Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
https://cas.tudelft.nl/

CAS-2023-516828

M.Sc. Thesis

Digital self-timed neuron design for Spiking Neuron Networks

Tianyu Du

Abstract

Spiking Neural Networks(SNN) have been widely leveraged by neuromorphic systems due to their ability to closely mimic biological neural behavior, where information is exchanged and received between neurons in the form of sparse events(spikes). Such neuromorphic systems are highly energy-efficient because the use of a global clock can be avoided by asynchronous event-driven operations. Neurons, as the basic processing units of neuromorphic systems, are required to be low-power and high-speed for the implementation of complex networks. In this work, two fully event-driven digital Integrate-and-Fire(IF) neuron design is presented. Both design exploits the hierarchical structure, which allows the synaptic weights can be accumulated by local compute units in parallel. Instead of using handshake protocols, the proposed design generates on-demand event pulses to drive the weight accumulation, so we call it self-timed. Both neurons are designed by SystemVerilog and synthesized in TSMC 28nm technology. According to the synthesis results, both designs can finish the accumulation of 1024 6-bit weights within 100ns, with a power consumption of 0.055pJ per spike and 0.23pJ per spike respectively.

Digital self-timed neuron design for Spiking Neuron Networks

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Electrical Engineering

by

Tianyu Du
born in Dalian, China

This work was performed in:

Circuits and Systems Group
Department of Microelectronics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Delft University of Technology
Department of
Microelectronics

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"Digital self-timed neuron design for Spiking Neuron Networks"** by **Tianyu Du** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: January 2023

Chairman: _____
prof.

Advisor: _____
prof.dr.ir. M.Y. Advisor

Committee Members: _____
dr.

_____
dr.

Abstract

Spiking Neural Networks(SNN) have been widely leveraged by neuromorphic systems due to their ability to closely mimic biological neural behavior, where information is exchanged and received between neurons in the form of sparse events(spikes). Such neuromorphic systems are highly energy-efficient because the use of a global clock can be avoided by asynchronous event-driven operations. Neurons, as the basic processing units of neuromorphic systems, are required to be low-power and high-speed for the implementation of complex networks. In this work, two fully event-driven digital Integrate-and-Fire(IF) neuron design is presented. Both design exploits the hierarchical structure, which allows the synaptic weights can be accumulated by local compute units in parallel. Instead of using handshake protocols, the proposed design generates on-demand event pulses to drive the weight accumulation, so we call it self-timed. Both neurons are designed by SystemVerilog and synthesized in TSMC 28nm technology. According to the synthesis results, both designs can finish the accumulation of 1024 6-bit weights within 100ns, with a power consumption of 0.055pJ per spike and 0.23pJ per spike respectively.

Acknowledgments

By finishing this thesis, I am concluding an unforgettable journey in my life, for which I shall thank many people that I have met along the way.

First of all, my sincere gratitude to Professor Rene van Leuken for providing me with the opportunity to work on this fascinating topic and work with the brilliant people in Innatera, thank you for all the helpful guidance and insights. I would also like to express my great appreciation to Professor Alexander de Graaf for his support and feedback throughout the internship.

I would like to express my deepest appreciation to Aditya Dalakoti, Kamlesh Kumar Singh, and Jinbo Zhou, thank you for being so patient and offering me help whenever I needed it, I'm so grateful for your trust and encouragement. Your knowledge and experience also help me to become a better engineer. I would also like to thank everyone in Innatera for their support and kindness, it has been a great fortune for me to work with them.

Special thanks to my dear friends and colleagues Yichen and Jiongyu, it is so lucky to have you by my side on this journey, I really miss the days when we worked together in the same room.

Words cannot express my gratitude and love for my family, thank you for your unconditional love, company and support. Finally, I want to say to my grandfather, I miss you so much, I know you will always be there for me, giving me the courage and strength to keep on the journey of my life.

Tianyu Du
Delft, The Netherlands
January 2023

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem statement

The development of brain-like architecture has become a grand challenge in the research of neuromorphic computation. The human brain has evolved over billions of years and it is able to solve complex problems with massively parallel computation and low-power consumption. As a result, neuromorphic engineering is seeking solutions to build a system to execute brain-like computation and respective hardware implementation. Numerous neuromorphic processors have emerged recently. Unlike Von-Neumann architecture-based computers, neuromorphic processors are highly energy-efficient, parallel and distributed[1]. Thus, these neuromorphic processors have shown better performance in many engineering applications such as signal processing and pattern recognition[2].

Neuromorphic system design leverage Spiking Neural Network(SNN) processing due to its ability to closely mimic biological neural behavior. Information is exchanged and received between billions of neurons in the form of sparse asynchronous events(spikes). Specifically, SNN consists of a large set of hardware neurons that replicate the behavior of neurons in brains, neurons are connected by specialized functional units called synapses[3]. Each synapse has an associated weight which represents the connection strength. Time is incorporated as an explicit dependency in SNNs, at a particular instant, neurons receive numerous spikes from connected synapses and start evaluation and generate spikes via local variables.

Asynchronous logic design has long been seen as the optimized method for modeling Spiking Neuron Networks because computation in biological neural networks is executed by asynchronous event-driven operations. In SNNs, redundant operations are skipped due to sparsity exploitation, so power is only dissipated when necessary[1]. Using asynchronous event-driven design allows neurons to evaluate their membrane potential against the threshold without the need for a global clock signal, which eliminates the power which is consumed by the continuously running clock. Besides, parallel and event-driven processing allow neurons to work independently of other neurons, which significantly simplifies the back-end timing closure. In addition, asynchronous logic can also eliminate the time margin and reduce the latency response[4].

Most asynchronous communication implemented in state-of-art neuromorphic processors is based on handshake protocols such as Loihi[5] and TrueNorth[6]. However, traditional handshake protocols require c-elements and control signals(req and ack) to get correct behavior which causes extra area and latency. Besides, the performance of handshake protocols depends on wire delay[4], as a result, the propagation delay and signal integrity issue would become even worse in large-scale neuromorphic systems. Another problem for the existing neuromorphic processors is that each spiking neuron

accumulates synapse weights sequentially, which leads to low time efficiency especially when the network becomes more complex. [1]proposed a self-timed event-driven SNN architecture, which relies on an oscillator and a novel delay cell to generate a local clock so the use of the global clock or handshake can be avoided. Inspired by the work in [1], this thesis presents two novel event-driven pulse-mode asynchronous architectures for digital Integrate-and-Fire(IF) neuron design in SNN. The proposed design is able to generate on-demand event pulses to drive the weight accumulation rather than using a global clock. Instead of one-by-one accumulation, the proposed design offers a hierarchical structure where weights are divided into multiple subgroups and accumulated by local compute units in parallel. The proposed design allows digital spiking neurons to evaluate with high speed and low power consumption.

1.2 Objectives

The objective of this work is to design high-speed Spiking Neural Network subcomponents(neurons) in self-timed asynchronous logic. The components need to be connected together in neuromorphic array structures to solve use case problems. The functionality of the hardware design should be verified, and relevant design parameters should be identified in order to perform trade-off analysis. The proposed design should be implemented in ASIC technology and the performance on power, area and delay should be analyzed. Finally, different connectivity patterns should be compared to determine the one with better properties.

1.3 Contributions

The main contributions of this thesis include:

- Hardware implementation of the two proposed neuron designs by SystemVerilog, the function is verified by Modelsim pre-synthesis simulation, critical timing analysis is given.
- The design of two novel pulse-based asynchronous pipelines that can be applied to the hierarchical structure of the proposed neuron designs.
- The synthesis results of the two neuron design is presented and compared, the post-synthesis simulation is given by the Xcelium simulator.
- Giving the trade-off analysis of different hierarchical structures and providing the methods for optimizations.

1.4 Thesis outline

- Chapter2 first summarizes the basic background knowledge related to this work and then some state-of-the-art design in the field is given.

- Chapter3 presents the design of two proposed self-timed neurons as well as their combined asynchronous pipeline design, related timing constraints are discussed and the pre-synthesis simulation results is given.
- Chapter4 provides the pos-synthesis simulation results of two neuron designs and discusses their performance, trade-off analysis is given to explore the effect of the hierarchical structure on the neuron's performance.
- Chapter5 concludes the thesis and lays out the possible future work.

2 Literature review

This chapter will first give a basic background knowledge related to Spiking Neural Networks, some state-of-the-art neuromorphic processors will be introduced. After that, the method of asynchronous logic design will be discussed, which includes the topic of self-timed logic and asynchronous pipeline design.

2.1 Spiking Neural Networks

2.1.1 Neuromorphic systems

The concept of neuromorphic computing was first proposed in 1989[7] and became a milestone of brain-like system development. Neuromorphic computing, also known as neuromorphic engineering, is the reverse engineering of the brain. It aims to use semiconductor technology to construct the basic units of the brain's nervous system—neurons and synapses, and connect them in the way that they are organized in the brain. The feasibility of neuromorphic engineering comes from the similar massively parallel structure of neural and neuromorphic networks, as well as the similarity of the working process between neural and transistors. Neuromorphic system designs based on analog, digital, and hybrid mixed-signal have emerged in recent years due to their desirable features.

2.1.2 Spiking Neural Networks

Spiking Neural Networks(SNNs) are exploited by neuromorphic systems for data exchange due to their highly sparse processing[1]. SNN was first proposed by Alan Lloyd Hodgkin and Andrew Huxley in 1952, where calculations are performed by using a model that is highly similar to the biological neuron mechanism. The development of SNN narrows the gap between the artificial neural network and the real biological neural system. An SNN consists of spiking neurons as the basic processing units, instead of using continuous values for data transmission as in conventional ANNs, data is transmitted via the precise timing of spikes or a sequence of spikes in SNNs. Due to the high sparsity of spiking events and the form of event-driven computation, SNN can provide excellent energy efficiency and is the best choice for neuromorphic systems[8].

SNNs are considered to have more potential than ANNs in terms of integration and energy consumption, this comes from three advantages brought about by the spike mechanism. Firstly, the bandwidth and power consumption of transmitting binary pulses are much lower than those of transmitting multi-bit continuous values. Secondly, the weight accumulation in SNNs can be completed simply by adders since the pulses are single-bit, while in ANNs multipliers are required which cost extra area. Thirdly,

since SNNs are event-driven, neurons are in an idle state when no input is received, while neurons in ANNs need to be in a working state all the time.

2.1.3 Neurons

Biological neurons consist of a cell body, dendrites, and axons, as shown in Figure 2.1. Information is transmitted between neurons in the form of electrical signals. The cell body is the center of neuron operations, which is used to process the electrical signals received from the dendrites and generate output. Dendrites are used to receive electrical signals from pre-neurons. A neuron has a large number of dendrites, and each dendrite can simultaneously receive electrical signals from other neurons. The spike signals are transmitted through the axon to the synapses, then each synapse applies its own weights to the spikes before they go to downstream neurons. A neuron has an average of 15,000 synapses, which can be connected to multiple neurons and receive electrical signals from multiple neurons simultaneously. The electrical signals received by synapses can promote or inhibit the membrane potential of the neuron. When the membrane potential overcomes the threshold, an output signal will be generated and transmitted to other neurons.

Some neuron models are developed to imitate the actual activity of neurons, mainly including the integrate-and-fire model, Hodgkin–Huxley model, and the Izhikevich model.

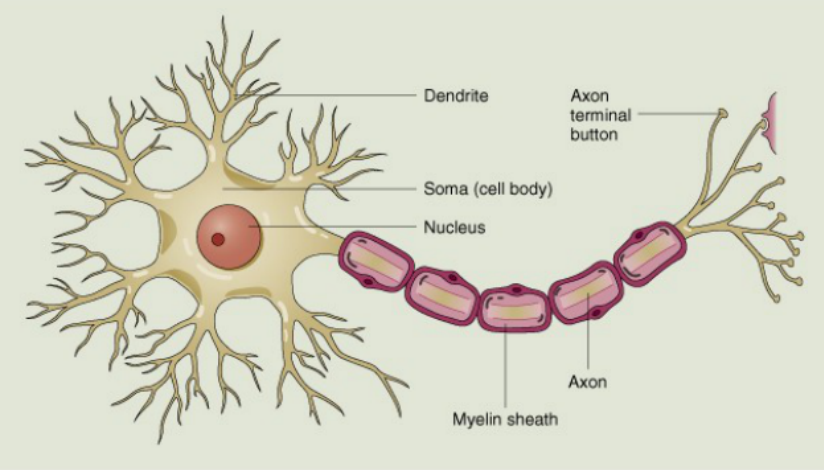


Figure 2.1: Overview of the neurons in the brain [9]

2.1.3.1 Integrate-and-fire model

Integrate-and-fire(IF) model is the simplest neuron model proposed by Lapicque in 1907[10]. In the IF model, the state of the neuron is defined by the membrane potential, which receives excitation or inhibition from synaptic inputs. The IF model can be regarded as an integrator, the input current of the neuron is integrated by a capacitor,

when the voltage over the capacitor reaches the threshold voltage, the capacitor is discharged and a voltage spike is transmitted at the output of the neuron.

The limitation of IF neuron model is that it cannot model the leakage of the membrane potential, so the Leaky Integrate-and-Fire(LIF) model is proposed to solve this problem. In LIF model a register is added to provide the leakage circuit, as shown in Figure 2.2. The LIF neuron model is one of the most widely used models for analyzing the behavior of neural systems.

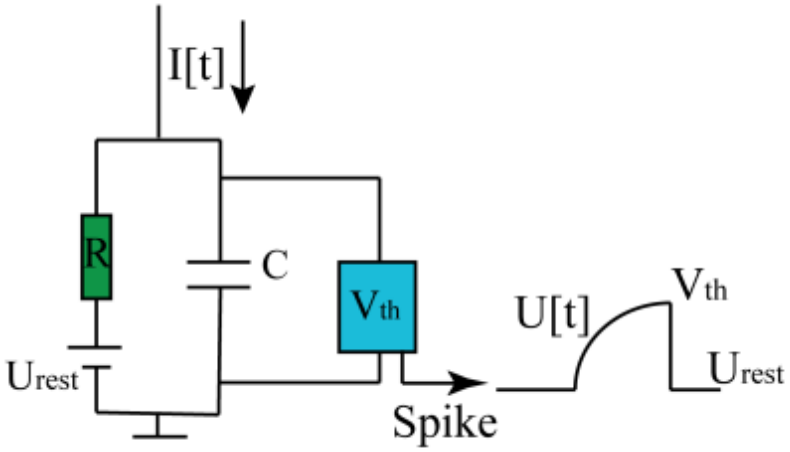


Figure 2.2: Equivalent circuit model of the LIF neuron
[11]

2.1.3.2 Hodgkin–Huxley model

The Hodgkin-Huxley (H-H) model is proposed by the quantitative study of ionic current and conductance on the cell membrane of squid neurons[12]. It has a high degree of biological similarity and is currently the closest model to biological neurons. Since the cell membrane is selectively permeable and only specific ions are allowed to pass through, so the ions flowing inside and outside the cell will affect the potential difference of the cell membrane. The H-H model can accurately represent the dynamic behaviour of the ion channels by a conductance-based model, but it needs to model sodium, potassium ions and leakage channels separately which is complex and the calculation process is cumbersome. The equivalent circuit of the H-H model is shown in Figure 2.3.

2.1.3.3 Izhikevich model

Izhikevich model was proposed in 2003, it is a simplified model which combines the biological plausibility of Hodgskin-Huxley dynamics and the computational efficiency of integrate-and-fire model[13]. The model uses second-order differential equations to describe the dynamic properties of the cell membrane, as shown in equations 2.1, where v represents the membrane potential of the neuron, u represents the inhibitory variable of sodium ion and the activation variable of potassium ion, which provide negative

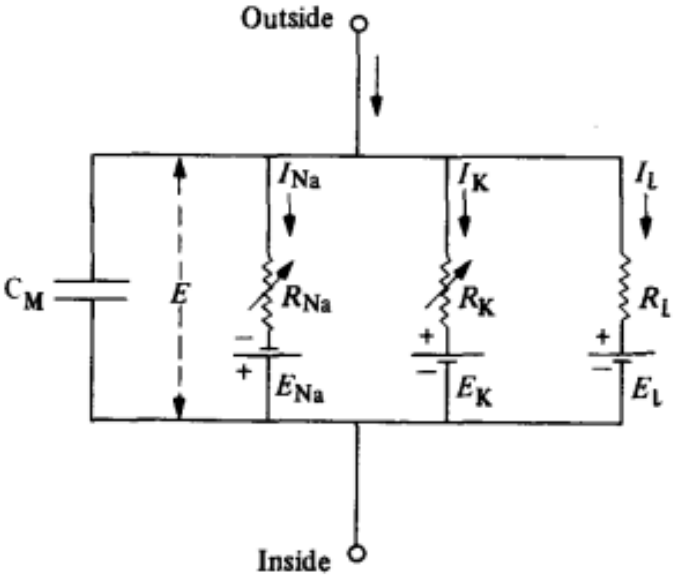


Figure 2.3: Equivalent circuit model of the H-H neuron [12]

feedback to the neuron membrane potential v, and I represents the external input current. If the neuronal membrane potential v exceeds the threshold of 30mV, then v and u will be reset by equation 2.2.

$$\begin{cases} \frac{\mathrm{d}v}{\mathrm{d}t} = 0.04v^2 + 5v + 140 - u + I, \\ \frac{\mathrm{d}u}{\mathrm{d}t} = a(bv - u), \end{cases} \tag{2.1}$$

$$\begin{cases} v \to c \\ \qquad\qquad , if v \geq 30mV \\ u \to u + d, \end{cases} \tag{2.2}$$

a, b, c, and d are dimensionless parameters, where a represents the recovery speed of the membrane potential; b represents the sensitivity of the recovery variable u to the of the membrane potential; c determines the reset value after the neuron fires a pulse; d determines the recovery variable u after the neuron fires a pulse. Different types of firing patterns can be obtained by adjusting these parameters[13].

2.1.4 AER

The neuron firing rates in a Spiking Neural Network range from tens of Hz to hundreds of HZ, even if there are thousands or even millions of neuron firing at the same time, the firing rates of the network is between KHz and MHz, which can be easily met by the digital system. Therefore, Address Event Representation (AER) is proposed for spike transporting in SNNs. AER leverages the time-multiplexing method to transport spikes between neuron arrays, spikes are encoded as addresses that indicate the source

neuron and destination synapses and sent over a bus(Figure 2.4). An arbiter is used to order the spikes if multiple spikes arrive simultaneously. The addresses are decoded at the input of the neuron array and then the spikes are sent to the destination neurons.

