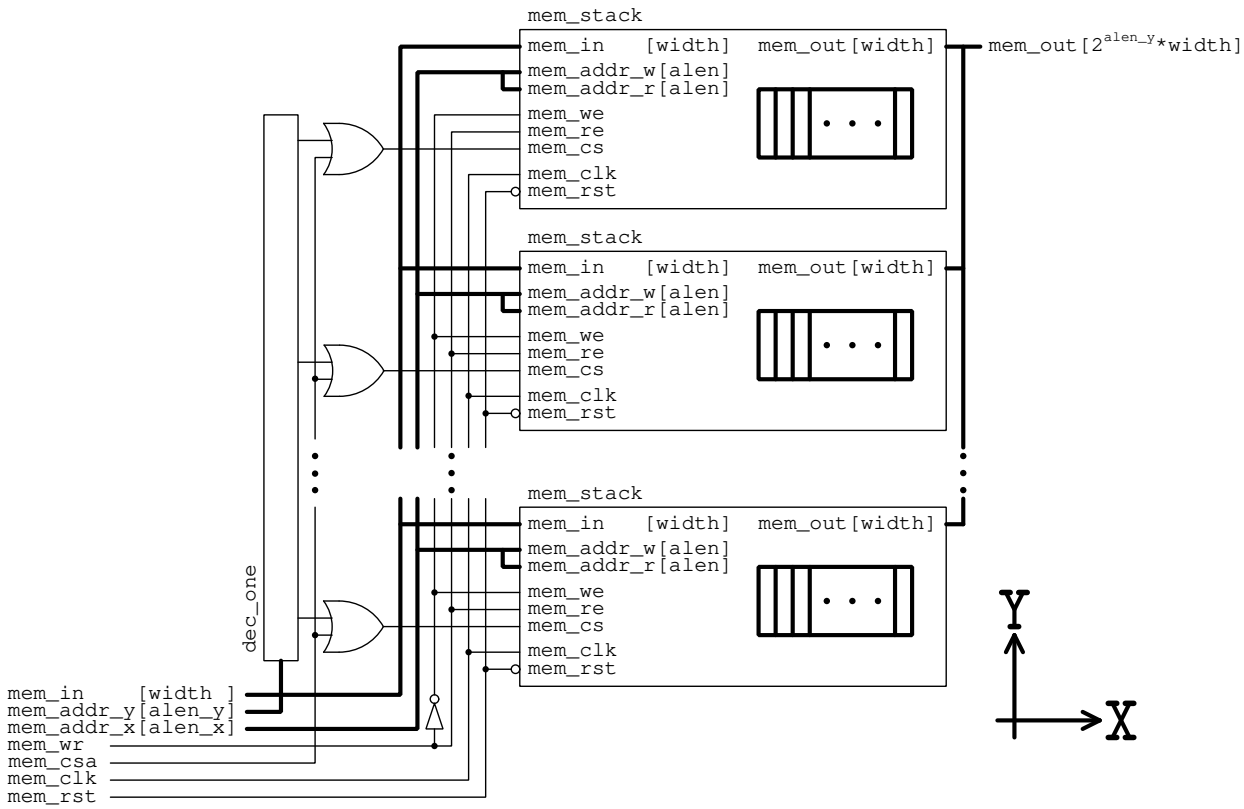


Figure 4.5: Memory grid schematic

The Figures 4.6 and 4.7 below demonstrate the usage of this memory circuit. In the top waveforms, the signals chip-select-all and write-read (S_CSA and S_WR) are set to logical '0' in order to write values. As indicated by the address signals S_ADDR_Y and S_ADDR_X, the programmer uses two nested *FOR*-loops to iterate through every entry of the memory grid.

The bottom waveforms demonstrate the accessing of the memory contents. After setting the signals MEM_CSA = 1 and MEM_WR = 1, the test-bench iterates through all the rows of all stacks simultaneously. As mentioned earlier, the outputs of each stack are concatenated into one large bus.

```
...
  for k in (2**C_ALEN_Y - 1) downto 0 loop    -- iterate memory stacks
    for l in (2**C_ALEN_X - 1) downto 0 loop  -- iterate individual rows
      s_in  <= v_strs(k * 2**C_ALEN_X + l);
      wait for 1 ns;

      s_addr_x <= std_logic_vector(unsigned(s_addr_x) + 1);
    end loop;

    s_addr_y <= std_logic_vector(unsigned(s_addr_y) + 1);
  end loop;
...
```

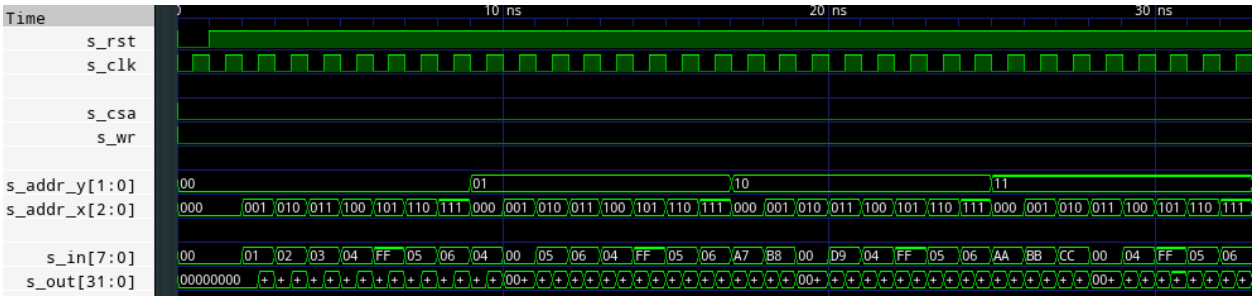


Figure 4.6: Memory grid operation (programming)

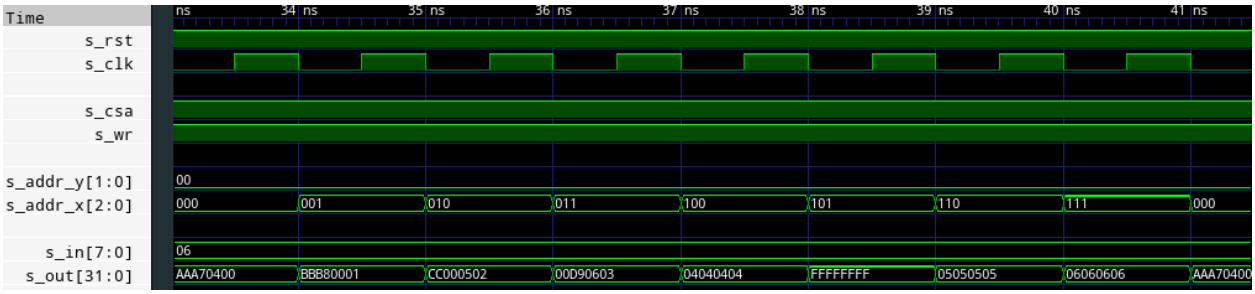


Figure 4.7: Memory grid operation (accessing)

4.1.2 Content-Addressable Memory

Content-Addressable Memory (CAM) is a hardware implementation of the dictionary data-structure [31]. Its function is to accept a keyword as an input and output an associated value. Such circuits are mainly used in networking routers [31]. Given that the delivery of spiking events is essentially a packet delivery problem, routing tables are implemented with CAMs. As mentioned earlier, the memory stacks are to be mapped to SRAM circuits.

Similarly, the CAMs are to employ custom cells and SRAM for keyword matching and data retrieval.

4.1.2.1 Match Cell

The smallest circuit in the implementation of CAM is a matching cell. Shown in Figure 4.8, it consists of a single memory register and a small combinatorial circuit. The latter consists of an array of $XNOR$ gates, the number of which equals the width of the register. Each memory bit is applied to a bit applied to the input bus `MC_KEY`. Hence, when both words have equal bits at the same index, the corresponding $XNOR$ gate outputs a logical '1'. A complete match requires for all bits to be equal; thus, all the $XNOR$ outputs are applied to an AND gate.

In Figure 4.9, the cell is first programmed with a specified key (`0xD5` in this test-case). Next, after enabling the read operation of the register, multiple values are applied to the key input `MC_KEY`. When the key is equal to the one that was initially programmed, the cell indicates a match by raising the pin `MC_MATCH`.

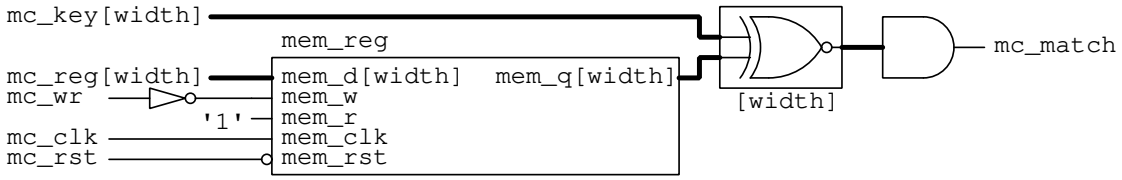


Figure 4.8: Match cell schematic

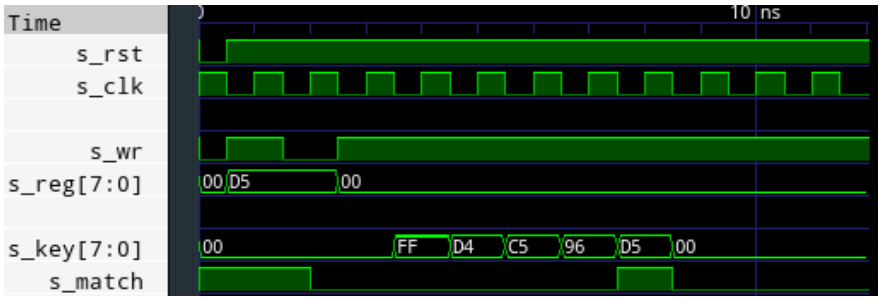


Figure 4.9: Match cell operation

4.1.2.2 Match Array

As the name indicates, the matching array circuit is simply an expansion of the matching cell, the number of which is specified by 2^{ALEN}, where `ALEN` is a generic parameter. The symbol labeled as `DEC_ONE` decodes an address into a one-hot representation, and is used for programming each memory register. The output vector is a concatenation of the match lines from every cell. Provided that every stored key is unique, the output vector `CAM_MATCH` can have at most one logical '1', the index of which corresponds to a matching address. As illustrated in Figure 4.10, the input key `CAM_KEY` is connected in parallel to the key inputs of every cell. In large CAM implementations, these parallel connections may be a source of very significant power consumption and/or delay.

If a given key is stored in multiple addresses of the same CAM, then a match would result in several logical '1's being output from the match array. Depending on how the value retrieval is performed, several values may be accessed at once causing corrupted readings.

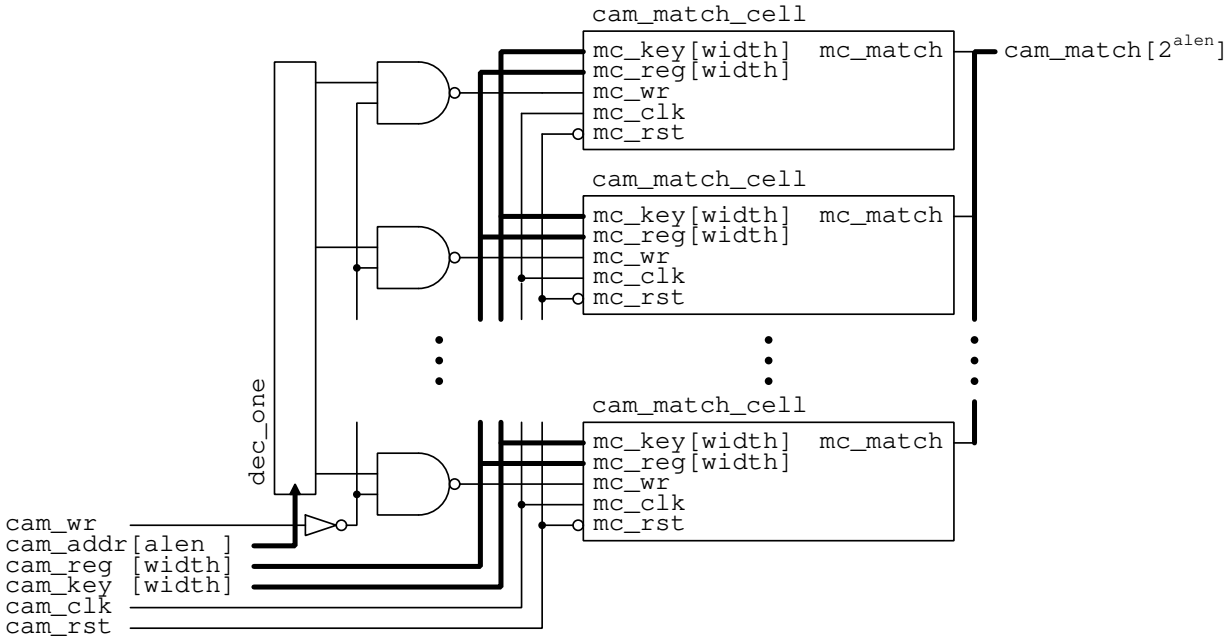


Figure 4.10: Match array schematic

Figure 4.11 shows the programming phase of the matching array, and Figure 4.12 demonstrates the matching of the stored keys.

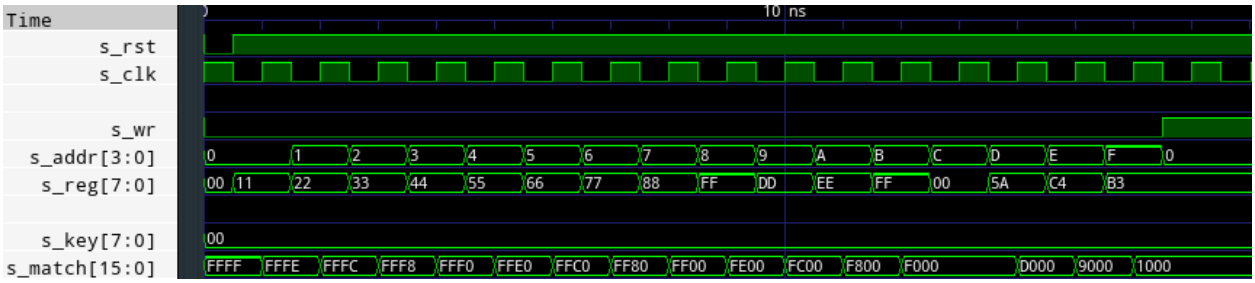


Figure 4.11: Match array operation (programming)

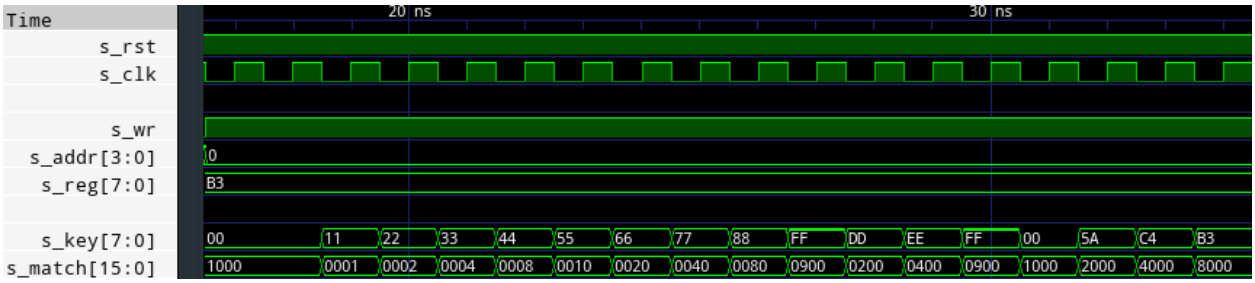


Figure 4.12: Match array operation (accessing)

4.1.2.3 CAM Stack

The CAM stack combines the matching array with the memory stack previously described. The output of the matching array is encoded from a one-hot representation into a binary address using the module `ENC_ONE` depicted in Figure 4.13; the latter is then used to access the memory circuit and the value associated to the input key is retrieved. The routing tables are implemented with this circuit.

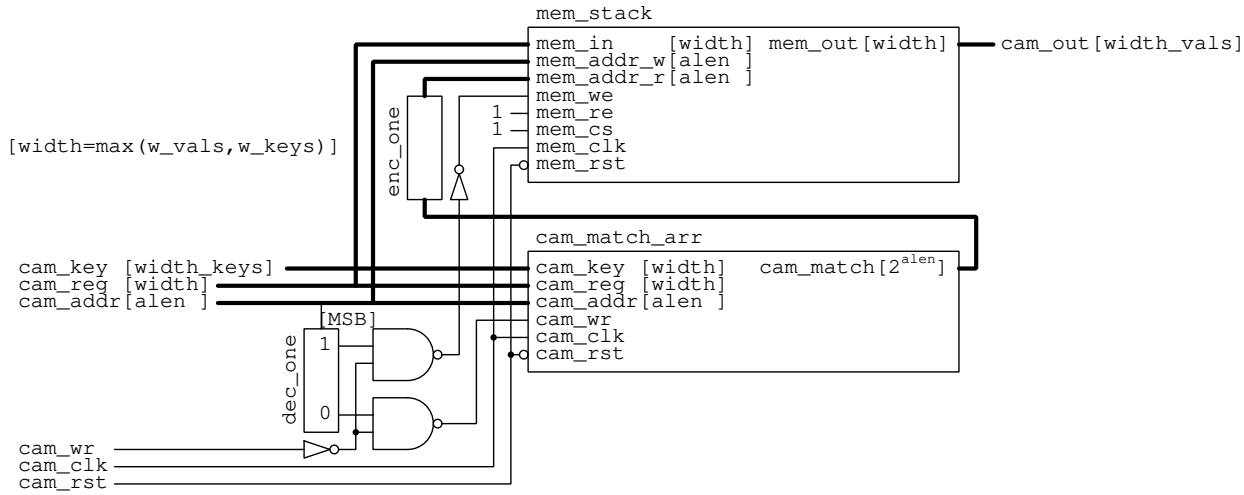


Figure 4.13: CAM stack schematic

The waveforms below demonstrate the operation of this module. In this particular test case, four key-value pairs are used. Both of these are programmed one after another: first the keys, then the corresponding values. Once the writing mode is disabled, each key (`0xA`, `0xB`, `0xC`, `0xD`) is used to retrieve all the stored values.

Table 4.1: Example dictionary

Keys [HEX]	11	22	33	44
Values [HEX]	A	B	C	D

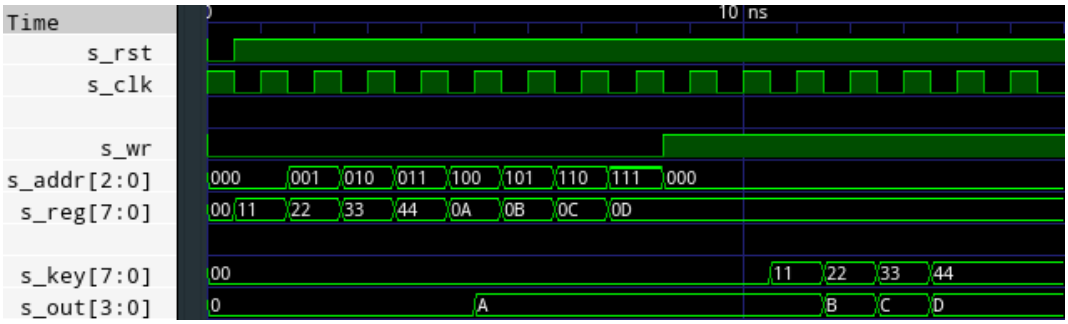


Figure 4.14: CAM stack operation

4.1.2.4 CAM Grid

Similar to the previous circuit, the CAM grid is a combination of a matching array with the memory grid, a two-dimensional memory arrangement. The main advantage of the memory

grid circuit is that the output busses of all the memory stacks are concatenated. In other words, when the chip-select-all signal `MEM_CSA` is set, all the stacks inside the grid are accessed at the same time.

