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

CAS-2023-00

M.Sc. Thesis

On-chip Self Timed SNN Custom Digital Interconnect System

Jiongyu Huang B.Sc.

Abstract

A Spiking neural network (SNN) is a type of artificial neural network which encodes information using spike timing, network structure, and synaptic weights to emulate the information processing function of the human brain. Within an SNN, it is always required to support the spike transmission that travels between neurons(array). This thesis aims to design a customized high-speed interconnect system which supports multi-point communication in a neuromorphic computing system. The burst-mode two-wire protocol in point-to-point communication is applied in this interconnect system, which is designed in high-level modelling with SystemC. In order to improve the utilization of hardware resources, a virtual channel system is involved. Furthermore, this system could be extended to a variable number of neuron arrays to support different types of spiking neural networks. Also, optimization methods are adopted to increase the transmission rate of the system and save unnecessary energy consumption. The interconnect system could achieve a throughput of 3.802 $Gbits/s$ with the given MNIST use case, based on the evaluation of simulation results.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science   Delft University of Technology

On-chip Self Timed SNN Custom Digital Interconnect System

My Subtitle

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Electrical Engineering

by

Jiongyu Huang B.Sc.
born in Shang'yu, 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
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 **"On-chip Self Timed SNN Custom Digital Interconnect System"** by **Jiongyu Huang B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 30.01.2023

Chairman:	Prof. Dr. Ir. Rene van Leuken
Advisor:	Dr. Aditya Dalakoti
Committee Members:	Dr. C. Galuzzi

Abstract

A Spiking neural network (SNN) is a type of artificial neural network which encodes information using spike timing, network structure, and synaptic weights to emulate the information processing function of the human brain. Within an SNN, it is always required to support the spike transmission that travels between neurons(array). This thesis aims to design a customized high-speed interconnect system which supports multi-point communication in a neuromorphic computing system. The burst-mode two-wire protocol in point-to-point communication is applied in this interconnect system, which is designed in high-level modelling with SystemC. In order to improve the utilization of hardware resources, a virtual channel system is involved. Furthermore, this system could be extended to a variable number of neuron arrays to support different types of spiking neural networks. Also, optimization methods are adopted to increase the transmission rate of the system and save unnecessary energy consumption. The interconnect system could achieve a throughput of 3.802 $Gbits/s$ with the given MNIST use case, based on the evaluation of simulation results.

Acknowledgments

I would like to thank my advisors, Prof. Rene Van Leuken, Ir. Alexander de Graaf, Dr. Aditya Dalakoti and Dr. Kamlesh Singh for their kind feedback and suggestions in every weekly meeting. I am especially grateful to Dr. Aditya Dalakoti and Prof. Rene van Leuken, who provided me with the directions for the thesis when I felt lost.

Also, I would like to thank my colleague master students, Yichen Yang and Tianyu Du, who provide firm support and work with me in Innatera Nanosystems.

Finally, I would like to thank my parents, Xiaomei Yan and Meilong Huang. Without their selfless support and encouragement, it would be impossible to complete my studies at TUDelft.

Jiongyu Huang B.Sc.
Delft, The Netherlands
30.01.2023

Contents

List of Figures

List of Tables

Introduction 1

1.1 Problem Statement

With the growth in transistor integration density and complementing of the multi-core architecture, the modern Von Neumann computing system continued to succeed in development since its birth. However, the CPU, memory and other components are placed separately physically in a traditional Von Neumann computing system. To connect the whole computing system, the communication bus must exist, which consumes a lot of energy and becomes the speed bottleneck for this system. Furthermore, due to the physical limitations, the trend of transistor size shrinking has slowed down. As the density of data escalates dramatically, the von Neumann architecture becomes fundamentally non-scalable and inefficient when extracting valuable data.

Inspired by the human brain's working mechanism, a non-traditional computing architecture, named neuromorphic computing system, was proposed in the 1980s to mimic mammalian neurology. It uses the very-large-scaled-integrated (VLSI) circuit to implement neural system whose architecture is based upon neurobiology. There are different types of neural networks, such as recurrent neural networks (RNN), artificial neural networks (ANN), convolutional neural networks (CNN), etc.

This thesis elaborates on the process of designing a high-speed, self-timed interconnect system between neuron arrays within in Spiking Neural Network (SNN). Asynchronous self-timed burst mode is a kind of methodology in which a small pulse signal (spike) is used for marking a certain time period to progress the processes in a circuit. Conventionally, the clock is widely used in circuits with the function of transmitting a signal. When applying the clock signal, it works by flipping on and off constantly, based upon the frequency of the clock cycle, during which the power is dissipated, whether data is moving or not. The clock on an SoC system is generally generated by a reference clock with the PLL(Phase-Locked Loop) and the digital frequency divider, and it may be connected to drive thousands of register clock terminals for the whole circuit. As it is impossible for any single cell to have such a large driving capacity, a large number of buffers must be inserted. Also, in order to ensure the same delay to each register, the clock tree needs to be balanced. With the development of the manufacturing process and the increase of the design scale, the power consumption generated by the clock tree accounts for a higher and higher proportion of the power consumption of the entire SoC, sometimes accounting for almost 50%.

However, for a neuromorphic system, its frequency of events happening is much lower than a traditional processor. Thus, a considerable amount of energy is unnecessarily consumed when a clock tree is used in an SNN system. Furthermore, as process nodes shrink, the process variability adds to the difficulty of satisfying timing closure for SoC design. This problem is exacerbated by clock tree balancing, jitter as well as clock skew, which make this issue dramatically more difficult to solve.

Consequently, in our interconnect system, the Self Timed Burst Mode Asynchronous Logic is used to eliminate the need for clocks. In a self-timed circuit, when a process is about to be triggered, a pulse/spike corresponding to it, which is called an event, will act like a signpost to make the circuit follow in time to coordinate its sequence of actions, replacing the function of the clock in a traditional circuit. These events will not be activated until their specific trigger conditions are met. Thus, a self-timed circuit only consumes little power during its idle state.

To develop this interconnect system and verify its functionality, both the SNN array as well as interconnects are involved. The SNN arrays refer to a large amount of data output from the separate dies on a chip, and the scale of the data varies with the changing application scenario. However, the implementation of the SNN array is not included in this thesis. To make up for this, a trained MNIST network is applied to simulate the SNN array. The interconnects between these 'virtual' SNN arrays are implemented in a self-timed logic. These interconnects are specifically focused on the design of transmitters, receivers as well as the protocol with which they connect to each other. To reduce the use of resources and ensure the stability of data, a two-wire SerDes link is used to transfer information, which could ensure high speed and low latency. In addition, a multi-point communication mechanism is designed in our system, as there are multiple neuron arrays in a neuromorphic system generally. Depending on the test case provided, it is expected to evaluate different types of encoding and decoding schemes, with the trade-off between power, area and other evaluation criteria.

1.2 Objective

The objective of this thesis is to design and implement a customized high-speed interconnect system in self-timed burst mode asynchronous logic. This interconnect system is applied to transfer data within a neuromorphic computing system and some optimization methods are proposed to enhance the performance of this system. Furthermore, MNIST user cases will be provided to verify its functions and evaluate relevant power, area and throughput features.

1.3 Contribution

The contributions of this thesis are:

- Implement a serial link in asynchronous self-timed logic to capture spikes and transfer data between neuron arrays.
- Present optimization methods to compress output data from neuron arrays.
- Combine the serial link with the data compression method to speed up data transfer between neuron arrays.

1.4 Outline

The following is the structure of the thesis:

- **Chapter 2:**
  It provides some background information about the neuromorphic computing system as well as the self-timed pulse-mode communication link.
- **Chapter 3:**
  It introduces the design flow of the whole interconnect link system. A detailed explanation has been provided for all the components implemented. How they cooperate with each other has been elaborated as well. Also, the Parameterization of this system and some optimization methods have been mentioned in this chapter.
- **Chapter 4:**
  In this chapter, the power, performance and area of the interconnect system have been evaluated. The simulation results are shown in detail as well.
- **Chapter 5:**
  This chapter concludes this interconnect system and suggests some future improvements.

Background 2

This chapter first describes the architecture of a neuromorphic system, which contains a huge amount of connectivity to transmit data among nodes within its inner network. Also, as mentioned before, the system's spike rate is at a relatively low level most of the time. Because of that, an event-based methodology could be more suitable for a neuromorphic system. Furthermore, different communication patterns are introduced and discussed to evaluate whether they can overcome the bottleneck of data throughput during the spike rate's peaking period.

2.1 Spiking Neural Networks

As transistor technology advances, a greater number of transistors may be accommodated in the same space within a chip. The switching frequency also increases with the improvement of the transistor technology. However, in terms of computation, memory, and communication, the traditional von Neumann architecture is inherently inefficient and nonscalable when trying to handle complex scenarios with massive data. As the figure 2.1 shows, with the improvement of the clock frequency, the power density of the processors keeps increasing as well, which brings greater heat dissipation pressure and power overhead to the system. To overcome this issue, a lengthy ambition [16] has been to use neuroscientific insights to create a versatile computer that is energy and space efficient, homogeneously scalable to large networks of neurons and synapses, and flexible enough to run complex behavioural models of the neocortex as well as networks inspired by neural architectures [6].

Figure 2.2 shows a cross-section of the neuron networks inside the cortex of the brain. The neurons in the figure are connected to each other and transfer information to each other through their connections. Inspired by natural neural networks, spiking neural network (SNN) processing is utilised in neuromorphic compute accelerator ICs, which use neuron models to communicate information in the form of asynchronous events [21]. SNN is a kind of artificial network which is made up of neurons, axons as well as synapses. Apart from that, SNN incorporates the time factor also into its working model. Not like what they do in traditional networks, in SNN, a neuron does not transfer data when reaching the end of each propagation cycle (if this SNN network has a clock). Instead, only when a neuron's accumulated value, which is known as membrane potential, exceeds its threshold, will this neuron generate a spike or pulse. Figure 2.3 shows a three-layer network with one input layer with p input spike trains, one output layer with n neurons, as well as a single hidden layer. In an SNN neuron network, each neuron has one long wire which snakes away to the neurons in the other layers. Based upon this long wire, which is known as an axon, spikes from the

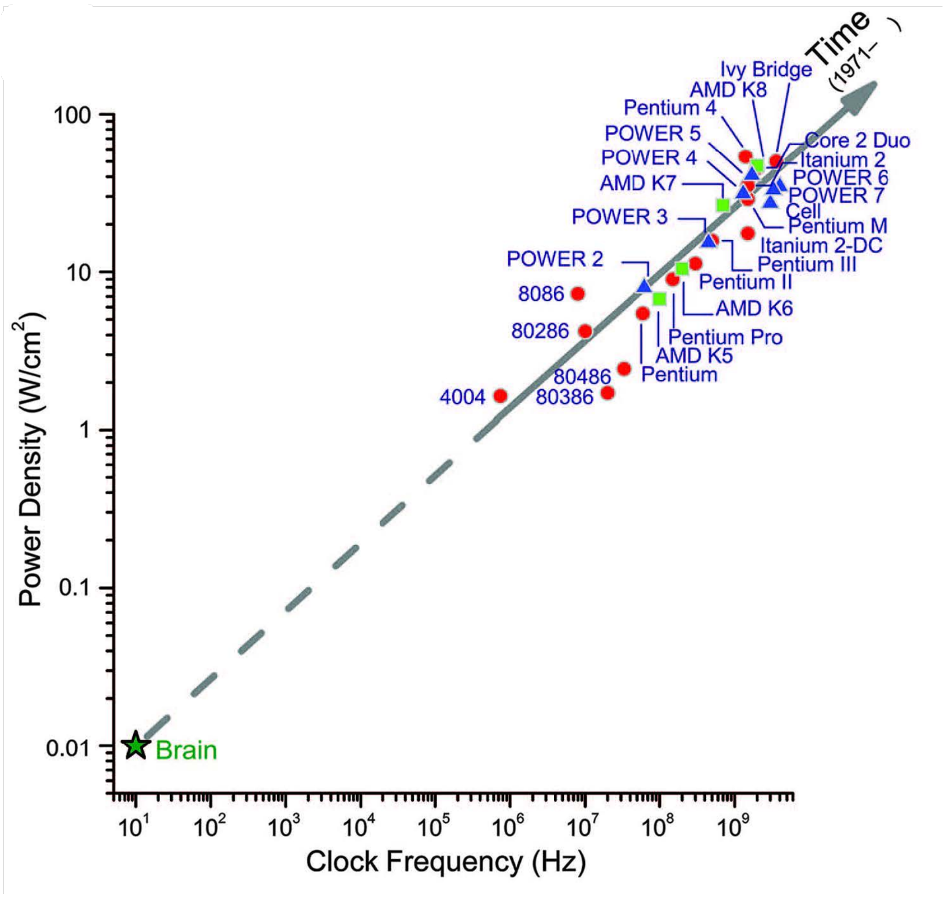


Figure 2.1: The trend of increasing power densities and clock frequencies of processors. [17]

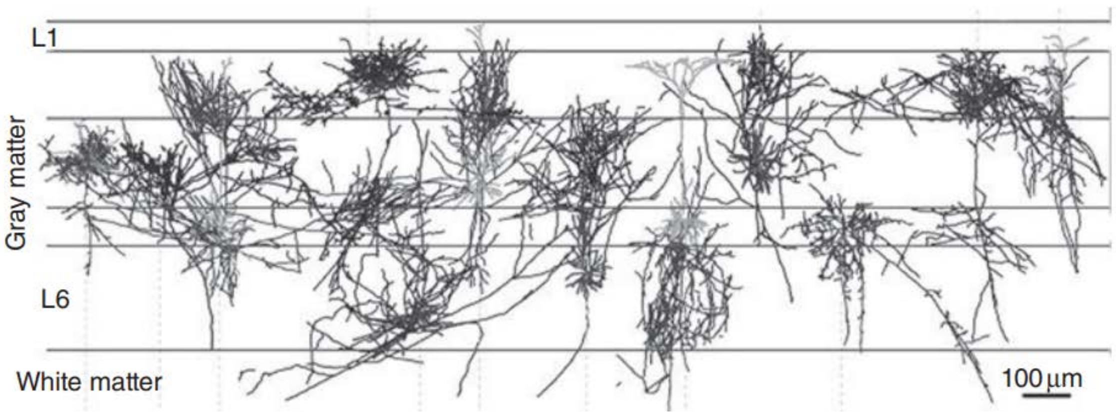


Figure 2.2: Neural network inside the human brain. [14]

previous layer can be transferred to the next layer. Neurons placed in different layers communicate through synapses, which work as connecting points between the axon on one side and the neuron body on the other.