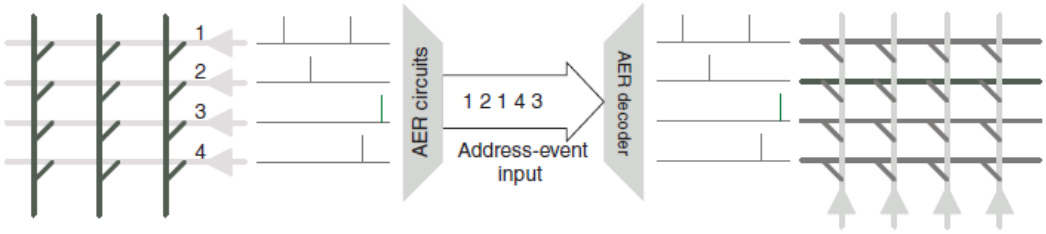


Figure 2.4: Conceptual diagram of AER working as a bus for spiking events [3]

2.1.5 State of the art

The State-of-the-art implementations of SNN emulators are mainly based on full digital and hybrid mixed-signal silicon technology[1]. Analog circuits can accurately simulate the dynamic characteristics of neurons and realize relatively complex dynamic models, but they are sensitive to temperature changes and have very weak programmability. Besides, analog neuromorphic ICs also suffer from high design cost and low neuron density. In contrast, digital neuromorphic ICs leverage logic gates to emulate neurons and synapses and dense memory, so the design can be synthesized and integrated in an SoC which results in high density and low cost. Neuromorphic ICs that are worth mentioning will be discussed in this section.

- **Neurogrid**[14] is a hybrid mixed-signal multi-chip system used for large-scale neural simulations, it consists of 16 neurocores and each core consists of 65k analog neurons with digital routing. The neuron circuits model the soma, dendritic trees, synapses, and axonal arbors. The neurocores are interconnected with a tree topology and a 4KB memory is implemented in each core for spike transmitting. A daughterboard with 256Mb RAM is applied to transmit the spikes to arbitrary locations in multiple Neurocores.

- **TrueNorth**[6] is a fully digital SNN implementation proposed by IBM. It consists of 4096 neurosynaptic cores and each core has 256 input axons and 256 neurons. The cores exploit a crossbar structure where the 256 axons are shared by 256 neurons, so each neuron can have up to 256 inputs. The axons and neurons are connected by the synapses on each cross-point(Figure 2.6). 4096 cores are distributed in a 64×64 array, the output spikes from one neuron can be transmitted to any other cores in the form of packets. Truenorth is designed by a mixed asynchronous–synchronous approach, handshake-based asynchronous design is used for the communication and control circuits, while the synchronous design is used for computations[15]. Truenorth is ideal for energy-efficient applications due to its event-driven process, which only consumes 72mW of power, corresponding to 26 pJ per synaptic event.[2].

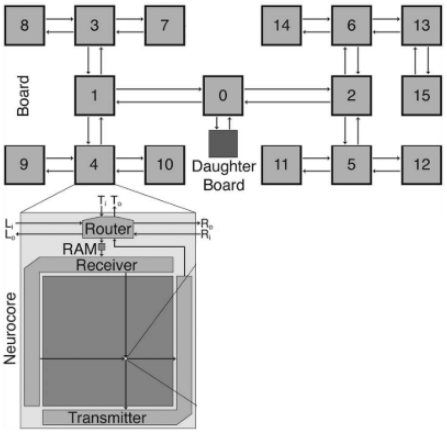


Figure 2.5: Overview of the Neurogrid architecture [14]

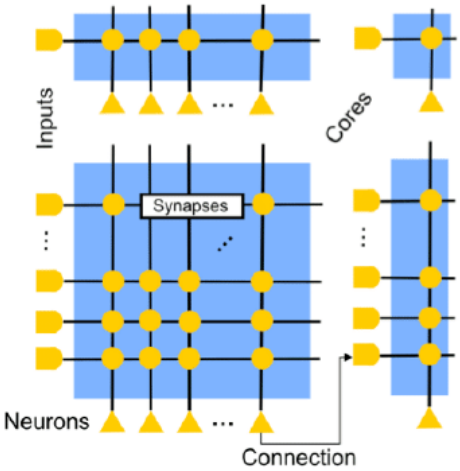


Figure 2.6: Overview of the TrueNorth architecture [16]

- **Loihi**[5] is a fully digital neuromorphic processor developed by Intel. The chip consists of 128 neuromorphic cores and each core integrates 1024 neurons. Loihi is implemented in an asynchronous bundled data design style which allows operations to be processed in an event-driven manner. A handshake-based hierarchical mesh protocol is implemented to support the spike transmission between the neuromorphic cores[17], which allows spikes can proceed at any rate without waiting for clock edges or consuming clock power during periods of idle.

- **μBrain**[1] is an asynchronous and fully-synthesizable digital neuromorphic processor design with self-timed cores. μBrain core consists of three layers of 336 fully-connected integrate-and-fire neurons. All the neurons work independently(no global clock) to accumulate the weighted incoming spikes and emit spikes when the accumulator overflows. μBrain provides ultra-low-power consumption by

event-driven processing. Instead of using handshake protocols, it generates an on-demand local clock by a lightweight oscillator to drive all the blocks, as shown in Figure 2.7. The oscillator is triggered only when the input spikes are detected by the input edge detector, low-power customized delay cells are implemented in the oscillator to warrant the correct pacing of its phases for ordered propagation of spikes. μBrain consumes around 26pJ per spike with 30% of static power[1].

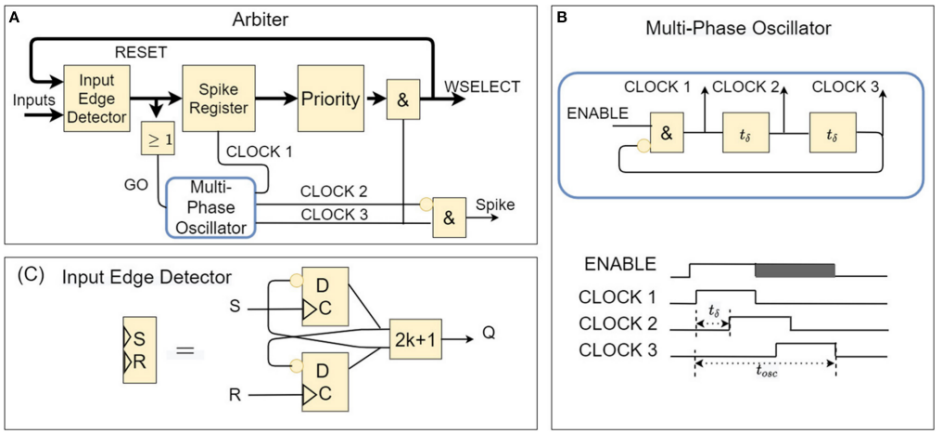


Figure 2.7: Basic components for self-timed operations in μBrain [1]

2.2 Asynchronous design

Most digital systems designs are based on synchronous logic. In recent years, with the rapid development of integrated circuit manufacturing technology, the size of transistors has been continuously reduced, which has increased the design scale and introduced many unexpected problems, such as clock skew and electromagnetic interference. As a result, asynchronous logic designs have shown many advantages compared with synchronous design in many aspects due to the cancellation of the global clock, and a number of novel application areas based on asynchronous design have recently emerged[18].

The main advantages of the asynchronous design can be included as follows:

- **Low power**
  Data transmission in an asynchronous circuit depends on the interaction of handshake signals rather than the global clock. The handshake signal is sent out only when the data is valid, and the circuit will stay idle if no valid is received, which reduces unnecessary power consumption.

- **Average case instead of worst-case performance**
  The speed of the synchronous circuit is determined by the clock frequency, and the clock frequency is determined by the length of the critical path of the data path, so the performance of the synchronous design is the worst-case performance. In contrast, the communication in asynchronous circuits depends on the local control signal, so the data can be sent out as long as they are valid rather than waiting

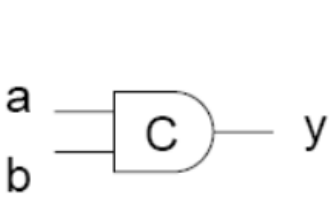
for the clock signal. Therefore, the performance of the asynchronous depends on the average performance rather than the worst-case performance[19].

- **No clock distribution and clock skew problems**
  In a synchronous system, the arrival time of the clock signal to each component may be different, which results in a clock skew. As the scale of the digital systems become larger, clock skew becomes a much greater concern. However, asynchronous systems do not have a global clock, so there is no need to consider the problem of clock skew.

- **Less emission of electromagnetic noise**
  The working pace of components is consistent in synchronous systems, which results in noise and electromagnetic interference[20]. However, the components in the asynchronous system work independently and locally, thereby reducing electromagnetic interference and noise.

2.2.1 Muller-C element

The Muller-C element is the fundamental element in asynchronous circuits, it is used as a state-holding device that prevents hazards and races in the asynchronous circuit. The logic function of the C-element is that when both inputs are '0', the output becomes '0'; when both inputs are '1', the output becomes '1'; when the two inputs are different, the output remains unchanged. The symbol and truth table of the C-element are shown in Figure 2.8.



a	b	y
0	0	0
0	1	no change
1	0	no change
1	1	1

Figure 2.8: The symbol and truth table of the C-element [21]

2.2.2 Classes of asynchronous circuits

Asynchronous circuits can be classified as delay-insensitive(DI), quasi-delay-insensitive(QDI), speed-independent(SI), and bundled delay depending on their delay assumptions[18].

In DI circuits, the delay of the gates and wires is assumed to be uncertain, only circuits consisting of C-elements and inverters are DI circuits. Most DI circuits are actually QDI, the only difference between DI and QDI is that all the wires in QSI circuits are assumed to have equal delays at each fan-out point, which means signal

transitions on the same node occur simultaneously. SI circuits assume that the delay of the gates is uncertain, but the delay of wires is zero, although it is unrealistic to assume that the line delay is ideal, the circuit can still be regarded as SI design by adding the wire delay to the gate delay[22]. Bundled delay design is also regarded as the self-timed design[21], it assumes that the delay of gates and wires is fixed and known. The bundled delay design is similar to the synchronous design, which requires additional timing constraints, and data is considered to be valid after a long enough delay.

2.2.3 Handshake protocols

The handshake protocol is an important concept in the asynchronous circuit, which is used to perform data transmission between the internal blocks, there are two types of protocols, as shown in Figure 2.9: four-phase handshake protocol and two-phase handshake protocol. In the 4-phase handshake mode, the occurrence of a request or acknowledgment is represented by the level of the handshake signals, the handshake signals are reset in turn, in the return-to-zero(RTZ) phase. In contrast, the occurrence of a request or acknowledge is represented by the transitions of the handshake signals, handshakes signals don't need to return to zero for each data transaction(NRZ), which consumes less power and delay than the 4-phase handshake protocol, but may involve more complex hardware design[18]

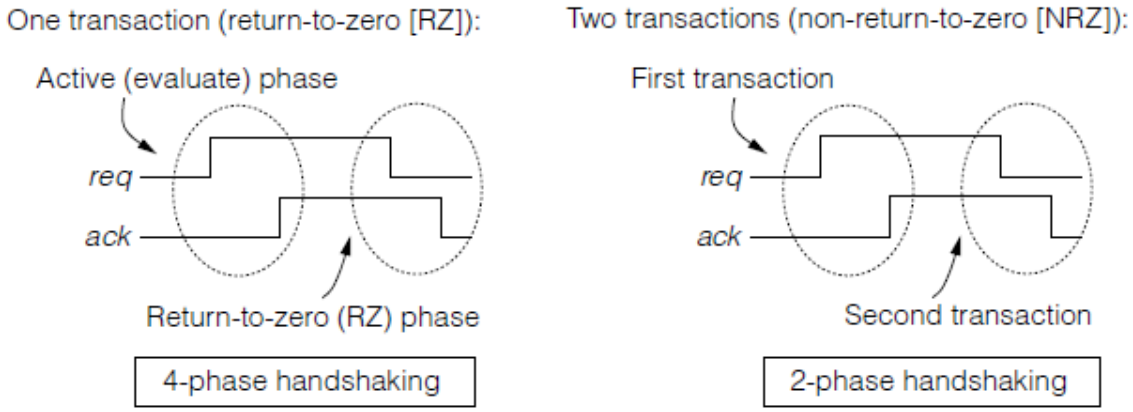


Figure 2.9: Basic asynchronous communication protocols [19]

2.2.4 Data encoding

2.2.4.1 Bundled data

Bundled data is also called single-rail data, where the data signals use normal Boolean levels to encode information[21], and where the request signal is bundled by the worst-case delay to indicate the data validity, as shown in Figure 2.10. The set-up and hold

time constraints can be satisfied by adding delays to the matched delay cell. The advantage of the single-rail data is that although it is a synchronous-style protocol, it provides the average case performance because of localization of the delay matching[18]. The single-rail data can be implemented with both 2-phase handshake protocol and 4-phase handshake protocol.

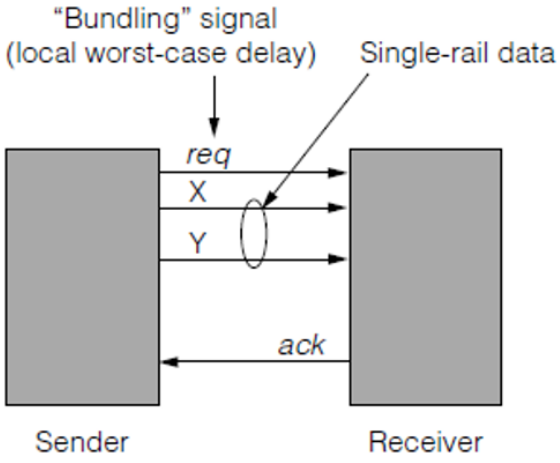


Figure 2.10: Bundled data encoding
[19]

2.2.4.2 Dual-rail codes

In dual-rail designs, data is encoded with two wires or rails X0 and X1, as shown in Figure 2.11. In the 4-phase dual-rail protocols, the combination of x0 and x1 is regarded as a codeword, and $\{X0, X1\}=\{1, 0\}$ and $\{X0, X1\}=\{0, 1\}$ respectively represent the valid logic '0' and logic '1'; the empty codeword $\{X0, X1\}=\{0, 0\}$ represents that the data is invalid; the codeword$\{X0, X1\}=\{1, 1\}$ is not used. A transition from one valid codeword to another valid codeword must go through an empty codeword. Since the validity of data is included in the dual-rail lines, only one ack signal is needed to complete the handshake between the blocks. Normally completion detectors (CD) are used to identify if every bit is valid[23]. The dual-rail 2-phase handshake protocol is similar to the above, the only difference is that the information is encoded as transitions[21].

2.2.5 Asynchronous pipeline

The pipeline is widely used in modern digital systems to increase parallelism and hence improve the system throughput. In a synchronous pipeline, combinational logic is divided into small blocks by registers all the registers are driven by the global clock. While in an asynchronous pipeline, the interaction of neighboring blocks is coordinated by the handshaking protocols. Asynchronous pipelines can be classified as static logic pipelines and dynamic logic pipelines depending on their encoding methods[18].

Static logic pipelines use bundled data and operate on static logic data paths. Sutherland's micropipeline is a design of static logic pipeline[24], which uses a 2-

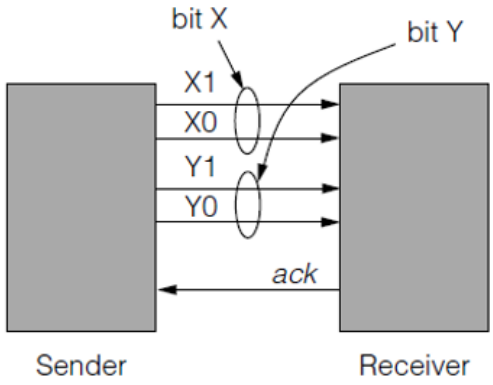


Figure 2.11: Dual-rail encoding
[19]

phase handshake protocol. The micropipeline consists of c-elements and capture-pass latches(Figure 2.12), which allows data can be transported to the next stage at each transition of the handshake signals. The mousetrap pipeline [25] is a variant of the micropipeline, which is designed by only standard cells and has less complex signaling and lower overhead.

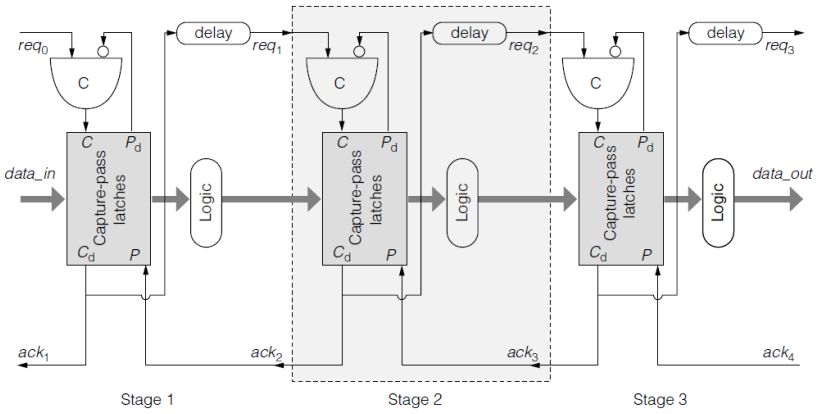


Figure 2.12: Sutherland's micropipeline
[18]

Dynamic logic pipelines use dual-rail encoding and operate on dynamic logic data paths. PS0 pipeline is the most influential dynamic pipeline which uses a 4-phase handshake protocol. The design only consists of function blocks and completion detectors(CD), without any storage elements(Figure 2.13). Each block switches between the evaluate state and the precharge state, and the state of the blocks is determined by the CDs. The validity of data is verified by OR'ing the two rails for each individual bit and then using a C-element to combine all the results[26].