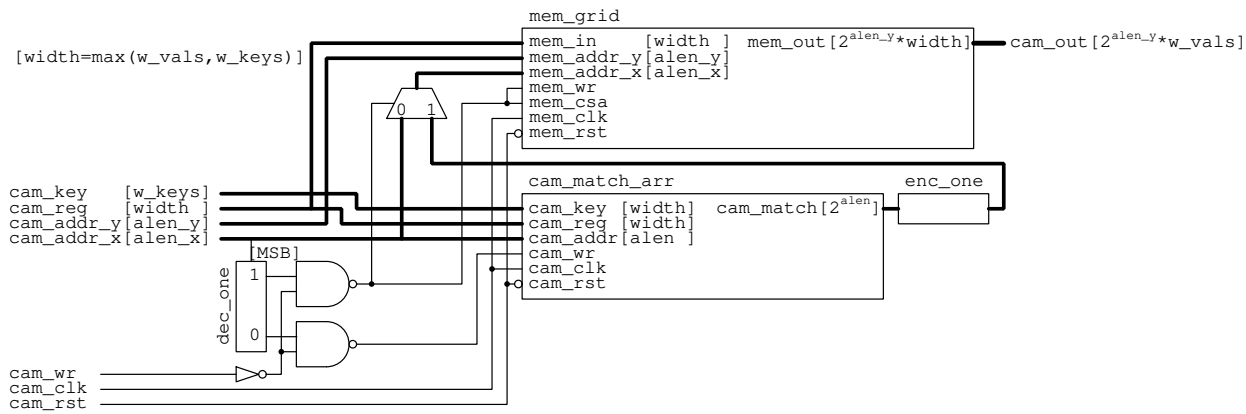


Figure 4.15: CAM grid schematic

Figures 4.16 and 4.17 demonstrate the operation of the circuit. It is similar to the CAM stack with the exception of having more values to be programmed into the grid. In the CAM stack, each key is associated with only one value. In the CAM grid, on the other hand, every key is linked to a set of values. The number of keys is specified by 2^{ALEN_X-1} and the number of values per key is specified by 2^{ALEN_Y}, where `ALEN_X` and `ALEN_Y` are both generic parameters.

The waveforms below are presented for a test-case with `ALEN_X = 3` and `ALEN_Y = 2`. Hence, there are 4 keys and each key retrieves a list of 4 values. The widths of keys and values are both specified by the parameters `W_KEYS` and `W_VALUES`, respectively.

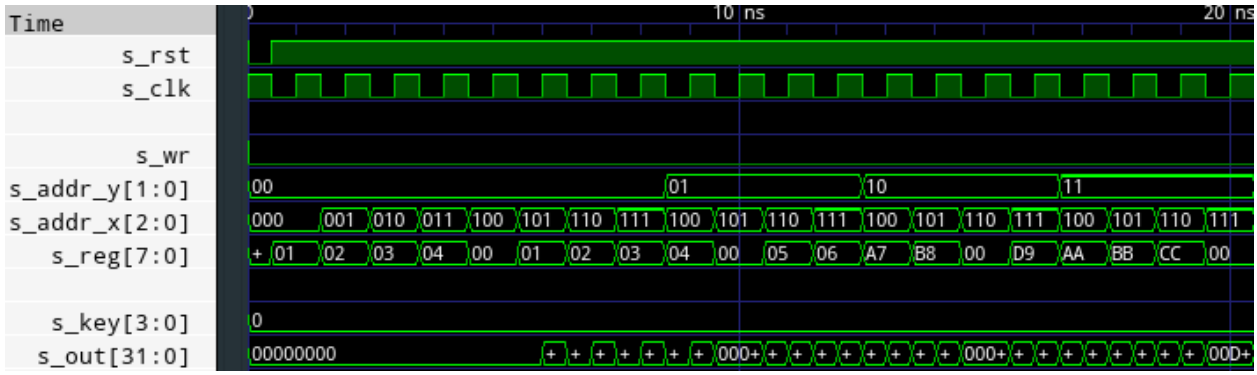


Figure 4.16: CAM grid operation (programming)

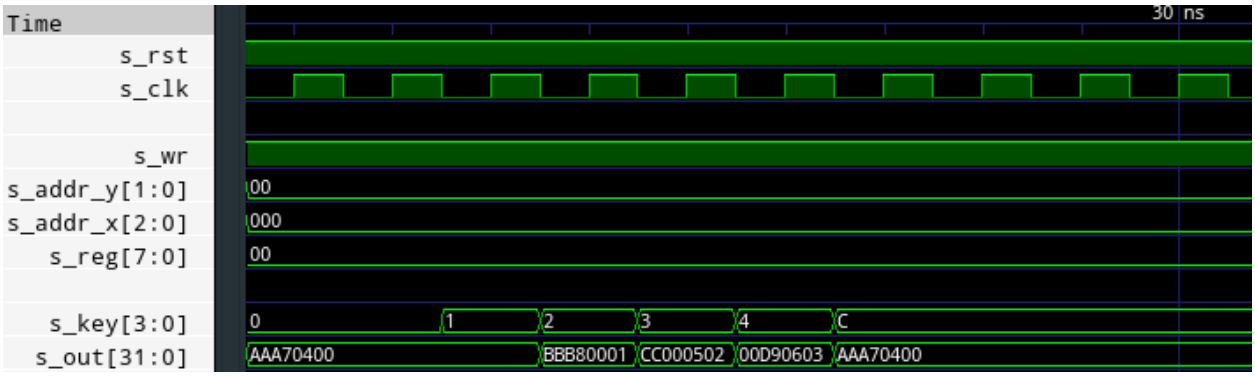


Figure 4.17: CAM grid operation (accessing)

4.2 Neurons & Synapses

The objective of this chapter is to elaborate on the VHDL implementation of neural and synaptic modules of the integrated circuit.

4.2.1 Neuron

In the first design iteration of the IC core, the chosen model for the spiking neuron is the Leaky Integrate-and-Fire [18]. This model is selected over the Izhikevich model due to having fewer configuration parameters and relative ease of implementation [16]. Also, as a stepping stone to the final design, the neural dynamics are written with floating-point representation. Although such a design is not fully-synthesizable, it allows one to observe the effects of spike delivery delays introduced by the core routers. In addition, this serves as a reference point for the comparison of various methods of implementation.

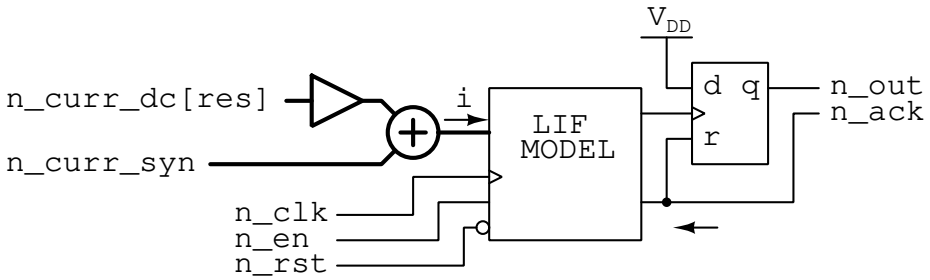


Figure 4.18: Neuron structure

The neuron module is controlled with the `RESET`, `ENABLE` and `CLOCK` pins. The function

of the enable signal is similar to the reset in that it pauses the operation of the neuron, but without resetting its state. Every positive-edge of the `CLOCK` signal triggers a `PROCESS` statement that updates the neural state variable using Euler's method, and compares the membrane voltage with the spike threshold.

There are also two inputs for currents. The current sourced by the pre-neuron synapse is analog and uses floating-point representation. The biasing current `CURR_DC` on the other hand is provided by a memory module, and uses a binary encoding scheme. As such, it requires a scaling parameter prior to being added with the synaptic current before being supplied to the neuron. In other words, using 2's complement representation, the largest code `CURR_DC = 01...11` is mapped to a value measured in milliamperes.

When the neuron is enabled, the value of the membrane voltage is updated with every positive-edge of the clock based on the model parameters for membrane resistance R_m and capacitance C_m. When a spike is emitted, a logical '1' is written into a D flip-flop to indicate an event. When this occurs, the operation of the neuron is halted, preventing it from integrating input currents. When an acknowledge signal `ACK` is sent by an arbitration device, the D flip-flop and the membrane voltage are both reset, and the neuron can resume its operation.

4.2.2 Synapse

The operation of the synapse is similar to that of the neuron. It has the same control pins for the reset, enable and clock signals. Given that both the synapse and the neuron are modelled with differential equations, both are clocked at the same rate. Each positive edge of the clock triggers an update in the states of the synapses and the neurons using the Euler method.

The synapse also has two inputs to indicate the transconductance values of each synapse. Assuming that the neural spikes all have the same voltage, synaptic strengths can be simply expressed in terms of currents.

Note that in this design, the synapse reads-in current values from the local memory grid and the content-addressable memory, which correspond to both local and external neuron events, respectively. Due to being retrieved from memory modules, both currents are stored in binary representation. Hence, before being fed to the synaptic equation, they are scaled in the same way as the biasing currents of the neurons. The output of the synaptic circuit is a single floating-point number that is connected directly to the neuron.

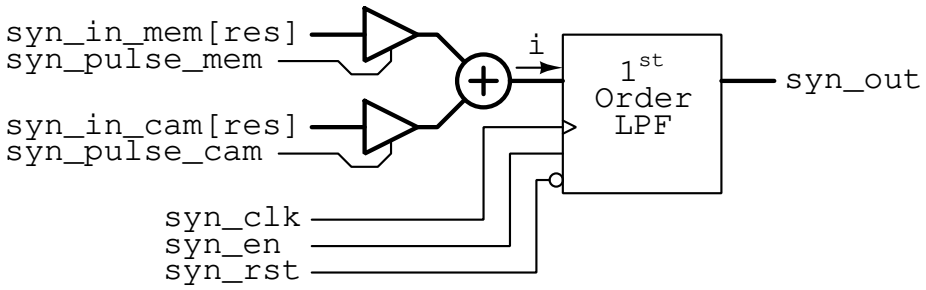


Figure 4.19: Synapse structure

The current inputs of the synapses are always connected to the memory modules' outputs. To restrict the inputs to impulses rather than constant values, two additional control signals are used: `PULSE_MEM` and `PULSE_CAM`, both of which are driven by the local router. To elaborate, when `PULSE_MEM = 1` at the instant of a positive clock edge, a short current pulse is injected from the memory grid into the synaptic circuit. This is demonstrated in Figure 4.20.

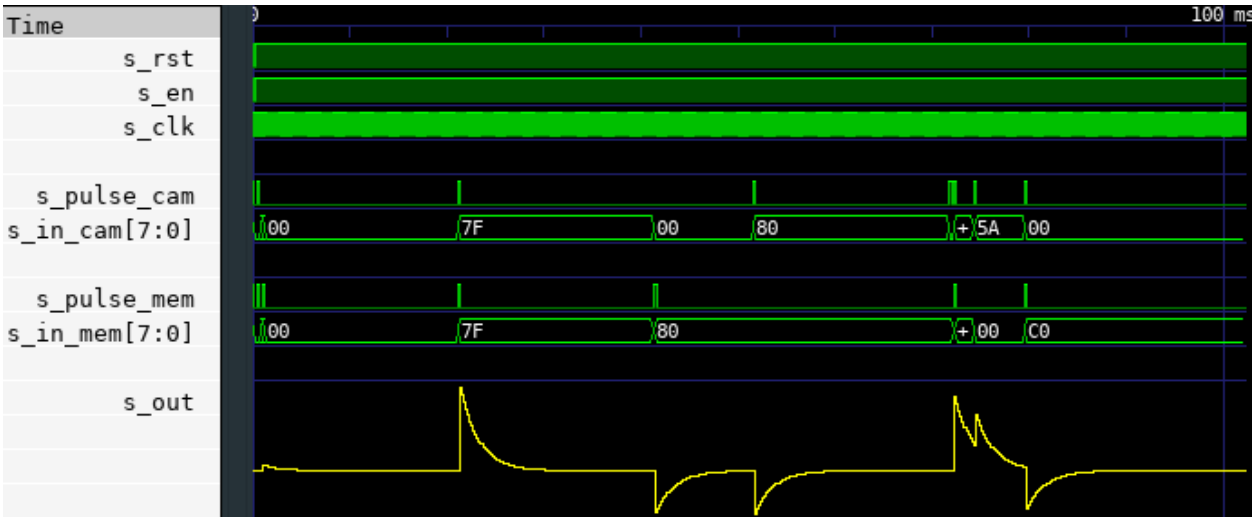


Figure 4.20: Synaptic circuit operation

4.2.3 *SN* Column

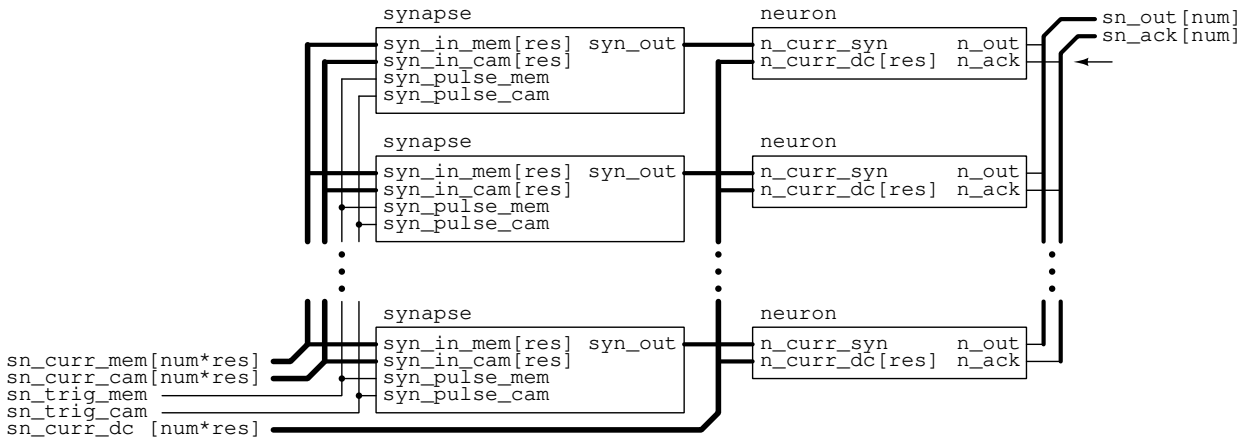


Figure 4.21: Synapse-Neuron Column

For ease of implementation, all the synaptic and neural pairings are placed in a column arrangement. In Figure 4.21, the control signals `RST`, `EN` and `CLK` are omitted for simplicity, but are connected to every synapse and neuron.

The pairs in the *SN* column receive inputs sourced by the external events, the local events, and finally the biasing currents for the neurons. The input vectors `SN_CURR_MEM`, `SN_CURR_CAM` and `SN_CURR_DC` all have the size of $NUM \times RES$, where NUM represents the number of synapse-neuron pairings, and RES stands for the resolution, or the number of bits, per current input. Hence, slices of each input are applied to the pairings.

The input signals labelled as `SN_TRIG_MEM` and `SN_TRIG_CAM` are connected to each synapse to simultaneously inject currents when events are delivered.

At the output of this module, the neuron outputs are bundled into a single bus `SN_OUT` with the size NUM. The input vector `SN_ACK` of the same size is used to transmit the acknowledge bits.

4.3 Arbitration

This section describes the arbitration module of the integrated circuit. In short, the objective of the arbiter is to serialize the events of the spiking neurons, encode their addresses and deliver them to the local router [32]. In addition, the arbiter sends release signals to the neurons that have spiked, allowing them to be reset. Ideally, the order in which the release

signals are sent is the same as the order in which the neurons have spiked.

As shown in Figure 4.22, the fundamental structure of the arbiter is that of a tree, where every node is responsible for giving priority to one of two inputs in case of colliding events [33]. All inputs and outputs have a bi-directional polarity: a neuron that has emitted a spike needs to be acknowledged by the arbiter after its address has been recorded. The arbitration node at the root of the tree has a short-circuited output in order to generate the acknowledge signals that get delivered to the corresponding neuron.

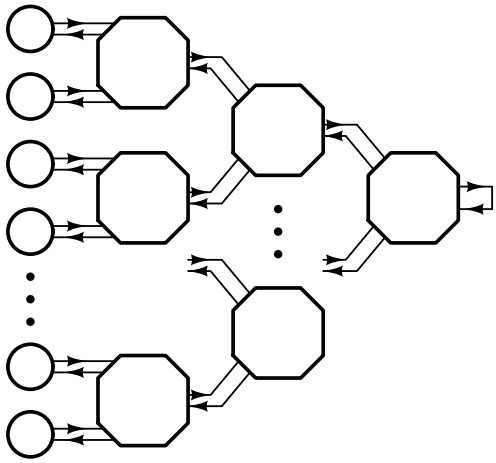


Figure 4.22: Arbitration tree

4.3.1 Arbitration Latch

The basic building block of the arbiter is the arbitration latch, whose purpose is to continuously monitor two neurons. As shown in Fig. 4.23, this latch has two inputs and two outputs. It also has three possible output combinations. When only one neuron outputs a spike, the latch simply transmits the logical '1' to the corresponding output. The output that has been set to '1' remains in this state until the spiked neuron is acknowledged (or released).

If both input neurons output two events with a small time difference, the arbitration latch can only keep one output active, and is only free to transmit the slower output once the first neuron has been acknowledged. This is depicted in Table 4.2 and Figure 4.24.

Table 4.2: Possible arbitration latch outputs

STATE	y_2	y_1
s_0	0	0
s_1	1	0
s_2	0	1
s_3	0	0

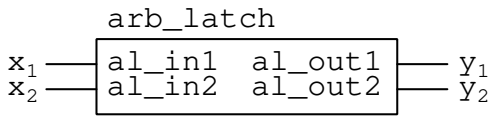


Figure 4.23: Arbitration latch interface

Simultaneous spikes need to be handled by the latch with a certain randomness. As a consequence, such a latch must be as symmetrical as possible in its physical implementation. The simplest method of implementation is done with behavioral modeling using an HDL. The corresponding finite-state machine is shown in Figure 4.25, where the arrows indicate the circuit inputs and the circles indicate its states. Although, only three states are required to encode the possible output combinations, four states are used for the purpose of balancing the outputs. The first time that both inputs spike together, priority is given to output the y_1; during the following simultaneous event occurrence, priority is given to the output y_2. This is done in order to ensure that, in the case of fast-spiking inputs, the arbiter allocates time for each neuron.

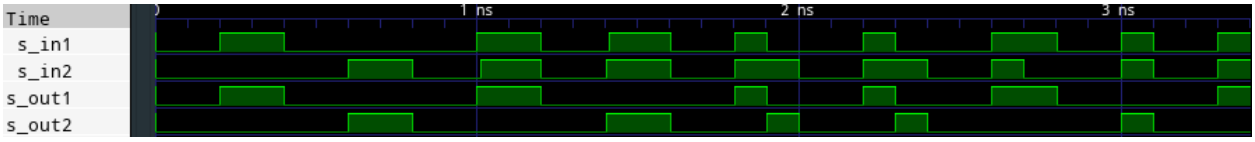


Figure 4.24: Latch operation

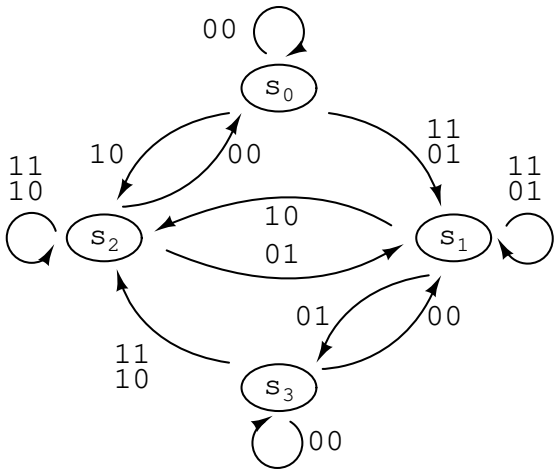


Figure 4.25: Finite-state machine

4.3.2 Arbitration Node

The arbitration node extends the functionality of the arbitration latch by funneling both latch output signals into one output, which allows the formation of the arbitration tree. In addition, this module transmits the acknowledge signals to the neuron whose address is being encoded and recorded by the router.

Other than the *OR*-ed outputs of the latch, the arbitration node also produces the release signals `AN_REL1` and `AN_REL2` that are used to acknowledge the latched-in neurons. The input `AN_ACK` shown to the right of Figure 4.26 sets one of the release signals with two *AND* gates driven by the arbitration latch. Finally, these signals are also used to indicate the address bit `AN_ADDR` of the spiked neuron: `AN_ADDR = 0` for the first neuron, and `AN_ADDR = 1` for the second.