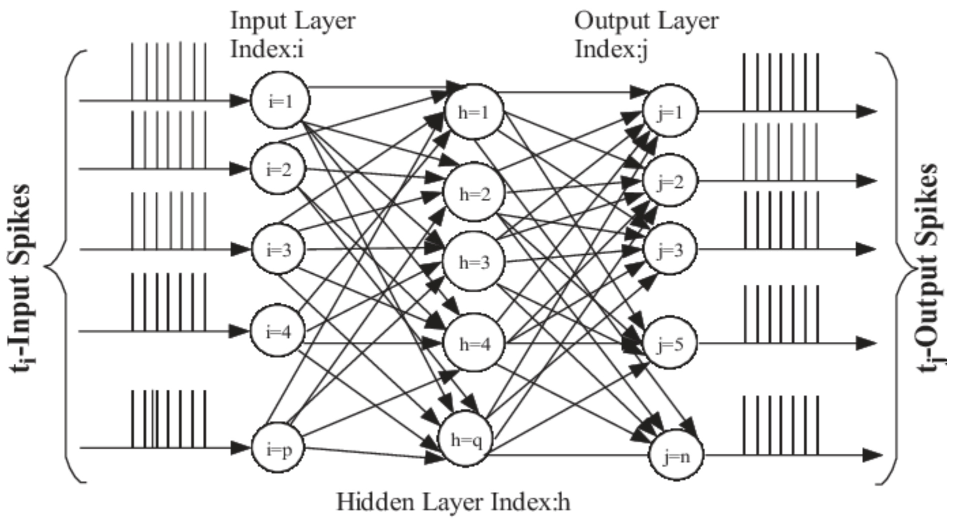


Figure 2.3: Architecture of a multilayer spiking neural network. [13]

Figure 2.4 illustrates how SNN imitates the work of biological neuron networks. From the point of view of computing machines, the human brain has many qualities that are superior to modern supercomputers [17]. For comparison, traditional computers operate in the range of Gigahertz (GHz), which leads to its high power density of around 100 W/cm2. This value is headed away from the operating point of the human brain, as figure 2.1 shows. when the human brain is at work, it only runs at an average firing rate of around 10 Hz with a relatively low power density at 10 mW/cm2. Also, the brain is highly efficient in how it processes information and tolerates faults. As mentioned before, the basic processing units in the human brain's network are neurons and synapses, which connect in a complex pattern. Apparently, the human brain's function is different dramatically from the traditional Von-Neumann architectures, which require far more space and power to operate. According to [1], the human brain is made up of about 86 billion neurons, forming a parallel computing system massively. Though every single neuron's inner structure is relatively simple, its immense parallelism leads to the neuron network's excellent performance. Based upon this, the behaviour of an individual neuron can be analyzed and imitated as a single computational unit which is independent of the other neurons.

Based on the features of the human brain mentioned above, simulating a neuromorphic system using traditional computing architecture is not only difficult but also inefficient. Brain-inspired neuromorphic architectures, which have parallel processors (neurons) have been proposed. These neuromorphic architectures operate at low-frequency and do relatively simple operations. For each neuron, it has locally distributed memory stored at the connection points to other elements, which corresponds to the synaptic connections to other neurons. To implement these neuromorphic networks, different communication methods have been evaluated in order to make use of the neuromorphic system's inherent and scalable parallelism.

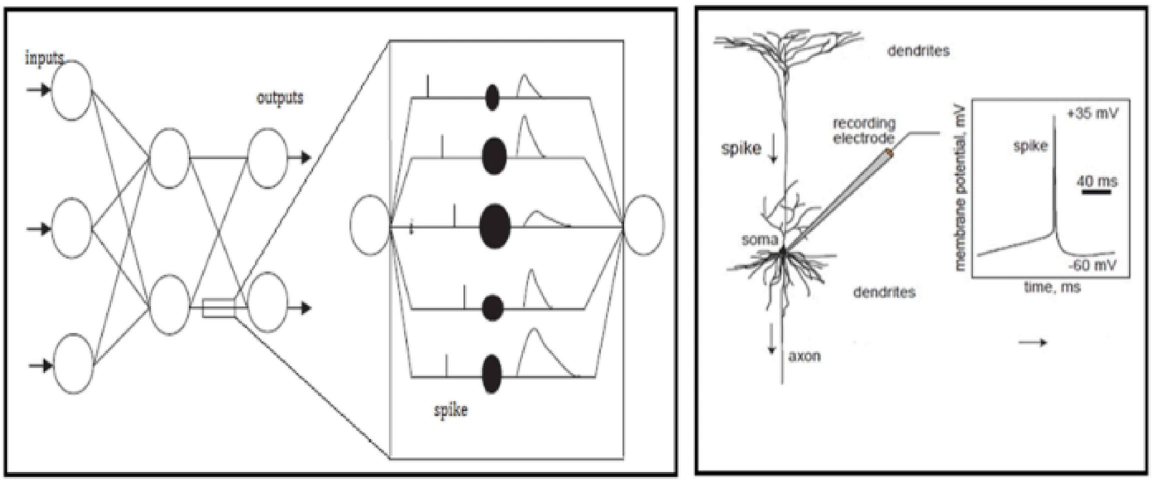


Figure 2.4: SNN with connections and Biological Neuron.

2.2 Address Event Representation

Address event representation (AER), first proposed in [15], is an event-based protocol which is commonly used in inter-chip communication between image sensors and neuromorphic processors to convey pulses. The mechanism of how the AER communication method works is shown in Figure 2.5, where the squares on both sides represent their respective neurons of the transmitting chip and receiving chip. As depicted, when the neural network is working, some neurons will produce some spikes at different times. The encoder encodes the address of the spiking neuron as an unique binary address. This binary address is transferred to the decoder through a digital bus. The decoder on the receiving chip asynchronous decodes the address and places the signal to the corresponding location. In AER, a spike is uniquely identified by two sets of information: the spiking neuron's position within the system, which is explicitly encoded as an address, and the moment that the spike occurs, which is implicitly encoded since the events are conveyed in real-time. The encoded information is called as an address-event, which is represented as a star in the square. As a data-driven digital multiplexing protocol, the AER is widely used in transmitting neural signals. It could send several pulses via a single wire(channel), providing a method which is more close to how human neurology works. The number of inter-neuron communication channels required by this AER method is reduced from n to about $\log_2 n$. For example, if a system had one million senders and one million receivers, one-to-one wiring would necessitate one million wires, whereas AER would necessitate it to around 20 ($\log_2 1,000,000$) wires.

However, as a point-to-point communication method, this solution may encounter some problems in practical application scenarios. As mentioned before, the encoder encodes the incoming pulses based upon their channel's address and the decoder receives these pulse asynchronously. Because of this asynchronous nature, there is no need for an AER circuit to process the timing information alone with the pulse. The time relationship between pulses can be expressed by the sequence of occurrence of the event [20].

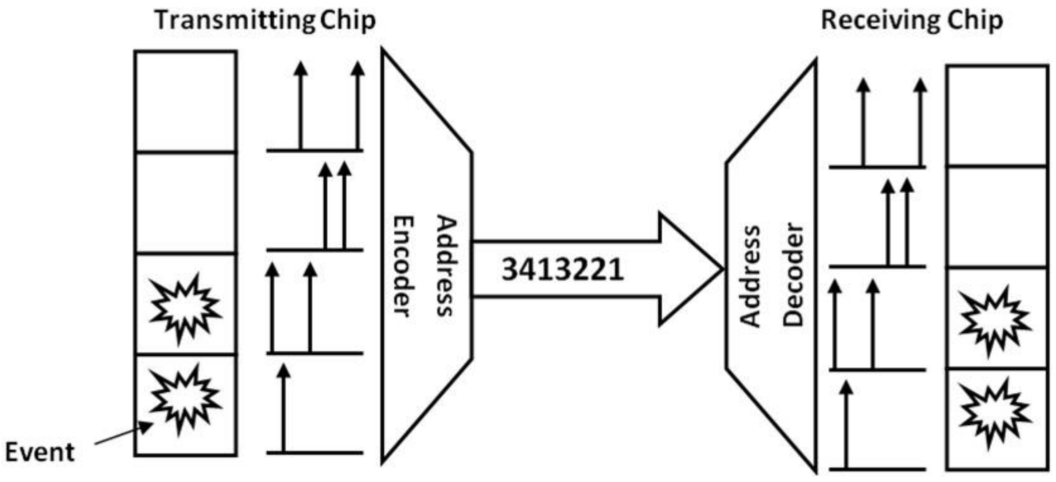


Figure 2.5: Standard address event representation (AER) protocol.

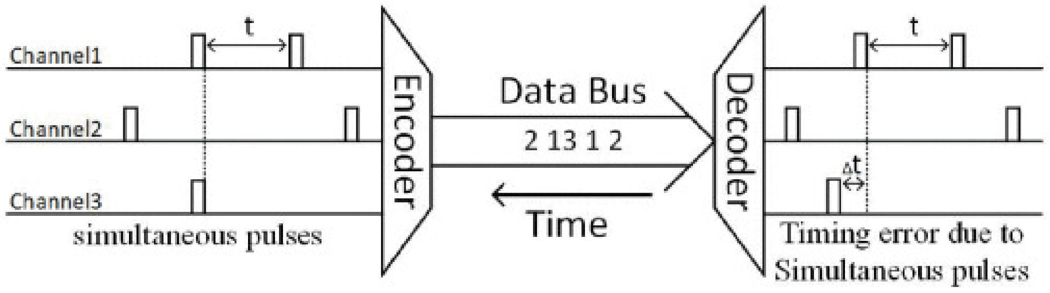


Figure 2.6: The mechanism of point-to-point AER communication.

The decoder receives the message and generates a fixed width pulse to the appropriate channel. In a neuromorphic network, it is quite common for multiple neurons to generate spikes at the same time. When pulses collide, in a classic AER circuit with fixed priority, the channel with the channel with the highest priority has much more chances to transmit pulses than the channel with the lowest priority. This could result in timing problems in the channels with low priority. For example, in figure 2.6, the channel1 has the highest priority while the channel 3 has the lowest priority. At a certain moment, both channel1 and channel3 need to transmit a pulse. After the point-to-point AER communication, the small timing error occurs at the channel3, whose priority is the lowest. Though this timing error is very small, in actual application scenarios, for example in AER-based image sensor field [27], this small error can have critical consequences.

2.3 Interconnect Strategies For SNN Implementations

For SNN implementation, several methodologies have been investigated, including software, firmware, and full-custom design. Table 2.1 summarises the neural system implementation strategies and highlights the trade-offs in terms of real-time constraints, area utilisation, power consumption, as well as architectural reconfiguration for implementation.

Table 2.1: Trade-off regarding the implementation of neural systems. [4]

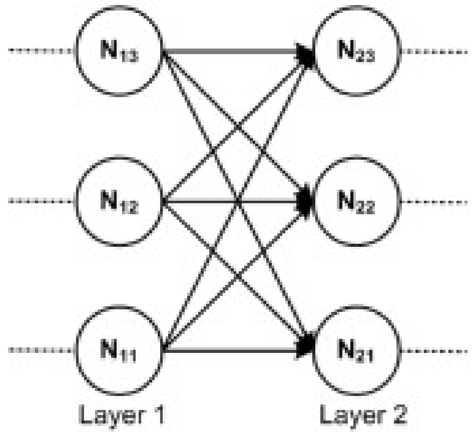
Implementation approach	**Area utilisation**	**Power consumption**	**Execution speed**	**Architectural reconfiguration**
Software	High	High	Low	High
Firmware	Medium	Medium	Medium	Medium
Hardware	Low	Low	High	Medium

In this section, related work regarding the interconnect techniques used in hardware SNN implementations is summarised. Network-on-chip (NoC) designs are discussed in particular, as a result of their suitability for enabling large-scale SNN hardware implementations.