The performance of the asynchronous pipeline is evaluated by two metrics, which are the *forward latency* and the *reverse latency*[27]. The *forward latency* is the time it takes a data item to flow forward through an empty pipeline stage, and the *reverse*

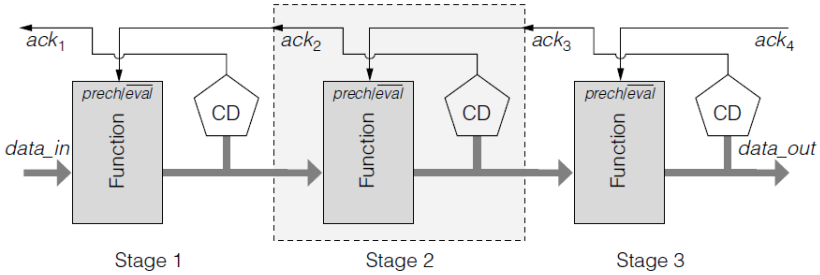


Figure 2.13: PS0 pipeline
[18]

latency is the time it takes the ack signal to flow back to the previous stage. The throughput of the pipeline is measured at its maximum capacity and determined by the cycle time, which consists of one *forward latency* and one *reverse latency*[18].

3 Implementation

In this chapter, two types of digital implementation of self-timed IF neurons will be illustrated. This chapter begins with an introduction to the basic concept of the IF neuron. Then there will be an overview of the proposed design. Next, the hardware implementation of self-timed neurons based on two types of accumulators and the corresponding pipeline design will be discussed. After that, the simulation result will be presented to verify the functionality.

The proposed design imitates the actual activity of the integrate-and-fire model neuron(IF neuron). The behavior of the IF neuron can be described as follows: the neuron receives multiple input spikes, which represent the action potential, from the pre-synaptic neurons, then the input spikes will be weighted and accumulated over time as the membrane potential of the neuron, when the membrane potential overcomes the threshold, the neuron will emit an action potential taking the form of a binary spike to the downstream neurons. The working process of an IF neuron can be illustrated in Figure 3.1. In this design, it is assumed that input spikes are fed into the neuron simultaneously at each t_n. The time interval between two groups of input spikes is Δt. The length of Δt depends on the speed of the neuron accumulation, a smaller Δt means a higher throughput that the neuron can achieve.

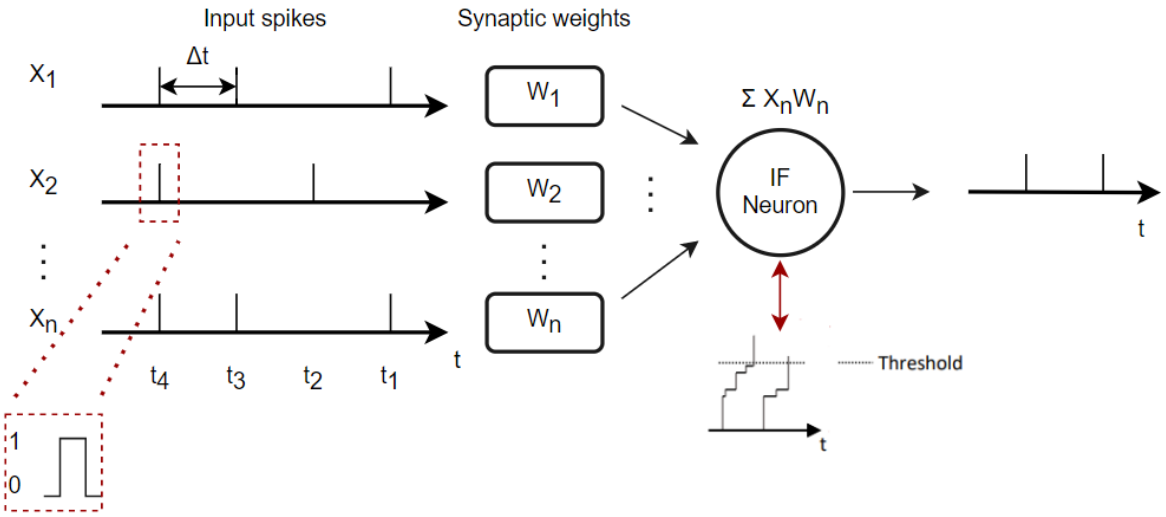


Figure 3.1: The working process of the spiking neuron

The proposed IF neurons will be implemented in an m×n SNN neuromorphic crossbar array, which is the most common structure for the SNN cores. The structure of the array is shown in Figure 3.2. The input spikes are received by the array at each

row and sent to all neurons in parallel, thus reducing the communication volume m times compared to a point-to-point approach. Each cross-point of the array is assigned a synaptic weight, if the weight of a particular input-neuron pair is non-zero and the input spike is fired, then the synaptic weight is activated. The neurons then accumulate all the activated weights and compare the result with the threshold value. The output spikes from the neurons will be transmitted to other arrays in the network.

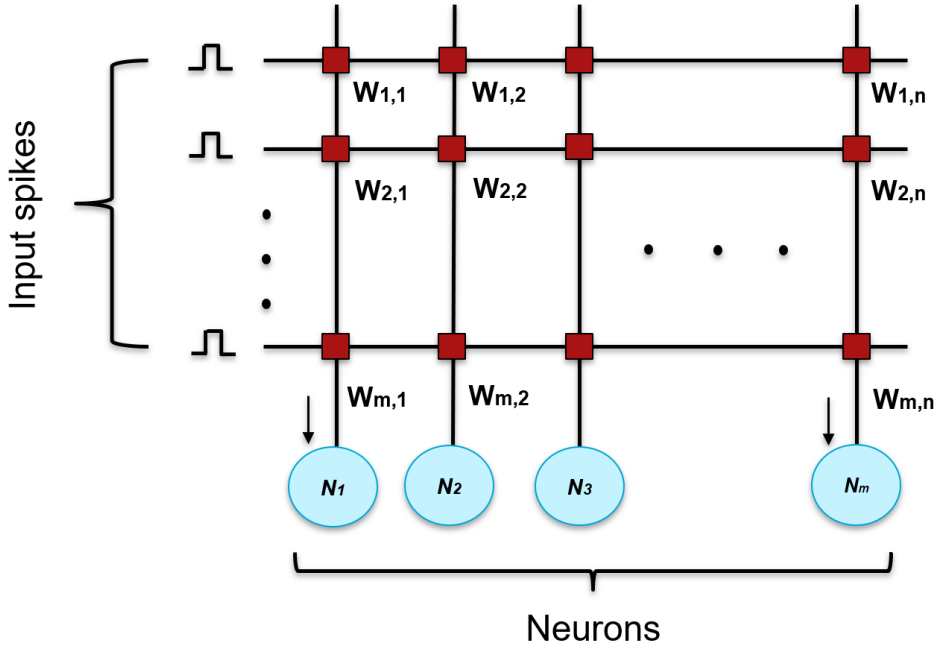


Figure 3.2: Schematic of SNN neuromorphic array

3.1 Architecture overview

A 1024-input self-timed neuron is proposed in this work, the schematic is shown in Figure 3.3. The accumulator is the most critical component in this design, which accumulates all the weights activated by the input spikes, the result from the accumulator will be accumulated again as the membrane potential. The accumulator will generate a finish spike when its accumulation is done, the finish spike is connected to the clock port of the register to update the membrane potential. The membrane potential will be reset to 0 when it overcomes the threshold. A delay cell is implemented to propagate the finish spike, it is allowed to send out only when the membrane potential overcomes the threshold. Some constraints should be followed to guarantee the neuron works properly, first the time interval between two finish spikes t_{finish} should be long enough to prevent the setup violation(inequality 3.1), besides, the delay of the delay cell should be within a proper range to guarantee that the control signal of the mux arrives before the finish spike, and the finish spike can be sent out before the next finish spike comes(inequality 3.2). The synaptic weights are sent to the neuron from the weight memory, a global reset signal is used to initialize all the registers inside the neuron. The proposed design is fully event-driven, the neuron starts accumulation only

when it receives input spikes, otherwise it will stay idle.

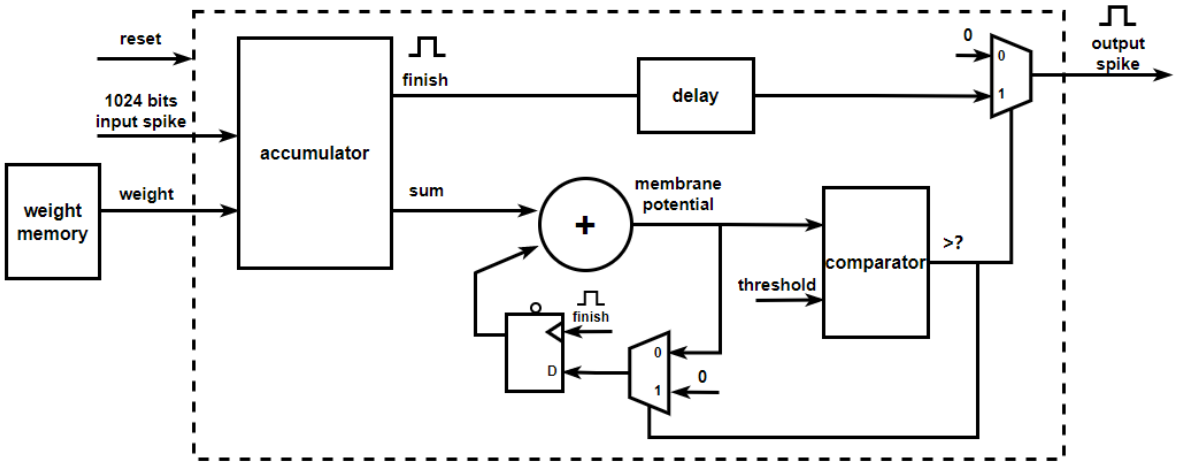


Figure 3.3: Schematic of proposed self-timed IF neuron

$$t_{finish} > t_{cq} + t_{adder} + t_{comparitor} + t_{mux} + t_{setup} \tag{3.1}$$

$$t_{finish} > t_{delay} > t_{adder} + t_{comparitor} \tag{3.2}$$

3.2 Accumulator design

Two high-speed, self-timed hierarchical accumulator designs are proposed in this work. Unlike other synchronous digital neuron implementations, it relies on delay cells to support its asynchronous event-driven processing, which avoids using a global clock. Since the activity in SNNs is highly sparse, the activity gating that comes with self-timed flow control eliminates the power that would be consumed by a continuously running clock. The hierarchical architecture of the proposed self-timed accumulator for 1024 weight accumulation is shown in Figure 3.4. It consists of 3 levels, 1024 input spikes are captured in the first level and separated into 64 groups, 16 spikes in each group. The synaptic weights activated in each group are accumulated by a local compute unit. The compute units in the same level work independently and parallelly, their output will be stored in the register as the input of the compute units in the next level.

The total delay of this design is the sum of the delay of 3 levels, which is much faster than one-by-one accumulation because of the parallel computing. The asynchronous pipeline is implemented to break the combinational logic so that the neuron is able to sample the input spikes with a higher frequency.

The compute unit is the core of the accumulator. In the rest part of this chapter, two different accumulator designs based on two types of compute units will be described as well as their corresponding pipeline design. The waveforms of the simulation will be posted to verify the functionality of each part.

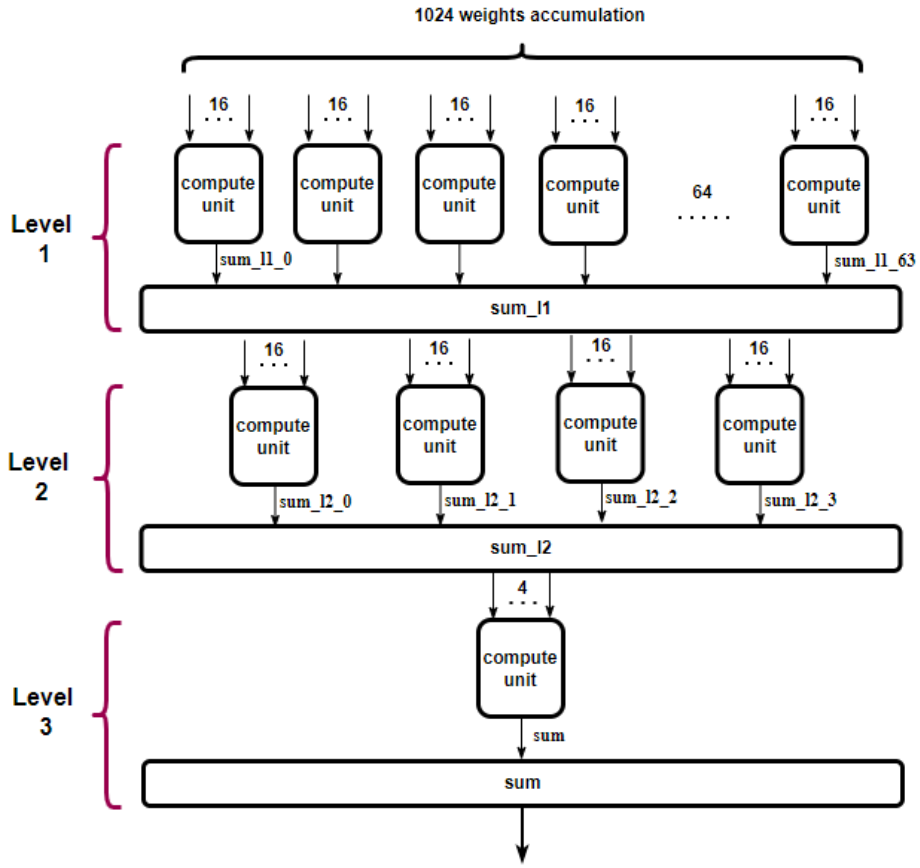


Figure 3.4: Hierarchical architecture of the self-timed accumulator

3.2.1 Adder tree based accumulator

In the first proposed design, the weight accumulation is done by the combinational logic, adder trees are used for multi-operand addition. 16-input adder trees are implemented in the first and second levels, a 4-input adder tree is implemented in the third level. The output of the adder trees of each level will be stored in the registers. In the absence of the global clock, the registers between the combinational logic of two levels are driven by event pulses, which are generated by a pulse-based pipeline. The input weights of the accumulator are determined by a capture layer which is implemented in front of the adder trees in the first level, the schematic of the capture layer is shown in Figure 3.5. The input spikes are sent to the clock port of registers, the data ports of the registers are consistently sent to logic '1', so whenever a spike is received by the register, the output of the register will be pulled up. The output of the register is used as the control signal for a mux to determine whether the weight is activated. If at least one input spike is detected, the or-gate tree will pull up the *req* signal and generate a start pulse, the pulse with of the start pulse is determined by the delay cell. The registers will be reset to 0 when the accumulation in the first level is done so that they are able to capture the next group of spikes. The output of the muxes is separated into 64 groups, 16 6-bit input data in each group will be sent to the corresponding adder

tree and accumulated.

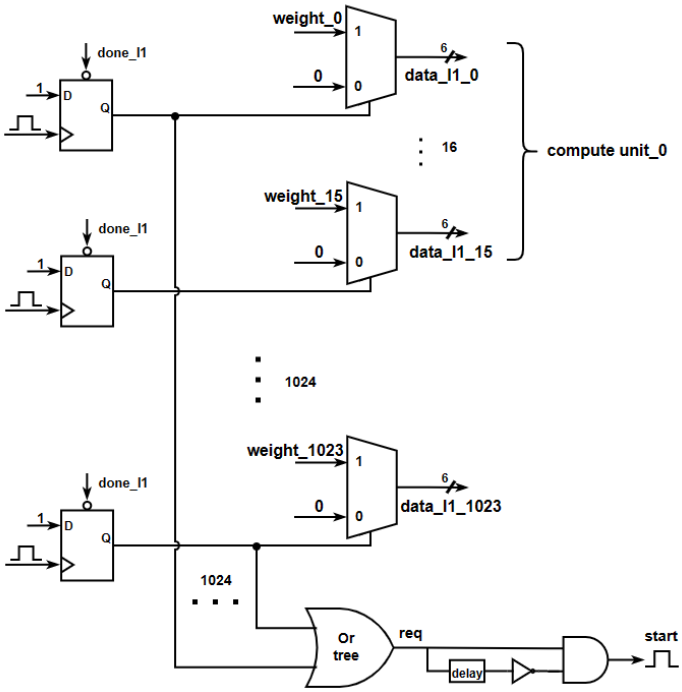


Figure 3.5: The schematic of capture layer

3.2.1.1 Compute unit design

Adder trees are widely used for fast multi-operand additions, most adder trees are designed by connecting two-operand adders in a tree structure for parallel computing. The delay, area and power consumption of the adder tree depends on the two-operand adder inside. Different kinds of adders can be used to construct an adder tree such as ripple carry adder(RCA), carry-select adder, carry-skip adder and carry-lookahead adder. RCA has less area and power consumption compared with other fast adders, so it is the most preferred choice for constructing an adder tree even though it has the highest delay, however, the connection inside the tree structure will compensate for the high latency of the single RCA. The structure of an N-operand RCA-based adder tree is shown in Figure 3.6, it consists of $\log_2 N$ stages with N-1 RCAs in total.