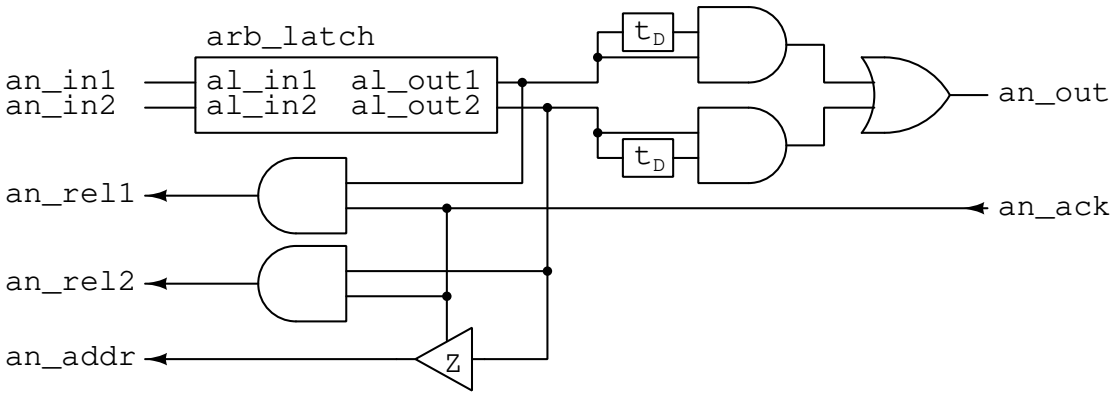


Figure 4.26: Arbitration node schematic

Prior to being combined with an OR gate at the output `AN_OUT`, the outputs of the arbitration latch are first sent through AND gates with single ends being delayed by buffers. This is done in order to introduce a glitch in the output of the arbitration node when the outputs of the latch directly transition between the states s_1 and s_2 of Figure 4.25.

Suppose that both neurons are spiking at high rates such that the outputs of the latch alternate without both of them going through an idle state. With an OR gate directly connected the outputs of the latch, the output of the arbitration module will always be a logical '1'. As a result, the arbitration tree will get locked on two specific branches, and will continue to ignore other potential spiking neurons. In other words, the acknowledge bits of the arbiter will be sent to only two neurons regardless of the states of the remaining network.

The AND gates with one buffered input cause the transition of the arbitration latch to go through a dead-time of duration t_D, regardless of the spiking frequency of the neurons. The duration t_D only needs to be only long enough to cause the output of the corresponding AND gate to glitch to a logical '0'.

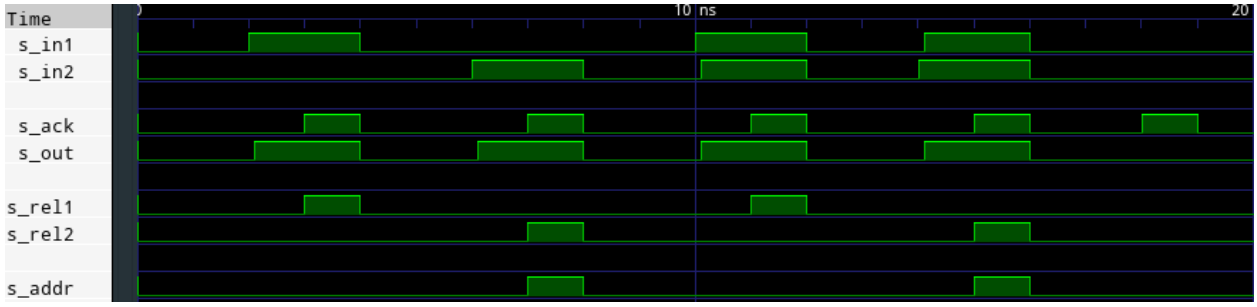


Figure 4.27: Node operation

4.3.3 Arbitration Column

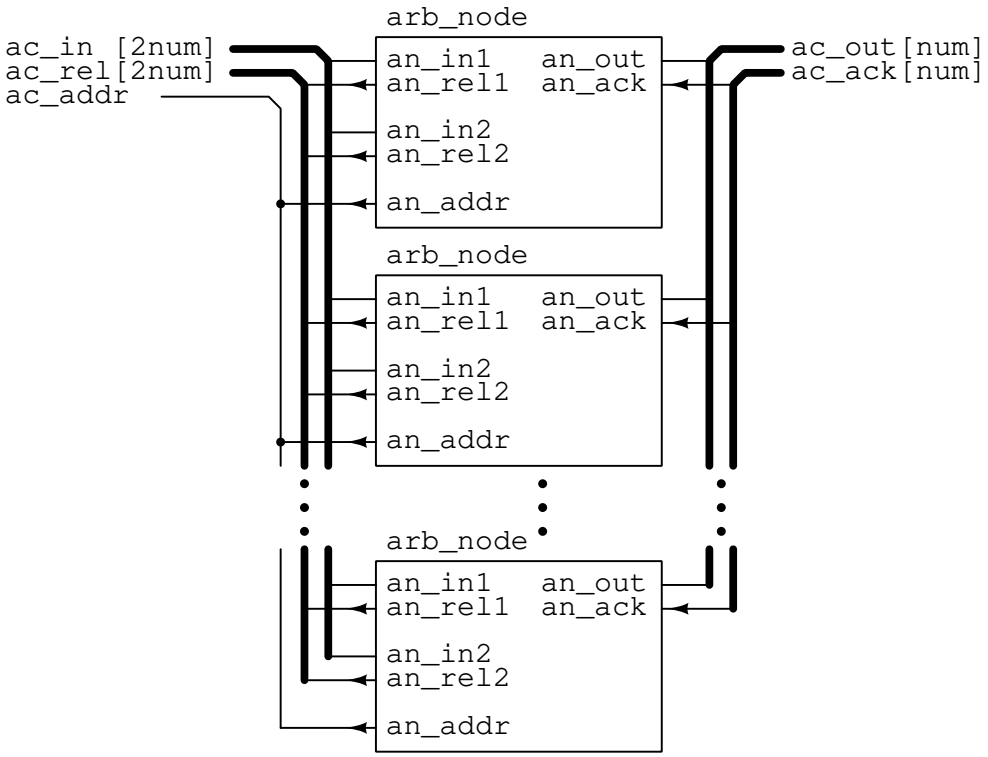


Figure 4.28: Arbitration column schematic

For the sake of simplicity in implementing the arbitration tree that is described in the following section, the individual nodes are first combined in columns. As shown in Figure 4.26, the address bit provided by each node is fed through a tri-state buffer. Hence, the outputs `AN_ADDR` of each node belonging to the same column can be simply connected together. Each column is responsible for encoding only one bit of the address event.

4.3.4 Arbitration Tree

The complete arbitration tree is built by cascading columns of arbitration nodes. The number of nodes in the first column is half the number of neurons. Each following layer has half as many nodes as the previous layer. The Figure 4.29 shows that the tree is expressed with a generic parameter `ALEN` indicating the length of the address event. Hence, the number of neurons on each integrated circuit is 2^{ALEN}.

As previously stated, each column of the tree encodes one bit of the spiking neuron address. The column at the input of the tree encodes the least significant bit (LSB), and the final

column provides the most significant bit (MSB).

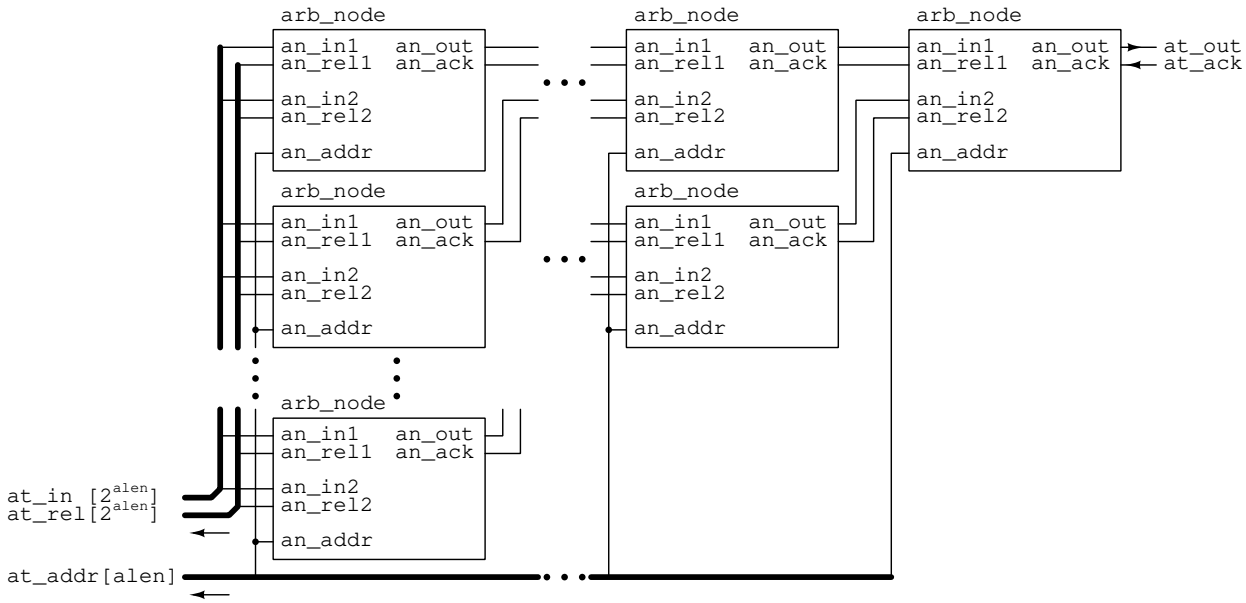


Figure 4.29: Arbitration tree schematic

In Figure 4.30, the input vector of the arbitration tree is first tested with random thermal codes (a string of '0's with a single '1'). The output goes $HIGH$ in the case of an event. A delayed acknowledge signal is supplied to the root of the tree, which results in a release vector `AT_REL` that matches the input. The address of the $HIGH$ bit is also encoded.

Moreover, multiple bits being set to logical '1's are also shown to be decoded one-by-one, with a dead-time glitch being present in the output signal `AT_OUT` around the time values $t = 18\ ns$ and $t = 26\ ns$.

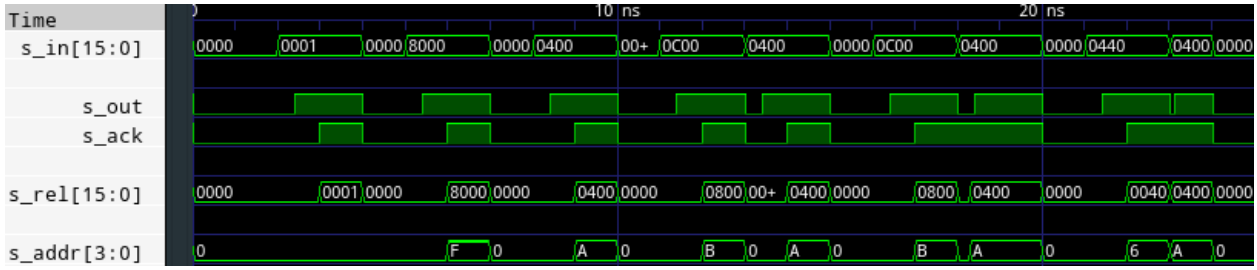


Figure 4.30: Tree operation with $N = 16$

4.3.5 Arbiter

The arbitration tree is implemented as an asynchronous circuit in order to allow the future use of analog neurons that can have arbitrary spiking times. However, for the arbiter to interface the local router of the IC, D flip-flops are used to buffer the release signals. The number of flip-flops is equal to the number of neurons in the circuit. This is shown in Figure 4.31.

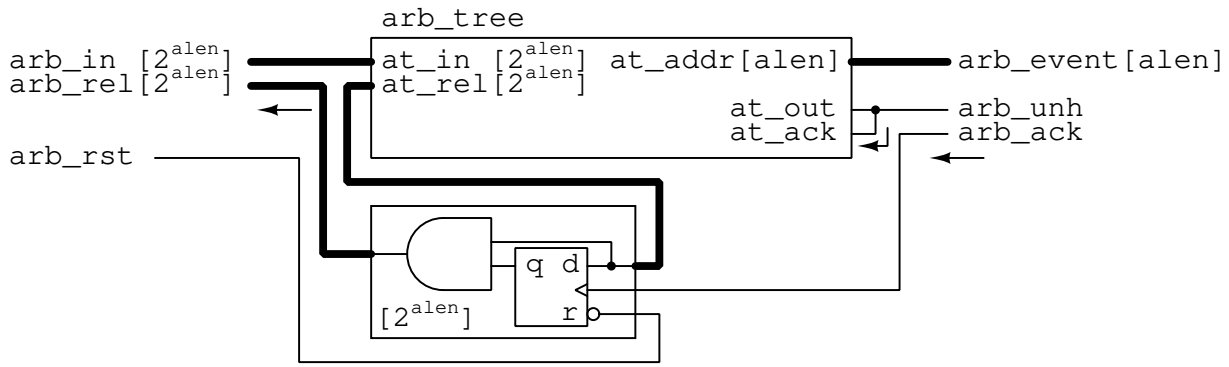


Figure 4.31: Complete arbiter

Assume that at time $t = 0ns$, one neuron outputs a spike and gets latched into a state awaiting an acknowledge signal. At this point, the input of the arbiter consists of a column of '0's with a single '1' at the index of the latched neuron. This event propagates through the arbitration nodes to the root of the tree, and backwards to the column of neurons through the input `AT_ACK`. By doing so, this signal encodes the address of the event. The outputs `ARB_EVENT` and `ARB_UNH` are supplied to the local router. Before the spiked neuron is released from its awaiting state, the acknowledge signal is buffered. Once the local router has registered the event, it sends a positive edge to the D flip-flops through the pin `ARB_ACK`, thus allowing the release signal to be finally delivered to the latched neuron. An AND gate is used so that the releasing signal can go to logical '0' without having to actively reset each flip-flop. The arbiter is now on stand-by for the next spiking event.

The interfacing of this device is shown in the simulation below. The signal highlighted in yellow demonstrates that the acknowledge signal provided by the local router results in a vector of bits `AT_REL`, consisting entirely of '0's with a single possible '1' at the index of the neuron to be released.

Figure 4.32: Arbiter operation

4.4 Routing

Neural networks are scalable by nature, and can have large numbers of neurons, which imposes the requirement of scalability on any hardware implementation of SNNs. This is achieved by interconnecting the desired number of cores through the use of routing circuitry. Following the example of the architectures presented in section 2.3, there may be multiple cores on every IC. However, the number of cores and their interconnection is yet to be decided.

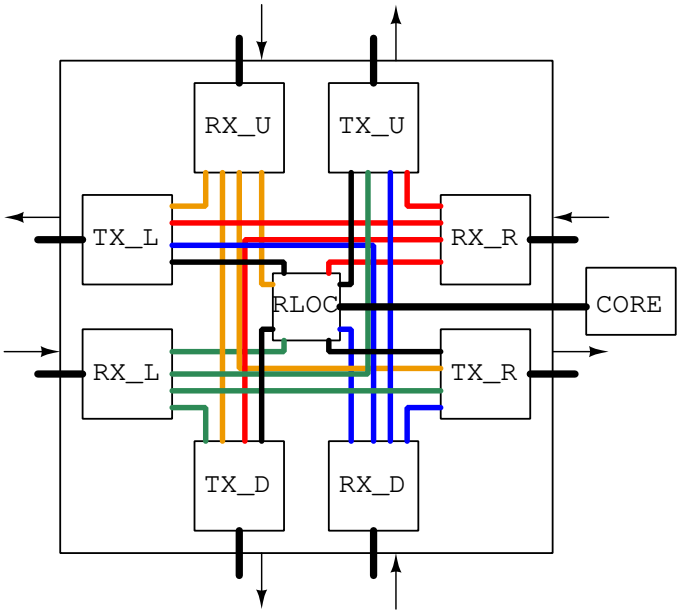


Figure 4.33: Router architecture

Given that all neural spikes have the same voltage magnitude, information is encoded in their timing. Hence, to minimize the distortion of information, the delays of the routers must be as short as possible relative to the time constants of the synapses and the neurons.

The chosen router architecture for a single core is shown in Figure 4.33. It consists of one local router, four receiving (RX) routers, as well as four transmitting (TX) routers. The equally colored traces are outputs of the RX routers, and the interior black traces are outputs of the local router. In addition, the module `RLOC` serves as an interface for the core and the whole router.

The local router is connected to all the receiving and transmitting routers. It captures all the local events, and routes them back to the local neurons. In addition, local events can also be sent to the transmitting routers depending on the contents of the local routing table.

The receiving routers accept external events. These can either be sent to local neurons, or re-directed to other transmitting routers, or both as decided by the routing tables. One thing worthy of note is that, if a spike is received on one side of the IC (e.g. left RX router), it cannot also be transmitted from the left TX router since that would indicate a sub-optimal routing structure.

The transmitting routers send spikes, which are encoded as source addresses, from the local router and three receiving routers.

4.4.1 Router Communication

The arbiter, local and external routers, etc. all employ the same communication method composed of the following signals:

- `EVENT` : signal bus transmitting the event address;
- `UNH` : signal indicating that the event is unhandled (not recorded);
- `ACK` : signal from receiving end indicating that the event is acknowledged.

The signals `EVENT` and `UNH` are both either inputs or outputs, and the signal `ACK` has the opposite polarity of the first two. In a transmitting port, `EVENT` and `UNH` signals are outputs, and `ACK` is an input. In a receiving port, the polarities are reversed. Figures 4.34 and 4.35 demonstrate the basic operation of this communication.

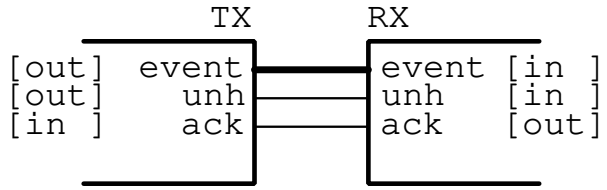


Figure 4.34: Communication port

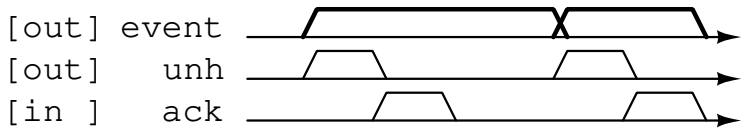


Figure 4.35: Communication pattern

When an event is being transmitted (by a TX router or arbiter), the spike address is written on the `EVENT` and the flag `UNH` is raised. When ready, a receiving port raises the signal `ACK` to indicate that it has stored the event in its own buffer, and the flag `UNH` is asynchronously reset.

4.4.2 Event Buffer

The event buffer is the core component in every type of router used in the present implementation. All neuron spikes are applied in parallel to the input of the arbiter. In order to reconcile collisions and to interconnect multiple ICs, all address events are serialized and stored in circular event buffers. The conceptual model is shown in Figure 4.36. This buffer essentially consists of a memory array and two pointers, `HEAD` and `TAIL`. A new event that is being recorded in the memory array is written to the address pointed to by the `HEAD` pointer, and an event that is being delivered is accessed by the `TAIL` pointer. Once the limit of the memory array is reached, the `HEAD` pointer simply restarts the writing procedure at the first entry.

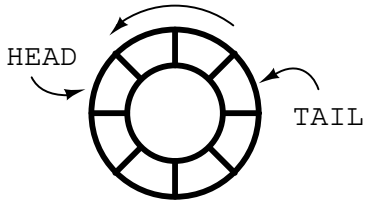


Figure 4.36: Circular buffer

The Figure 4.37 below described the implementation of this buffer. The memory array is implemented with the memory stack that has a dual port for writing and reading. The number of entries is equal to 2^{ALEN}, where `ALEN` is a generic parameter indicating the address length of the memory. The size of each entry is specified by the parameter `WIDTH`, which indicates the length of each address event.

The two components labelled as INCR correspond to counters. The widths of these counters are equal to the number of address bits of the memory stack. At start-up, both of these counters store a value of 0. Each positive edge at the input up increments the saved value by 1.