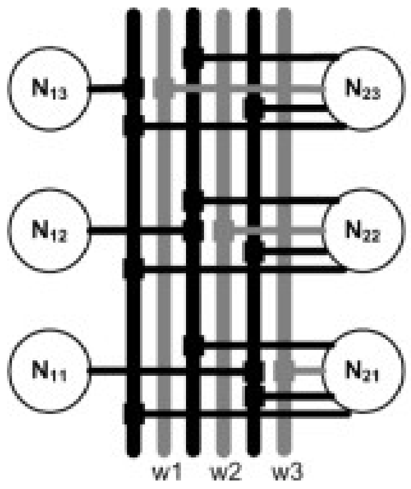
2.3.1 Shared bus topology

Using direct neuron-to-neuron communication via a common bus architecture provides a straightforward mechanism for neuron interaction. However, for a fully connected network using this shared bus topology, the number of bus lines necessary to link neurons is equal to the number of neurons in the pre-synaptic layer multiplied by the number of neurons in the post-synaptic layer. Thus, the bus-shared approach is not scalable. Furthermore, a bus-based system has switching needs which expand non-linearly with the size of the network mesh [8]. In Figure 2.7(a), there is a fully connected 2x3 neuron network topology. Its neural network implementation is shown in Figure 2.7(b), which utilizes a bus-based connectivity technique. In this diagram, the dark transfers signal from a pre-synaptic neuron to a post -synaptic neuron, and the grey lines are utilized to provide corresponding synaptic weight (w) to each neuron.

This two-layer fully interconnected feed forward neuron network, where n neurons are placed in each layer, arranges an interconnect density of n^2, making this architecture concept unworkable in terms of space utilisation. Besides, the real-time execution cannot be guaranteed by this bus-oriented design, since the network latency grows proportionately to the number of neural processing parts linked to the common bus [4]. When constructing a large neural network, using this method will make it difficult to meet the overall timing constraints of the system.



(a) A 2×3 fully connect network



(b) Bus-based interconnect scheme of a 2×3 network topology.

Figure 2.7: (a) A two-layer feed forward fully interconnected network with neurons per layer and (b) its bus-based interconnect scheme representation. [4]

2.3.2 Network-on-chip

A System on Chip (SOC) is a single ship which incorporates many capabilities required for a system onto a single chip. This may include one or more processor cores, memory subsystems, IO subsystems, and other similar functional logic/IP (Intellectual properities), all of which are integrated as a single IC device. For a traditional SoC computing architecture, it becomes more and more difficult to meet modern massive data processing scenarios, where high throughput and interconnection capacity between each subsystems are required. To alleviate this dilemma, several researches [2] [7] have presented the network-on-chip (NoC) interconnect architecture as a possible solution to the on-chip communication issues encountered in SoC.

A Network on Chip (NOC) is an on-chip interconnect technique applied on SOC designs to connect diverse design blocks (or IPs) effectively. Inspired by the personal computer (PC) network, the NoC architecture tries to imitate the method of how data is transferred in a PC network. Within a computer network, PCs are connected to each other typically, delivering information from one location to another under dedicated protocols and policies. To allow seamless integration of the computers inside the network, information is often carried via many communication layers, separating computing from communication. Nonetheless, while the NoC interconnect paradigm was inspired initially by the computer network, applying traditional computer network algorithms and methods directly into the NoC paradigm is not acceptable [2]. This is because that there exist certain constrains in terms of power consumption and area utilisation that are not of primary concern for typical network computer applications. In order to implement a NoC network on hardware, the aforementioned constraints

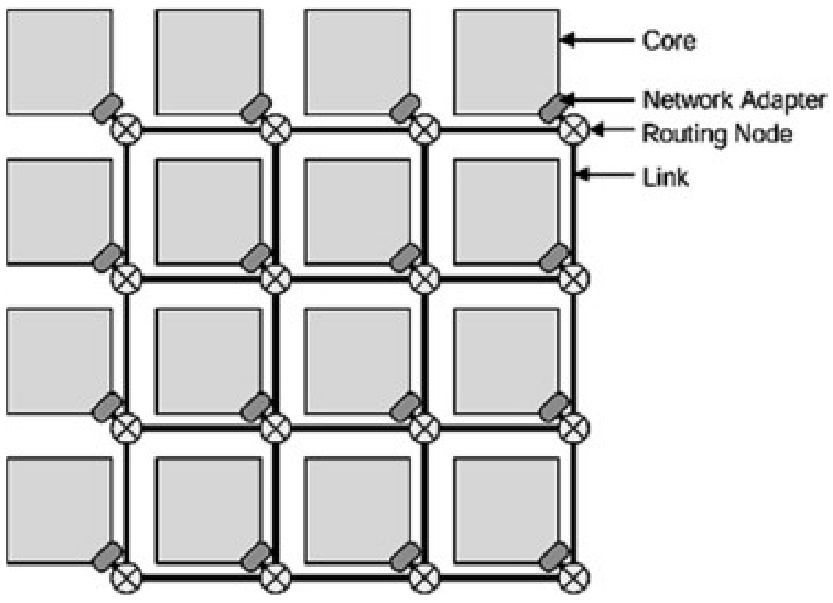


Figure 2.8: A general representation of a NoC-based architecture [8].

need to be considered.

Generally, a NoC architecture is made up of a collection of shared cores or processing units, network adapters, routers as well as links or connections that are organized in a specific topology based on the application scenarios. The goal of a NoC connection fabric is to reduce wire routing congestion on chip, making timing closure easier, and offering a standardized mechanism for adding or removing numerous IPs in the SOC design. Mapped to the SNN network, the processing units refer to the neuron models which are attached to the NoC routers through the neural network. Network adapters provide interface methods as well as communication mechanisms which allow spiking neuron models to communicate with one another via routers. Link's working principle is inspired spiking neuron synapses/axons. The SNN topology defines how spiking neurons models in a network are connected. The Figure 2.8 illustrates the block diagram of a NoC-based SNN network, where a conventional 4x4 SNN topology and its associated fundamental NoC components such as core or processing units, network adapters, routing nodes as well as links are included.

2.3.3 Current NoC-based SNN Approaches

A general reconfigurable SNN platform based on NoC is presented in [25]. The platform connects up to 108 digital integrate-and-fire (I&F) spiking neurons using a 2D mesh topology as well as some NoC routers. As the first attempt to leverage the NoC paradigm to facilitate SNN interconnections in hardware, this SNN was developed on an FPGA board. The platform outperforms in terms of real-time needs for a variety of applications including principle component analysis (PCA) classification as well as character and face recognition. The architecture, however, is built on non-adaptive

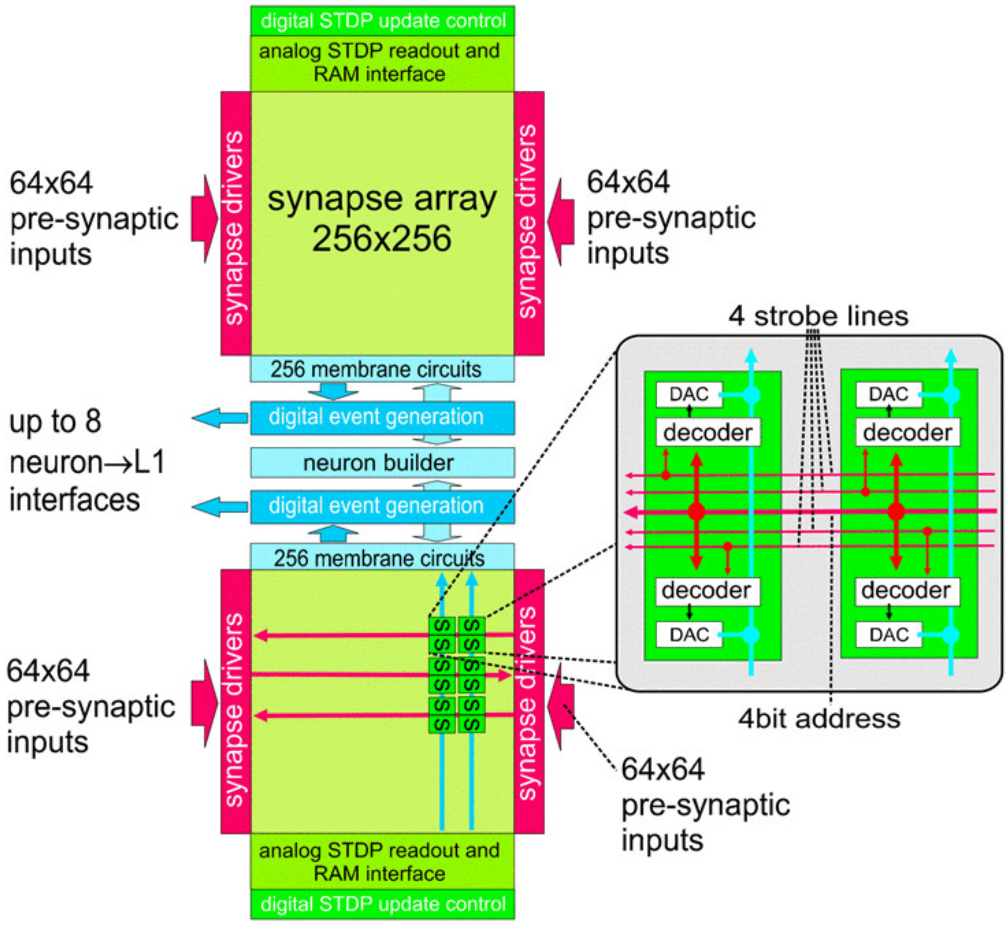

digital STDP update control
analog STDP readout and RAM interface
64x64 pre-synaptic inputs
synapse drivers
synapse array 256x256
synapse drivers
64x64 pre-synaptic inputs
4 strobe lines
256 membrane circuits
DAC
decoder
DAC
decoder
up to 8 neuron→L1 interfaces
digital event generation
neuron builder
digital event generation
256 membrane circuits
decoder
DAC
decoder
DAC
64x64 pre-synaptic inputs
synapse drivers
S
synapse drivers
4bit address
64x64 pre-synaptic inputs
analog STDP readout and RAM interface
digital STDP update control

Figure 2.9: Block diagram of the FACETS network core with connection pattern between synapses [18].

routers that lack traffic congestion control methods. Furthermore, because the digital neuron implementation employs an on-chip look-up table, the scalability of this platform is limited because the look-up table size grows with the number of neurons.

FACETS system is a high-density hardware neural network design that leverages multilayer communication, as described in [18]. For a FACETS system, its core processing building unit is a wafer module with 180,000 leaky integrate-and-fire (LI&F) neurons and 256 synapses per neuron. As shown in figure 2.9, in order to link many FACETS wafers based on a 2D torus topology, this platform employs a mix of hierarchical buses for managing neuron communication inside the wafer and an NoC router for providing the wafer-to-wafer connectivity fabric.

In [10], an EMBRACE architecture is proposed. This custom-embedded mixed signal scalable SNN architecture combines the programmability features of FPGAs and the scalable interconnectivity of NoC routers to implement large-scale artificial neural networks with a custom low-area/power programmable analogue CMOS neuron/synapse cell. Figure 2.10 illustrates the architecture of this EMBRACE network. The left side

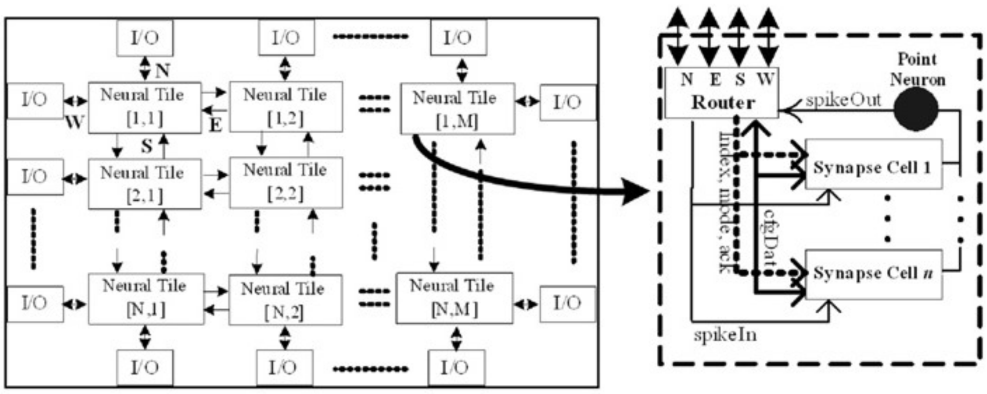


Figure 2.10: The architecture of the EMBRACE network [10].

of this figure shows a 2D EMBRACE NoC-based implementation for an SNN network. The Brain Tile, seen on the right side, is made up of a digital NoC router and analogue neural cells and synapses. One at a time, the NoC sends incoming spikes to a neural tile. Each spike is sent to the destination synapse through the signal *Spike_in*. When a neuron fires, a spike pulse is created on the signal *Spike_out*, which the NoC routes to several synaptic destinations.

Design 3

From chapter 2, the basics of the neuromorphic system as well as some typical communication patterns have been explained. Also, these different strategies have been discussed in terms of their features and limitations. Based on that, a new on-chip interconnect system is proposed to transfer data between an SNN network in this chapter. According to the specific requirements of an SNN network, the approaches how to design this system are elaborated on in detail.