The delay of the RCA-based adder tree $T_{RCA-tree}$ can be calculated by equation 3.3, $T_{RCA,k\,bit}$ represents the delay of a k-bit RCA and T_{FA} represents the delay of full adder. It can be seen from the equation that the total delay of the RCA-based adder tree is not the sum of the RCA delay of each stage, except for the first stage, it takes only two full adder delays at each stage.

$$T_{RCA-tree} = T_{RCA,k\,bit} + (\log_2 N - 1) \times 2T_{FA}[28] \tag{3.3}$$

In addition to the RCA-based adder tree, another adder tree scheme based on carry-save adders(CSA) can also be used for weight accumulation. A CSA can reduce the

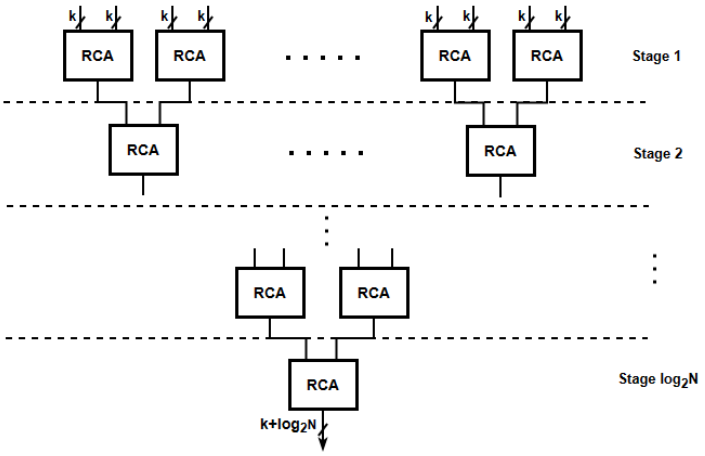


Figure 3.6: N-operand adder tree based on ripple carry adders

three operands to two by breaking the connections between the full adders in an RCA [29], then the two operands are sent to CSA in the next level. The final two operands of the adder tree are added by a two-operand adder. Figure 3.7 shows the CSA-based adder tree for seven k-bit operands. Each CSA only introduces one full adder delay, so the total delay of the CSA-based adder tree can be calculated by equation 3.4, where T_A represents the delay of the two-operand adder in the final stage and m represents the number of stages of CSA in the adder tree.

$$T_{CSA-tree} = T_A + m \times T_{FA}[28] \tag{3.4}$$

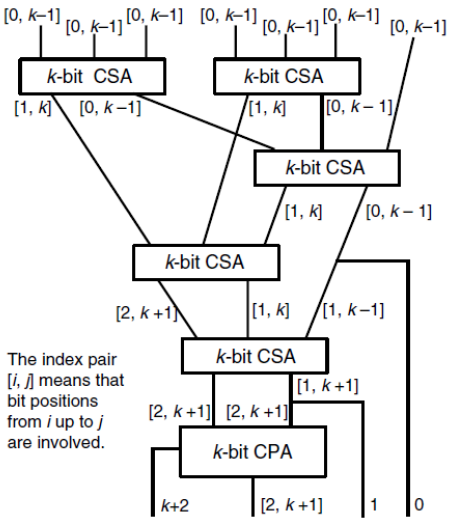


Figure 3.7: Adding seven k-bit operands with CSA-based adder tree [29]

In the proposed neuron design, RCA-based adder trees are chosen for the weight accumulation because of their parameterizable and regularity. Even though a CSA-

based adder tree has better delay performance than an RCA-based adder tree, the design of the former one is more complex, because the structure of the CSA-based adder tree is irregular, it depends on the number of operands. The connection between CSAs needs to change for different numbers of operands.

3.2.1.2 Pipeline design

In the absence of the global clock, an asynchronous pipeline is implemented to break the combinational logic and improve the throughput of the neuron. Most asynchronous pipeline designs rely on the handshake protocol, which costs less power consumption and latency compared with the synchronous pipeline. The drawback of the handshake-based asynchronous pipeline is that it requires extra control logic to provide the control signals, which consumes more area and power. Besides, commercial electronic design automation (EDA) tools cannot fully support the timing analysis of the handshake asynchronous design because of the existence of combinational loops. The above two main reasons make asynchronous design still not fully accepted in the industry. In this design, an asynchronous pipeline design without handshake protocols is proposed, the control signal of each pipeline stage is propagated by the delay cells. The design of the neuron based on the proposed asynchronous pipeline is shown in Figure 3.8.

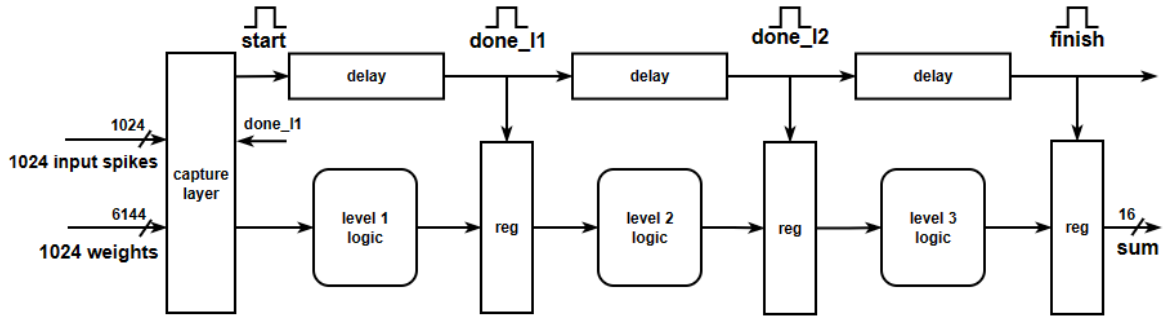


Figure 3.8: Asynchronous pipeline design without handshake protocols

A start pulse is generated by the capture layer whenever an input spike is received, then it is delayed until the output of the combinational logic is stable, the delayed pulse is used to update the registers and trigger the downstream levels. In order to guarantee that the pipeline works properly, some constraints should be given. First, the delay of the delay cells of each level should match the critical path of the combinational logic to prevent the setup violation, as shown in inequality 3.5. Second, the data of the current level should not be propagated to the next level until the next level completes its accumulation, which means the pulse should be delayed longer in the current level than in the next level. The delay cells should be designed in a manner that is shown in 3.6, where the T_{delay_l4} is the delay cell in Figure 3.3, which can be seen as the fourth stage of the pipeline. Besides, the registers in the capture layer are reset by the pulse signal propagated by the delay cell in the first level so that they can capture the next group of input spikes, so the time interval between two groups of spikes Δt should be

long enough to ensure that the registers are reset before the new spikes arrive, as shown in 3.7, where $T_{recovery}$ is the recovery time of the register.

$$T_{cq} + T_{comb} + T_{setup} \leq T_{delay} \tag{3.5}$$

$$T_{delay\_level1} \geq T_{delay\_level2} \geq T_{delay\_level3} \geq T_{delay\_level4} \tag{3.6}$$

$$\Delta t \geq T_{delay\_level1} + T_{recovery} \tag{3.7}$$

3.2.1.3 Simulation of adder tree based neuron

The Modelsim pre-synthesis simulation of the adder tree based self-timed neuron is shown in Figure 3.9. The input spikes are fed into the neuron every 10ns with random distribution, then the *req* signal is pulled up and the *start* pulse is generated by the capture layer, the pulse width of all the spikes is set to 100ps. The 1024-bit *control* vector determines the input data sent to the adder trees in the first level. The start pulse is propagated by the delay cells of each level which are set to 6ns, signal *done_l1*, signal *done_l2*, signal *finish* show the start pulse arrives at different levels at different timings, and the accumulation result of each level is sampled by these delayed start pulses. The result of the third level is sent out with the *finish* pulse when the accumulation is done. The result from the accumulator is accumulated again as the membrane potential of the neuron which is stored in *membrane_temp*. The threshold value in this simulation is set to 10000, when the membrane potential overcomes the threshold value, the *membrane_temp* is reset to 0 and the *finish* pulse is allowed to send out as the output spike.

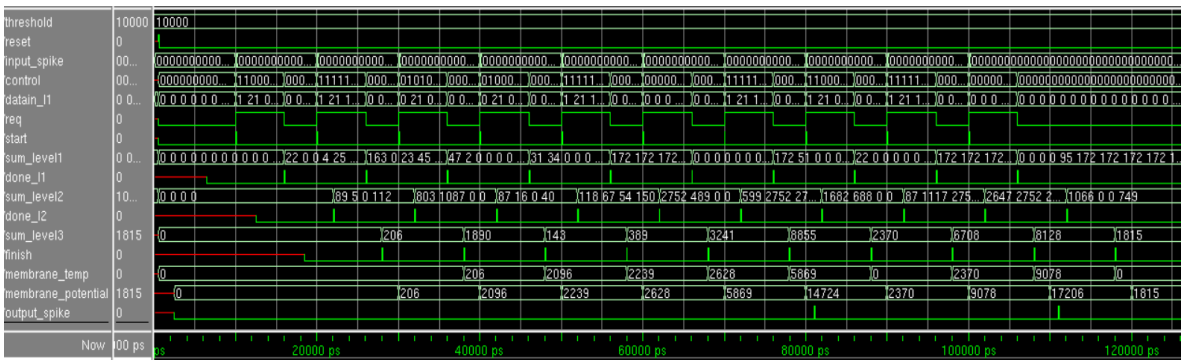


Figure 3.9: Pre-synthesis simulation of the adder tree based neuron

There are two problems with the adder tree based neuron design. The first one is the area consumption, plenty of adders are required for the adder trees, even though RCA consumes less area than other adders, it still consumes a big amount of area. Besides, big delay cells are needed to match the critical path of the adder trees, which may cost timing uncertainties. Since the big delay cells are composed of small delay cells in series, the variations of the small delay cells will be accumulated when the signal is propagating through the delay chains and lead to more timing uncertainties. So the

big delay cells are more unpredictable and uncontrollable compared with small delay cells. Both problems can be solved by the next type of design.

3.2.2 Time-multiplexed accumulator

In addition to using combinational logic to accumulate synaptic weights, an alternative accumulator design with time-multiplexed method is proposed. In this design, time-multiplexed compute units are used for weight accumulation instead of an adder tree. In the time-multiplexed accumulator, there is only one adder in each compute unit, the accumulation of weights is achieved by reusing this adder in time. In the absence of a global clock, the time-multiplexed compute unit generates an on-demand local clock, also called an event train, which warrants the correct pacing of adder accumulation. Therefore, it is the key to the self-timed operation of the neuron design. The primary component that enables this functionality is the two-line structure, where small delay cells are implemented instead of big delay cells to prevent timing uncertainties.

3.2.2.1 Compute unit design

The architecture of the time-multiplex compute unit is depicted in Figure 3.10. The whole process is triggered by the *start* pulse, the *start* pulse is obtained by ORing the input spikes with an or-gate tree. Meanwhile, the input spikes are sent to the clock ports of the registers, and the data ports of the registers are connected to logic '1', so if there is an input spike comes, the corresponding register will capture the spike and the output of register will be toggled to '1'. The register output will then be sent to the two demuxes in the corresponding stage of the two-line structure as the control signal.

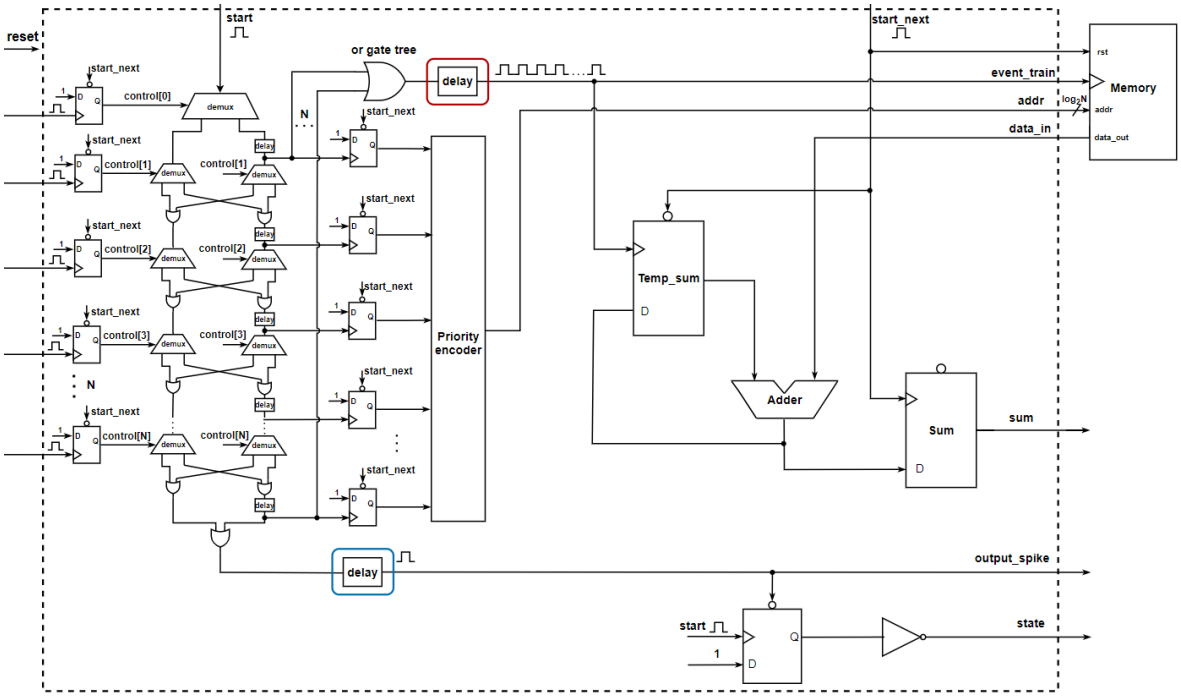


Figure 3.10: The schematic of time-multiplexed compute unit

The two-line structure consists of a bypass line and a delay line, on the delay line there are delay cells implemented to space the event pulses. The control signal of the demuxes determines the data flow inside the two-line structure. Figure 3.11 shows an example of data flow inside the two-line structure. Whenever *start* pulse is generated by the or-gate tree, it will flow through the two-line structure via a specific path determined by the input spike distribution, the demuxes on each stage will switch the path to the delay line if the input spike is captured by the corresponding register of that stage, if no input spike is captured, the path will be switched to the bypass line. As a result, the two-line structure will generate the same number of event pulses as input spikes during the *start* pulse flows through the two-line structure. Finally, another or-tree is applied to merge all the event pulses into a serial event train, which can be seen as the local self-generated clock signal for the compute unit.

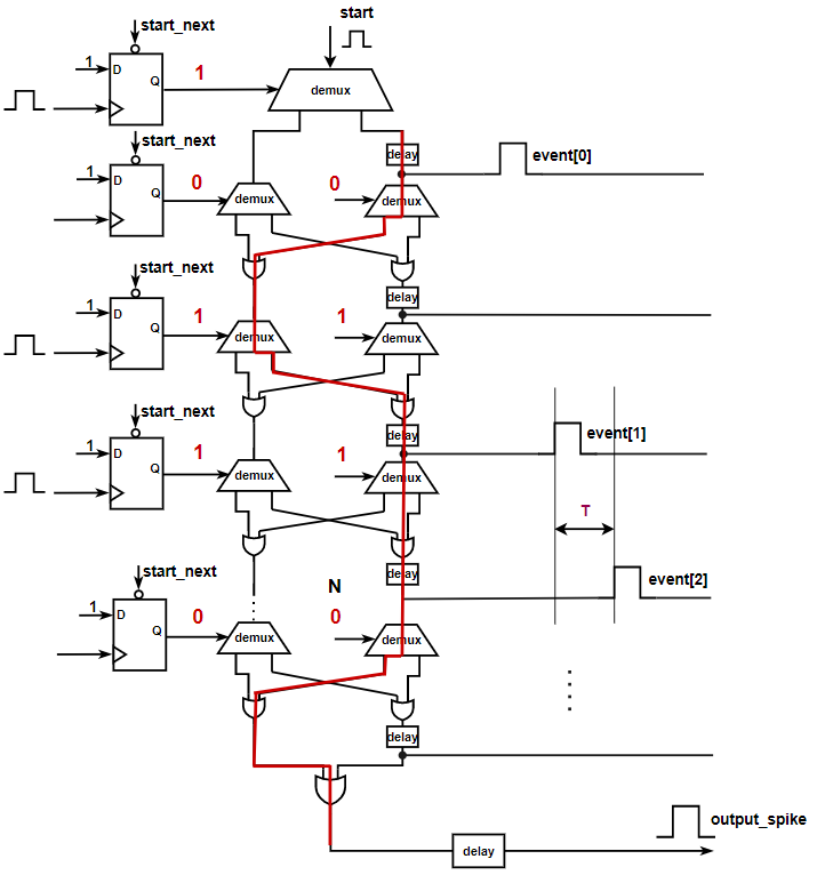


Figure 3.11: Data flow in the two-line structure

Event pulses will be captured sequentially by registers, and the output of these registers is connected to the input port of the priority encoder. The encoder will then generate the address of the corresponding weight for each input spike and send them to the outside weight memory. The self-generated clock will drive the weight memory to pick the synaptic weights activated by the incoming spikes and accumulate them by the adder. The time interval between two event pulses T is determined by the delay cells

on the delay line of the two-line structure, the delay of the delay cells should satisfy the condition below to guarantee that the output of the adder is already stable before the next data arrives. The left side of inequality 3.8 is the minimum time interval T between two event pulses, the right side of the inequality is the time for the adder output to be stable. If the delay of demuxes, logic gates and memory, set up time of registers is ignored, the delay of the delay cells in the two-line structure equals T, and it only depends on the critical path of the adder.

$$t_{demux} + t_{or} + t_{delay} > t_{memory} + t_{adder} + t_{setup} \tag{3.8}$$

In addition to the delay cells on the two-line structure, two extra delay cells are implemented, which are shown in the red box and blue box in Figure 3.10. The purpose of the delay cell in the red box is to guarantee that the event train can capture the address correctly, the rising edge of the event train should be located in the middle of the address valid region, as shown in Figure 3.12. The delay of registers, encoder and logic gates is ignored, so the delay in the red box t_{red} is set to T/2. The pulse flows through the two-line structure will finally be sent to the next level as the output spike of the compute unit after propagating by the delay cell in the blue box. The output spike is also a signal which indicates that the weight accumulation is completed, so the delay in the blue box t_{blue} should be calculated by equation 3.9 to synchronize with the event train.

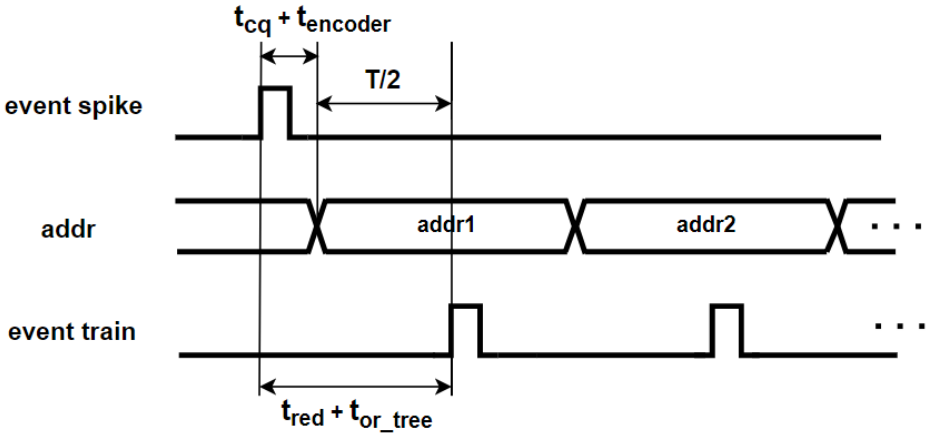


Figure 3.12: Timing analysis of t_{red}

$$t_{blue} = t_{red} + t_{or\_tree} + T \approx 3T/2 \tag{3.9}$$

The output spike of the compute unit is connected to the reset port of a register to pull up the *state* signal. When the *state* signal is '1' means the compute unit is idle, '0' means the compute unit is busy. The clock port of the register is connected to the *start* pulse of the compute unit, so the *state* signal will be pulled down by the *start* pulse at the beginning of each accumulation as shown in Figure 3.13. The start pulse of the next level is used to update the final result of the accumulation and reset the registers and input data to 0 so that they can be ready for the next accumulation.