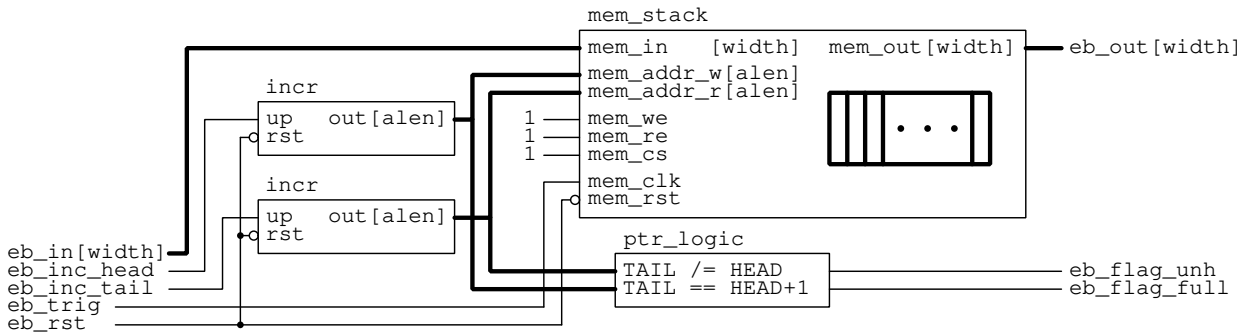


Figure 4.37: Event buffer schematic

Before any event has been recorded, the HEAD and TAIL pointers are both accessing the memory array at index 0. When both pointers store the same address, the pointer logic circuit keeps the flags EB_FLAG_UNH and EB_FLAG_FULL deasserted. When a new event appears at the input of the buffer EB_IN, a positive edge is applied to EB_TRIG in order to save it in the memory array. Next, the value of the HEAD pointer is incremented, and the flag EB_FLAG_UNH is raised, thus indicating that there is now an undelivered event stored in the buffer. This is demonstrated in Figure 4.38.

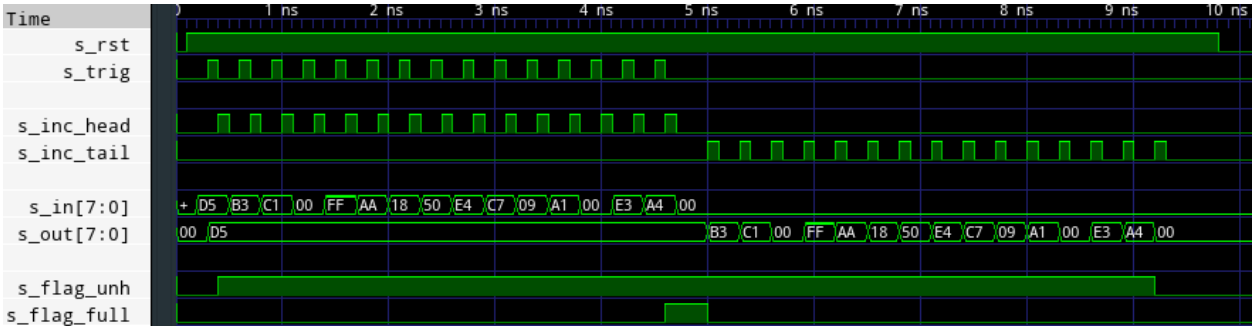


Figure 4.38: Event buffer operation (ALEN = 4, WIDTH = 8)

In Figure 4.38, the buffer is first loaded with 15 entries, at which point the HEAD pointer is right behind the TAIL pointer, and the flag EB_FLAG_FULL is raised to indicate that the buffer is full.

A module that is reading the contents of the event buffer is given control of the TAIL pointer.

In the simple example above, the buffer iterates through all its contents by receiving positive edges that increment the `TAIL` pointer. Once the value of `TAIL` catches up with the value of `HEAD`, the flag `EB_FLAG_UNH` indicating unhandled entries is reset.

Due to the finite register width of the counters, the addresses stored in the pointers wrap-around after reaching the last index of the memory array.

4.4.3 Transmitting Router (TX)

The transmitting router is the simplest of the three due to its lack of routing tables. Every TX router has four input ports and one output port. In Figure 4.39, the four input ports are shown in red with each port consisting of the signals `EVENT`, `UNH` and `ACK`. The output port is indicated in blue.

As explained at the beginning of this chapter in Figure 4.33, each TX router is connected to the local router as well as the three opposing RX routers. For this reason, the input ports of a TX router are labelled as `LOCAL`, `LEFT`, `FW`, and `RIGHT`, all of which are relative directions. For example, in the case of a TX router sending events at the right edge of the IC, the RX routers' events are received from the `UP`, `LEFT`, and `DOWN` sides of the IC.

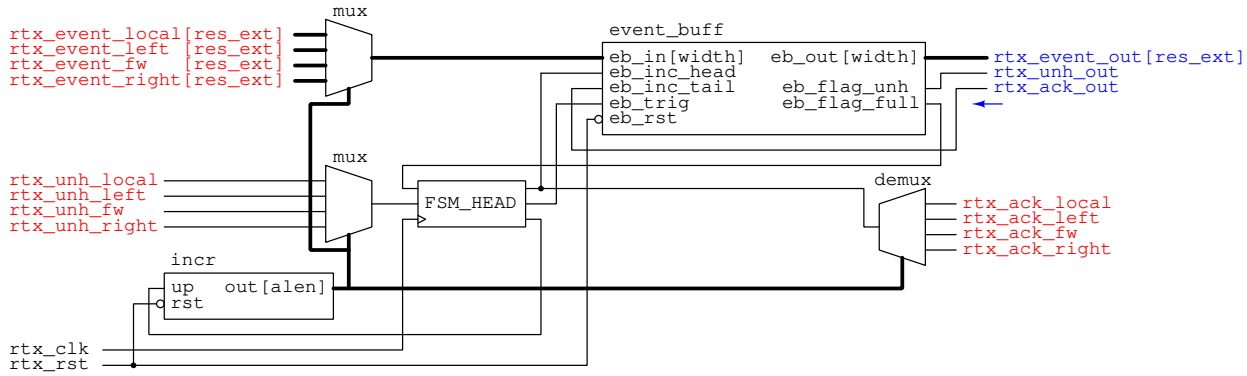


Figure 4.39: Transmitting router (TX) schematic

The counter `incr` shown in the figure above has 4 states and is used to iterate through all four input ports using two multiplexers for the `EVENT` busses and `UNH` flags, as well as a demultiplexer for the `ACK` signals. This iteration is managed by the finite state machine in Figure 4.40.

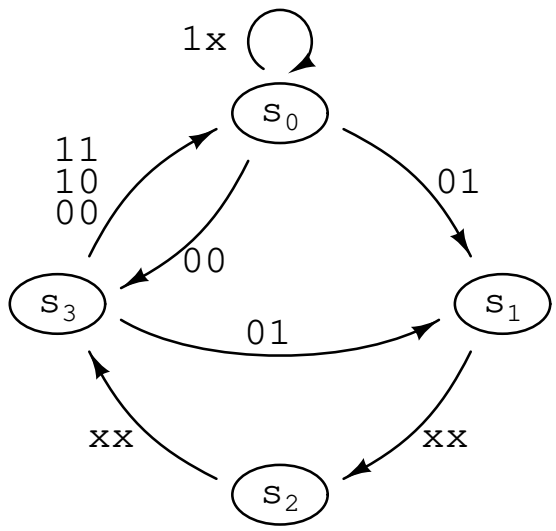


Figure 4.40: Finite-state machine

At each positive-edge of the clock, the FSM reads the values of the event buffer flags: `EB_FLAG_FULL` and `EB_FLAG_UNH`, which are indicted on the transitions of the FSM. Based on the present state, the FSM writes the values for three control signals `up`-signal for the counter increment, `EB_TRIG` to latch-in the buffer input, and `EB_INC_HEAD` to increment the `HEAD` pointer of that buffer. The outputs of the FSM are summarized in the Table 4.2.

Table 4.3: Outputs of the finite-state machine

STATE	`INC_UP`	`EB_INC_HEAD`	`EB_TRIG`
s_0	0	0	0
s_1	0	0	1
s_2	0	1	0
s_3	1	0	0

The objective of this FSM is to save incoming events into the buffer.

1. At the start of the operation, if there is an event coming from the LOCAL router, the FSM transitions from state s_0 to state s_1, and a positive-edge is sent to the input `EB_TRIG` saving the local event into the buffer.
2. At the next clock edge, the FSM goes to state s_2 regardless of the inputs, and increments the `HEAD` pointer of the event buffer.
3. Again, regardless of the inputs, the FSM transitions to state s_3 and increments the counter, thus changing the port being fed-through by the MUXes and the DEMUX.

4. Now processing the LEFT port, the FSM repeats the steps 1-3 if the flag `UNH` of that port is asserted, or else, it returns to state s_0.
5. If for a given port the flag `UNH` is deasserted, the FSM simply jumps to state s_3 in order to increment the address of the MUXes and the DEMUX and continue iterating through the ports without registering any events.
6. Finally, if the event buffer becomes full and its flag `EB_FLAG_FULL` is asserted, the FSM remains at state s_0 until memory becomes available once again.

In Figure 4.41 below, multiple events are presented to all the input ports. Note that the events without an asserted `UNH` flag are ignored by the FSM and not registered into the event buffer. The signals highlighted in yellow show the reading of all the events from the output port. Thus, each recorded event is successfully serialized and transmitted.

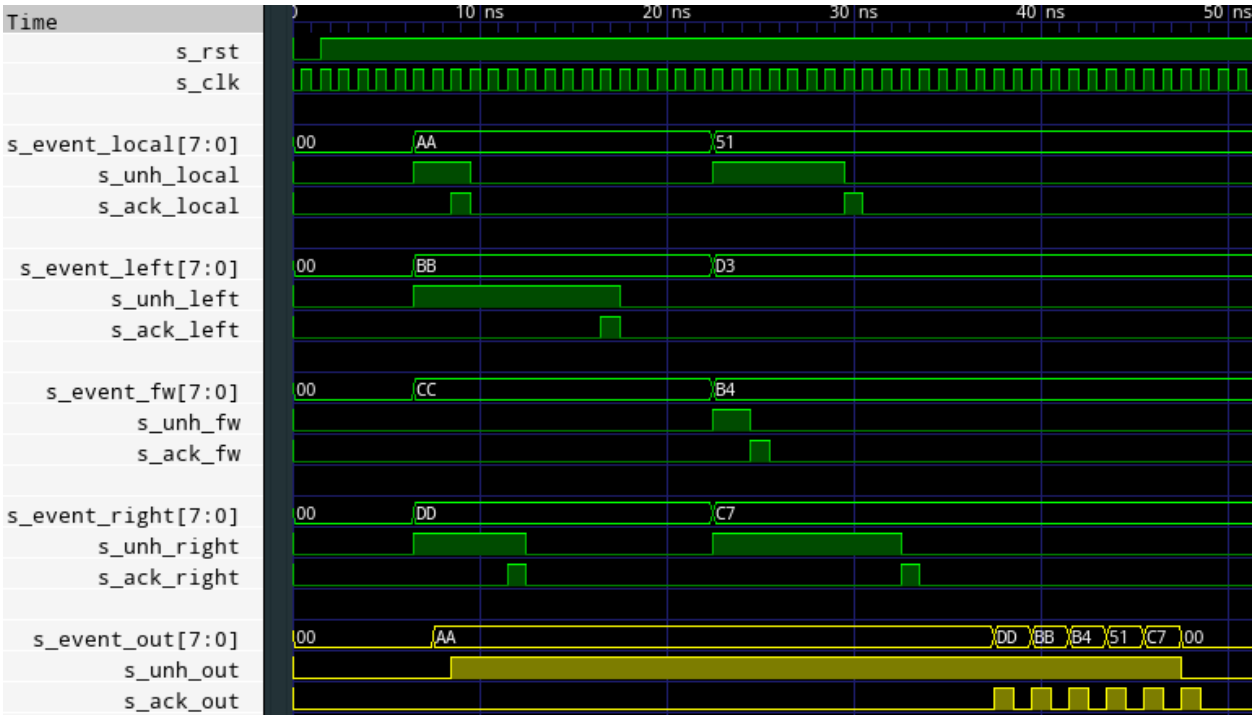


Figure 4.41: Transmitting router operation

4.4.4 Receiving Router (RX)

The receiving router functions similarly. It has one input port and four output ports that correspond to the local router and three TX routers. The main addition is the routing table implemented with content-addressable memory (CAM).

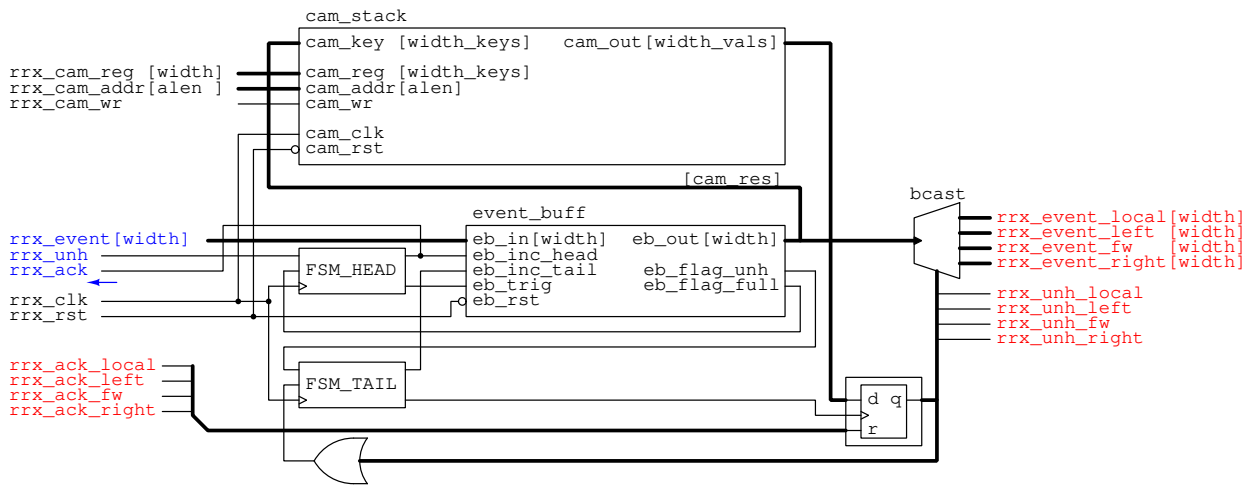


Figure 4.42: Receiving router (RX) schematic

As seen in Figure 4.42, the output of the event buffer is used as a key at the input of the CAM stack described in Chapter 3. Moreover, the multiplexers and the demultiplexer of the TX router are replaced with a single broadcasting module. This device, labelled as `BCAST`, has one input bus, four output busses and four select lines. In a demultiplexer, two select lines would be used to choose the output bus that would be made equal to the input bus. In the broadcasting device, each select bit can enable one output bus. The Table 4.4 below summarizes its operation. Most '0's have been omitted from the table.

Table 4.4: Broadcasting device truth table

s_3	s_2	s_1	s_0	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	x
		1				x	
		1	1			x	x
	1				x		
	1		1		x		x
1				x			
1	1	1	1	x	x	x	x

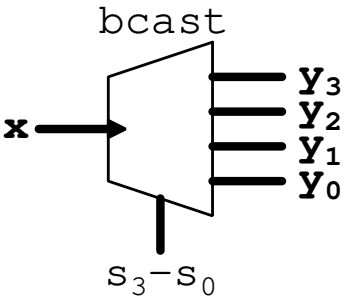


Figure 4.43: Broadcasting device symbol

The routing table is sized with three generic parameters:

- `ALEN` sets the number of key-value pairs to 2^{ALEN-1};
- `WIDTH_KEYS` sets the number of bits for each key, which is made equal to the address

length of an external spiking neuron;

- WIDTH_VALS sets the number of bits for each value.

In the chosen architecture design, the last parameter is always set to WIDTH_VALS = 4 due to the broadcasting to four ports. When an event at the output of the event buffer is being delivered, it is first used to retrieve its directions from the routing table. Next, the value obtained from the latter is latched into an array of D flip-flops, whose outputs are applied to the select lines of the broadcasting device. The four bits at the output of the routing table indicate the directions LOCAL, LEFT, FORWARD, and RIGHT from the most significant bit to the least significant bit.

As an example, suppose that an RX router on the LEFT edge of the IC receives the event EVENT = 0x74 with a routing table entry associated with the value 1010. These bits are first entered into four D flip-flops, which enable the local (RRX_EVENT_LOCAL) and forward (RRX_EVENT_FW) output busses of the broadcast unit. The latter corresponds to the TX router on the RIGHT edge of the IC. The same select lines are used as asserted UNH flags for the relevant ports. When an acknowledge signal is received for the local port, the positive-edge on the input RRX_ACK_LOCAL resets its D flip-flop, thus disabling the output bus RRX_EVENT_LOCAL and its flag RRX_UNH_LOCAL. In the meantime, the forward output port RRX_EVENT_FW is handled in the same fashion by the TX router on the opposite side of the IC. Once all the D flip-flops have been reset (and their OR-ed signal is '0'), the RX router proceeds with the next event in its buffer.

As seen in Figure 4.42, the RX router has two FSMs. The finite-state machine FSM_HEAD is responsible for recording incoming events while ensuring that the event buffer is not full (EB_FLAG_FULL = 0). The finite-state machine FSM_TAIL monitors for any unhandled events in the buffer. As described above, it is responsible for writing the outputs of the routing table into an array of four D flip-flops, and to increment the TAIL pointer.

Table 4.5: Outputs of FSM_HEAD

STATE	EB_INC_HEAD	EB_TRIG
s_0	0	0
s_1	0	1
s_2	1	0

Table 4.6: Outputs of FSM_TAIL

STATE	EB_INC_TAIL	S_LATCH_DFF
s_0	0	0
s_1	0	1
s_2	1	0

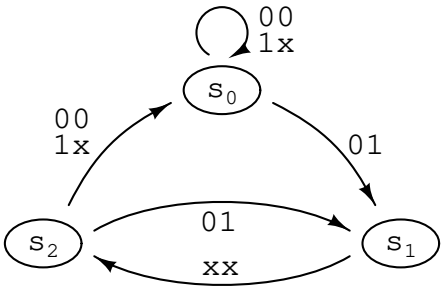


Figure 4.44: FSM for `HEAD` pointer

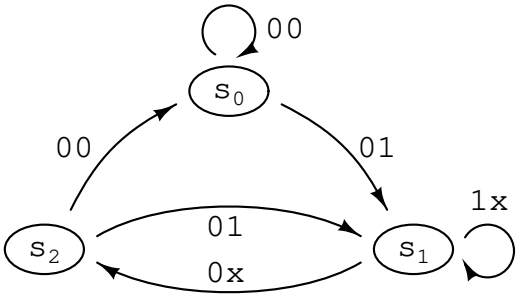


Figure 4.45: FSM for `TAIL` pointer

The operation of the TX router is shown in Figure 4.46, where the yellow traces show the input port. On the output traces, all ports are shown to handled simultaneously per event. The routing is shown in Table 4.7, where the keys are the four most significant bits of the event bus. **Note:** The width of each key in the routing table could be smaller than the width of the events. This can be used to optimize the sizes of the routing tables.