3.1 Methodology Flow

A detailed explanation of the design workflow of the proposed link system is as follows:

1) propose an overall multi-point communication link structure.
2) make a behaviour-level description (SystemC) of the event-driven cell.
3) make a behaviour-level description (SystemC) of the two-wire encoding protocol.
4) make a behaviour-level description (SystemC) of the arbitration mechanism.
5) make a behaviour-level level description (SystemC) of the SerDes link.
6) Verify the logical correctness of this link system.
7) Inject an MNIST data set to each neuron array to run a real-case testbench. Estimate the throughput of the system.
8) Estimate the area and power consumption of each gate by consulting the TSMC library.

3.2 Overview

This thesis aims to propose a self-timed on-chip high-speed interconnect link which supports N-to-N multi-point communication within an SNN network system. As this link system works in a neural network, excessive energy consumption is unacceptable. Though the average frequency of a neural network is much lower than a traditional processor, there exist peak periods where the spiking rate is very high. As a result, this interconnect link is required to be low-power, low-latency as well as high-speed. By using SystemC, this interconnect link is designed in system-level modelling. This means the components that make up this link system are mainly described at the behaviour level, mixed with some amount of RTL-level description. As mentioned in 2, a neuromorphic computing system generally includes multiple neuron arrays. These neuron arrays accumulate the input spikes and generate the output spike to another neural array. This process can be presented as the data propagation between SNN layers. As the spiking neural networks are event-driven in this scenario, there are no clock signals as well as related timing blocks (PLL/DLL) required in the whole architecture. Thus, the design method is defined as asynchronous. Since there is no clock, all components consume no additional energy while in an idle state. This makes this interconnect link power-efficient, especially in a neuromorphic system where the average frequency is quite low.

This communication link is designed to realize the information exchange between the layers of an SNN network. In this given neural network application scenario, there are four neuron arrays placed on a single chip. Each neuron array can then be mapped to a layer of neurons (input layer, hidden layer or output layer) in a neural network. Each array includes 256 neurons, which means that each array has at least 256 input ports and output ports when communicating from layer to layer.

Figure 3.1 depicts the overall architecture of this low-power, low-latency and high-speed link system. There are four neuron arrays placed on a single chip, each neuron array has a receiver RX at its input side, receiving and decoding the serial information transferred through the SerDes link. As mentioned before, when the accumulated value of a neuron exceeds the specified threshold, the neuron will release a spike and clear the value accumulated before. The pulse width of this generated spike is less than *1ns*, which is too short to be captured by a traditional flip-flop. For this reason, a custom 'pulse-latch' gate needs to be used to latch a pulse signal from the neuron to high logic. A previous contributor [26] has already implemented this pulse gate so we could assume that the pulses have been captured successfully. Multiple 'pulse-latch' gates are connected together to form a memory, which is the block named Pulse Latch0, Latch1 and etc. These Pulse Latches can capture and store the output of each neuron array temporarily.

After capturing the spikes from the neuron arrays and storing them as logic high in the Pulse Latches, these data will transfer to the Controller system, which are blocks named CS0 to CS3 in the figure. Each neuron array has its own TX, which is included in the Controller system, as well as RX. Each Controller system can receive all data from 4 Pulse Latches, while each Pulse Latch can only capture the spikes from its corresponding neuron array. This is because, in a neural network, it is very common for multiple neuron arrays to send information to the same target at the same time. Considering the worst case, each transmitter needs to have the ability to transfer data from any neuron array to its corresponding receiver. This is also the structural basis for this connection system to realize the multi-point communication function. However, as depicted in Figure 3.1, there are thousands of wires needed to build connections between Pulse Latches and the Packet Generator systems. Because of induction, cross-coupling, and other factors, the signal in a specific wire may be reduced or disrupted when data is transferred. As a result, inaccuracy increases considerably, necessitating additional processing at the receiver. Furthermore, it is unrealistic to place so many wires in the same place for a chip layout. This problem will be discussed in chapter 5 as a future work. In the following simulations, it is assumed that the receiver can

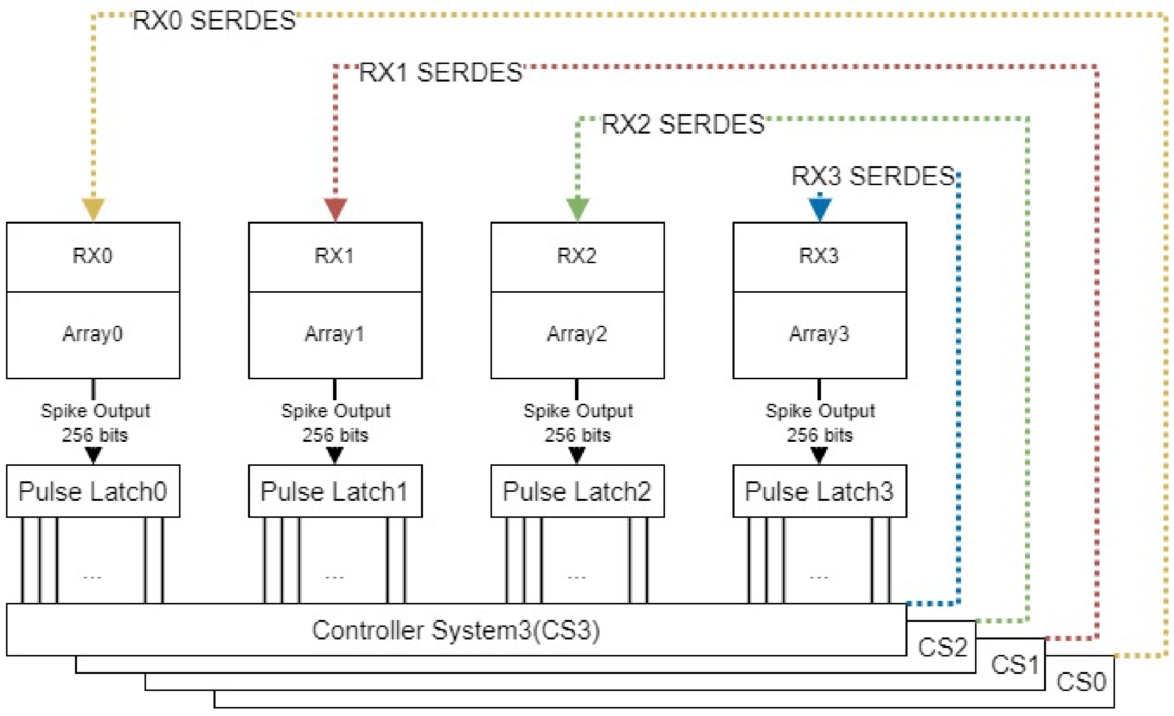


Figure 3.1: Architecture of the communication link.

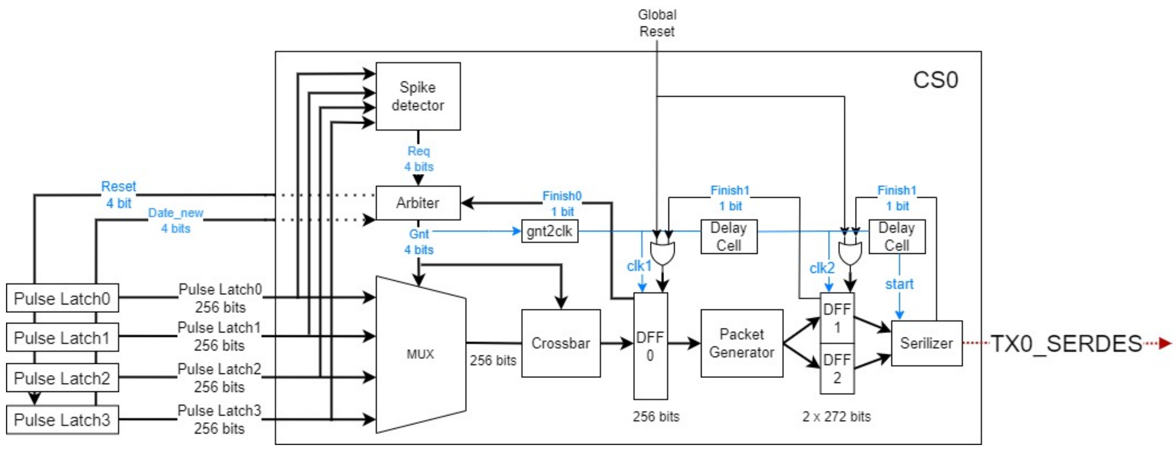


Figure 3.2: Inner structure of a Control System.

accurately and successfully receive all spike signals from the neuron arrays.

The inner structure of a packet generator system is shown in Figure 3.2. Once the block *Spike detector* detects the data from Pulse Latches, the *Arbiter* and *MUX* will cooperate to determine the sequence of data transmission. The 256-width packet is then transferred to the *Crossbar* to do mapping operations between two neuron arrays. The processed packet is stored in a DFF0, which is positively triggered by the signal from the *Arbiter*. The *Packet Generator* extracts the effective information in DFF0 and separates them into two opposite packets to provide to the *Serilizer* for parallel-to-serial data transmission. The mechanism of this process will be explained in more detail in this chapter.

In the following chapter, we will first introduce the method of constructing a self-time logic utilizing the features of SystemC. After that, the SerDes link between one transmitter and one receiver is explained. Then, the structure of Virtual Channels between multiple neuron arrays and their working mechanism is illustrated. Additionally, an optimization method to increase the transmission efficiency of a SerDes link is proposed. In the end, the parameterization of this system is explained to show the system's scalability and flexibility.

3.3 Design of an Event-driven Mechanism

In a spike neural network, the communication information is in the form of spikes. Since there is no clock used, the communication process is event-driven. Thus, communication in SNN is completely asynchronous. Without an event, which could be a spike from the external environment or other components in this case, the components within this communication system will stay in an idle state, and the information will not be transferred until the firing of a spike.

Since this communication link is built in a system-level modelling, many SystemC unique features which are specially developed for simulating circuits can be helpful to realize the basic functions of circuit event-driven.

In order to better explain the role of events in SystemC, some explanations need to be made to help understand related concepts. SystemC is actually a collection of classes and libraries based on the C++ programming language. As a result, the SystemC language's core grammar is derived straight from C++. There is a very important basic concept in SystemC called process. SystemC processes are similar to C++ methods or functions, except that they execute simultaneously. This is because it is needed to represent the fundamental behaviour of a system, which might have several circuits working in parallel. Therefore, a typical programme that runs sequentially cannot completely represent the behaviour of the circuit system.

In SystemC, there are three macros which can be used to declare a process - SC_THREAD, SC_CTHREAD and SC_METHOD. These macros correspond to threads, clocked threads and methods respectively. These macros are used to register the processes within their module. The code sample below demonstrates the typical syntax for registering a process in SystemC.

```
// Registering a method
SC_METHOD(<process_name>);
sensitive << <signal1> << <signal2>;
```

```

// Registering a thread
SC_THREAD(<process_name>);
sensitive << <signal1> << <signal2>;

// Registering a clocked thread
SC_CTHREAD(<process_name>, <event_name>);
```

In this example, the ⟨*process_name*⟩ specifies which C++ procedure is being registered as a process. And the *sensitive* keyword is utilized to declare a sensitivity list for methods and threads. This is significant since it defines when the process runs. The code written in a standard C++ method runs sequentially once the method is invoked, which means that the method body's statements are executed in order until reaching the last line. After that, this method is completed and all the associated memory with it is released. However, this is not the representative behaviour of a real genuine circuit, which remains in a steady state until the change of one of the input signals. To match the behaviour of real hardware circuits in a more precise way, the sensitivity list is used to imitate this behaviour. To achieve this, listing all of the signals which can trigger the process in the *sensitive* keywords is a good solution.

In the syntax of SystemC, there is also an object called *event*, which is a completely software-level concept. The *event* object is created by the keywords *sc_event*. *sc_event* is a class declared in the SystemC library, and it is used for process synchronization. In SystemC, any declared event is an object of the class *sc_event*. As shown in the last line of the code example, a process instance can be triggered or resumed when an event occurs, i.e., when the event is notified. The notification of an event is also encapsulated as a function in the class *sc_event*. There are multiple ways to call this method, which is explained as follows:

1) void notify(): this creates an immediate notification.
2) void notify(SC_ZERO_TIME): this creates a notification with Delta.
3) void notify(value, sc_time_unit): this creates a notification at the given time. For example, there is a statement:

   ev1.notify(1, SC_NS);

   This line of code notifies an event named ev1 after 1ns.