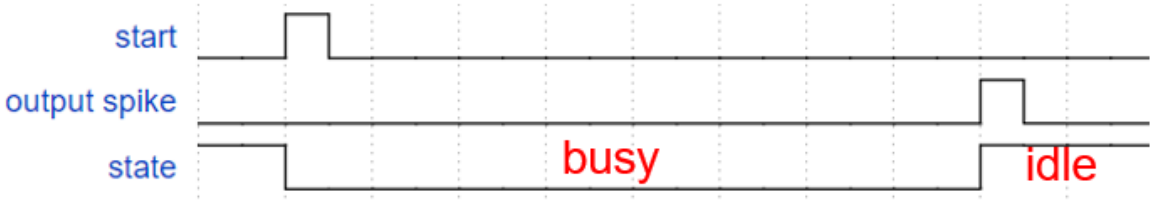


Figure 3.13: The state signal

3.2.2.2 Pipeline design and simulation

Based on the time-multiplexed compute unit design, another asynchronous pipeline design is proposed. In the former pipeline design, the delay cells of each pipeline stage should match the critical path delay, which is not suitable for the time-multiplexed compute unit, since the delay of the time-multiplexed compute unit depends on the number of input spikes, and the density of spikes is highly sparse in SNNs, so the probability of the time-multiplexed compute unit triggering the worst-case delay is very small(when all the input are fired), using the worst-case delay as the delay of each pipeline stage is not time-efficient. Therefore, a pulse-mode 4-phase handshake protocol for the interaction of neighboring levels is defined to reduce the unnecessary delay in each pipeline stage. An overview of the proposed pipeline is shown in Figure 3.14, the whole pipeline consists of 3 stages corresponding to the 3 levels in the accumulator, each level can receive new data from the previous level only when two conditions hold: the data from its previous level is ready(when *req* from the previous level is high);and its next level has stored the data from the current level(when *ack* from next level is high). The data transmitted between two levels come from the compute units, which include the accumulation results and the output spikes. The delay of each pipeline stage depends on the number of received spikes rather than the critical path delay, which results in an average case performance of the pipeline design.

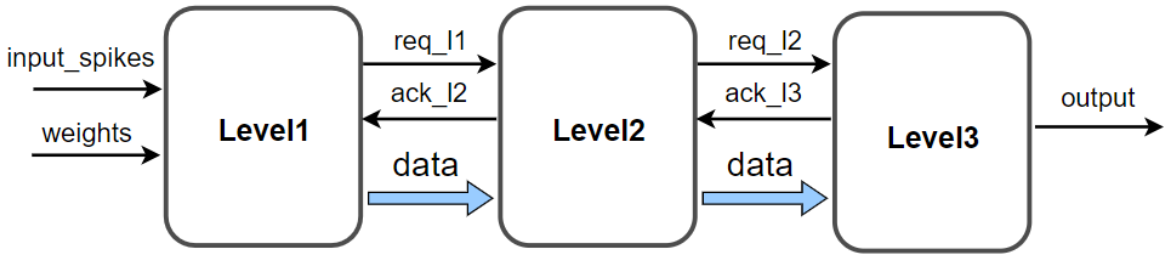


Figure 3.14: Handshake pipeline

The *req* and *ack* are the two key signals of the handshake protocols, both of them are generated by the registers. As discussed in 3.2.2.1, the *state* signal indicates whether the compute unit is accumulating or idle, in the proposed design, all the *state* signals in the same level are connected to the input of an and-gate tree, the output of the and-gate tree is the *done* signal of that level, so when all the *state* signal become '1',

the *done* signal will be pulled up, which indicates that the accumulation of the current level is done. As shown in Figure 3.15, The *done* signal is connected to the clock port of the register, the data port of the register is connected to '1', so whenever the register detects the rising edge of the *done* signal, the *req* signal will be pulled up and received by the next level. The start pulse of the next level is also used as the feedback signal which connects to the reset port of the register, so the *req* signal will be reset to zero when the next level starts the accumulation.

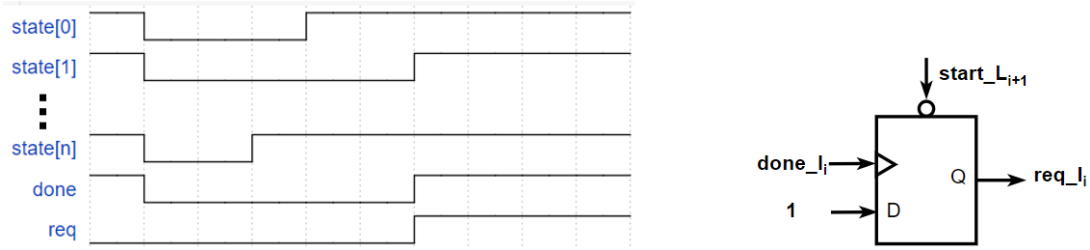


Figure 3.15: *Req* signal of level i

The *ack* signal is generated by a register with the start pulse of the previous level and current level, as shown in Figure 3.16. The *ack* signal will be pulled down when the rising edge of the start pulse of the previous level $start\_L_{i-1}$ is detected, and it will be pulled up by the start pulse of the current level $start\_L_i$. Since the start pulse is also used to update the result of the previous stage, the *ack* signal will be pulled up as soon as the current level receives the data from the previous level.

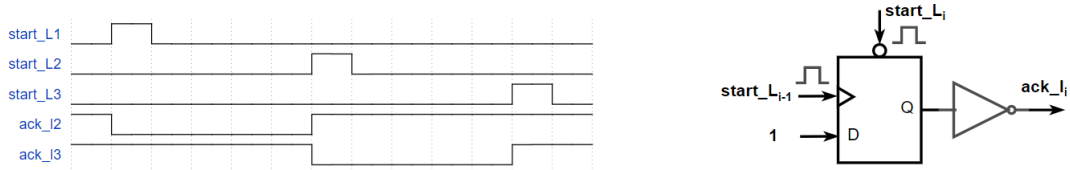


Figure 3.16: *Ack* signal of level i

The handshake process between two neighbouring levels is shown in Figure 3.17. When level i receives the input spikes from the previous level, the start pulse of level i $start\_L_i$ will be generated and the *ack* signal from the next level will be pulled down. When the accumulation of level i is finished, the *req* signal will be sent to the next level, if level i+1 is ready to receive the data from the previous level, the start pulse of level i+1 $start\_L_{i+1}$ will be generated, it will reset the *req* signal and pull up the *ack* signal. The data from level i will be captured at the rising edge of $start\_L_{i+1}$ for the accumulation of level i+1.

The schematic of the first level is shown in Figure 3.18. The 1024 bits input spikes will be first sent to the input spike filter, the input spike filter will decide whether the coming input spikes should be sent to the 64 compute units and generate the respective start pulses(*start_l1[n]*) for them. The proposed design is event-driven, only the compute unit that receives the input spikes will be triggered by the start pulse. If one of the 64 start pulses is fired, the *start_L1* will be fired, which indicates that

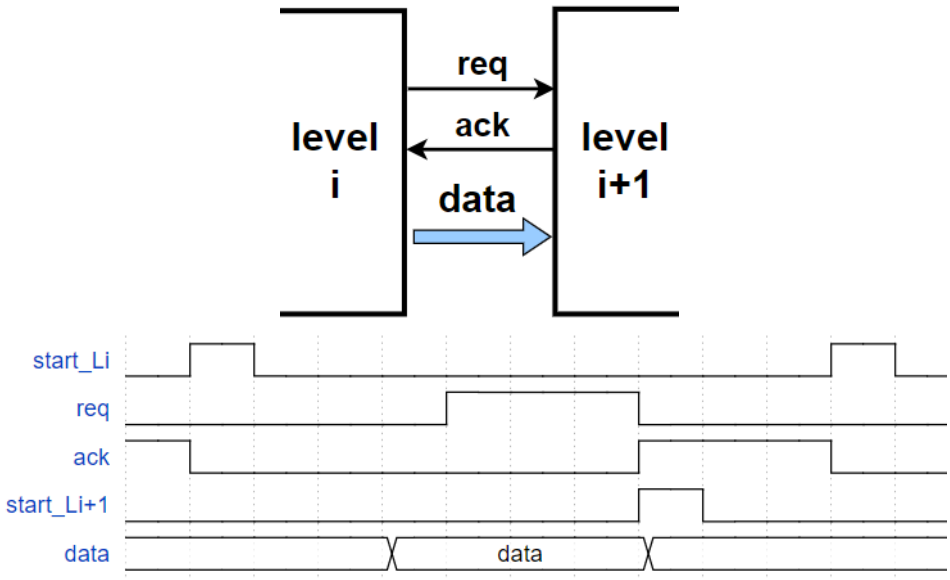


Figure 3.17: The process of the pulse-mode handshake pipeline

the accumulation of the first level has begun. In the worst case (all compute units are triggered), the 64 compute units will generate 64 event trains and 64 4-bit addresses to the corresponding weight memories, 64 6-bit signed weights are sent to the compute units as their respective input data. The output spikes of the compute units will be sent to the second level as the input spikes, a 10-bit signed result will be generated by each compute and stored in the arrays as the input data for the second level.

The role of the input spike filter is to prevent the compute units in the first level from receiving new spikes during the accumulation process. The schematic of the input spike filter is shown in Figure 3.19, the input spikes are separated into 64 groups, they can be sent to the compute units only when the ack signal from the second level is high. The start pulse of each compute unit is generated by ORing the 16 input spikes. The use of the delay cells is to guarantee that the input spikes arrive compute unit before the start pulse, so that the path of the two-line structure is stable when the start pulse comes.

The Modelsim simulation of the first level is shown in Figure 3.20. The signals in the red box belong to one of the compute units in the first level, which provides the worst-case delay, so the *done_l1* signal is pulled up with its *state* signal at the same time. In this simulation, the pulse width of the spikes is 100ps, the time interval between two events T is set to 2ns, the output of each accumulation is assumed to be stable after 1ns. The input spikes are captured by the registers and result in a 16-bit control signal '0000000101011111', which means 7 spikes are received by the compute unit. Then 7 event pulses will be generated by the two-line structure to drive the weight memory and registers. The priority encoder will generate the corresponding addresses for the input spikes and send them to the weight memory. The temporary result from the adder will be stored in the register driven by *event_train*. When the accumulation is finished, the *state* signal will be pulled up by the *output_spike*. The *req_l1* will be pulled up because

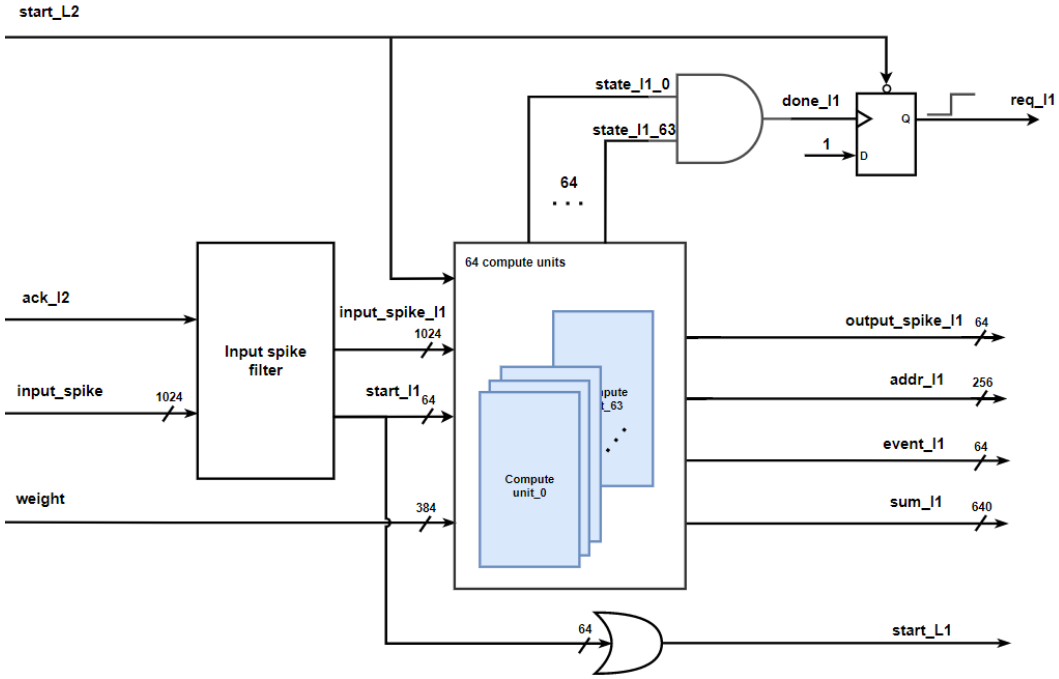


Figure 3.18: Digital implementation of the first level

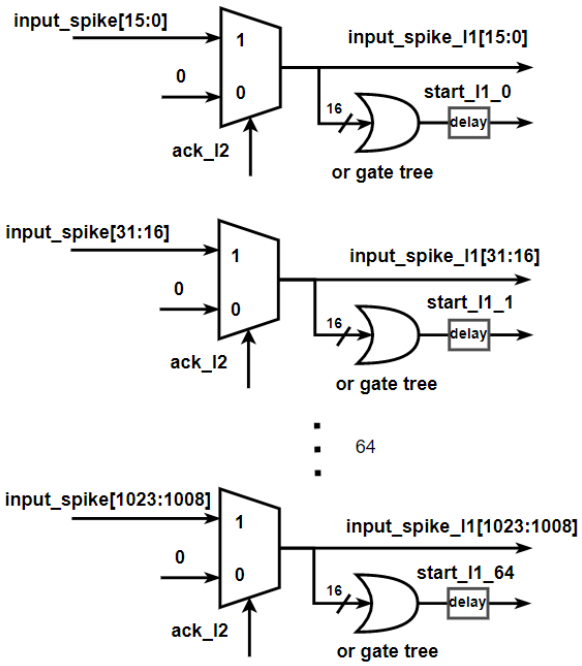


Figure 3.19: Input spike filter

all the compute units in the first level will be idle by this time. The start pulse of the second level *start_L2* will store the accumulation result and reset the registers to 0.

The schematic of the second level is shown in Figure 3.21, the start pulse signal

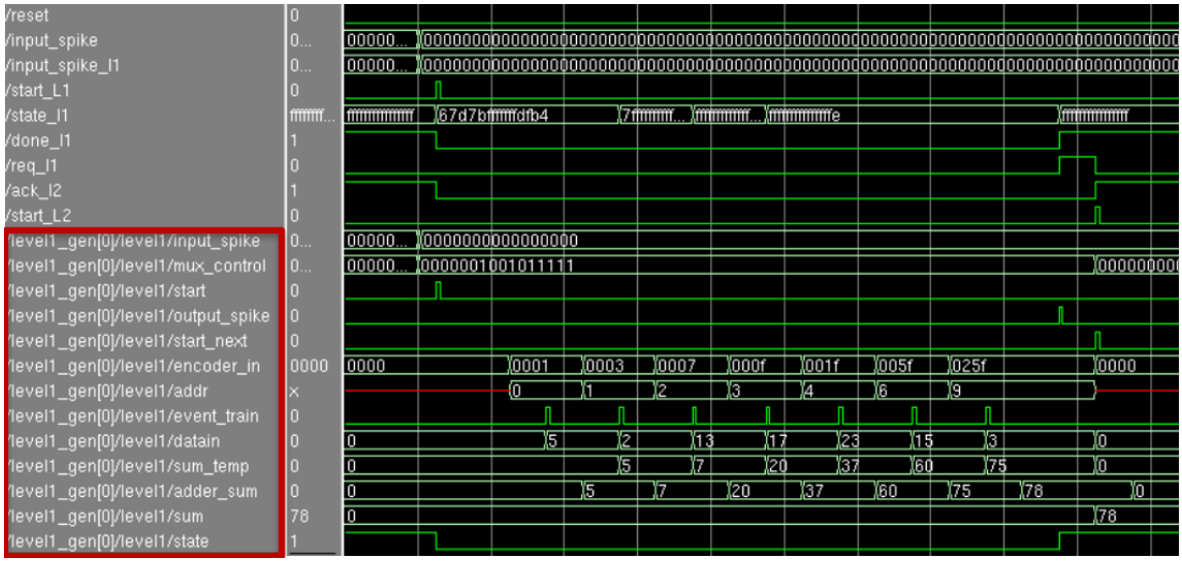


Figure 3.20: Pre-synthesis simulation of the first level

of the second level *start_L2* will be generated when *req_l1* and *ack_l3* are both '1', then it will pull up the *ack_l2* and feedback to the previous level to reset the *req_l1* and update the accumulation results. The pulse width of the *start_L2* is determined by the delay cell in the yellow box. Since the output spikes from the first level are not synchronized, so a spike synchronizer is implemented to synchronize the incoming spikes and generate the input spikes and start pulses *start_l2[n]* to the compute units. The delay cell in the green box provides the delay to the feedback loop so that the register which generates *req_l1* is able to reset itself, it also guarantees that the last output spike or spikes arrive at the spike synchronizer earlier than *start_L2*, the delay in the green box should be at least larger than the pulse width so that the start pulse can be generated correctly(assumed that the pulse width is larger than removal time of the register). Each compute unit reads the input data from the corresponding array, which stores the results of the previous level. The four state signals of the compute units are connected to the and tree, when all the state signals become '1', the request signal of the second level *req_l2* will be pulled up. Each compute unit will send an output spike and a 14-bit signed result to the next level when their accumulation is done.