Table 4.7: Sample routing table

KEYS	A	B	C	D	E	F	8	9
VALUES	8	6	F	B	4	5	3	4

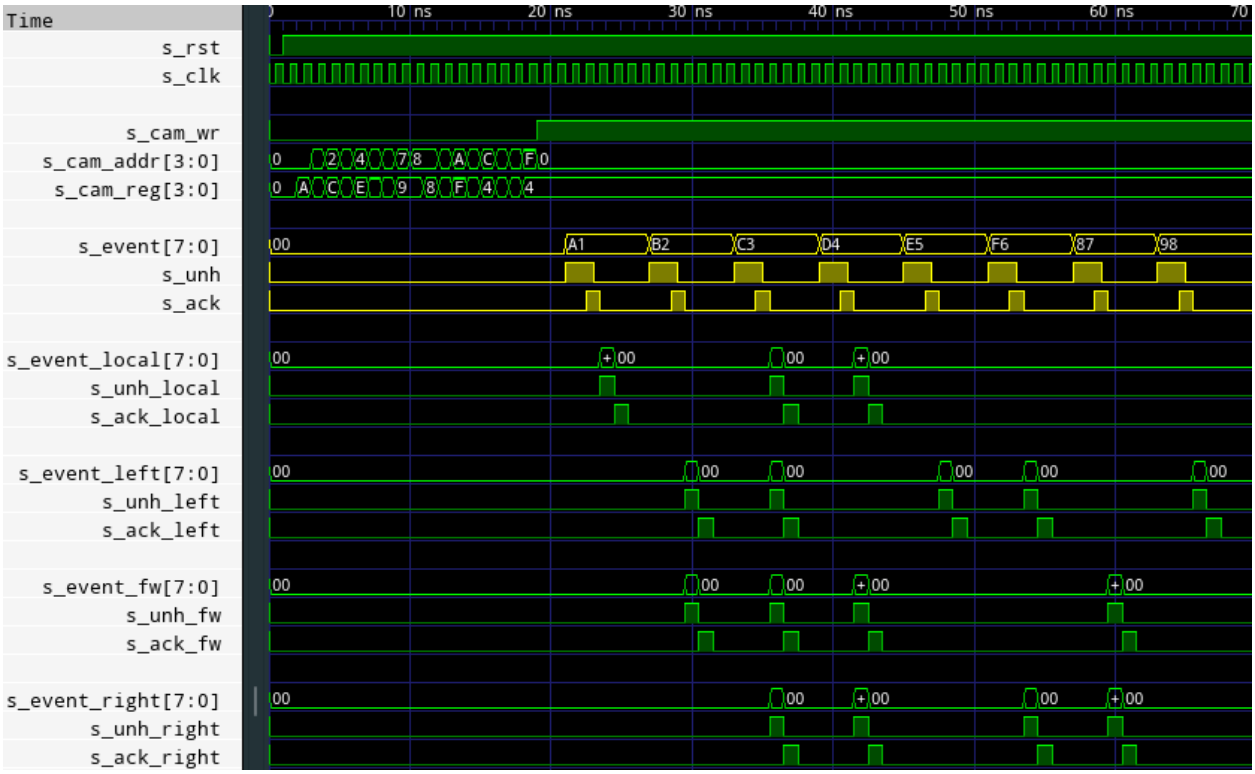


Figure 4.46: Receiving router operation

4.4.5 Local Router

As shown in Figure 4.33, the local router is at the core of the full router. It receives events from all four RX routers, and transmits to all TX routers. In addition, it also routes the events to and from the local neurons. Its design is essentially a combination of one TX router (upper portion of the figure), and one RX router (lower portion of the figure). One main difference is the presence of two event buffers: one is used to deliver spiking events back to the local neurons, and the second is responsible for broadcasting all local events outwards.

RX ports are shown in red, TX ports - in blue, and local ports - in purple. The signal bus `R_CAM_EVENT` delivers external events, and therefore, accesses the CAM grid for the synaptic values. The signal bus `R_MEM_EVENT` delivers local events, and retrieves synaptic values from the MEM grid. The associated signals `R_CAM_SYN_TRIG` and `R_MEM_SYN_TRIG` send positive edges to activate the synaptic circuits.

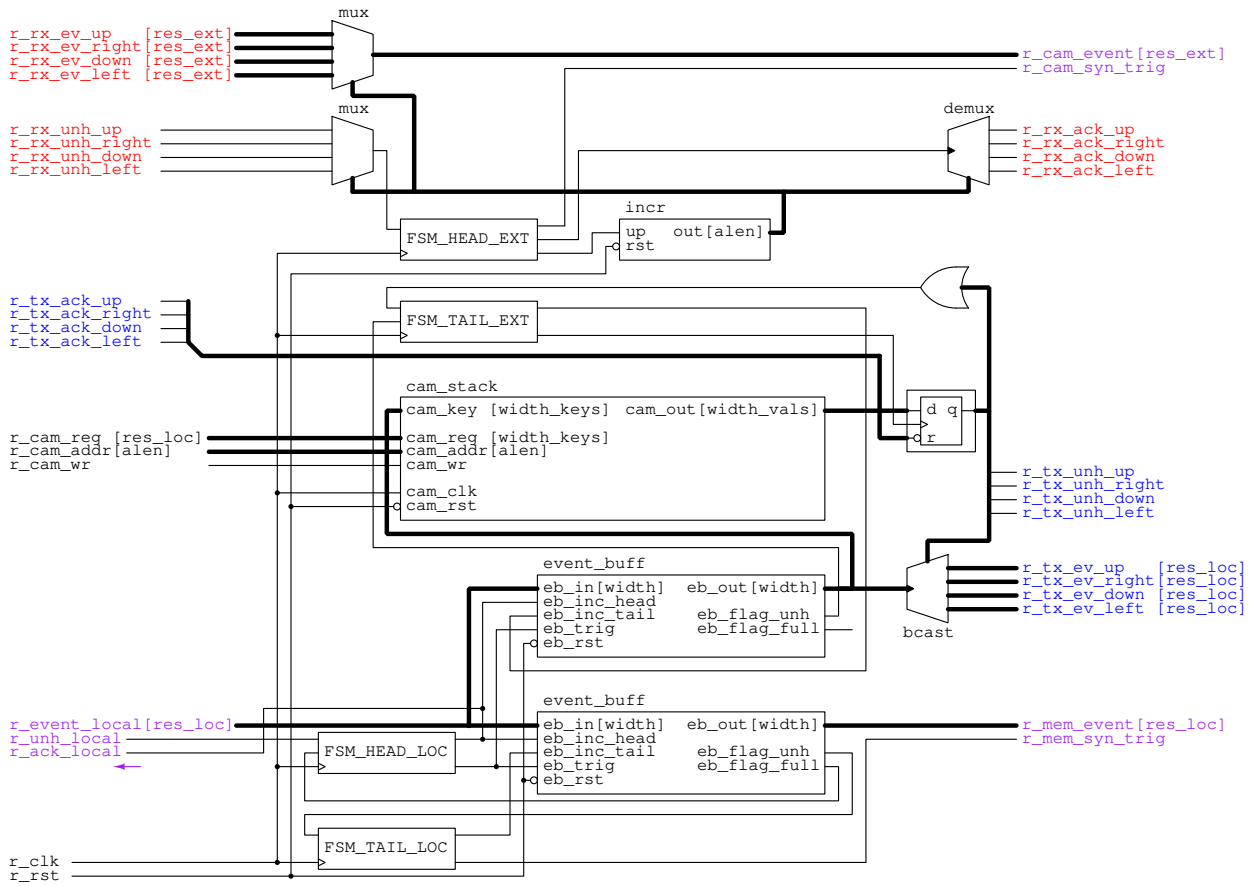


Figure 4.47: Local router schematic

In the simulation shown in Figure 4.48, the routing table is first programmed. Next, local events are sent to the input. The yellow traces show spiking events being delivered to local synapses and neurons.

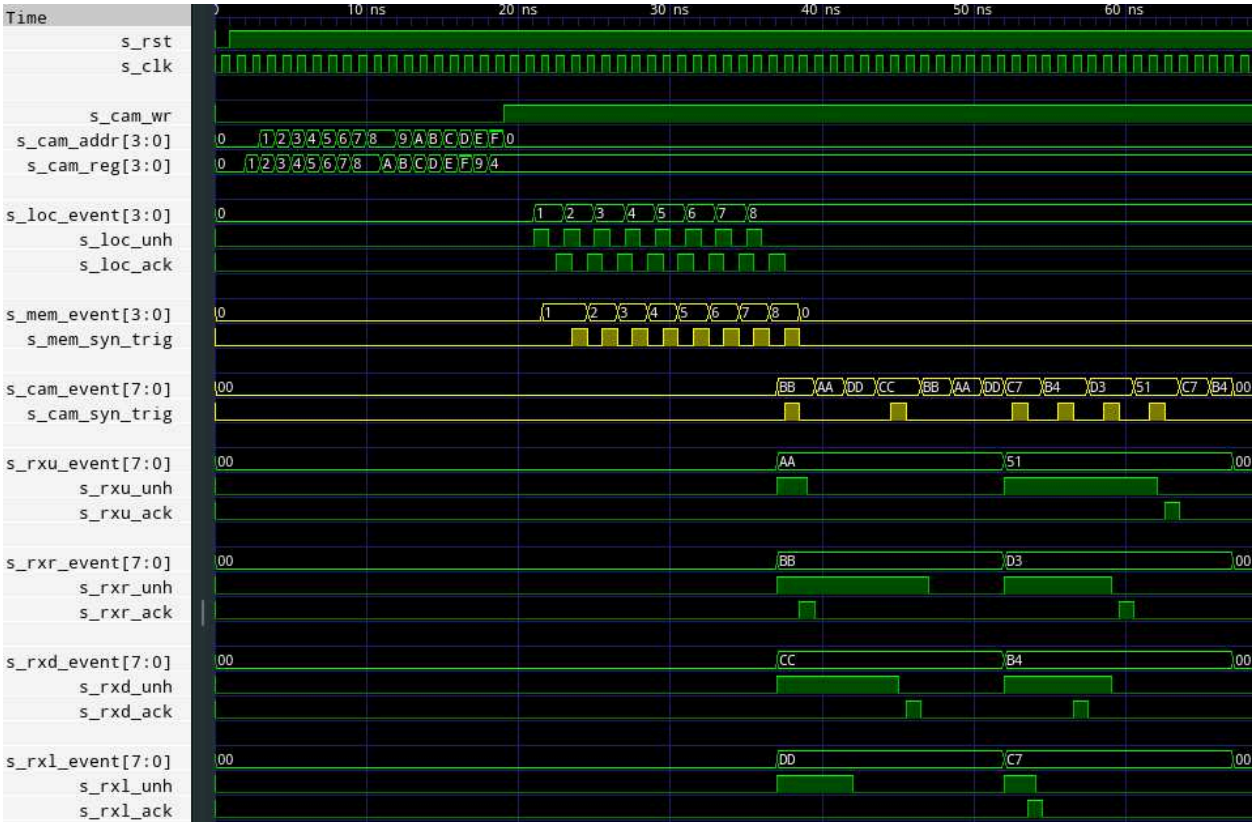


Figure 4.48: Local router operation

4.4.6 Combined Router

The full router is presented in Figure 4.49 and is the detailed implementation of the router architecture shown in Figure 4.33 at the beginning of the current chapter. Note that a register is added to the first select line of the address decoder. This register is used to store the address of the IC. Prior to an event being transmitted from the TX router, the address of each IC is appended to the address of the firing neuron to indicate the full origin of each event.

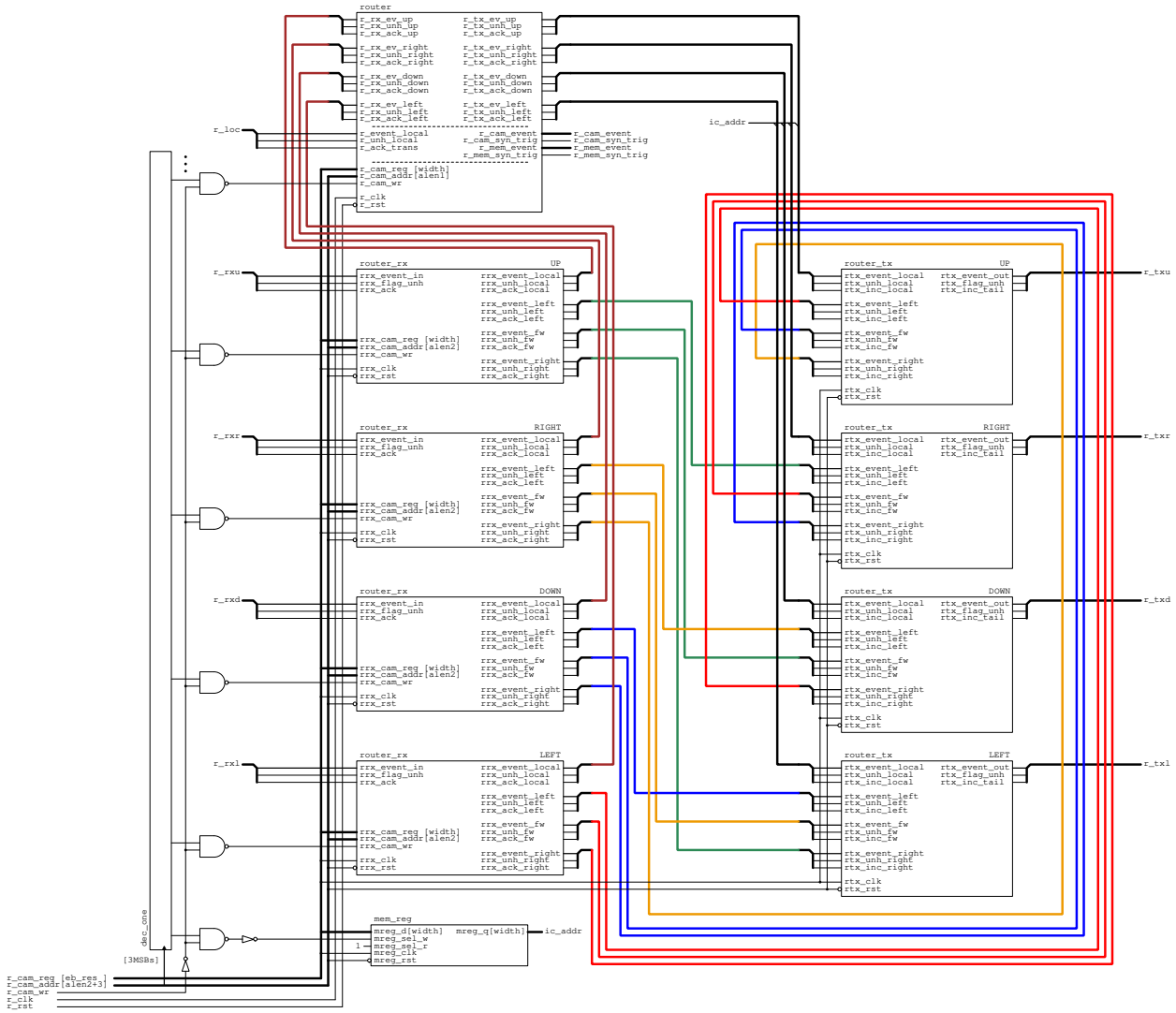


Figure 4.49: Full router schematic

4.4.7 Local Router Bridge

In Section 3.3, a method for optimizing the routing delays was presented, which is applicable to SNNs with clusters of neurons that can be isolated. As an example, the neurons in the output layer of a network do not have any destination neurons. Shown in Figure 4.50 is the schematic for the jumper module. The pin `J_EN` is connected to the select pin of every multiplexer and demultiplexer.

When this circuit is disabled, the arbiter's output port is connected to the input port of the router, where each port is composed of the signals `EVENT`, `UNH` and `ACK`. The router directly accesses the memory grid with the bus `MEM_ADDR` to retrieve synaptic strengths. It also has a direct connection to the synaptic trigger signal `SN_TRIG` that is used to inject the synaptic circuits with current values present at their inputs. As such, there needs to be a short delay between accessing the synaptic values, and activating the synaptic circuits.

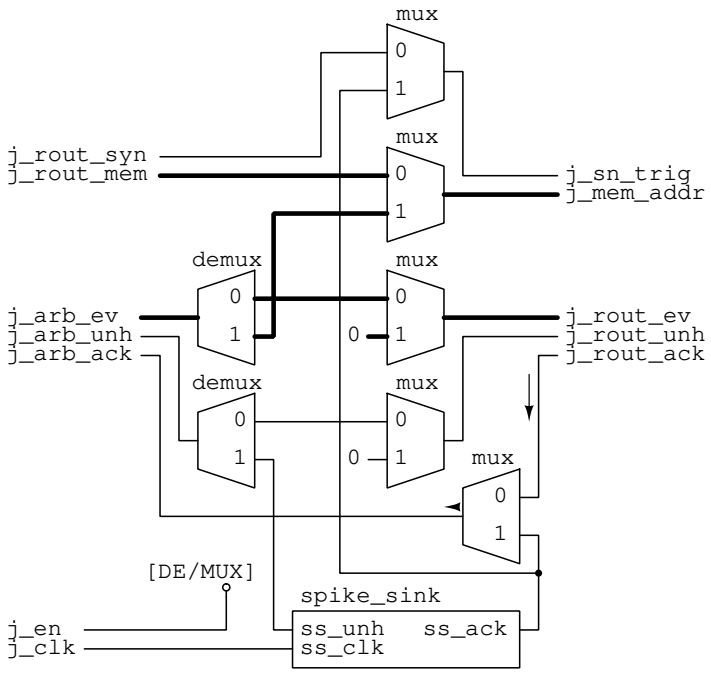


Figure 4.50: Schematic for jumper over local router

The device `SPIKE_SINK` seen at the bottom of the diagram receives an interrupt on the `UNH` pin and sends a positive-edge to the `ACK` pin to acknowledge the event with a delay of one clock cycle. Hence, when the jumper module is enabled, the `EVENT` bus of the arbiter is used to access the synapses in the local memory grid, and the `SPIKE_SINK` device uses its `ACK` signal to trigger these synapses as well as to release the spiked neuron from its latched state.

Chapter 5

Interface

It was stated in Chapter 2 that a neural network has an input layer, one or more hidden layers, and an output layer. The present chapter describes how a neural network fits within the context of a benchmark problem, such as MNIST classification or the control of a plant.

5.1 Input Encoding

As explained in Chapter 2, in the case of Artificial Neural Networks, the input layer does not have any neurons and is instead composed of a column of inputs. Technically speaking, ANNs only have hidden and output layers. The case is different with Spiking Neural Networks since the trans-conductive synapses convert voltage spikes into decaying pulses of current. In other words, an SNN must have an input layer in order to convert a set of network inputs into voltage spike trains. This operation is demonstrated in Figure 5.1 where a sinusoidal input current with a positive offset is transformed into an impulse train with a varying density of spikes.

Note: It is possible to apply a vector of inputs directly into the hidden layer of an SNN. But, this would require a matrix multiplication with the first set of weights, which is incompatible with the general architecture of Spiking Neural Networks.

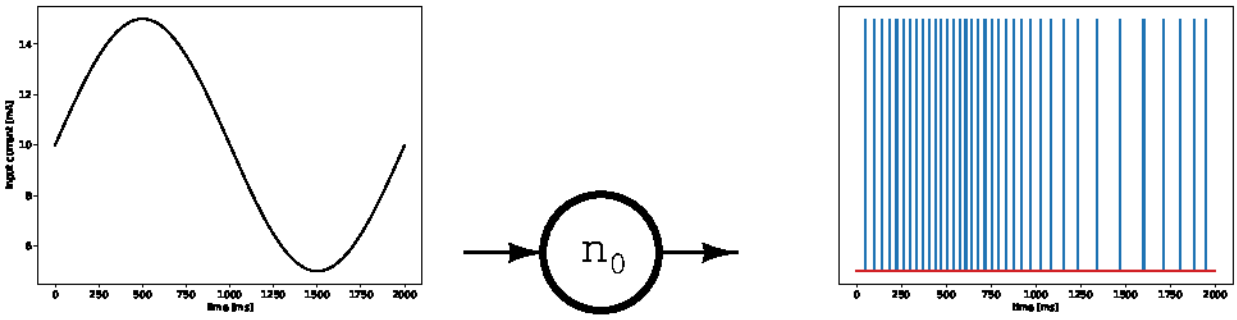


Figure 5.1: Encoding of a signal into a spike train

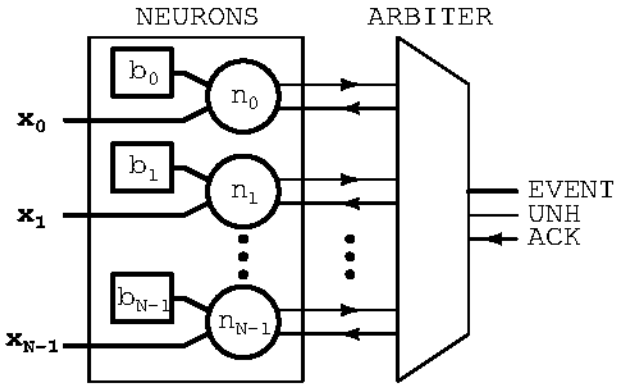


Figure 5.2: Spike generator

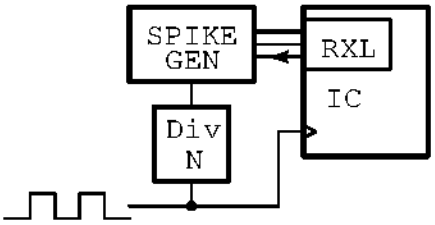


Figure 5.3: SNN IC with input encoded by one spike generator

A spike generator has been designed in order to accommodate an array of encoding neurons. Its architecture, presented in Figure 5.2, is essentially a portion of the complete IC architecture. It is composed of an array of neurons, a memory module for storing biasing currents, and an arbiter. The biasing inputs have a limited resolution due to having a finite number of bits; also, they are constant and do not change unless the memory of the spike generator is re-programmed. The input vectors $\boldsymbol{x_0}$, $\boldsymbol{x_1}$, $\cdots$, $\boldsymbol{x_{N-1}}$ correspond to time-varying current inputs that use floating-point representation. The states of the neurons are updated with every clock cycle. The arbiter funnels all the address events into a single port. Thus, up to four generators can be directly connected to the RX ports of a core.