By using the features introduced above, the event-driven features of the circuit can be realized. According to the characteristics of the process in SystemC, the processes within a component are divided into two parts, one is the 'event generating' processes and the other is the 'functional' processes. As illustrated in Figure 3.3, for an 'event generating' process, the context in its sensitive list is all input signals from outside or other components. The purpose of an 'event generating' process is to generate

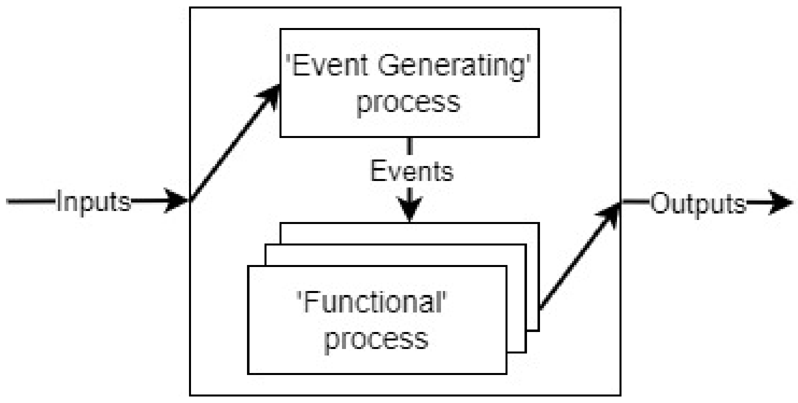


Figure 3.3: Working mechanism of an event-driven component.

events based on the inputs to this component. In this case, once the process detects a spike from the neurons, this process will generate an event, which is dedicated to activating other processes. A 'functional' process is used to implement the functions that the component should have. The sensitive list of this process can only be the event generated by the 'event generating' process. In other words, this component will not have any other actions until the event it sets occurs. This is exactly the working principle of an event-driven circuit. Furthermore, by using the function call *void notify(value, sc_time_unit)*, the delay time of the notification of an event can be controlled, which is very helpful when doing the timing analysis of the circuit.

3.4 Design of a SerDes Link

In our user case, there are multiple neuron arrays placed within the same chip. The input and output bandwidths of these arrays are both 256 bits, so this standard array is also called a square array. Providing 256 wires to connect two neuron arrays seems to be the most straightforward solution. However, by design, this parallel transmission requires all signals from the transmitter to arrive at the receiver at the same time. This cannot be ensured during the peak period with high frequencies, since the signal transit time for all signal lines cannot be guaranteed to be the same. The greater the frequency, the greater the importance of these tiny differences. As a result, the receiver must wait until all signal lines are established, which obviously reduces the transmission rate. Also, when doing parallel transmission, sending hundreds of bits simultaneously produces noise and leaves scope for error.

In serial transmission, there is no crowding as at a time only one bit is sent, which eliminates the crowding and chances of noise and error. In the case of serial transmission, this indicates that the crosstalk between signals is negligible. Thus, for data transmission between two neuron arrays, serial transmission is applied in this project. As shown in Figure 3.4, the data is converted from parallel to serial at the transmitter side. This packet is then sent in serial through the link and converted from serial to parallel at the receiver side. Because of this mechanism, the transmitter works as a

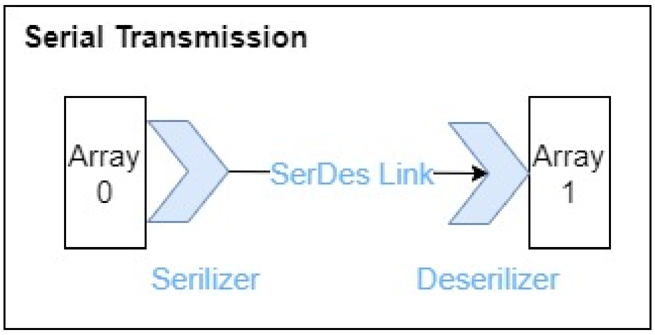


Figure 3.4: The SerDes Link Between Two Neuron Arrays.

serializer and receiving end plays the role of a deserializer.

SerDes is the abbreviation of serializer and deserializer. The expansion of demands for hardware signals has prompted people to pursue the improvement of high-speed signal transmission efficiency. As data transmission requires higher and higher bus bandwidth, parallel transmission technology is hindered by a series of problems such as difficult timing synchronization, serious signal offset, weak anti-interference ability and high design complexity. Compared with parallel transmission technology, serial transmission technology has fewer pins, strong scalability, point-to-point connection, and can provide higher bandwidth than parallel transmission, so it has been widely used in the field of embedded high-speed transmission.

3.4.1 Two-wire encoding

In order to ensure the stability of data during transmission and further reduce the number of pins and wires, a serial link [22] is implemented to transfer the parallel data into serial bits and then transfer them in order through the link. This link adopts the feed-forward architecture [3], which removes the need for a handshake protocol to control data flow between the data source and the data sink. Though this is a slight benefit for short-distance communications, it is critical for long-distance communications. In order to make up for this problem, a two-wire burst-mode logic using dual-rail encoding protocol [9] is introduced, where signals are presented using pulse mode and transmitted through two wires. In this protocol, one line is used to present the logic '0' and the other line is used to present the logic '1'.

In this project, a dual-rail encoding protocol is used to transfer the data through the SerDes link. By utilizing a two-wire burst mode protocol, multiple-bit information could be transferred from parallel to serial. Figure 3.5 illustrates how the two-wire burst mode protocol is used to convey a 7-bit packet from parallel to serial. This protocol utilizes two wires, which are named zero line and one line, for signal transmission within the chip. Every pulse which occurs on the zero line means a logical '0' for that signal, whereas the pulse on the one line corresponds to a logical '1'. The sequence between the pulses implicitly reflects the sequence between the bits in the signal. Also, because of this asynchronous mechanism, it is impossible to have pulses on both lines at the same

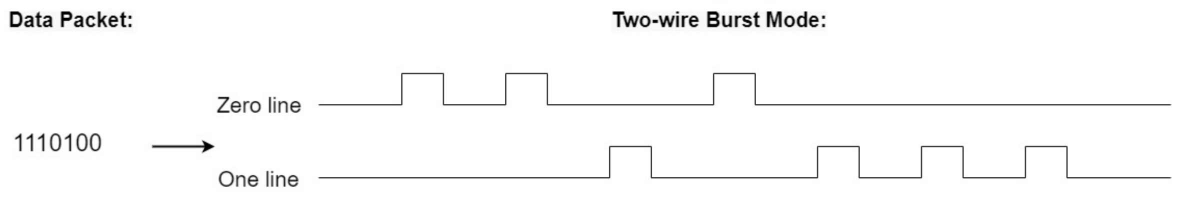


Figure 3.5: Two-wire Burst-mode Encoding.

interval. As shown in the figure, the data packet which needs to be sent is 1110100. The least significant bit in this information is '0', which is sent firstly on the zero line. After a short time interval, the next bit is transferred in a similar way. From the lowest bit to the highest bit, this string of multi-bit information is serialized and transmitted from the transmitter to the receiver. There is no clock involved in this method, as the sequence of the spikes could represent the relative timing information for these spikes.

Compared with the direct parallel transmission of information, converting the signal to serial and then decoding it at the receiving side theoretically takes more time. In this neuromorphic system, it is assumed that all the logic gates use the TSMC 28nm technology, where the minimum time interval resolution to ensure that the circuit can work properly is *250ps*. Take this value as the pulse width and the minimal time interval between pulses, each bit needs to consume *250ps+250ps=0.5ns* while encoding. The bit width of each neuron array's output is 256 bits. Considering the worst case, all 4 arrays need to transfer their packet through the SerDes at the same time, utilizing a single two-wire encoding link, the time consumed in this case can be calculated as:

$$T = 4 \times 256 \times 0.5ns \approx 0.5\mu s = 5 \times 10^{-7}s \tag{3.1}$$

Though converting the parallel information into serial takes more time compared with the parallel transmission, this extra time consumption is acceptable under this specific scenario. In a neuromorphic system, the average frequency of events occurring is around 10Hz, which means that pulses will be generated every 0.1s. Compared with this value, the time consumed in equation 3.1 could be negligible. Thus, sending the packet generated from the neuron arrays in serial will not cause serious timing issues in this SNN network.

3.4.2 Design of the Transmitter

In this SerDes link system, the transmitter is mainly responsible to convert the data from parallel into serial. Figure 3.6 shows the transmitter block in our system as well as its input and output ports. The port *Packet_out* from the previous component *Packet Generator* in Figure 3.2, and the port *clk2* is used to activate the inner DFFs in the transmitter. Once the DFFs have captured the signals, a start signal is generated inside the transmitter, indicating the start of the serialization conversion. As long as this start signal is generated, the TX will start to transfer the data from parallel to serial. The output port *One_line* corresponds to the logic high from port *Packet_out*, and the port

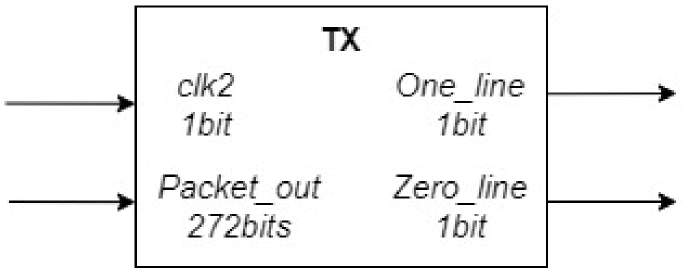


Figure 3.6: The Digital Block of Transmitter.

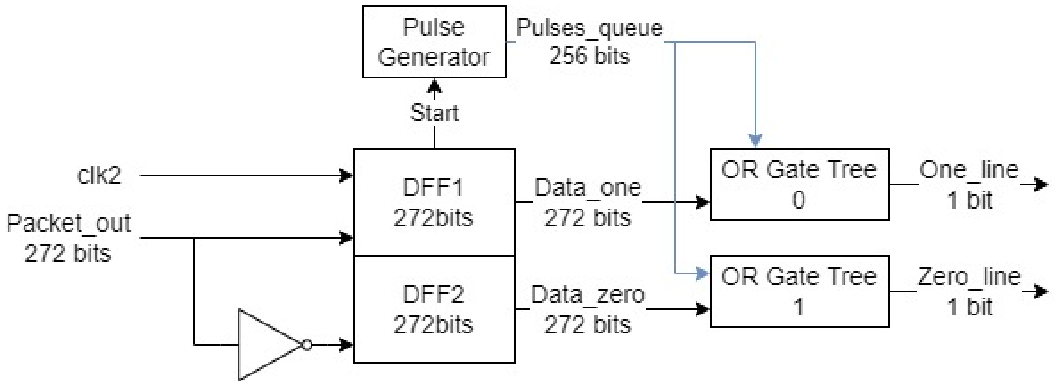


Figure 3.7: The Inner Structure of a Transmitter.

Zero_line corresponds to the logic low. At one time, there is one spike on one and only one of the two data lines, as the information represented by these two lines is opposite.

As mentioned above, a two-wire encoding protocol is applied there to convert the input data from parallel to serial. Figure 3.7 illustrates the inner structure of a transmitter. Based on this figure, the working principle of this component can be explained in a more intuitive way. In order to achieve the functionality of a transmitter, there are basically three sub-components needed:

- DFF with double size: this component is placed at the input side to process and temporarily store the inputs from the *Pulse Generator*.
- Pulse Generator: this component is used to generate parallel pulse signals with predefined time intervals.
- OR Gate Tree: this component extracts valid information from a 256 bits width packet and concentrates them on one line.

Figure 3.8 shows the function of the subcomponent *DFF with double size*, which is to do a bitwise NOT on the input data and then store the data before and after the operation in two DFFs respectively. The reason for this is that there is a delay in the actual level of the inverters, and the time spent on NOT operations of hundreds of input bits is not uniform. If the DFF is not inserted to store stable data, it may cause timing-related problems during subsequent conversions. With this subcomponent, it can be ensured that the outputs of the two signals *Data_one* and *Data_zero* are stable

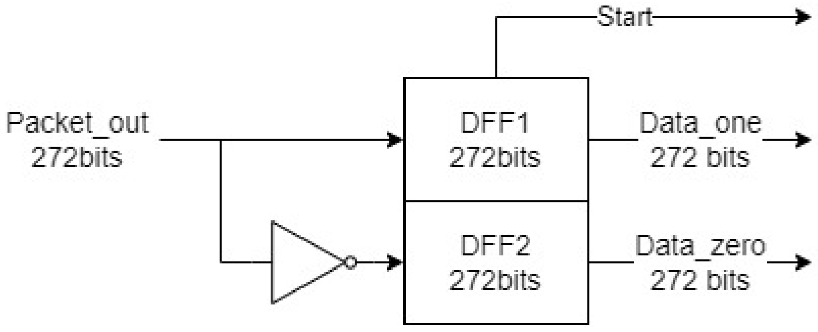


Figure 3.8: DFF with double size.

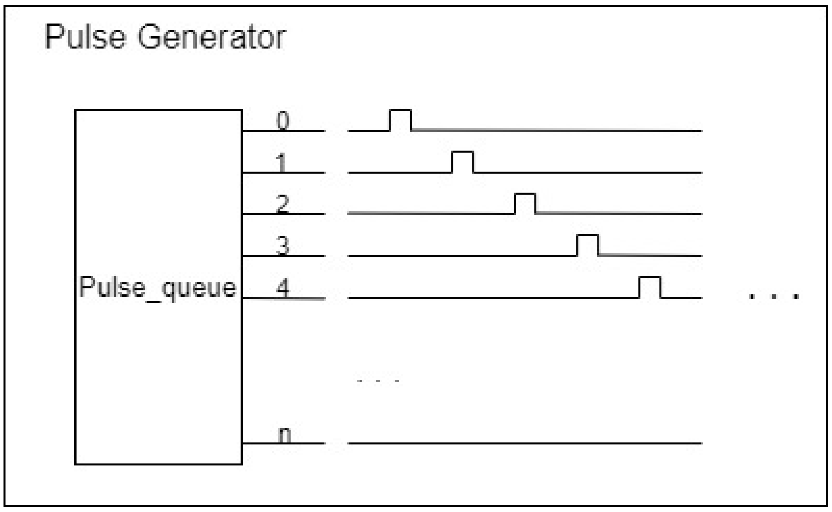


Figure 3.9: Output from a Pulse Generator.

and being produced simultaneously. When this subcomponent finishes processing the input signal, it generates a start signal inside the TX to activate the pulse generator.