The input spikes of the compute unit should be synchronized otherwise the compute units cannot be driven correctly. The schematic of the spike synchronizer is shown in Figure 3.22, the output spikes from the previous level are captured by registers, the flag signal will be pulled up if an output spike is generated by the compute unit from the previous level. If the flag is '1', then the start pulse of the second level is allowed to pass the mux and sent to the compute units as the input spikes. The 64 input spikes are separated into four groups, corresponding to the four compute units in the second level. The start pulse of each compute unit is generated by ORing the 16 input spikes. The flag signal will be reset by the start pulse of the first level.

Figure 3.23 shows the Modelsim simulation result of the second level, the signals in the red box belong to one of the compute units in the second level, its input flag and

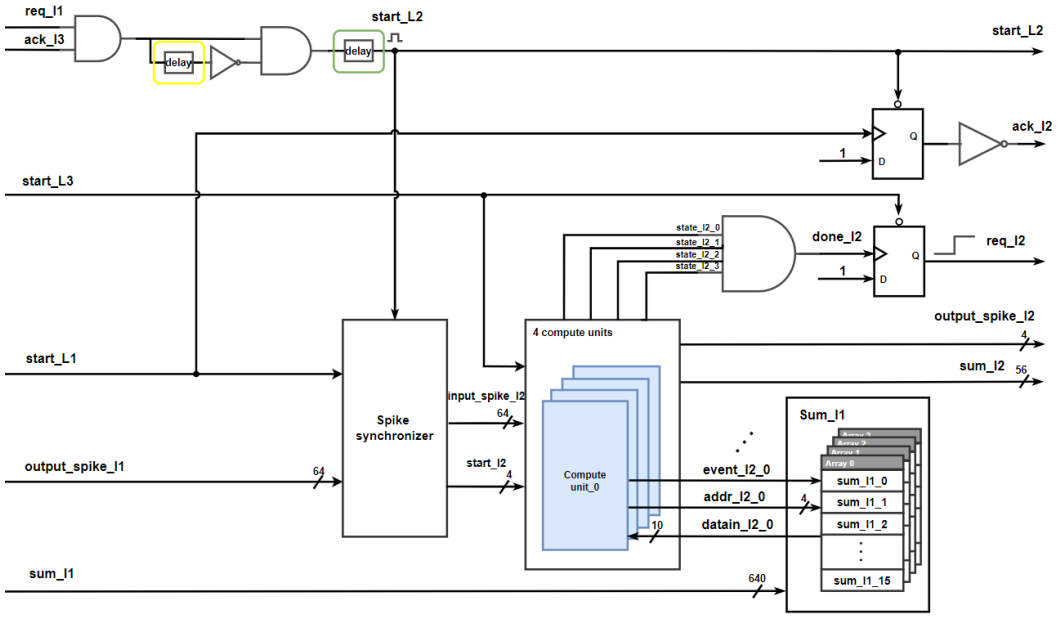


Figure 3.21: Digital implementation of the second level

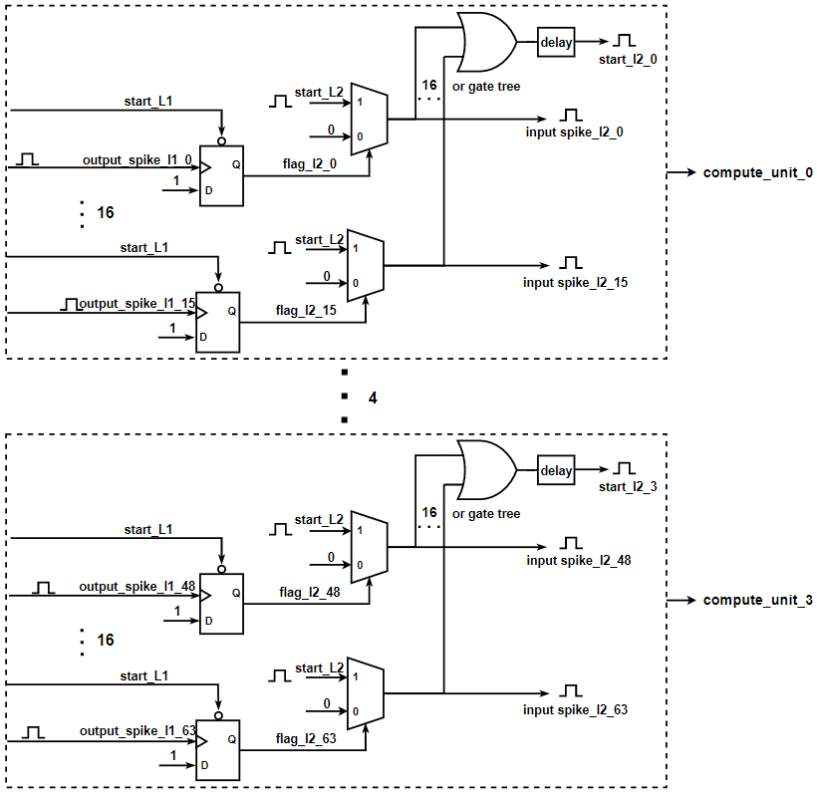


Figure 3.22: The spike synchronizer

the input data array are shown in the blue box. There are five '1' in the flag signal, which means five compute units in the previous level are driven by the input spikes, their results '48', '17', '31', '20', '15' are stored in the array, then the compute unit in the second level generates five events to read the input data from the array and accumulate them. The request signal *req_l2* is pulled up when all the state signals of the four compute units become '1', which means the accumulation of the second level is done, the delay in the feedback loop is set to 1ns, so the *req_l2* is reset to '0' after 1ns.

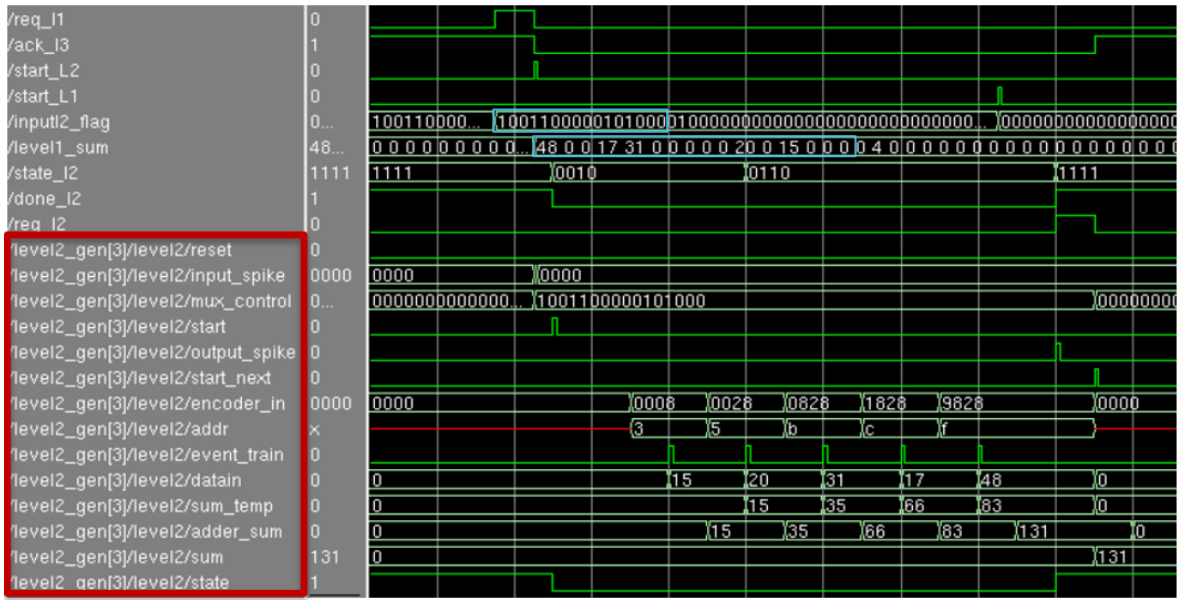


Figure 3.23: Pre-synthesis simulation of the second level

The schematic of the third level is shown in Figure 3.24. There is only one compute unit in this level, receiving the 4-bit output spike from the previous level. Since it is the last level of the accumulator, there is no feedback signal from the next level, so the state signal of the compute unit replaces the ack signal to generate the start pulse *start_L3*, the start pulse can be generated only when the compute unit is idle. The output spike of the compute unit is sent out directly as the finish signal of the accumulator, which indicates that the accumulation of the whole accumulator is done, a 16-bit accumulation result will be added to the membrane potential. The ack signal of the third level *ack_l3* will be sent back to the previous level as the feedback signal. The simulation of the third level is shown in Figure 3.25

In order to guarantee that there is no timing violation in the membrane potential accumulation logic, the minimum time interval between two finish spikes should meet inequality 3.1. The minimum time interval occurs only when two conditions are met, first, the data from the previous level should be ready, which means the *req_2* signal should be '1' when *state_l3* signal is pulled up by the finish spike, second, the third level provides the shortest delay, which means the compute unit only receives one output spike from the previous level, because the delay of the compute unit depends on the number of received spikes. The shortest delay of the third level can be calculated by

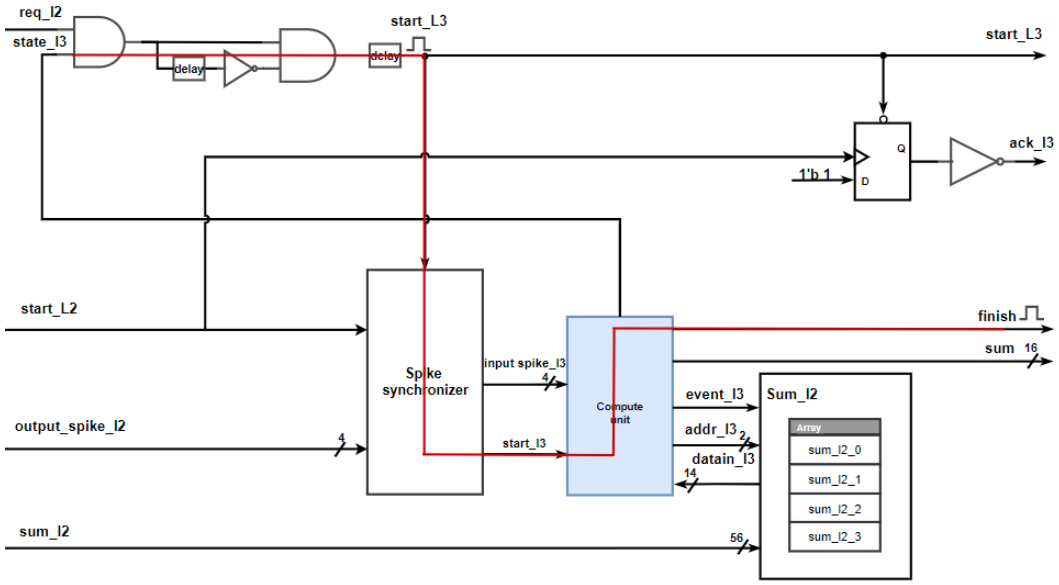


Figure 3.24: Digital implementation of the third level

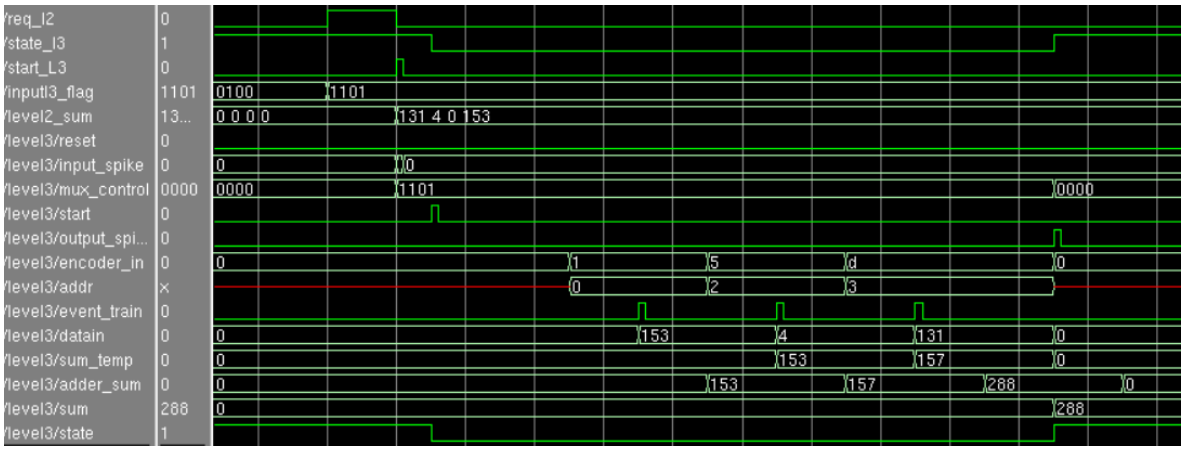


Figure 3.25: Pre-synthesis simulation of the third level

adding all the delays in the red path in Figure 3.24. In order to simplify the analysis, only the delay in the compute unit is considered. The shortest data path in the compute unit is shown in Figure 3.26, if we ignore the delay of the demuxes and or gates, the shortest delay can be calculated by adding the delay of the two delay cells on the path, which is 2.5T. We can replace the t_{finish} in inequality 3.1 with 2.5T, and combined with inequality 3.8, then ignore the delay of registers, muxes, demuxes, memory and logic gates, we can get two constraints for the delay cells on the two-line structure:

$$T > t_{adder\_cu} \tag{3.10}$$

$$T > \frac{2}{5}(t_{adder\_mp} + t_{comparator}) \tag{3.11}$$

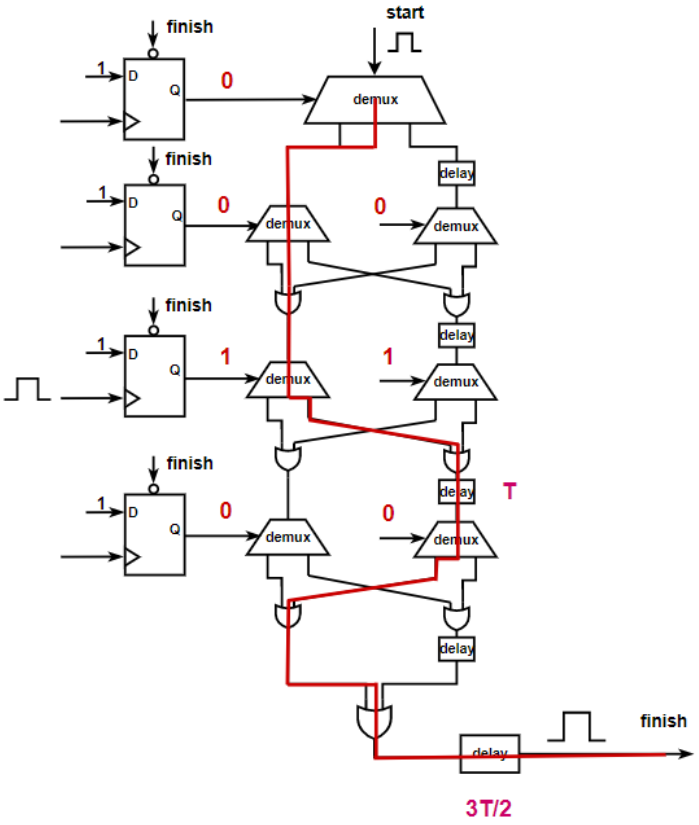


Figure 3.26: Shortest path in the compute unit in level3(one input spike received)

t_{adder_cu} in inequality 3.10 represents the critical path delay of the adder in the compute unit, t_{adder_mp} in inequality 3.11 represents the critical path delay of the adder used for the membrane potential accumulation. T can be different in different levels, inequality 3.11 only applies to the delay cells in the third level. It can be concluded that in the first and second levels, the delay of the delay cells in the compute units depends on the critical path of the adder in the compute unit, while in the third level, the delay of the delay cells in the compute units not only depends on the critical path of the adder in the compute unit, but also the adder for the membrane accumulation.

3.2.2.3 Simulation of time-multiplexed neuron

The Modelsim simulation of the time-multiplexed self-timed neuron is shown in Figure 3.27. The 1024-bit input spikes are fed into neuron every 40 ns, the compute units that receive the input spikes are triggered and then start the accumulation. The result of the first level is updated by the *start_L2* signal and stored in the array *level1_sum*, same for the second and third levels. The result of the third level is sent out with the finish spike when the accumulation of the input spikes is done. The result from the accumulator is accumulated as the membrane potential which is stored in *membrane_temp* driven by the finish spike. The threshold value in this simulation is set to 3000, when the membrane potential overcomes the threshold, *membrane_temp* is reset to 0 and the finish spike is allowed to send out as the output spike.

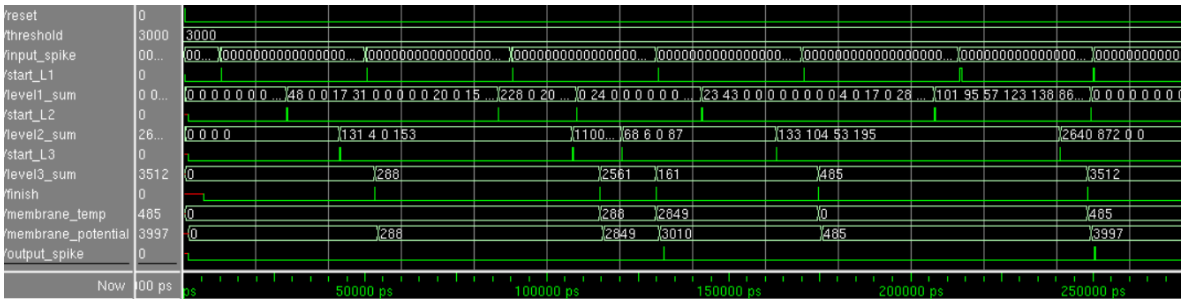


Figure 3.27: Pre-synthesis simulation of the time-multiplexed neuron

3.3 Summary

In this chapter, the hardware implementation of the self-timed neuron based on two types of accumulator is illustrated and some critical timing constraints are given, the functionality has been verified by Modelsim pre-synthesis simulation. The synthesis result of the self-timed neuron will be discussed in the next chapter.

4 Results

This chapter presents the synthesis result of the two proposed self-timed neurons. Both designs are described by using SystemVerilog language and synthesized in TSMC 28nm technology by Cadence genus, the power, area and delay performance of the two designs will be evaluated and compared, trade-off analysis will be discussed as well.