Rather than having multiple identical ICs, it is preferable to use stand-alone spike generators for the encoding of input signals due to the fact that the input neurons do not accumulate any post-synaptic currents. In addition, the neurons of the spike generator sample external analog signals, which is significantly harder to integrate with the existing architecture of the IC. Lastly, the spike generators can be clocked at a slower rate, thus reducing overall power consumption. However, the programmable time-constants must be adjusted accordingly.

5.2 Output Decoding

The methods employed for decoding the spike patterns of output neurons depends on the context of the problem.

Regarding image classification tasks, the input to the neural network is typically a column of constant values that represent pixel colors and intensities. In such cases, the simplest method for interpretation is an accumulator. This method consists of applying an input to the SNN and allowing it to operate for a certain time frame to allow the neurons to settle at constant spike rates. Then, one may simply pick the neuron that has emitted the greatest number of spikes within that time frame [34]. The index of the selected neuron would be the classification result.

Another possible option is to employ the first-to-spike decoding. Rather than allowing the SNN to operate for a pre-determined time interval, a classification is terminated when a single spike is produced at the output layer [34]. Similarly, the index of the neuron would correspond to the result. However, this kind of methodology would either require large membrane capacitances in the neurons in order to mitigate the transient effects.

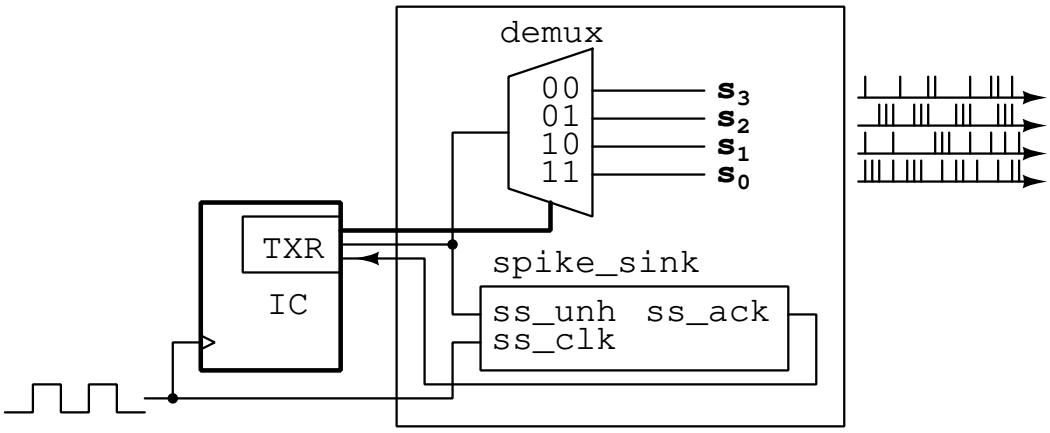


Figure 5.4: Spike decoder for four output neurons

The diagram in Figure 5.4 depicts a spike decoder that would be connected to a transmitting TX port of the IC. The `EVENT` bus is applied to the select lines of a demultiplexer, and the flag `UNH` is transmitted to the pin of the corresponding output neuron. After a delay of one clock cycle, an `ACK` bit is sent to the TX port, allowing the `UNH` status to be reset.

For problems that require time-dependent solutions, such as the generation of control signals, the resulting spike trains can be processed with low-pass filters.

Chapter 6

Benchmarks

This chapter focuses on the benchmarks used to evaluate the performance of Spiking Neural Networks in system-level simulations (Python), as well as in hardware-level simulations (VHDL). One of the main objectives is to observe the effects of synaptic strength quantization and the delays introduced by the routing architecture.

In the Python simulations, simultaneous neuron spikes are detected and delivered to the synapses at the same time. In other words, the SNNs are implemented in such a way that does not require the use of an arbiter or a router. In VHDL, however, simultaneous spikes are encoded by an arbiter and routed to their destinations one at a time, resulting in a distortion of the time-encoded information. This distortion can be mitigated by increasing the clock rate relative to the time constants of the neurons and synapses.

6.1 MNIST Handwritten Digit Classification

The classification of handwritten digits is a standard benchmark used in the field of Machine Learning [35], and is performed on the MNIST dataset, which consists of 60,000 images for training and 10,000 images for testing [36]. Each handwritten digit is centered in an image of 28x28 pixels with an 8-bit grayscale encoding [36]. Since each MNIST case consists of constant inputs to the network, this benchmark is useful in testing the performance of rate-based coding.

In this work, a simple neural network architecture is used to compare the performances of artificial and spiking networks. Both ANN and SNN have [784, 30, 10] neurons in the input, hidden and output layers, respectively. Backpropagation and stochastic gradient descent are

used to train the ANN with the ReLU activation function, and the resulting weight matrices are scaled-up to obtain the set of synaptic strengths for the SNN. The scaling is done in order to compensate for the fast decaying membrane voltages of the neurons. The quantization of the weights is done with the following procedure:

1. All the synaptic values are scaled such that the largest magnitude is made equal to 2^{res-1} (e.g. $W_{MAX} = 128$ if a resolution of $res = 8b$ is used).
2. Next, all the synapses are rounded to the nearest integer.
3. Finally, they are scaled once again in order to make the largest synapse strong enough to trigger a spike in a post-synaptic neuron.

Regarding the hardware-level implementation, let M and N be the number of cores and neurons, respectively. Then, the widths of their addresses are obtained as follows:

$$m = \lceil \log_2 M \rceil \tag{6.1}$$

$$n = \lceil \log_2 N \rceil \tag{6.2}$$

where $\lceil\rceil$ is the ceiling function. Table 6.1 provides an estimate for the required memory size. The variable res represents the number of bits used in the quantization of each synapse.

Table 6.1: Memory Estimate

Module	**Complexity**	**MNIST**
Routing table (LOC)	$5N$	320
Routing table (RX)	$MN(m+n+4)/4$	3584 (14336)
External synapses	$MN(m+n+Nres)$	534528
Local synapses	N^2res	32768
Total		581952
		72744 bytes

The performance of these networks is summarized in Table 6.2. A classification accuracy of 96.42% is relatively poor compared to the results reported in [35]. In part, this is due to the chosen architecture with only 30 hidden neurons, whereas the best performing results in [35] were obtained with Convolutional Neural Networks. In addition, the backpropagation was only performed to train the weight matrices without adjusting the biasing vectors. As mentioned in subsection 2.2.5, this was done in order to avoid the issues related to the combination of constant currents and synaptic pulses.

Note: The results of the last column were obtained for 1000 MNIST cases due to the long simulation time.

Table 6.2: Classification Accuracy [%]

Resolution	**ANN**	**SNN (Python)**	**SNN (VHDL)**
FP	96.42	96.22	–
FP_F	–	96.19	–
FP_B	–	96.12	–
8-bits	96.33	96.28	95.70
6-bits	96.14	95.93	94.50
5-bits	95.29	95.46	94.00
4-bits	84.31	83.35	73.80

In Table 6.2, the row with the resolution FP indicates the network performances with floating-point representation for the synapses. The rows with the labels FP_F and FP_B are used to evaluate the effects of spike de-synchronization that was described in subsection 2.2.5. Typically, each digit classification is done with an SNN that begins from a state of rest. Hence, the input neurons that read the values of pixels with equal grayscale intensities output synchronized spike trains. In the test labelled FP_F, the entire MNIST test iterates from case 0 to case 9999 without resetting the neural and synaptic states between each classification. Hence, the membrane voltages and synaptic currents at the end of one classification are preserved for the start of the classification that follows. In other words, the SNN starts with non-zero initial conditions, which introduce delays into the spike trains of input neurons. The test labelled FP_B is performed in the same manner, but with an iteration from case 9999 back to case 0.

One may see from these results that the SNN is fairly robust against spike de-synchronization. However, it needs to be mentioned that the simulated time of each test-case is extended from $t_F = 50ms$ to $t_F = 80ms$. This is done to allow the low-pass filtering of the synapses to mitigate these effects.

Table 6.2 shows that there is a drastic drop in classification accuracy as the synaptic resolution is lowered from 5 bits to 4 bits. Figures 6.1 and 6.2 show the unique synaptic values that are spread over an almost linear range, as well as the binning of these values for the resolutions of 8 bits, 5 bits and 4 bits. One of the reasons for the drastic drop in performance is the large number of synaptic strengths that get rounded down to zero. In the first and second subplots, there are approximately 2000 and 3000 repeated synapses in the smallest non-zero bins. However, in the third subplot, this number drops by more than half. This

happens because the thresholds for rounding-down are further away from the zero, which causes most synapses to be truncated.

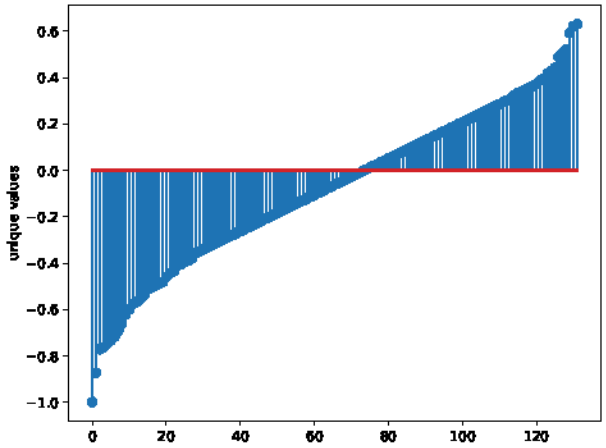


Figure 6.1: Unique synaptic values with 8-bit resolution

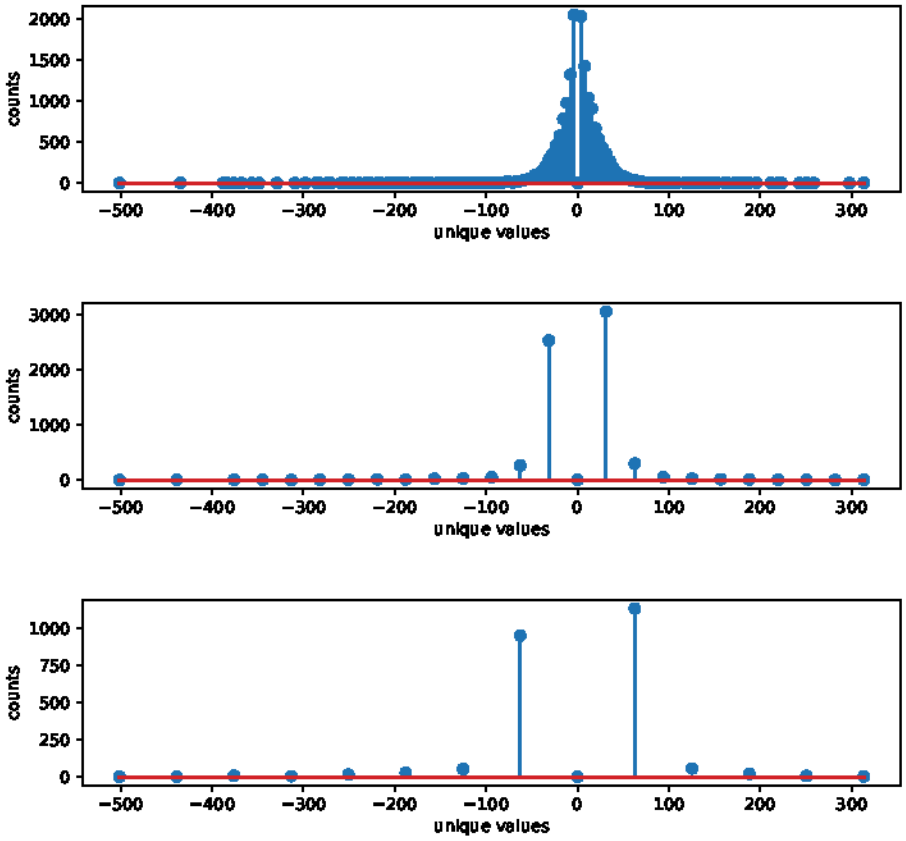


Figure 6.2: Histograms for synaptic values with resolutions of 8 bits, 5 bits and 4 bits

Note: The bin corresponding to a strength of zero has been omitted from these plots to prevent it from reducing the scale of the vertical axes.

Figures 6.3 and 6.4 demonstrate the raster plots and the spiking profiles of the system-level (Python) and hardware-level (VHDL) simulations. Note that the spiking rates on the right side have only been shown for the neurons of the hidden and output layers. One may see that the routing architecture introduces differences in the spiking rates of some neurons, but the overall network behaviors are very similar.

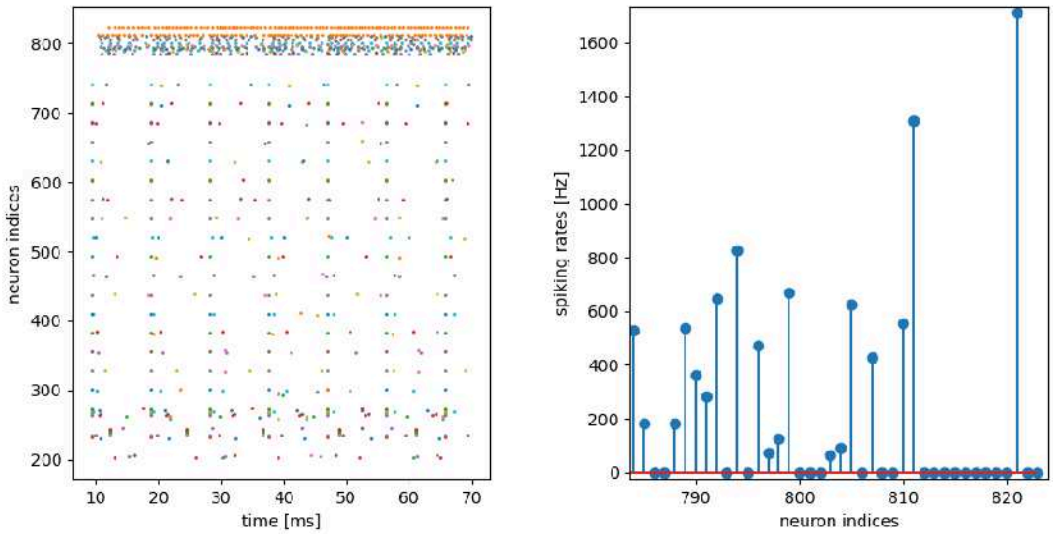


Figure 6.3: Sample MNIST test case (Python)

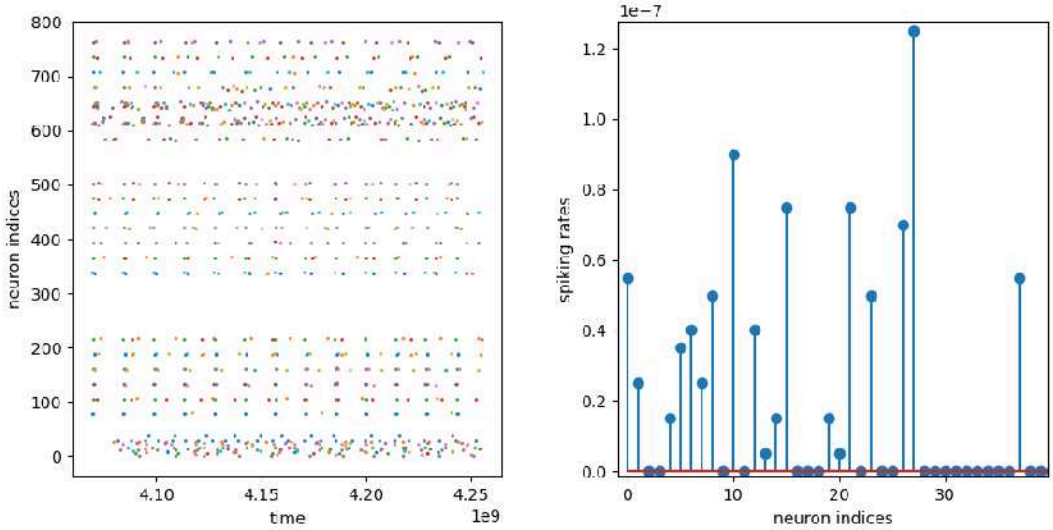


Figure 6.4: Sample MNIST test case (VHDL)

6.1.1 Impact of Parametric Variation

A fabricated integrated circuit is expected to have parametric variations, and the membrane resistances and capacitances of the neurons will vary from one another. The effects of these variations on the accuracy of the MNIST classification are observed and summarized in Table 6.3. Each entry corresponds to a complete MNIST evaluation with an 8-bit synaptic quantization. The value at the top-left is obtained from Table 6.2 and serves as a point of reference.

As an example, each neuron would have a membrane resistance of $R_m = 10\ \Omega$. The percentages in the left column (and top row) represent one standard deviation away from the desired value. At the start of a classification, each resistance is set to $R_i = 10 + \Delta R$ where ΔR is a random value obtained by sampling from a Gaussian distribution with a mean $\mu = 0$ and a standard deviation of $\sigma = 0.5 = 0.05 \times 10$. Note that the parameters are randomly selected for each handwritten digit.

These simulations show that, even with variances in both membrane resistance and capacitance, the classification accuracy does not degrade by more than 2%.

Table 6.3: Accuracy with neural parametric variation

Res \ *Cap*	0 %	5 %	10 %	15 %
0 %	96.28	96.08	95.79	94.80
5 %	96.23	96.02	–	–
10 %	96.19	–	95.53	–
15 %	96.18	–	–	94.97

6.1.2 Modeling DAC Variation

The retrieval of a synaptic value from memory needs to be followed by a Digital-to-Analog Conversion (DAC) into a current value. One possible implementation is the binary weighted current source [37]. The final design of the integrated circuit requires the use of bipolar current sources that employ both NMOS and PMOS devices. However, for the sake of simplicity, the issues related to parametric variation and mismatch are illustrated with a unipolar DAC.

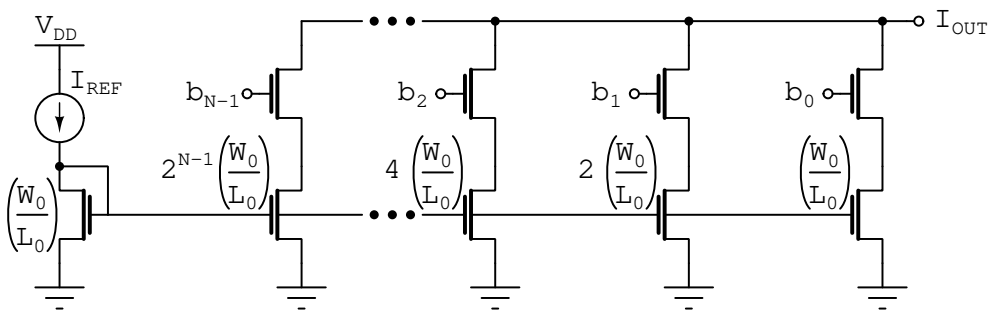


Figure 6.5: Programmable current source

As shown in Figure 6.5, this current source is composed of a series of binary weighted transistors. The right-most transistor corresponds to the least-significant bit, and has the same geometry as the diode-connected transistor. Thus, when $b_0 = 1$, these devices conduct the same current I_{REF}. Going from right to left, the aspect ratio of each NMOS device doubles. The transistors shown at the top are used as digital switches. Ideally, the threshold voltages of all the devices are equal, and an N-bit binary code produces the following current output:

$$I_{OUT} = I_{REF} \sum_{k=0}^{N-1} 2^k b_k \tag{6.3}$$

The CMOS process variations lead to a variance in the threshold voltage, which is inversely proportional to the area of the device [38]. As a consequence, the NMOS devices no longer conduct currents that are powers of 2.

If the length is kept constant across all devices in the DAC, the variances of the threshold voltage and aspect ratio are proportional to the inverse of their widths [38], [39]:

$$\sigma^2(V_{t0}) \propto \frac{1}{W} \tag{6.4}$$

$$\frac{\sigma^2(K')}{(K')^2} \propto \frac{1}{W} \tag{6.5}$$

where $K' = \mu_n C_{OX}(W/L)$. As such, the non-ideal transfer characteristic of a DAC can be expressed with the relation:

$$I_{OUT} = I_{REF} \sum_{k=0}^{N-1} (2^k + \nu_k) b_k \tag{6.6}$$

where ν_k is an indicator of the current coefficient mismatch. Figure 6.6 indicates the standard deviations or each bit coefficient (i.e. 1, 2, 4, etc.) of the DAC, going from the least significant to the most significant bit. Although the relative transistor matching improves with an increase in width, the overall coefficient mismatch still increases due to the fact that larger transistors conduct greater currents.