As illustrated in Figure 3.9, once receiving the start signal from the previous subcomponent, the *Pulse Generator* block generates 272 consecutive spikes in 272 separate lines. A suitable pulse spacing should be provided between every two pulses. As the TSMC 28nm library is applied in this project, this time interval is 500ps. These sequential pulses are used to do bitwise AND operations with two data packets from ports *Data_one* and *Data_zero*.

Figure 3.10 shows the inner structure of the *OR Gate Tree* subcomponent. In this example, the process of how the data in *Data_one* is generated is demonstrated. The working principle for data in *Data_zero* is duplicated as *Data_one* and not shown in this figure. There are primarily two parts, with bitwise AND gates on the left side and the OR gate tree on the right side. This figure demonstrates how the packet from *Data_one* port is transferred from parallel to serial and finally outputs at the port *One_line*. First, a big AND gate is placed to do a bitwise AND operation between the consecutive spikes from a *Pulse Generator* and the packet data from *DFF1*. Following that, if the logic

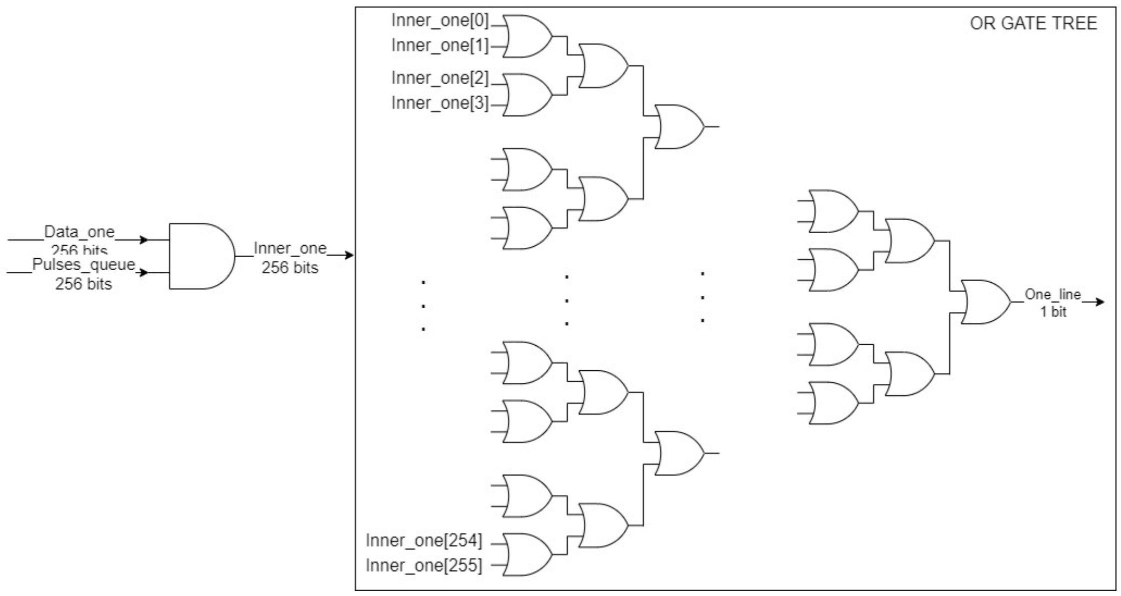


Figure 3.10: Inner Structure of an OR Gate Tree.

high appears in the *Data_one*, the pulse is assigned to the intermediate signal *Inner_one*, with the same pulse width as the pulse from *Pulse Generator*. Otherwise, the logic high is in the packet data from *DFF2*, which is completely opposite to the *DFF1*. So, the pulse is assigned to the port intermediate signal *Inner_zero*, which is connected to another *OR Gate Tree* that is responsible for port *Data_zero*. After that, the parallel intermediate signal *Inner_one* is injected into the gate tree, which converts the 256-bit signal into a single 1-bit line. The same data processing occurs on another data line as well. These two OR gate trees essentially generate the transmitter's two-bit output, converting the input data from parallel to serial and delivering it out through the SerDes link.

3.4.3 Design of the Receiver

Figure 3.11 shows the receiver block in our communication system as well as its input and output ports. The receiver is used to decode the spike packet from two ports *One_line* and *Zero_line*, as seen in Figure 3.5, converting this 2 bits serial data into a 256 bits parallel data packet and outputting the result on the port *RX_out*. To achieve this, several components are needed, which will be demonstrated step by step in the following content.

Figure 3.12 demonstrates how the receiver works. The two input ports *One_line* and *Zero_line* are connected with the same-named ports from the transmitter. Once the receiver detects a new spike packet, an internal start signal is triggered to start the decoding process, which transfers the serial packet into parallel. After decoding, an internal signal named *RX_Done* jumps to logic high, indicating the completion of this conversion. Then the receiver creates the parallel spike output at the port *Spike_out*.

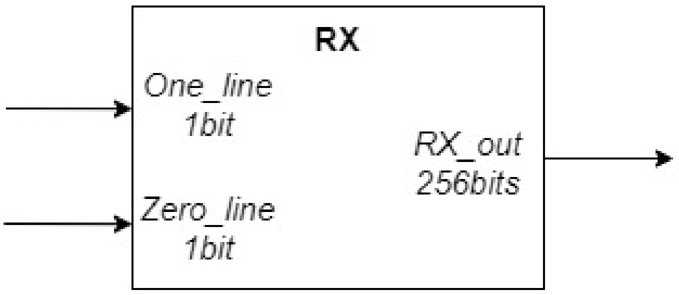


Figure 3.11: The Digital Block of Receiver.

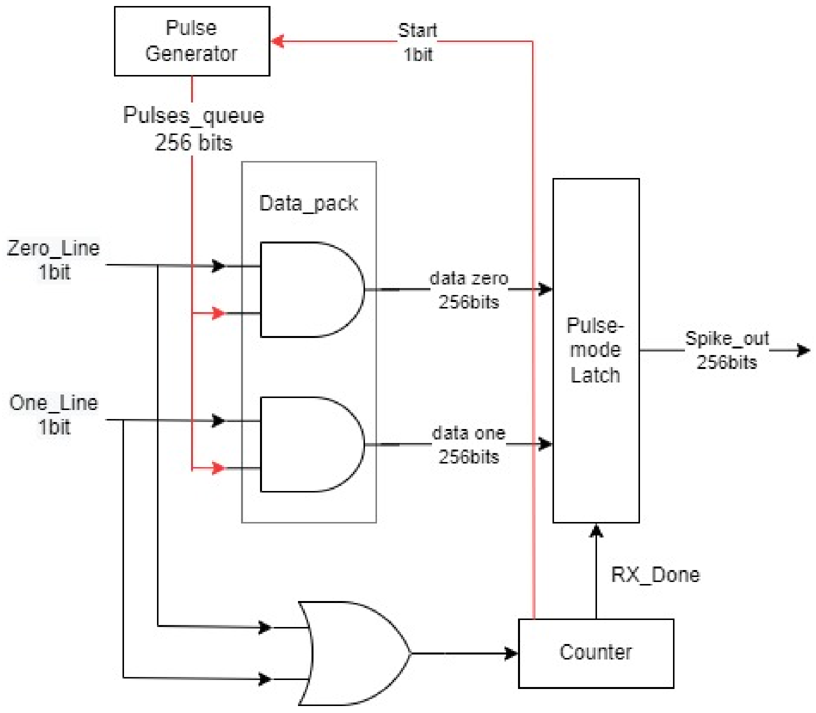


Figure 3.12: The Inner Structure of a Receiver.

In order to achieve the functionality of a receiver, there are mainly 4 subcomponents needed:

- Counter: this subcomponent is used to count the number of pulses from the ports *One_line* and *Zero_line*. To achieve this function, the signals from these two ports connect to an OR gate, whose output connects to the input of this *Counter*. When the value of this counter has just switched from 0 to 1, it generates a start signal to activate the *Pulse Generator*. When it reaches the maximum value, the intermediate signal *RX_Done* turns to logic high, indicating the accomplishment of the receiving process. As a result, this counter acts as a monitor in the receiver.
- Data_Pack: The inner structure of the *Data_Pack* block is shown in Figure 3.13. It seeks to turn the serial data into parallel while also distinguishing the pulse from *Zero_Line* and *One_Line*. When the counter starts working, the *Pulses_queue*

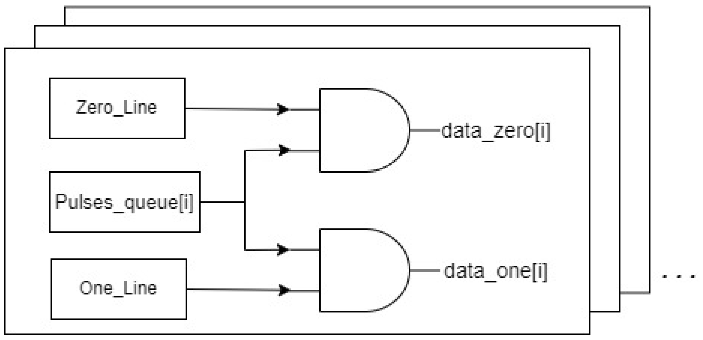


Figure 3.13: The Inner Structure of the *Data_pack* block.

signal from the Pulse Generator does the logic AND operation with two data inputs. For each pulse, the data from the two input lines are identified as logic high or logic low using the AND operator, and then output its corresponding position at the output port, which is allocated through the pulse signal *Pulses_queue* from the Pulse Generator.

- Pulse Generator: following the same design as in the transmitter, this component generates parallel pulse signals with predefined time intervals. The waveform generated by it is the same as that in Figure 3.9.
- Pulse-mode Latch: This subcomponent store the output results from the *Data_Pack*. Depending on whether a logic high appears in *data_zero* or *data_one* signal, it determines one bit of the resulting packet at a time. Once receiving the *RX_out* signal, the result is outputted at the port *Spike_out*.

3.5 Design of the Multi-Point Communication

As indicated in Section 3.4, a SerDes link connection has been designed to achieve point-to-point communication. However, in this neuromorphic system, there are multiple neuron arrays working simultaneously. As a result, it is quite common when multiple arrays require to transfer data at the same time. For a basic point-to-point communication method, this conflict can lead to blockage of the data transmission path. Furthermore, the destination of each data packet can be the same or different, which requires the multi-point processing capability of the communication system. In order to alleviate this problem, a solution which supports multi-point communication is proposed in this chapter.

The simplest method to solve this problem can be adding more SerDes links at the hardware level. By doing that, multiple data packets from different neuron arrays can be sent to their destination at the same time. However, this method has several drawbacks. Firstly, in an SNN network, a neuron array may receive signals from any other neuron array, including itself. Considering the worst case, the number of SerDes links that each neuron array needs to be assigned to is equal to the total number of arrays on the chip. Figure 3.14 illustrates an example of the direct connection between two neuron arrays.

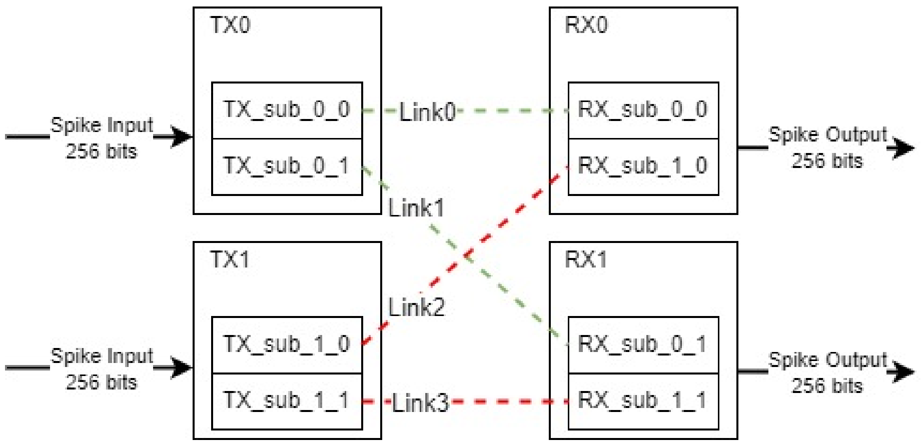


Figure 3.14: Direct Connection between two neuron arrays.

We could see that there are $2 \times 2 = 4$ links needed. If the communication link system is designed using this methodology, the number of SerDes links required is the square of the number of neuron arrays, which is $4 \times 4 = 16$ in our user case where 4 neuron arrays are placed on the same chip. With the expansion of the number of neuron arrays, the number of SerDes links requirements grows exponentially, dramatically increasing the area and the power consumption of the whole system. Furthermore, these extra SerDes links will be in an idle state most of the time due to the low average frequency of neural network operation, which is a waste of resources on the chip.