4.1 Post-synthesis simulation

The post-synthesis simulation is done by Xcelium with the netlist and sdf file provided by the synthesis tool. The functioning of the proposed self-timed neurons depends on the delay cells providing the correct delay to ensure a sufficiently long interval between pulses. In order to find the proper delay of the delay cells, the critical path delay of the related combinational logic should be determined first. The virtual clock is used to wrap the combinational logic so that the synthesis tool is able to analyze the timing path of the constrained combinational logic. As the common part of both designs, the delay of the delay cell which is used to propagate the finish pulse(Figure 3.3) is analyzed first. A 17-bit ripple carry adder is implemented to accumulate the results from the accumulator. The timing report below shows that the critical path delay from the output port of the accumulator to the output port of the comparator is about 1.75ns, so a 2ns delay is implemented to provide a safe margin. The delay cells are provided by the TSMC 28nm standard cell library, *set_dont_touch* command is used to prevent the delay cell from being optimized by the synthesis tool.

Timing Point	Flags	Arc	Edge	Cell	Fanout	Load (fF)	Trans (ps)	Delay (ps)	Arrival (ps)	Instance Location
input2[0]	-	-	F	(arrival)	2	2.2	0	0	1	(-,-)
add_93_21_g429__7410/Z	-	A2->Z	F	AN2D0BWP30P140	2	3.0	36	36	38	(-,-)
add_93_21_g427__2346/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	93	131	(-,-)
add_93_21_g426__2883/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	227	(-,-)
add_93_21_g425__9945/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	323	(-,-)
add_93_21_g424__9315/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	419	(-,-)
add_93_21_g423__6161/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	516	(-,-)
add_93_21_g422__4733/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	612	(-,-)
add_93_21_g421__7482/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	708	(-,-)
add_93_21_g420__5115/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	805	(-,-)
add_93_21_g419__1881/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	901	(-,-)
add_93_21_g418__6131/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	997	(-,-)
add_93_21_g417__7098/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	1094	(-,-)
add_93_21_g416__8246/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	1190	(-,-)
add_93_21_g415__5122/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	1286	(-,-)
add_93_21_g414__1705/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	1382	(-,-)
add_93_21_g413__2802/CO	-	CI->CO	F	FA1D0BWP30P140	1	1.3	36	90	1472	(-,-)
add_93_21_g412/ZN	-	I->ZN	R	CKND0BWP30P140	1	2.1	43	42	1515	(-,-)
add_93_21_g411__1617/S	-	CI->S	F	FA1D0BWP30P140	1	1.3	34	123	1638	(-,-)
add_93_21_g410/ZN	-	I->ZN	R	CKND0BWP30P140	2	1.7	37	38	1676	(-,-)
g461__6783/ZN	-	A1->ZN	F	NR2D0BWP30P140	1	1.3	26	26	1702	(-,-)
g455__2398/ZN	-	A1->ZN	R	AOI31D0BWP30P140	1	1.0	68	46	1748	(-,-)
out	<<<	-	R	(port)	-	-	-	0	1748	(-,-)

Figure 4.1: The critical path from the output port of the accumulator to the output port of the comparator

Besides, the pulse width of all the pulse signals should be big enough so that they can be propagated successfully. A pulse signal will be narrowed when it propagates through the logic gates if the pulse width is not big enough, and the pulse will be filtered by the delay cell if its pulse width is smaller than the inertial delay of the delay cell[30]. In the proposed designs, the pulse width of the generated pulse signals is about 700ps, which is big enough to propagate through the logic gates and delay cells successfully.

4.1.1 Post-synthesis simulation of the adder tree based neuron

The delay of delay cells in the adder tree based accumulator is related to the critical paths of the adder trees in each level. Among the three levels of the accumulator, the adder trees in the second level have the longest critical path because they have the most stages and larger ripple carry adders, the critical path delay of the second level is about 1.9ns as shown in Figure 4.2, so the propagation delay of the delay cells in each pipeline stage is set to about 2.5ns, which is also the cycle time of the pipeline.

Timing Point	Flags	Arc	Edge	Cell	Fanout	Load (fF)	Trans (ps)	Delay (ps)	Arrival (ps)	Instance Location
sum_level1[10]	-	-	F	(arrival)	2	2.2	0	0	50	(-,-)
satge1_gen_1_adder_stage1/g675__1666/Z	-	A2->Z	F	AN2D0BWP30P140	2	3.0	36	36	86	(-,-)
satge1_gen_1_adder_stage1/g673__2883/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	93	180	(-,-)
satge1_gen_1_adder_stage1/g672__9945/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	276	(-,-)
satge1_gen_1_adder_stage1/g671__9315/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	372	(-,-)
satge1_gen_1_adder_stage1/g670__6161/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	468	(-,-)
satge1_gen_1_adder_stage1/g669__4733/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	565	(-,-)
satge1_gen_1_adder_stage1/g668__7482/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	661	(-,-)
satge1_gen_1_adder_stage1/g667__5115/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	757	(-,-)
satge1_gen_1_adder_stage1/g666__1881/CO	-	CI->CO	F	FA1D0BWP30P140	1	1.3	36	90	847	(-,-)
satge1_gen_1_adder_stage1/g665/ZN	-	I->ZN	R	CKND0BWP30P140	1	2.1	43	42	890	(-,-)
satge1_gen_1_adder_stage1/g664__6131/S	-	CI->S	F	FA1D0BWP30P140	1	1.3	34	123	1013	(-,-)
satge1_gen_1_adder_stage1/g663/ZN	-	I->ZN	R	CKND0BWP30P140	1	2.6	50	46	1059	(-,-)
stage2_gen_1_adder_stage2/g699__6161/CO	-	A->CO	R	FA1D0BWP30P140	1	1.3	40	87	1146	(-,-)
stage2_gen_1_adder_stage2/g698/ZN	-	I->ZN	F	CKND0BWP30P140	1	2.1	35	35	1181	(-,-)
stage2_gen_1_adder_stage2/g697__4733/S	-	CI->S	R	FA1D0BWP30P140	1	1.3	40	137	1318	(-,-)
stage2_gen_1_adder_stage2/g696/ZN	-	I->ZN	F	CKND0BWP30P140	1	2.6	41	39	1357	(-,-)
stage3_gen_1_adder_stage3/g793__5122/CO	-	A->CO	F	FA1D0BWP30P140	1	1.3	36	99	1456	(-,-)
stage3_gen_1_adder_stage3/g792/ZN	-	I->ZN	R	CKND0BWP30P140	1	2.1	43	42	1499	(-,-)
stage3_gen_1_adder_stage3/g791__1705/S	-	CI->S	F	FA1D0BWP30P140	1	1.3	34	123	1622	(-,-)
stage3_gen_1_adder_stage3/g790/ZN	-	I->ZN	R	CKND0BWP30P140	1	2.6	50	46	1668	(-,-)
adder_stage4/g957__5107/CO	-	A->CO	R	FA1D0BWP30P140	1	1.3	40	87	1755	(-,-)
adder_stage4/g956/ZN	-	I->ZN	F	CKND0BWP30P140	1	2.1	35	35	1790	(-,-)
adder_stage4/g955__2398/S	-	CI->S	R	FA1D0BWP30P140	1	1.3	40	137	1927	(-,-)
adder_stage4/g954/ZN	-	I->ZN	F	CKND0BWP30P140	1	1.0	23	26	1953	(-,-)
out[12]	<<<	-	F	(port)	-	-	-	0	1953	(-,-)

Figure 4.2: The critical path of the adder tree in the second level

The post-synthesis simulation of the adder tree based neuron is shown in Figure 4.3, input spikes are sent to the neuron every 5ns, the start pulse is generated when the input spikes are detected and then propagated to the next level by the delay cells at each level. The time interval between two markers is the total delay of the accumulator, which is about 7.6ns. The output of the accumulator *sum_level3* is accumulated again as the membrane potential, the finish pulse is propagated by a 2ns delay cell and is allowed to send out as the output spike only when the membrane potential overcomes the threshold(10000), so the total delay of the neuron is about 9.6ns.

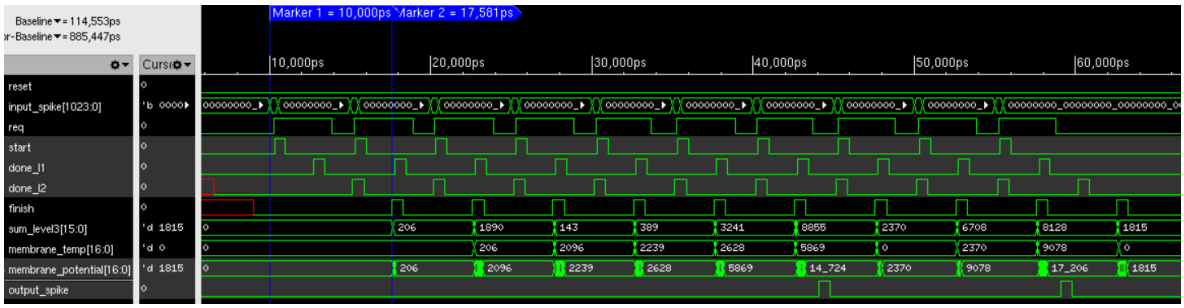


Figure 4.3: The post-synthesis simulation of the adder tree based neuron

4.1.2 Post-synthesis simulation of the time-multiplexed neuron

In order to simplify the design, the delay cells with the same delay are implemented in the compute units of all three levels, the delay of these delay cells is related to the critical path delay of the adders in the third level, because they have the longest critical path. Ripple carry adders are implemented in the compute units for the accumulation, the critical path delay of the ripple carry adder in the third level is about 1.6ns as shown in Figure 4.4, so the delay of the delay cells in the two-line structure is set to about 2ns. Figure 4.5 shows the post-synthesis simulation of the time-multiplexed compute unit with the worst-case stimulus(all inputs are fired), 16 event pulses are generated. The time interval between two markers is the worst-case cycle time of the pulse-mode handshake pipeline, which is about 36.4ns. The synthesis doesn't include the weight memory, so the delay of the weight memory is not annotated in the sdf file.

Timing Point	Flags	Arc	Edge	Cell	Fanout	Load (fF)	Trans (ps)	Delay (ps)	Arrival (ps)	Instance Location
input2[0]	-	-	F	(arrival)	2	2.2	0	0	50	(-,-)
add_12_19_g404__5115/Z	-	A2->Z	F	AN2D0BWP30P140	2	3.0	36	36	86	(-,-)
add_12_19_g402__6131/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	93	180	(-,-)
add_12_19_g401__7098/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	276	(-,-)
add_12_19_g400__8246/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	372	(-,-)
add_12_19_g399__5122/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	468	(-,-)
add_12_19_g398__1705/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	565	(-,-)
add_12_19_g397__2802/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	661	(-,-)
add_12_19_g396__1617/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	757	(-,-)
add_12_19_g395__3680/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	854	(-,-)
add_12_19_g394__6783/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	950	(-,-)
add_12_19_g393__5526/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	1046	(-,-)
add_12_19_g392__8428/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	1142	(-,-)
add_12_19_g391__4319/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	1239	(-,-)
add_12_19_g390__6260/CO	-	CI->CO	F	FA1D0BWP30P140	1	2.1	43	96	1335	(-,-)
add_12_19_g389__5107/CO	-	CI->CO	F	FA1D0BWP30P140	1	1.3	36	90	1425	(-,-)
add_12_19_g388/ZN	-	I->ZN	R	CKND0BWP30P140	1	2.1	43	42	1468	(-,-)
add_12_19_g387__2398/S	-	CI->S	F	FA1D0BWP30P140	1	1.3	34	123	1591	(-,-)
add_12_19_g386/ZN	-	I->ZN	R	CKND0BWP30P140	1	1.0	26	31	1622	(-,-)
sum[15]	<<<	-	R	(port)	-	-	-	0	1622	(-,-)

Figure 4.4: The critical path of the ripple carry adder in the third level

The post-synthesis simulation of the time-multiplexed neuron is shown in Figure 4.6 The elasticity of the handshake asynchronous pipeline allows input spikes to arrive at any time, but if the input rate is high, the time interval between two groups of input spikes will be tight, which results in that the input spikes may be filtered by the input spike filter as shown in Figure 4.6, the two groups of input spikes indicated by the arrows are filtered and not able to sent to the compute units in the first level. The

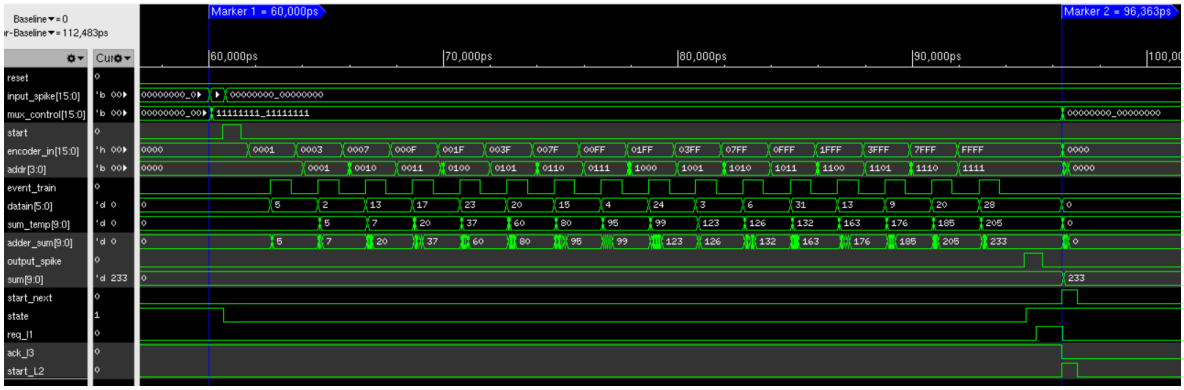


Figure 4.5: The post-synthesis simulation time-multiplexed compute unit

total delay of the time-multiplexed neuron depends on the number and distribution of the input spikes, it is smaller when the input spikes are few and dispersed. The time between two markers in Figure 4.6 is the worst-case delay of the time-multiplexed neuron, it starts from the moment input spikes arrive to the rising edge of the output spike, which takes about 83.3ns. The output spike is allowed to send out only when the membrane potential overcomes the threshold value(3000).

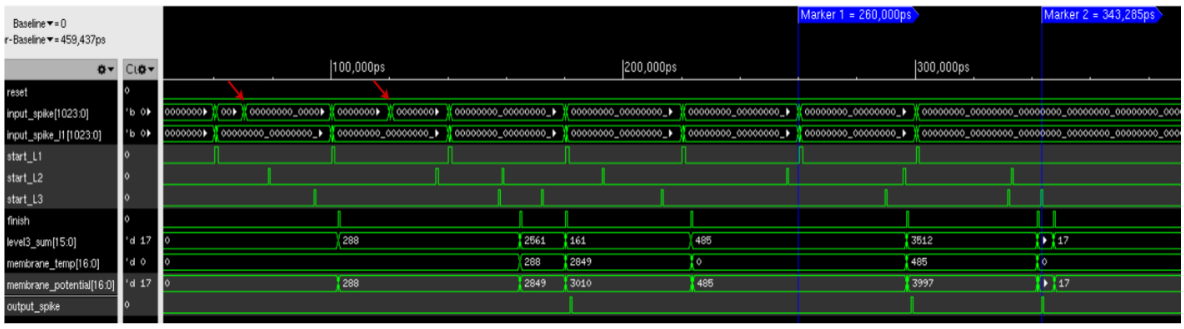


Figure 4.6: The post-synthesis simulation time-multiplexed neuron

4.2 Synthesis results

The proposed two neurons are synthesized using TSMC 28nm technology, the obtained results depicted in table 4.1 show the comparison between two designs in terms of area, power and latency performance. The area is reported in μm^2 and kilo gate equivalents(kGE), where a gate equivalent corresponds to the area of a nominal drive strength 2-input NAND (or NOR) gate in the standard cell library.

The toggle count format(.tcf) file is used for accurate power estimation, it contains switching activity in the form of toggle count information and static probability for the specific stimulus. The .tcf file can be generated by running the testbench on the RTL or synthesized gate-level netlist. The power performance of the proposed designs is evaluated by giving the average dynamic energy consumption per spike of the neurons.

The process of the power estimation is shown in Figure 4.7, the gate-level netlist and the sdf file generated by the synthesis tool are used for the input file for the post-synthesis simulation, then the simulator runs the simulation and generates the tcf file with the accurate switching activity of the input stimulus. Finally, the switching activity is annotated back to the synthesis tool by loading the tcf file, the power report will give the accurate power consumption of this simulation. The average energy consumption per spike is obtained by multiplying the dynamic power with the runtime and then dividing by the number of input spikes.

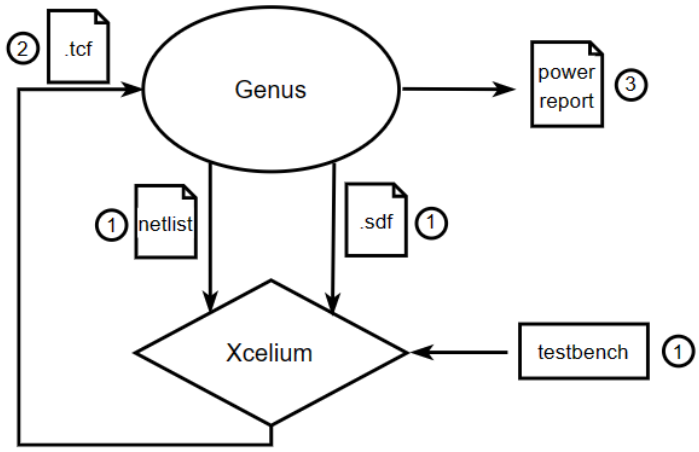


Figure 4.7: The flow of the power estimation

	Adder tree based	Time-multiplexed
Total area(μm^2)	31514.83	38109.9
Total area(kGE)	83.37	100.82
Cell area	59.47	83.57
Net area	23.9	17.25
Leakage power(nW)	5414.65	8037.67
Average dynamic energy consumption per spike(pJ)	0.055	0.23
Delay(ns)	9.6	83.1(worst-case)
Throughput(Gbps)	400	27.5
Operating voltage(V)	0.72	0.72

Table 4.1: Sybthesis result of two proposed neurons

It can be seen from table 4.1 that both designs can achieve competitive power consumption and delay with a relatively large area. Since both designs are fully event-driven, so they consume dynamic power only when input spikes are received. The

adder tree based neuron shows a better performance in power, area, and latency than the time-multiplexed neuron. Although the purpose of the time-multiplexed neuron is to reduce the number of adders to obtain a lower area, the area consumption of the time-multiplexed neuron is larger than the adder tree based neuron. This is due to the fact that the generation of the local clock requires plenty of delay cells which consume even more area than the area saved from the adders. Figure 4.8 shows the gate reports of two designs from the synthesis tool, the area consumption of the delay cells is marked by red boxes. It can be seen from the report that the cell area of the time-multiplexed neuron is dominated by buffers, which are the delay cells. Nearly half of the area comes from the delay cells, only 22% of the area comes from the logic gates. In contrast, the area is dominated by the logic gates in the adder tree based design, only 0.3% of the cell area is delay cells, therefore, even though it consumes more than twice the area of logic gates than the time-multiplexed design, its total cell area is still smaller. Besides, time-multiplexed design requires more registers to store the temporary data, which also brings extra area. Since the delay cells in the standard cell library are energy-consuming, so the large amount of delay cells in the time-multiplexed design also makes it consume more leakage power and dynamic power than the adder tree based design. The throughput shows the maximum number of spikes that the neuron can receive in one second, which is given in Gigabits per second(Gbps). Since the pipeline design in the adder tree based neuron has a smaller cycle time, so it has higher throughput than the time-multiplexed design.