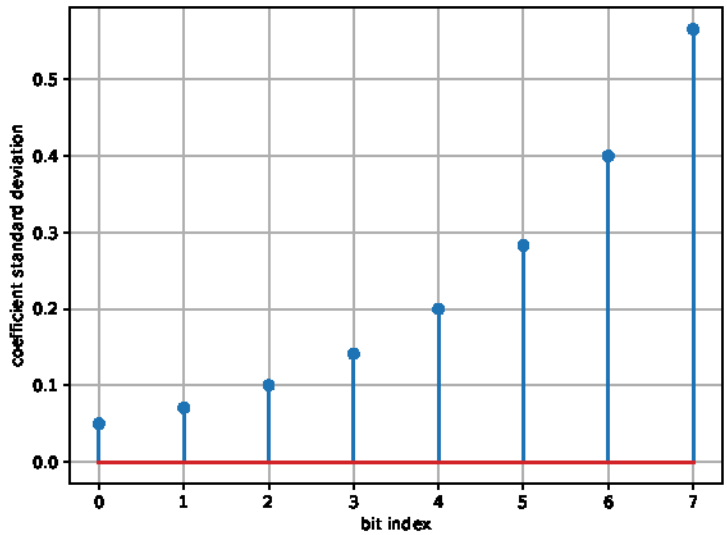
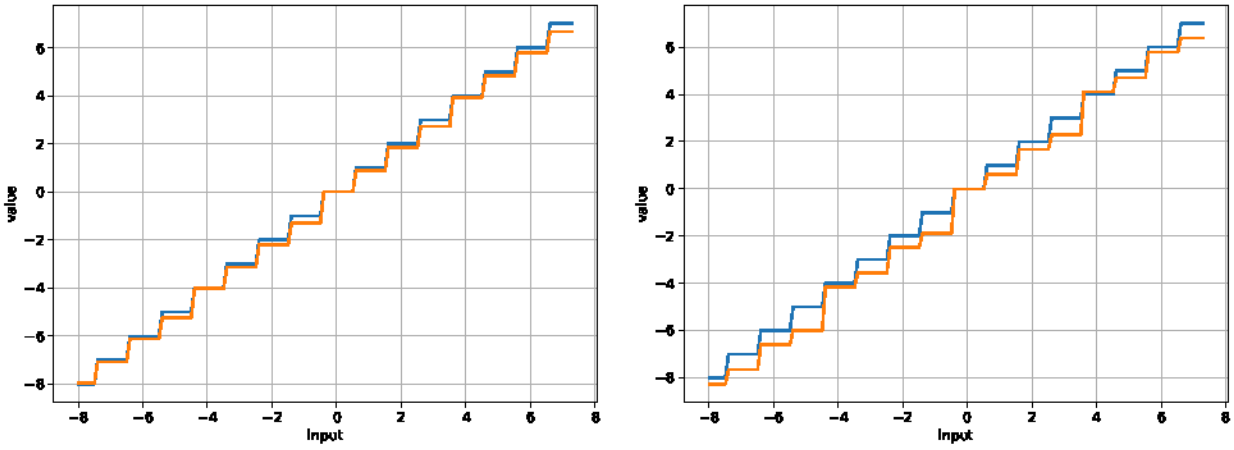


Figure 6.6: Standard deviations for mismatch of each bit

In this subsection, the MNIST classification accuracy of the SNN is performed with non-ideal DACs. As described in Chapter 3, the designed architecture employs one synapse per neuron. Hence, prior to every MNIST classification, a set of 40 non-ideal DACs (30 hidden and 10 output neurons) is generated. In every DAC, the weight of each bit is selected as $A_k = 2^k + \nu_k$ where 2^k is derived from the bit position, and ν_k is sampled from a Gaussian distribution with a mean $\mu = 0$ and $\sigma = C_k$. The variable C_k is obtained from Figure 6.6. In this graph, the standard deviation of the LSB relative error is $\epsilon_0 = (0.05/2^0) \times 100 = 5\%$, and the MSB relative error is $\epsilon_7 = (0.57/2^7) \times 100 \approx 0.45\%$.

Figures 6.7a and 6.7b show two possible transfer characteristics for the non-ideal DACs obtained using this method. The blue curves are plotted to show the ideal transfer characteristic. Chapter 4.2 explains that, although the synaptic weights are obtained from a memory module that stores quantized values, the mathematical model of the synapses still uses floating-point representation. Hence, it is only the initial value of a given decaying exponential that is fed through a nonlinear DAC.



(a) (b)

Figure 6.7: Sample DAC transfer characteristics (4-bit)

Figure 6.6 only shows one set of mismatch values. This test has been repeated several times for varying severities in the process variation. The results are summarized in Table 6.4 below. One may see that the accuracy of the MNIST classification is far more sensitive to parametric mismatch seen in the synaptic DACs than in the variations of neural parameters.

Table 6.4: Accuracy with mismatch in DAC current sources

LSB error (%)	**MSB Error (%)**	**Accuracy**
0	0	96.28
1.0	0.09	96.18
2.0	0.18	95.87
3.0	0.27	95.15
4.0	0.35	94.39
5.0	0.44	90.33

6.2 Cart-Pole Balancing

The balancing of the cart-pole (inverted pendulum) is a standard benchmark problem used to evaluate and compare methods of Control Theory and Reinforcement Learning [40], [41]. As shown in Figure 6.8, this system is composed of a cart with mass m_1, and a rod of mass m_2 and length $2l$. The state-vector consists of four variables: the position and velocity of the cart, and the angle and angular velocity of the pole.

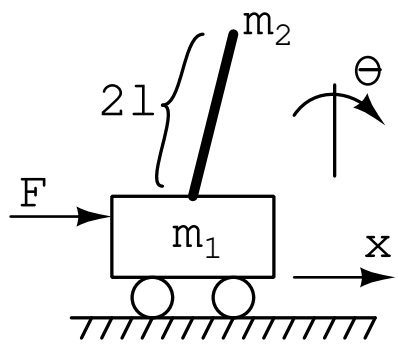


Figure 6.8: Cart-pole system [40], [42]

Assuming a frictionless system, the dynamics of the cart-pole are governed by the following equations, where θ is the angle of the pole with respect to the equilibrium, and x is the position of the cart [43]:

$$\ddot{\theta} = \frac{g \sin\theta + \cos\theta \left(\dfrac{-F - m_2 l \dot{\theta}^2 \sin\theta}{m_1 + m_2} \right)}{l \left(\dfrac{4}{3} - \dfrac{m_2 \cos^2\theta}{m_1 + m_2} \right)} \tag{6.7}$$

$$\ddot{x} = \frac{F + m_2 l (\dot{\theta}^2 \sin\theta - \ddot{\theta} \cos\theta)}{m_1 + m_2} \tag{6.8}$$

The cart-pole is in an unstable (upward) equilibrium when $\theta = 0°$, $\dot{\theta} = 0°/s$, $\dot{x} = 0\ m/s$. While in a state of equilibrium, the cart can be at any position x. Thus, this is categorized as a line equilibrium.

In this benchmark, the cart-pole starts with an initial angle of $\theta = 10°$, and the objective of a controller is to bring the system back to the origin while ensuring that the position of the cart does not exceed a pre-determined range. The only input to the system is the horizontal force F applied to the cart.

Figure 6.9 presents the setup for the simulation experiment. The SNN has an input layer that reads the state-vector of the cart-pole system, and an output layer consisting of two neurons that produce spike trains with time-varying frequencies. These two spike trains are fed through low-pass filters (4th-order Butterworth) and applied to a comparator. The latter produces a bang-bang voltage signal $\pm V_{DD}$, which is then linearly converted to a mechanical force of an actuator that drives the cart.

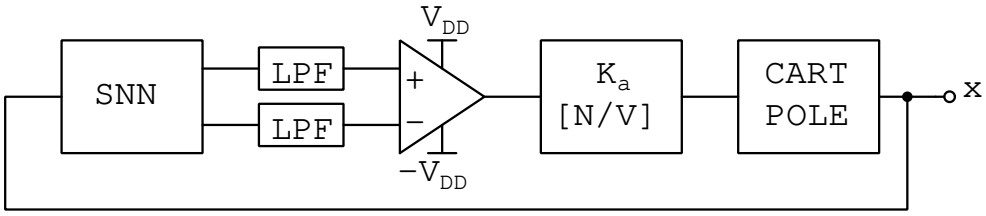


Figure 6.9: Simulation setup [42]

The SNN shown in Figure 6.10 has been manually designed in a way that ensures symmetric operation of the controller. Before being applied to the network, the state-vector x is scaled by a constant coefficient A_0. The product A_0x is applied to the top four neurons of the input layer n_{0-3}, and the product $-A_0x$ is applied to the bottom portion of the input layer n_{4-7}. As an example, the neurons n_2 and n_6 both read the pole angle, and have inputs of equal magnitudes but opposite polarities. Thus, when $\theta > 0°$, the neuron n_2 is active and n_4 is quiet. The opposite is true when $\theta < 0°$.

The four synapses that interconnect the neurons n_{0-3} to n_8 are the same as the synapses that interconnect the neurons n_{4-7} to n_9, and can be expressed with a vector K. Additionally, two inhibitive synapses have been added in the output layer in order to implement the Winner-Takes-All approach [44].

Using an ideal mathematical model of the SNN, the performance of the controller is demonstrated below in Figure 6.11. It takes approximately five seconds for the pole to reach equilibrium, and ten seconds for the cart to be pushed back to origin. The nature of the bang-bang force generation with the use of a comparator introduces much noise into the system.

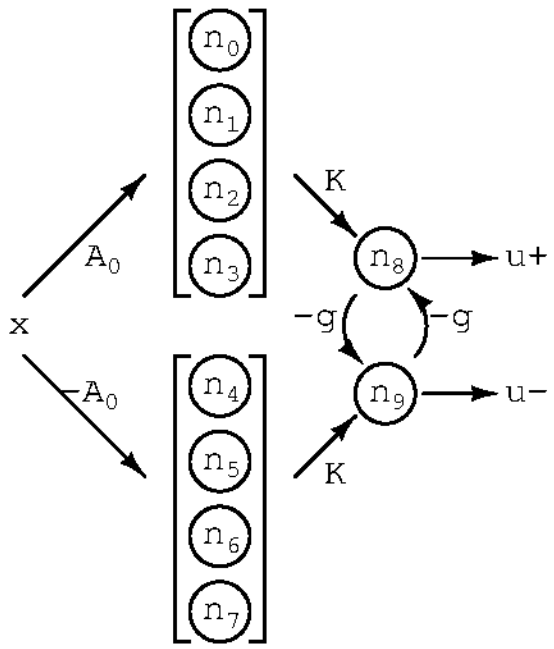


Figure 6.10: SNN for cart-pole control [42]

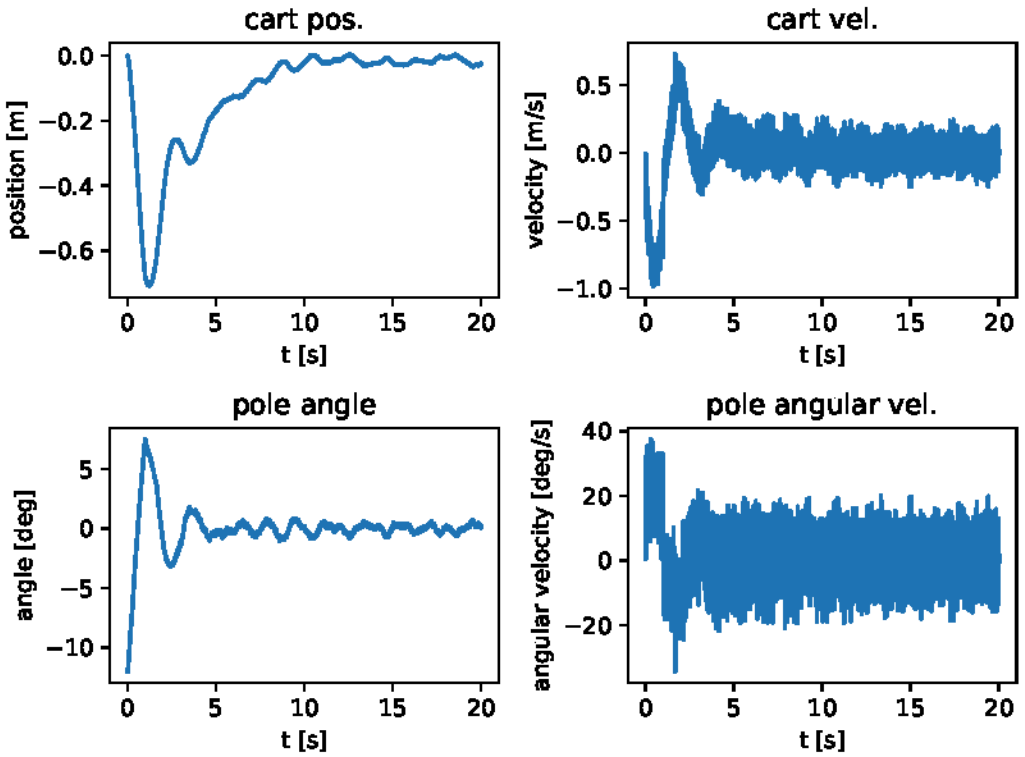


Figure 6.11: Cart-pole balancing simulation (Python) [42]

Figures 6.12 and 6.13 demonstrate the operation of the SNN simulated at the hardware level. From top to bottom, the graphs show the position and velocity of the cart, and the angular position and velocity of the pole. Given the small number of neurons employed in this network, the integrated circuit is clocked with a slow frequency of $f_{CLK1} = 4\ kHz$. The cart-pole and the spike generator with input layer neurons are both clocked with $f_{CLK2} = 1\ kHz$.

This simulation is repeated twice: with and without the local router bridge introduced in section 3.3. In the first simulation, a local event is received and re-distributed in only one clock cycle, and the controller brings the cart back to origin at time $t < 10\ s$. In the second simulation, however, three clock cycles are needed, and the performance of the controller is much worse due to the delays of the local router. Thus, the use of the bridge makes for a significant improvement for problems that require a relatively fast response.

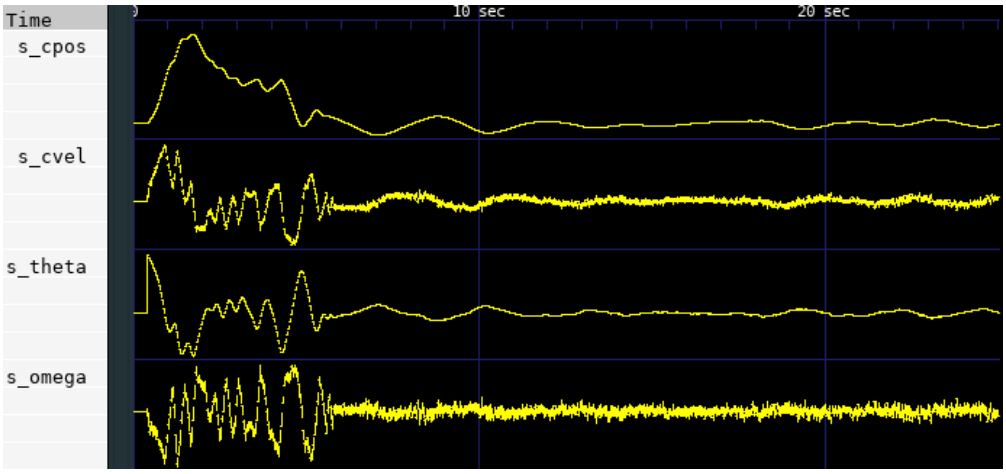


Figure 6.12: Cart-pole balancing with local router bridge (VHDL)

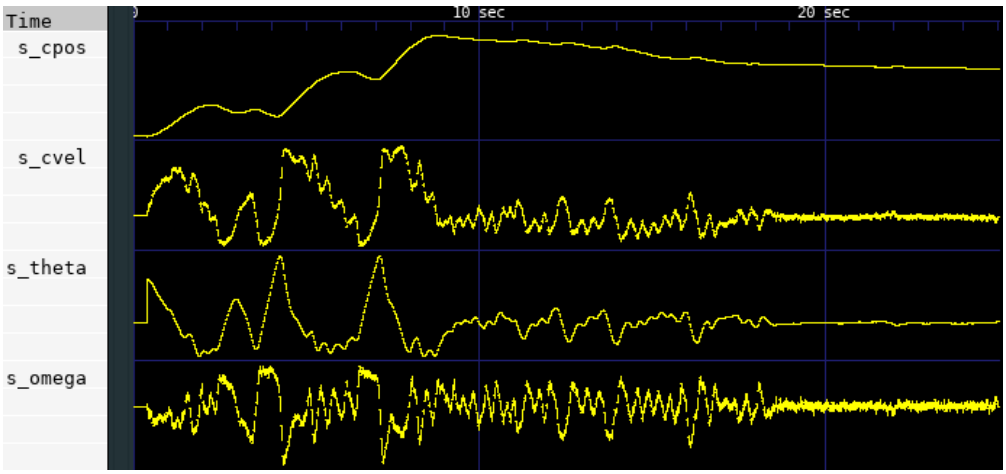


Figure 6.13: Cart-pole balancing without local router bridge (VHDL)

Chapter 7

Conclusion

Spiking Neural Networks (SNNs) are a new generation of neural networks with greater biological plausibility. When translated to hardware implementations, this fact may lead to significant reductions in power consumption and improve scalability due to the high parallelism of such network topologies. The focus of this work was to present an architecture for a hardware platform that can be used to simulate the behavior of SNNs, to study the impact of various design variables on the performance of SNNs, and to demonstrate the robustness of such networks at accomplishing benchmark tasks.

Chapter 2 compared the basic functionality of artificial and spiking neural networks, and showed some key differences in their behaviors. And, section 2.3 made a brief summary of existing ICs.

Chapter 3 provided a high-level overview of the proposed SNN core architecture. Moreover, this chapter explained the functionality of a circuit that allows spiking events to bypass the local router with the goal of reducing the number of clock cycles required to handle an event.

Chapter 4 described in detail the design and functionality of each module composing the SNN core.

Chapter 5 gave an overview of spike generators (or input encoders) and spike train converters that establish an interface between the SNN and the systems/circuits that are external to the IC.

Finally, Chapter 6 demonstrated the performance of the system-level and hardware-level spiking neural networks. The system-level simulations, based on mathematical models of the neurons and synapses, were first used to study the effect of synaptic strength quantization. The MNIST classification accuracy was tested with strength resolutions of 8 bits, 6 bits, 5

bits and 4 bits. The performance degraded slowly from 8 bits to 5 bits, and had an abrupt drop in accuracy at 4 bits.

The effect of spike de-synchronization was also observed on the accuracy, and it was noted that, for MNIST, the accuracy of the SNN barely decreased. This implies that a rate-based spike-train decoding scheme is sufficient for this application.

Next, the impact of variation in neural and synaptic parameters was evaluated. From a set of multiple MNIST tests, it was concluded that the SNNs are very robust against variations in the neural parameters (membrane resistance and capacitance). But, the variations of the synaptic strengths had a much greater effect, and lead to a fast degradation of classification accuracy.

Finally, at the hardware-level, the cart-pole (inverted pendulum) stabilization problem was used to compare the performance of the SNN with and without a router bypassing circuit. Given that the SNN used in this benchmark has a small number of neurons that fit on a single core, the delivery of neural spikes did not require the use of a router. Thus, each spike required fewer clock cycles to be propagated. This led to a better stabilization of the cart-pole system.

7.1 Future Work

The mathematical model simulations as well as the hardware-level design serves as a foundation and a platform for conducting further research in the area of Spiking Neural Networks. In order to fabricate a working IC, the following course of action needs to be taken:

1. In the current design, when a neuron awaits an acknowledge signal, it stops the accumulation of input currents. One possible design extension could allow the neurons to resume their operation without requiring the arbiter's permission.
2. Modify the design of the local router bypass such that some address events can be propagated through the bridge, and others could be broadcast through TX routers.
3. Add a clock divider that allows neurons and synapses to be clocked at a different rate than the routers.
4. Compare existing circuits for neurons and synapses, propose potential improvements, and incorporate them within the complete architecture of the core.
5. Implement calibration circuitry to improve the robustness against process variations.