To conclude, directly increasing the number of SerDes links is not an appropriate solution to achieve multi-point communication. This communication system should be designed in a more resource-efficient way. Hence, based on the SerDes link, we propose a solution which multiplexes a physical channel using virtual channels (VCs) to reduce congestion during packet transmission. VCs provide multiple buffers for each physical channel, which could reduce latency and increase the network throughput. The following content elaborates on the design details of this VCs scheme step by step.

3.5.1 Virtual channel for a single physical channel

As mentioned above, it is not a good choice to increase the number of transmission channels directly from the hardware level. So the method of VCs is introduced to improve the reuse rate of the channel. VCs topology is widely used by makers of commercial parallel computers due to its simplicity, low cost, and distance insensitivity. By doing that, a physical channel could support several virtual channels multiplexed across the physical channel. Its architecture is illustrated in Figure 3.15.

In this virtual channel circuit, a single physical SerDes link can be shared by four virtual channels. Inside this VCs system, there are several components needed to achieve its functionality. The collaboration of these components ensures an appropriate packet transfer mechanism. The following content demonstrates how these three components interact locally and how this controller interacts with Transmitter and Receiver.

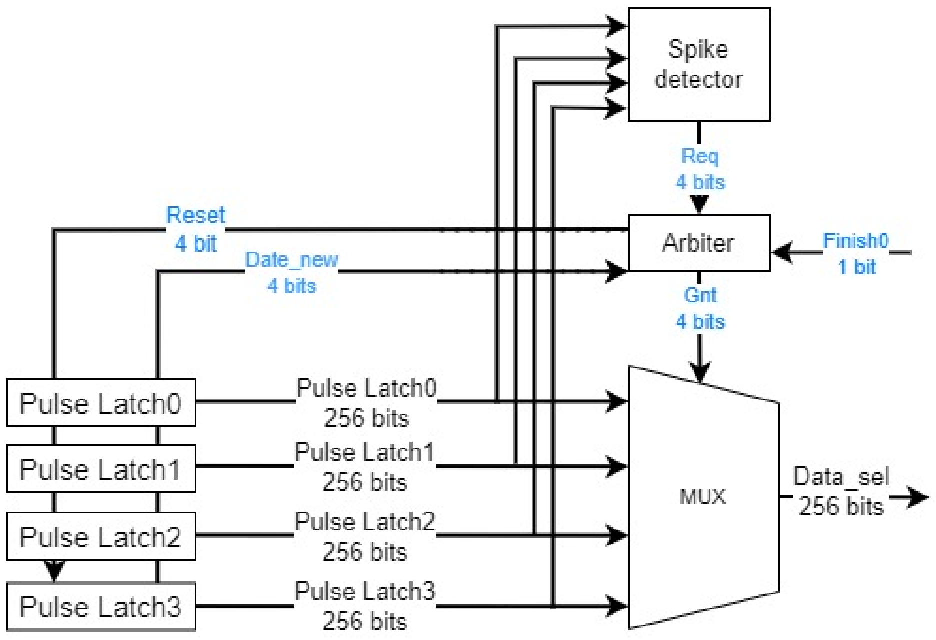


Figure 3.15: A physical channel divided into four virtual channels.

- Pulse Latch
  The pulse latch from figure 3.16 is connected with a neuron array at the input side of this virtual channel system. After capturing the spikes from its corresponding neuron array, the Pulse latch could convert the received spikes into a logic high and store them in proper positions within the 256-bit bandwidth. By doing so, the pulse latch records the short-duration pulses and stores them in a stable way. When the data is stored in the latch, an intermediate signal, which is named as *Dara_new* in the figure, is pulled to logic high to trigger the arbiter connected to the latch.

  Furthermore, the pulse latch performs as a buffer within this virtual channel system. As there are multiple virtual channels sharing the same physical SerDes link, there will be conflicts when multiple virtual channels try to occupy the link at the same. However, as there is only one link at the output, only one data packet from a virtual channel could be transferred through the SerDes link at one time. The data packet from the other channels has to wait in its own buffer until it is the channel's turn to use the physical link. Without this pulse latch or buffer, several data packets will be missed when they try to occupy the physical channel simultaneously. Once finishing the data transmission, the pulse latch will receive a reset signal to clear the data, waiting for new spikes from the neuron array.

- Spike Detector
  The spike detector in this virtual channel system is used to judge and generate the application of each virtual channel to the physical channel. It receives the outputs from the four pulse latches and outputs the *Req* signal to the arbiter. It is placed between the Pulse Latches and the arbiter, monitoring the output of

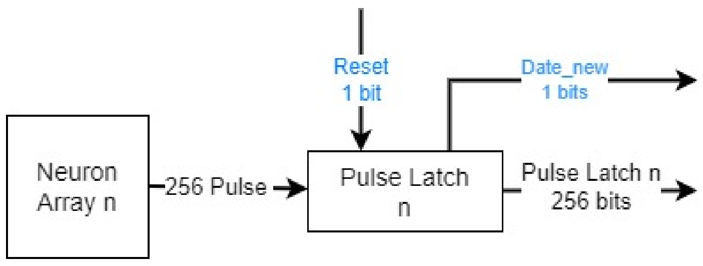


Figure 3.16: The Digital Block of a Pulse Latch.

four pulse latches to determine whether there is a need to use the SerDes link. Take one pulse latch's output as an example, as long as it detects one logic high value within these 256 bits, the detector will pull the corresponding bit high at its output signal *Req*, prompting the arbiter that the data packet from this channel has a request to use the SerDes link.

- Arbiter
  In this virtual channel system, when multiple neuron array requests to send the data packet simultaneously, an arbiter is needed to determine their sequence. The arbiter is a very common module in digital design and has a wide range of applications. The definition of an arbiter is that when two or more modules need to occupy the same resource, an arbiter is needed to decide which module will occupy the resource. In general, a module that proposes to occupy a resource needs to generate a request signal. After all the requests are sent to the arbiter, it will return a grant. The most important point of the arbiter is that only one module can be granted. because this resource can only be occupied by one module at a time.

  In this scenario, in order to provide arbitration among several neuron arrays, a round-robin arbiter is designed. The arbiter receives a 4-bit *Req* signal from the Spike Detector, representing the request of each neuron array. Once the value of the signal *Data_new* changes, which indicates the new spikes from the neuron array, the arbiter starts to process the *Req* signal. The *Gnt* signal will then be generated at the output port, indicating the authorization for transmission of a specific packet whose priority is the highest. When this packet transmission is complete, subsequent components will send a *Finish* signal to the arbiter. Once receiving this signal, the data stored in the corresponding pulse latch will be cleared. Then the value of *Gnt* signal changes, allowing second-priority packets to transmit data. By repeating this process, all data packets can use the physical channel for data transmission in an orderly manner until new data from the neuron array arrives. The above is the process of the arbiter to achieve round-robin. The formula for calculating the *Gnt* signal by *Req* signal is as follows:

$$Gnt = Req\&\overline{Req-1}, \tag{3.2}$$

$$Req_{new} = Req\&\overline{Gnt} \tag{3.3}$$

When a virtual channel requests to transmit a packet, its corresponding bit in *Req* turns to logic high. Then the arbiter does the arbitration to permit the virtual channel with the highest priority. After the transmission of this virtual channel is completed, the *Req* signal will change. How the Req_{new} is calculated is shown in the formula 3.3, and the *Gnt* output will also change accordingly. By default, array0 has the highest priority and array3 has the lowest priority.

- MUX
  In a digital circuit, a data selector or multiplexer (referred to as MUX) is a device that selects one of the digital input signals for output. It is used here to increase the amount of data within a certain amount of bandwidth. Once receiving the arbitration result from the Round Robin Arbiter, the MUX selects the path from the pulse latch to the transmission line, without having to have a SerDes link for each input signal.

3.5.2 Overall Controller

The communication system is designed to link four neuron arrays and allows the propagation of packets between them. As these neuron arrays are placed on the same chip, this communication system is designed under an on-chip scenario. Compared with the off-chip scenario, on-chip cores could connect with each other for a high-bandwidth, low-latency communication. More wires could be used within the chip to help control the data transmission between the neuron arrays.

In Figure 3.1, each neuron array is assigned to a SerDes link, which refers that the number of SerDes links needed is equal to the number of neuron arrays on this chip. By applying this methodology, the requirement of resources for building the SerDes links has a linear growth relationship with the number of neuron arrays, which is much more efficient than the original solution depicted in Figure 3.14.

After determining the number of SerDes Links needed for this link system, the main problem faced by the control system is that one neuron array may need to send packets to multiple destinations. In other words, a data packet may send requirement signals to multiple physical SerDes Links. Figure 3.15 illustrates a basic virtual channel system with a local controller which is responsible for only one physical channel. However, there are four SerDes links within this communication link system, and the order and time consumption of each packet is different. If we simply duplicate the system shown in Figure 3.15 by 4 times, it will affect the reset time of the data buffered in the pulse latch. If a packet is reset before being transmitted, no data will be transferred, increasing the data miss rate of the system. Conversely, if the data is not reset in time after the transmission is completed, subsequent data will not be correctly stored in the corresponding buffer, which will also affect the subsequent data transmission.

To solve that, the global switching controller is designed to ensure the correctness of the data reset time. As Figure 3.17 illustrates, all the local controllers are integrated into one global controller. Each pulse latch is assigned an individual counter. Only when the number of grants received by a latch is equal to the number of requirements

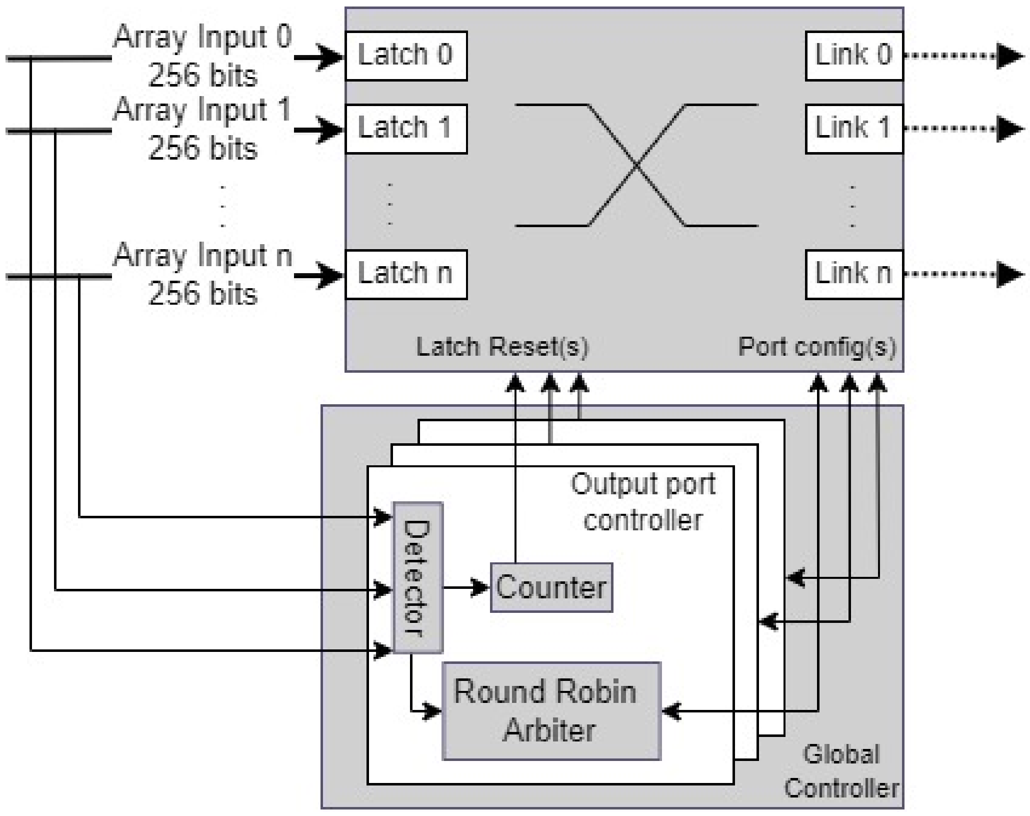


Figure 3.17: Block Diagram of the Overall Controller.

sent by it, that cache will be reset.

3.6 Data Processing Before Transmission

In the previous design, during data transmission, all packets were completely transmitted. In this way, when a SerDes link transmits a data packet, it needs to convert the whole 256-bit information from parallel to serial. However, in subsequent testing cases, a quarter MNIST dataset is used in our testbench. For this MNIST dataset, the average spiking rate is around 20%. For a 256 bits neuron array, every time new data is generated, only 20% of neurons will generate spikes, which means that a large part of the values in a data packet is logic low. Furthermore, the value of this part of logic 0 will consume the same time and power as logic 1 when it is transmitted through the SerDes link, which is undoubtedly a waste of resources for the system.

In this section, an optimization regarding this problem is proposed. Compared with the original method, the new method analyzes and compresses the internal data of each packet before data transmission. In the previous solution, the whole packet is transmitted, which involves unnecessary energy and latency, considering that most of the values within a packet are logic low. The new method divides the entire packet into several segments. If the data stored in a segment is all 0, the segment will not be sent by the SerDes link, which improves the resource utilization and transmission efficiency of the system. From figure 3.2, it can be seen that this data processing part is mainly realized by two components: *Crossbar* and *Packet Generator*. The rest of this chapter will elaborate on the design and functionalities of these two components.