Type	Instances	Area	Area %	Leakage Power (nW)	Leakage Power %
sequential	3820	8700.048	27.5	2558.357	33.8
inverter	2480	626.850	2.0	198.655	2.6
buffer	4862	15313.284	48.5	2950.410	39.0
logic	11655	6950.034	22.0	1852.783	24.5
physical_cells	0	0.000	0.0	0.000	0.0
total	22817	31590.216	100.0	7560.204	100.0

(a) Time-multiplexed neuron

Type	Instances	Area	Area %	Leakage Power (nW)	Leakage Power %
sequential	1753	3975.804	17.7	1202.464	23.9
inverter	3140	791.280	3.5	206.973	4.1
buffer	21	65.142	0.3	12.660	0.3
logic	14706	17647.056	78.5	3617.713	71.8
physical_cells	0	0.000	0.0	0.000	0.0
total	19620	22479.282	100.0	5039.810	100.0

(b) Adder tree based neuron

Figure 4.8: The gate reports of the two proposed designs

4.3 Performance of different hierarchical structures

The compute units in the two neuron designs are parameterizable, which provides flexibility and customizability for different accumulator hierarchy designs. The proposed designs divide the input spikes into 64 subgroups, each 16 spikes are handled by a

16-input compute unit, which results in a 3-level hierarchical accumulator. In order to explore the effect of the hierarchical structure on the neuron's performance, this work also provides a 2-level hierarchical accumulator consisting of multiple 32-input compute units. 1024 input spikes are divided into 32 subgroups, with 32 spikes in each subgroup. The overview structure of the 2-level accumulator is shown in Figure 4.9. The comparison of the performance of two neuron designs at N=16 and N=32 is shown in table 4.2 and table4.3, where N is the number of spikes in each subgroup.

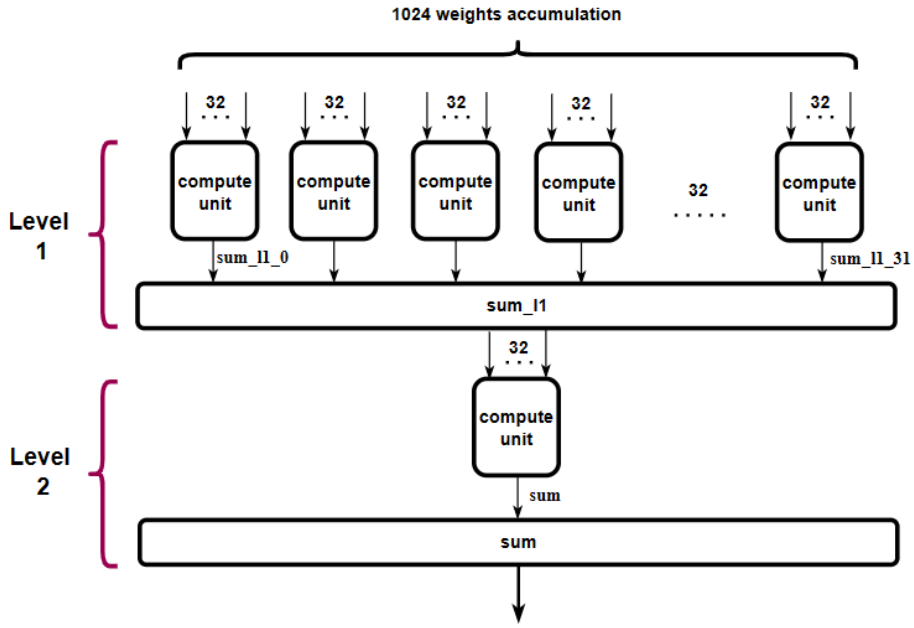


Figure 4.9: Hierarchical structure of the accumulator with N=32

	N=16	N=32
Total area(μm^2)	38109.9	33456.36
Total area(kGE)	100.82	88.5
Cell area	83.57	72.99
Net area	17.25	15.51
Leakage power(nW)	8034.67	7069.37
Average dynamic energy consumption per spike(pJ)	0.23	0.24
Throughput(Gbps)	27.5	14.6
Delay(ns)	83.1(worst-case)	135.6(worst-case)

Table 4.2: Performance comparison between N=16 and N=32 for time-multiplexed neuron

For the time-multiplexed neuron, the area is reduced by 12% at N=32, because fewer delay cells and registers are required. The changing of the structure has little effect on power consumption, however, it has a significant effect on the total delay. Since the worst-case delay of the time-multiplexed compute unit is almost doubled when the number of input spikes changes from 16 to 32, even with fewer levels in the accumulator, the total delay is still increased. The throughput of the time-multiplexed neuron when n=32 is only nearly half of that when n=16 because the cycle time of the pipeline is doubled.

	N=16	N=32
Total area(μm^2)	31514.83	30626.31
Total area(kGE)	83.37	81.02
Cell area	59.47	57.35
Net area	23.9	23.67
Leakage power(nW)	5414.65	5165.95
Average dynamic energy consumption per spike(pJ)	0.055	0.05
Throughput(Gbps)	400	333.3
Delay(ns)	9.6	7.6

Table 4.3: Performance comparison between N=16 and N=32 for adder tree based neuron

For the adder tree based neuron, the changing of the structure doesn't show a significant effect on the area and power but decreases the total delay. The critical path delay of the adder tree doesn't change a lot as input doubles, so fewer levels result in a smaller total delay. The delay of the delay cells in the two levels is set to 3ns, which is the cycle time of the pipeline. The throughput of the neuron is decreased because of the longer cycle time.

4.4 Optimizations

Although the adder tree based neuron shows better performance than the time-multiplexed neuron in many aspects, it has very limited room to optimize. In contrast, the performance of the time-multiplexed design can be optimized in different ways. In this section, we mainly discuss some methods that can be applied to the time-multiplexed to improve its performance.

As we discussed in the previous section, the area and power consumption of the time-multiplexed neuron mainly come from the delay cells, so its performance can be improved by applying delay cells with smaller area or power consumption to replace the delay cells given by the standard cell library. A new type of delay cell is proposed in [1], it is designed by two thyristors in a cross-coupled configuration to limit the current in the delay cell so that it is able to provide considerable delays with low power

consumption. Besides, the delay cells proposed in [31] and [32] can also be considered to be applied in the neuron designs.

The adders in the compute units also play a crucial role in the time-multiplexed neuron, the critical path of the adder determines the time interval between two event pulses(T), a smaller T can not only reduce the total delay and the pipeline cycle time but also reduce the area consumed by the delay cells. In the proposed prototype, a ripple carry adder is applied in the compute unit and it has the longest critical path than any other types of adders, so T can be reduced by replacing the RCA with other fast adders. However, normally fast adders consume more area than the RCAs, so the trade-off between the area of adders and delay cells needs to be explored in the future work.

Besides, pulse gates can be applied to the proposed neuron design to replace the conventional logic gates for high-performance pulse propagation. Pulse gates are designed by self-resetting CMOS, which is able to maintain stable, narrow pulse signals by the reset loop inside[33]. The pulse gates family consists of pulse buffer, pulse OR, pulse AND and pulse latch, the behavior of the pulse gates is shown in Figure 4.10. By applying the pulse gates, the neuron is able to operate at a very high speed.

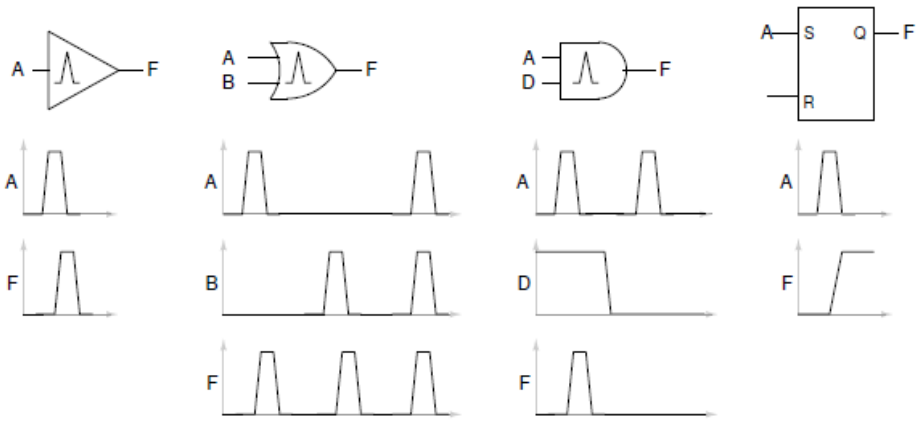


Figure 4.10: Basic pulse hates
[34]

4.5 Summary

In this chapter, the post-synthesis simulation of the two proposed neuron designs is presented, then the synthesis results of the designs are discussed and compared, as well as the trade-off analysis between two hierarchical structures. Finally, some methods to optimize the time-multiplexed neuron are given.

5 Conclusion and Future work

5.1 Conclusion

Our human brains have evolved over billions of years to be very energy efficient to handle massive information-processing as well as the tremendous metabolic energy consumption inside, as a result, the Spiking Neural Networks are designed to mimic the biological neural behaviour and have been widely applied to the neuromorphic computing systems to obtain extremely low power consumption. The compute efficiency of the brain-like neuromorphic systems can be achieved by asynchronous design and event-driven operations, which abandon the global clock to save energy from redundant operations. The neurons are the basic processing units of the SNNs, which accumulate the synaptic weights and generate spikes to the neighbouring neurons. In this work, two self-timed digital IF neuron designs are proposed, both designs are fully event-driven and synthesizable. The two proposed neurons exploit the hierarchical structure to achieve the high-speed, parallel accumulation for 1024 6-bit synaptic weights.

The accumulator is the core component of the self-timed neuron, which consists of multiple parallel-processing compute units, the only difference between the two proposed neurons is the design of the accumulator. In the adder tree based neuron design, the weight accumulation is achieved by the combinational logic, the accumulator is composed of multiple adder trees, the input of the accumulator is determined by the existence of the input spikes. A pulse-based acyclic asynchronous pipeline is implemented to increase the sample rate and the throughput of the neuron, the cycle time of the pipeline is determined by the delay of the delay cells in each pipeline stage. While in the time-multiplexed neuron design, the local weight accumulation is achieved by the compute unit which generates an on-demand local clock. Delay cells are used to provide sufficient spacing between the self-generated clock pulses. A pulse-mode handshake asynchronous pipeline is applied to the time-multiplexed neuron to obtain better time efficiency.

Both designs are implemented with SystemVerilog and synthesized by Cadence genus in TSMC 28nm technology. The synthesis results show that the adder tree based neuron consumes less area than the time-multiplexed design, and it is able to accumulate 1024 synaptic weights with 9.6ns delay and energy consumption of 0.055pJ per spike, better than the 83.3ns delay and 0.23pJ per spike of time-multiplexed design. However, since the performance of the time-multiplexed neuron is dominated by the delay cells, so it can be significantly improved by applying delay cells with better performance or reducing the number of delay cells, while the adder tree based neuron has very limited room for optimization.

5.2 Future work

For the time-multiplexed neuron, different types of adders can be implemented in the compute units to replace the ripple carry adder for a shorter critical path, which results in less area consumed by the delay cells, but it comes with a cost that more area consumed by the adders. As a result, the balance between the area of delay cells and adders should be explored in future work.

Besides, the proposed neuron designs can be applied to an SNN array as one column. Normally a group of input spikes is received by the array every few microseconds. Since the proposed neuron designs are able to finish the accumulation within 100ns or even 10ns, which means most of the time the neurons are idle, so it is possible to time-multiplex one column instead of implementing every column in the array, the output spike generated by each accumulation can be captured first and send out parallelly when each column completes their accumulations. By time-multiplexing one column, the area consumption of the SNN array can be reduced tens or even hundreds of times.

Bibliography

[1] J. Stuijt, M. Sifalakis, A. Yousefzadeh, and F. Corradi, "μbrain: An event-driven and fully synthesizable architecture for spiking neural networks," *Frontiers in neuroscience*, vol. 15, p. 538, 2021.

[2] C. S. Thakur, J. L. Molin, G. Cauwenberghs, G. Indiveri, K. Kumar, N. Qiao, J. Schemmel, R. Wang, E. Chicca, J. Olson Hasler, *et al.*, "Large-scale neuromorphic spiking array processors: A quest to mimic the brain," *Frontiers in neuroscience*, vol. 12, p. 891, 2018.

[3] S.-C. Liu, T. Delbruck, G. Indiveri, A. Whatley, and R. Douglas, *Event-based neuromorphic systems*. John Wiley & Sons, 2014.

[4] M. Chu, B. Kim, S. Park, H. Hwang, M. Jeon, B. H. Lee, and B.-G. Lee, "Neuromorphic hardware system for visual pattern recognition with memristor array and cmos neuron," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 4, pp. 2410–2419, 2014.

[5] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *Ieee Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[6] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.

[7] L. S. Smith, "Neuromorphic systems: Past, present and future," *Brain Inspired Cognitive Systems 2008*, pp. 167–182, 2010.

[8] S. Ghosh-Dastidar and H. Adeli, "Spiking neural networks," *International journal of neural systems*, vol. 19, no. 04, pp. 295–308, 2009.

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

[10] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. homogeneous synaptic input," *Biological cybernetics*, vol. 95, no. 1, pp. 1–19, 2006.

[11] S. Shafayet Chowdhury, C. Lee, and K. Roy, "Towards understanding the effect of leak in spiking neural networks," *arXiv e-prints*, pp. arXiv–2006, 2020.

[12] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of physiology*, vol. 117, no. 4, p. 500, 1952.

[13] E. M. Izhikevich, "Simple model of spiking neurons," *IEEE Transactions on neural networks*, vol. 14, no. 6, pp. 1569–1572, 2003.

[14] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, and K. Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, 2014.

[15] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. 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.

[16] T. Hwu, J. Isbell, N. Oros, and J. Krichmar, "A self-driving robot using deep convolutional neural networks on neuromorphic hardware," in *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 635–641, 2017.

[17] M. Davies, A. Wild, G. Orchard, Y. Sandamirskaya, G. A. F. Guerra, P. Joshi, P. Plank, and S. R. Risbud, "Advancing neuromorphic computing with loihi: A survey of results and outlook," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 911–934, 2021.

[18] S. M. Nowick and M. Singh, "High-performance asynchronous pipelines: An overview," *Ieee design & test of computers*, vol. 28, no. 5, pp. 8–22, 2011.

[19] S. Hauck, "Asynchronous design methodologies: an overview," *Proceedings of the IEEE*, vol. 83, no. 1, pp. 69–93, 1995.

[20] J. Spars and S. Furber, *Principles asynchronous circuit design*. Springer, 2002.

[21] J. Sparsø and S. Furber, "Principles of asynchronous circuit designa systems perspective, 2001," *Google Scholar Google Scholar Digital Library Digital Library*.

[22] M. Davies, A. Lines, J. Dama, A. Gravel, R. Southworth, G. Dimou, and P. Beerel, "A 72-port 10g ethernet switch/router using quasi-delay-insensitive asynchronous design," in *2014 20th IEEE International Symposium on Asynchronous Circuits and Systems*, pp. 103–104, IEEE, 2014.

[23] J. Spars and S. Furber, *Principles asynchronous circuit design*. Springer, 2002.

[24] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[25] M. Singh and S. M. Nowick, "Mousetrap: High-speed transition-signaling asynchronous pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, 2007.

[26] M. Singh and S. Nowick, "High-throughput asynchronous pipelines for fine-grain dynamic datapaths," in *Proceedings Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC 2000) (Cat. No. PR00586)*, pp. 198–209, 2000.

[27] T. E. Williams, *Self-timed rings and their application to division*. Stanford university, 1991.

[28] S. K. Patel and S. K. Singhal, "Area–delay and energy efficient multi-operand binary tree adder," *IET Circuits, Devices & Systems*, vol. 14, no. 5, pp. 586–593, 2020.

[29] M. Vlăduţiu, *Computer arithmetic: algorithms and hardware implementations*. Springer Science & Business Media, 2012.

[30] J. Juan-Chico, P. Ruiz de Clavijo, M. Bellido, A. Acosta, and M. Valenia, "Inertial and degradation delay model for cmos logic gates," in *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, vol. 1, pp. 459–462 vol.1, 2000.

[31] M. Kurchuk and Y. Tsividis, "Energy-efficient asynchronous delay element with wide controllability," in *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 3837–3840, 2010.

[32] D. Z. Turker, S. P. Khatri, and E. Sanchez-Sinencio, "A dcvsl delay cell for fast low power frequency synthesis applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 58, no. 6, pp. 1225–1238, 2011.

[33] M. Miller, C. Segal, D. Mc Carthy, A. Dalakoti, P. Mukim, and F. Brewer, "Impolite high speed interfaces with asynchronous pulse logic," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pp. 99–104, 2018.

[34] F. Brewer, D. McCarthy, and M. Miller, "Automated timing constraint generation for pulse gate circuits," 2021.