6. Complete the physical layout of the integrated circuit.

In addition, significant research contributions may be accomplished by investigating the topics presented below:

1. Implement spike-timing dependent plasticity (STDP), and study its ability to optimize an existing SNN.
2. Explore learning algorithms for SNNs.
3. Study neural models with greater biological plausibility, such as the Izhikevich model [16], and compare them to the LIF model.
4. Investigate alternative routing schemes and circuit topologies (e.g. asynchronous, synchronous, tree topology, etc.). Review architectures that transmit multiple spikes in a single packet.
5. Investigate dynamic element matching for digital-to-analog conversion of the synaptic circuits to improve the robustness of SNNs.

The first three objectives may be key to utilizing SNNs to their full potential.

Bibliography

[1] J. Heaton, *Applications of deep neural networks*, 2021. arXiv: 2009.05673 [cs.LG].

[2] K. Hunt, D. Sbarbaro, R. Żbikowski, and P. Gawthrop, "Neural networks for control systems - a survey," *Automatica*, vol. 28, no. 6, pp. 1083–1112, 1992, ISSN: 0005-1098. DOI: https://doi.org/10.1016/0005-1098(92)90053-I. [Online]. Available: http://www.sciencedirect.com/science/article/pii/000510989290053I.

[3] A. B. Nassif, I. Shahin, I. Attili, M. Azzeh, and K. Shaalan, "Speech recognition using deep neural networks: A systematic review," *IEEE Access*, vol. 7, pp. 19 143–19 165, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2896880.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[5] D. A. Vaccari and E. Wojciechowski, "Neural networks as function approximators: Teaching a neural network to multiply," in *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, vol. 4, 1994, 2217–2222 vol.4. DOI: 10.1109/ICNN.1994.374561.

[6] W. Maass, "Networks of spiking neurons: The third generation of neural network models," *Neural Networks*, vol. 10, no. 9, pp. 1659–1671, 1997, ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(97)00011-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608097000117.

[7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," in *Neurocomputing: Foundations of Research*. Cambridge, MA, USA: MIT Press, 1988, pp. 696–699, ISBN: 0262010976.

[8] J. Lazzaro and J. Wawrzynek, "A multi-sender asynchronous extension to the aer protocol," in *Proceedings Sixteenth Conference on Advanced Research in VLSI*, 1995, pp. 158–169. DOI: 10.1109/ARVLSI.1995.515618.

[9] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, *Activation functions: Comparison of trends in practice and research for deep learning*, 2018. arXiv: 1811.03378 [cs.LG].

[10] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, *Efficient processing of deep neural networks: A tutorial and survey*, 2017. arXiv: 1703.09039 [cs.CV].

[11] W. Gerstner, "What is different with spiking neurons?" In *Plausible Neural Networks for Biological Modelling*, H. A. K. Mastebroek and J. E. Vos, Eds. Dordrecht: Springer Netherlands, 2001, pp. 23–48, ISBN: 978-94-010-0674-3. DOI: 10.1007/978-94-010-0674-3_2. [Online]. Available: https://doi.org/10.1007/978-94-010-0674-3_2.

[12] J. Pospíchal and V. Kvasnička, "70th anniversary of publication: Warren mcculloch & walter pitts - a logical calculus of the ideas immanent in nervous activity," in *Emergent Trends in Robotics and Intelligent Systems*, P. Sinčák, P. Hartono, M. Virčíková, J. Vaščák, and R. Jakša, Eds., Cham: Springer International Publishing, 2015, pp. 1–10, ISBN: 978-3-319-10783-7.

[13] M. T. Hagan and H. B. Demuth, "Neural networks for control," in *Proceedings of the 1999 American Control Conference (Cat. No. 99CH36251)*, vol. 3, 1999, 1642–1656 vol.3. DOI: 10.1109/ACC.1999.786109.

[14] Xin Jin, S. B. Furber, and J. V. Woods, "Efficient modelling of spiking neural networks on a scalable chip multiprocessor," in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 2008, pp. 2812–2819. DOI: 10.1109/IJCNN.2008.4634194.

[15] A. Gilra and W. Gerstner, *Non-linear motor control by local learning in spiking neural networks*, 2017. arXiv: 1712.10158 [q-bio.NC].

[16] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on Neural Networks*, vol. 14, no. 6, pp. 1569–1572, 2003. DOI: 10.1109/TNN.2003.820440.

[17] Dhanya E, N. Pradhan, Sunitha R, and A. Sreedevi, "Analysis of the dynamic behaviour of a single hodgkin-huxley neuron model," in *2015 International Conference on Emerging Research in Electronics, Computer Science and Technology (ICERECT)*, 2015, pp. 441–446. DOI: 10.1109/ERECT.2015.7499056.

[18] L. Abbott, "Lapicque's introduction of the integrate-and-fire model neuron (1907)," *Brain Research Bulletin*, vol. 50, no. 5, pp. 303–304, 1999, ISSN: 0361-9230. DOI: https://doi.org/10.1016/S0361-9230(99)00161-6. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0361923099001616.

[19] W. Wang, S. Zhou, J. Li, X. Li, J. Yuan, and Z. Jin, *Temporal pulses driven spiking neural network for fast object recognition in autonomous driving*, 2020. arXiv: 2001.09220 [cs.CV].

[20] N. Rotem, E. Sestieri, J. Hounsgaard, and Y. Yarom, "Excitatory and inhibitory synaptic mechanisms at the first stage of integration in the electroreception system of the shark," *Frontiers in Cellular Neuroscience*, vol. 8, p. 72, 2014, ISSN: 1662-5102. DOI: 10.3389/fncel.2014.00072. [Online]. Available: https://www.frontiersin.org/article/10.3389/fncel.2014.00072.

[21] H. Markram, W. Gerstner, and P. J. Sjöström, "Spike-timing-dependent plasticity: A comprehensive overview," *Frontiers in Synaptic Neuroscience*, vol. 4, p. 2, 2012, ISSN: 1663-3563. DOI: 10.3389/fnsyn.2012.00002. [Online]. Available: https://www.frontiersin.org/article/10.3389/fnsyn.2012.00002.

[22] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, "Deep learning in spiking neural networks," *Neural Networks*, vol. 111, pp. 47–63, Mar. 2019, ISSN: 0893-6080. DOI: 10.1016/j.neunet.2018.12.002. [Online]. Available: http://dx.doi.org/10.1016/j.neunet.2018.12.002.

[23] S. A. Aamir, Y. Stradmann, P. Muller, C. Pehle, A. Hartel, A. Grubl, J. Schemmel, and K. Meier, "An accelerated lif neuronal network array for a large-scale mixed-signal neuromorphic architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 12, pp. 4299–4312, Dec. 2018, ISSN: 1558-0806. DOI: 10.1109/tcsi.2018.2840718. [Online]. Available: http://dx.doi.org/10.1109/TCSI.2018.2840718.

[24] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri, "A scalable multicore architecture with heterogeneous memory structures for dynamic neuromorphic asynchronous processors (dynaps)," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 12, no. 1, pp. 106–122, 2018. DOI: 10.1109/TBCAS.2017.2759700.

[25] F. C. Bauer, D. R. Muir, and G. Indiveri, "Real-time ultra-low power ecg anomaly detection using an event-driven neuromorphic processor," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 6, pp. 1575–1582, 2019. DOI: 10.1109/TBCAS.2019.2953001.

[26] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown, "Overview of the spinnaker system architecture," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2454–2467, 2013. DOI: 10.1109/TC.2012.142.

[27] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The spinnaker project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, May 2014, ISSN: 1558-2256. DOI: 10.1109/JPROC.2014.2304638.

[28] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, 2015. DOI: 10.1109/TCAD.2015.2474396.

[29] M. Davies, N. Srinivasa, T. .-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. .-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. .-H. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018. DOI: 10.1109/MM.2018.112130359.

[30] S. Schmitt, J. Klähn, G. Bellec, A. Grübl, M. Güttler, A. Hartel, S. Hartmann, D. Husmann, K. Husmann, S. Jeltsch, V. Karasenko, M. Kleider, C. Koke, A. Kononov, C. Mauch, E. Müller, P. Müller, J. Partzsch, M. A. Petrovici, S. Schiefer, S. Scholze, V. Thanasoulis, B. Vogginger, R. Legenstein, W. Maass, C. Mayr, R. Schüffny, J. Schemmel, and K. Meier, "Neuromorphic hardware in the loop: Training a deep spiking network on the brainscales wafer-scale system," in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 2227–2234. DOI: 10.1109/IJCNN.2017.7966125.

[31] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, Mar. 2006.

[32] J. Wei, J. Zhang, X. Zhang, Z. Wu, C. Dou, T. Shi, H. Chen, and Q. Liu, "An asynchronous aer circuits with rotation priority tree arbiter for neuromorphic hardware with analog neuron," in *2019 IEEE 13th International Conference on ASIC (ASICON)*, 2019, pp. 1–4. DOI: 10.1109/ASICON47005.2019.8983508.

[33] "Communication," in *Event-Based Neuromorphic Systems*. John Wiley & Sons, Ltd, 2015, ch. 2, pp. 7–36, ISBN: 9781118927601. DOI: https://doi.org/10.1002/9781118927601.ch2. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118927601.ch2. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118927601.ch2.

[34] A. Grüning and S. Bohté, "Spiking neural networks: Principles and challenges," in *ESANN*, 2014.

[35] A. Baldominos, Y. Saez, and P. Isasi, "A survey of handwritten character recognition with mnist and emnist," *Applied Sciences*, vol. 9, no. 15, 2019, ISSN: 2076-3417. DOI: 10.3390/app9153169. [Online]. Available: https://www.mdpi.com/2076-3417/9/15/3169.

[36] Y. LeCun, C. Cortes, and C. J. C. Burges, "The MNIST database of handwritten digits," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[37] S. Hanfoug, N.-E. Bouguechal, and B. Samir, "Behavioral non-ideal model of 8-bit current-mode successive approximation registers adc by using simulink," *International Journal of u- and e- Service, Science and Technology*, vol. 7, pp. 85–102, Apr. 2014. DOI: 10.14257/ijunesst.2014.7.3.09.

[38] K. W. M. T. C. Carusone D. A. Johns, *Analog Integrated Circuit Design*, 2nd ed. Hoboken, NJ, USA: Wiley, 2011, p. 100.

[39] P. R. Kinget, "Device mismatch and tradeoffs in the design of analog circuits," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 6, pp. 1212–1224, 2005. DOI: 10.1109/JSSC.2005.848021.

[40] O. Boubaker, "The inverted pendulum: A fundamental benchmark in control theory and robotics," *International Conference on Education and e-Learning Innovations*, Jul. 2012. DOI: 10.1109/iceeli.2012.6360606. [Online]. Available: http://dx.doi.org/10.1109/ICEELI.2012.6360606.

[41] S. Nagendra, N. Podila, R. Ugarakhod, and K. George, "Comparison of reinforcement learning algorithms applied to the cart-pole problem," *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, Sep. 2017. DOI: 10.1109/icacci.2017.8125811. [Online]. Available: http://dx.doi.org/10.1109/ICACCI.2017.8125811.

[42] A. Syutkin, "Balancing of an Inverted Pendulum with a Spiking Neural Network - Case Study and Report". Montreal, QC, Canada: Concordia University, Dec. 2020.

[43] R. Florian, "Correct equations for the dynamics of the cart-pole system," Aug. 2005.

[44] N. Lynch, C. Musco, and M. Parter, *Winner-take-all computation in spiking neural networks*, 2019. arXiv: `1904.12591 [cs.DC]`.

Appendices

Appendix A

Python Source Code

A.0.1 Spiking Neural Network

snn.py

```
#!/bin/python

import matplotlib.pyplot as plt
import numpy as np

from neurons  import Neurons
from synapses import Synapses
# from ma_filter    import MA_Filter

# network of spiking neurons
class SNN:
  def __init__(self, num, tf, ts=1, model='izhikevich'):
    self.ts = ts                    # time step [ms]
    self.tf = tf                    # duration of simulation [ms]
    self.t = np.arange(0, tf, ts)   # time array
    self.tn = len(self.t)           # number of time samples

    self.num = num                        # number of neurons
    self.nrns = Neurons(num, ts, model)   # array of neurons
    self.syns = Synapses(num, ts)         # grid of synapses
    self.raster = np.zeros((num, self.tn), dtype = np.int8)  # allocated grid for raster plot


```

```
  # advance simulation by one time-step
  def step_forward(self, t_idx, curr):
    self.raster[:, [t_idx]] = self.nrns.step_forward(curr + \
                                                     np.sum(self.syns.i,
                                                     ↪  axis=0).reshape((self.num, 1)))
    self.syns.step_forward( self.raster[:, t_idx] )
    return self.nrns.spikes


  # reset state of network
  def reset(self):
    self.nrns.reset()
    self.syns.reset()
    self.raster.fill(0)


  # compute spike rates
  def rates(self):
    self.spike_rates = np.zeros((self.num, 1))
    for k in range(self.num):
      spike_idxs = np.where(self.raster[k])[0]
      if (spike_idxs.size == 0) or (spike_idxs.size == 1):
        self.spike_rates[k] = 0
        continue

      deltas = np.diff(self.t[ spike_idxs ])
      self.spike_rates[k] = (np.average(deltas) / 1000)**(-1)
    return self.spike_rates


  # process raster matrix before plotting
  def prep_raster(self):
    self.raster = self.raster.astype(float)
    for t_idx in range(self.tn):
      fired_idx = np.where(self.raster[:, [t_idx]] == 1)[0]
      self.raster[fired_idx, [t_idx]] = fired_idx + 1
    self.raster[self.raster == 0] = np.nan


  # plot raster diagram
  def plot_raster(self):
    self.rates()
    self.prep_raster()
```

```

    plt.subplot(121)
    plt.plot(self.t, self.raster.transpose(), marker='.', markersize=2)
    plt.title('raster')
    plt.xlabel('time [ms]')
    plt.ylabel('neuron index')

    plt.subplot(122)
    plt.plot(self.spike_rates)
    plt.title('spiking rates')
    plt.xlabel('neuron indices')
    plt.ylabel('rates')

    plt.tight_layout()
    plt.show()


  # save raster
  def save_raster(self, filename):
    file_raster = open(filename, 'w')

    time = self.t.transpose()
    for k in range(len( time )):
      file_raster.write(str( time[k] ) + ',')
    file_raster.write('\n')

    size = self.raster.shape
    for k in range(size[0]):
      for l in range(size[1]):
        if (l == size[1] - 1):
          file_raster.write(str( self.raster[k, l] ))
        else:
          file_raster.write(str( self.raster[k, l] ) + ',')
      file_raster.write('\n')
    file_raster.close()
```

neurons.py

```python
#!/bin/python

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

# class for network neurons
class Neurons:

  ##########################################################################################
  # initialize Izhikevich neurons
  def init_neurons_izhikevich(self, num):
    # regular spiking model
    self.a = 0.02  * np.ones((num, 1))
    self.b = 0.20  * np.ones((num, 1))
    self.c = -65.0 * np.ones((num, 1))
    self.d = 8.0   * np.ones((num, 1))

    # initialize state variables
    self.v = self.c * np.ones((num, 1))  # membrane voltage; USE COPY FUNCTION
    self.u = self.b * self.v             # recovery variable


  # update states
  def step_forward_izhikevich(self, curr):
    self.fired_idx = np.where(self.v >= 30)[0]            # identify indices of neurons that have spiked
    self.v[ self.fired_idx ]  = self.c[ self.fired_idx ]  # reset membrane voltages
    self.u[ self.fired_idx ] += self.d[ self.fired_idx ]  # update recovery variable

    # update neurons' states using Euler method
    # membrane voltage is updated twice with a half-step (as done in simple_model paper by Izh.)
    self.v += 0.5*self.ts * (0.04 * (self.v)**2 + 5 * self.v + 140 - self.u + self.biases + curr)
    self.v += 0.5*self.ts * (0.04 * (self.v)**2 + 5 * self.v + 140 - self.u + self.biases + curr)
    self.u += self.ts * (self.a * (self.b * self.v - self.u))

    self.spikes.fill(0)                # array of 0s for spiking neurons
```

```
36        self.spikes[ self.fired_idx ] = 1  # set to 1 elements that correspond to fired
          ↪  indices
37        return self.spikes                 # indices of spiked neurons
38
39
40      # reset states
41      def reset_izhikevich(self):
42        self.v = self.c * np.ones(self.num).reshape((self.num, 1))  # membrane voltage; USE
          ↪  COPY FUNCTION
43        self.u = self.b * self.v                                    # recovery variable
44        self.fired_idx = []                                         # empty list for fired
          ↪  indices
45        self.spikes.fill(0)                                         # array of 0s for
          ↪  spiking neurons
46
      ↪  ##########################################################################################
47
48
49
      ↪  ##########################################################################################
50      # initialize LIF neurons
51      def init_neurons_lif(self, num):
52        self.tau = 150.0             # time constant
53        self.r = 10.0                # resistance for current
54        self.v = np.zeros((num, 1))  # membrane voltage
55
56
57      # update states using Euler's method
58      def step_forward_lif(self, curr):
59        self.fired_idx = np.where(self.v >= 30)[0]  # identify indices of spiking neurons
60        self.v[ self.fired_idx ] = 0                # reset membrane voltages
61        self.v += self.ts * (-self.v + self.r * (self.biases + curr)) / self.tau
62        self.v[ self.v < 0 ] = 0                    # clip negative voltage membranes to
          ↪  zero
63
64        self.spikes.fill(0)                 # array of 0s for spiking neurons
65        self.spikes[ self.fired_idx ] = 1  # set to 1 elements that correspond to fired
          ↪  indices
66        return self.spikes                 # indices of spiked neurons
67
68
69      # reset states
70      def reset_lif(self):
```

```
    self.v.fill(0)       # reset all neurons to 0mV
    self.fired_idx = []  # empty list for fired indices
    self.spikes.fill(0)  # array of 0s for spiking neurons

  ↪ ##########################################################################################


  # initialize parameters
  def __init__(self, num, ts=1, model='izhikevich'):
    self.model = model  # neuron model (Izhikevich or LIF)
    self.ts = ts        # time-step [ms]
    self.num = num      # specify number of neurons

    # assign function pointers for corresponding model
    if (model == 'izhikevich'):
      self.init_neurons_izhikevich( num )               # initialization
      self.step_forward = self.step_forward_izhikevich  # simulation
      self.reset = self.reset_izhikevich                # resetting
    elif (model == 'lif'):
      self.init_neurons_lif( num )                      # initialization
      self.step_forward = self.step_forward_lif         # simulation
      self.reset = self.reset_lif                       # resetting
    else:
      print('ERROR: specified model is invalid.')
      sys.exit()

    self.biases = np.zeros((num, 1))                 # biasing currents for neurons
    self.fired_idx = []                              # empty list for fired indices
    self.spikes = np.zeros((num, 1), dtype=np.int8)  # array of 0s for spiking neurons


  def err_handler(type, flag):
    print("Floating point error (%s), with flag %s" % (type, flag))
  saved_handler = np.seterrcall(err_handler)
  save_err = np.seterr(over='warn')
```

synapses.py

```python
#!/bin/python

import matplotlib.pyplot as plt
import numpy as np

# grid of synapses that interconnect neurons
# element syn_{ij} is a synapse from neuron i to neuron j
class Synapses:
  def __init__(self, num, ts=1):
    self.ts = ts                    # time-step [ms]
    self.num = num                  # number of neurons
    self.tau = 2.0                  # decaying time constant [ms]
    self.g = np.zeros((num, num))  # [pA] synaptic transconductance in units of current
    self.i = np.zeros((num, num))  # [pA] current - state of synapses


  # reset currents of synapses
  def reset(self):
    self.i.fill(0)


  # update value of current in synapses
  # input: grid of 1s and 0s indicating spiking neurons
  def step_forward(self, spikes):
    temp = np.copy(self.g)
    for k in range( len(spikes) ):
      temp[k] = spikes[k] * self.g[k]
    self.i += self.ts * (-self.i / self.tau) + temp  # first order diff equation
    return np.transpose(np.sum( self.i, axis=0 ))    # currents going into same neuron
    ↪  are combined
```