3.6.1 Packet Generator

No matter how the data changes in a packet, as long as this packet is recognized as valid (that is, there is at least one logic high within it), the previous solution will send the entire packet to the destination through the SerDes link. For example, suppose there are two packets to be transmitted, the first packet contains 1 logic high value and 254 logic low values, and the second packet contains 255 logic high values. Although there is a huge difference in the amount of information contained in the two packets, the time and energy consumed by transmitting these two packets are exactly the same.

The improved method divides the entire packet into several segments, and then checks these segments separately to determine whether they are valid or not. By default, the 256 bits are divided into 16 segments, and each segment contains 16 bits. The value of the number of segments can be changed to 4, 8, 32 and so on. How this value can be parameterizable is explained in chapter 3.7. At present, this value is set as 16 to facilitate subsequent explanations.

After dividing the packet into 16 equal parts, the first thing to do is to judge whether each segment is all 0. This judgment can be realized by a series of OR Gate trees. Figure 3.18 illustrates this judgement process. Each segment is assigned an OR Gate tree, and the 16 bits contained in each segment pass through their assigned OR Gate tree. The output of each OR Gate tree is gathered together in sequence according to the respective segments to form a 16-bit wide output signal *Valid*. Each bit within this signal represents the validation of this segment. If one of the bit *Valid[n]* is zero, it means that all the values in the segment represented by this bit are 0. In other words, the segment is judged to be invalid and will not be transmitted by the SerDes link. Within a segment, as long as at least one value is 1, it can be judged as valid.

After judging each segment, the next step is to integrate the *Valid* signal and values in each segment according to the result to generate a new packet. As Figure 3.19 shows, the newly generated packet *Packet_out* has a bit width of 272 bits. The *Valid* signal is placed in the header of the *Packet_out*, indicating the location and number of valid segments. The segments judged to be valid will be inserted into the *Packet_out* sequentially according to the original order, but the segments judged to be invalid are omitted, which reduces the amount of information transmitted. Considering the worst case, all segments are judged to be valid, plus the *Valid* signal, a total of 272 bits of signals need to be transmitted, which is why the bit width of the output signal *Packet_out* is 272 bits. Considering the 20% spike rate factor, when the system is running, most of the values in signal *Packet_out* are 0 in most cases.

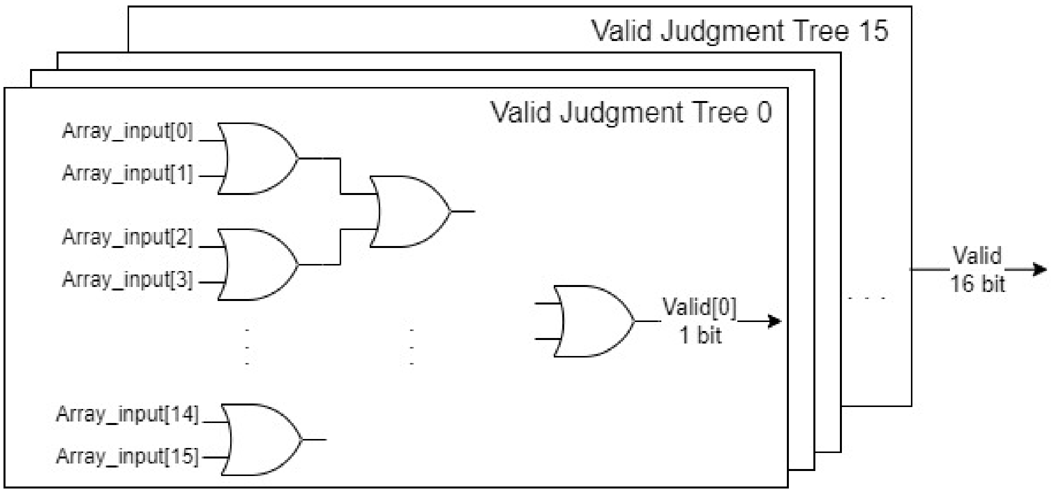


Figure 3.18: OR Gate Tree to judge validation.

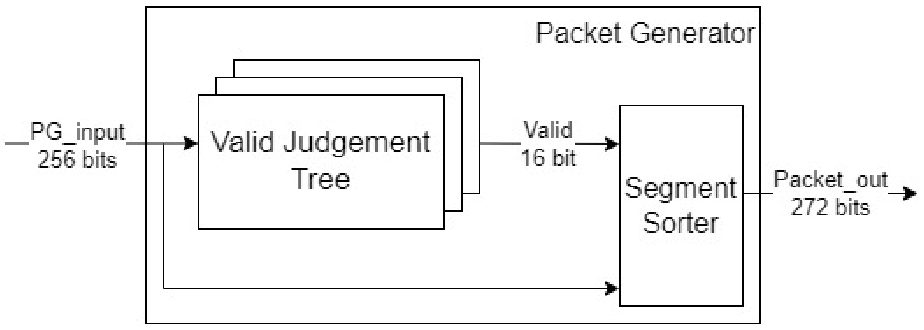


Figure 3.19: Inner Structure of a Packet Generator.

3.6.2 Crossbar

In a neuron network, the output from the previous neuron array is not in one-to-one correspondence with the input at the receiving end, which means that any output bit may be connected to any input bit. And the connection mode of neuron arrays varies between different arrays. Moreover, these connections are required to be configurable under different circumstances. To solve that, a crossbar mechanism needs to be proposed to solve the mapping problems between the neuron arrays properly.

In a network, a crossbar switch is used to transport data or signals between two separate sites. The crossbar setup is a matrix in which each crossbar switch runs between two locations, in a design which is intended to connect every component of an architecture to every other component. To conclude, crossbar switches are designed to implement all permutations of connections among any inputs to any outputs.

In our communication system, the crossbar is designed as a generic SystemC model to allow easy customization of the number of ports needed for a given use case. Consistent with the previous design, the crossbar divides a 256-bit-width packet into 16 segments and builds a 16-16 connection system. As this component is designed in a behaviour

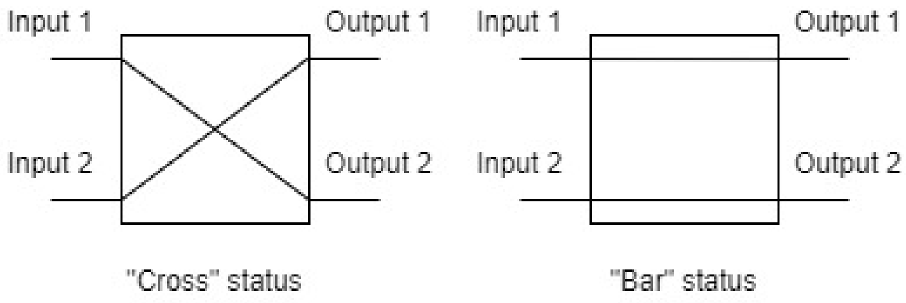


Figure 3.20: Crossbar switch states. [19]

model, the code implementing it does not involve the actual structure of the crossbar switch. For this reason, we propose two possible designs based on the crossbar's functionality: one is the Switches and the other is the Programming Logic Array(PLA).

- Switch

  Originally, a crossbar switch is developed for interconnection networks of multiprocessors. In the beginning, a crossbar is a 2x2 buffer-less switch. It could be in one of the two states, cross or bar, which is the reason why it is named crossbar. Figure 3.20 shows the input-to-output permutation of a basic crossbar in two states.

  By implementing any input-to-output permutation with additional inputs and outputs, the concept of a crossbar switch was extended to larger sizes of switches. A crossbar switch's architecture is simple yet resource-intensive, as it must implement all possible permutations of inputs to outputs. One of the typical designs of a crossbar switch is shown in Figure 3.21, where 4 switches are used to implement this basic 2x2 crossbar switch. With the growth of the number of switch terminals, the cost of the crossbar switch increases at a rate of $O(N^2)$, where N is the number of terminals. To make matters worse, the large-sized crossbar switch is composed of the basic switch cascaded in Figure 3.21. The larger the size of a crossbar switch, the higher the physical interconnect distance and delays, which led to the development of networks of crossbar switches [19]. Due to the above defects, when considering the Resister Transfer Level design of this component, this traditional crossbar switch topology will not be applied in our system.

- PLA

  A programmable logic array (PLA) is a type of logic device that is used to construct combinational logic circuits. As shown in Figure 3.22, the PLA consists of a set of programmable AND Gates and OR Gates, which will then be conditionally complemented to form an output. For a PLA with N inputs variables, it requires to have 2^N. And for M outputs from a PLA, there should be M OR Gates, each having programmable inputs from the output of the AND Gates. Many logic functions may be synthesised in the sum of products canonical form using this arrangement.

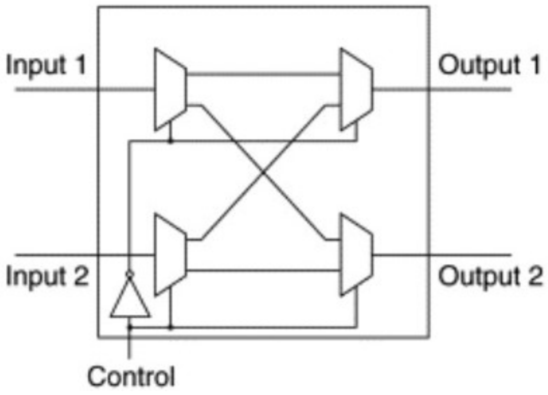


Figure 3.21: Typical design of crossbar switch.

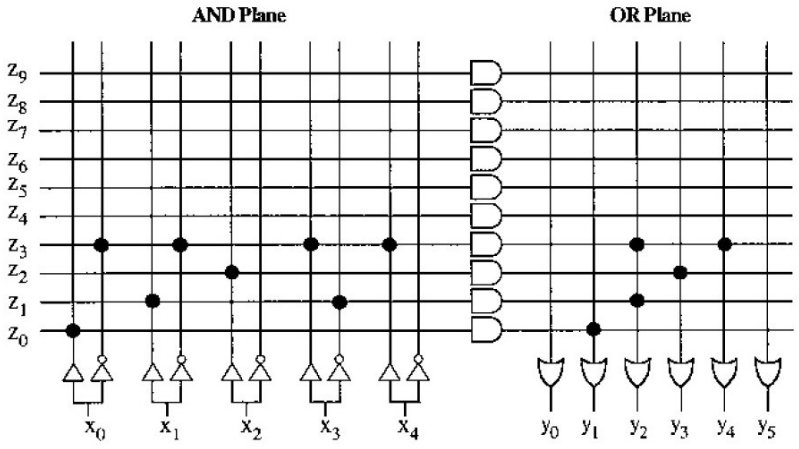


Figure 3.22: PLA with an AND array followed by an OR array. [23]

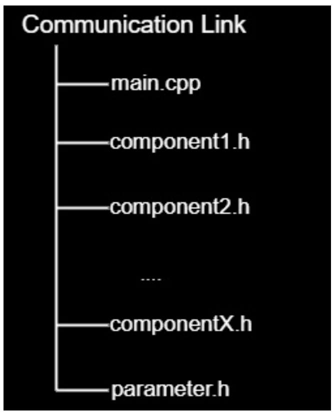
Enters from the lower left-hand side, the inputs are named as $x_0, x_1, ..., x_{N-1}$ in turn. Along the vertical lines, their complemented and uncomplemented values are fed into the AND array. The term z_i from the horizontal line indicates a product term, which is the logic AND of all the inputs linked to it by the connections (represented by the black points in AND Plane).

In Figure 3.22, for example, the first term z_0 is equal to z_0, the second term z_0 is equal to $x_1\overline{x_3}$, and the fourth term z_3 is equal to $\overline{x_0}\overline{x_1}x_3x_4$. Each vertical line in the OR Gate Plane represents a sum of the term z linked to it, which is also represented by the black points. For example, in Figure 3.22, the term y_2 equals to $z_1 + z_3$. When these two arrays are combined, they offer a programmable method for constructing any three-level not-and-or logic function of the input signals.

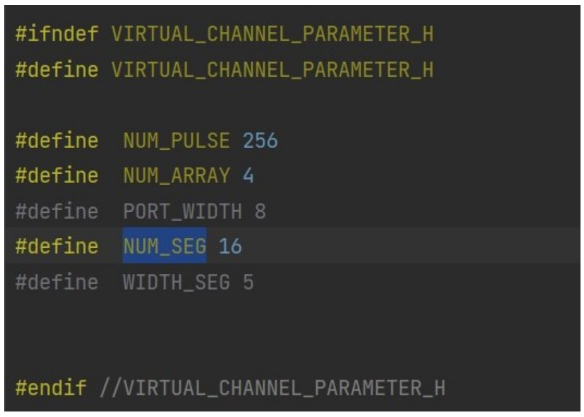
Compared with the purely switching method mentioned in the previous chapter. PLAs have more flexibility than it does since the connections between the AND and OR gates are programmable. Also, the structural design of PLA has a higher utilization rate for its gates, making it more power-efficient than the previous method [11], which is crucial in our low-power system design.

3.7 System Parameterization

This multi-point communication link is designed to be configurable and parameterizable. Some criteria should be developed to determine the sequence of priority When data from different arrays need to utilize the same SERDES link. Moreover, as the average spiking rate for a neuron array is around 20%, sending the whole packet generated through the SERDES link wastes a great amount of unnecessary energy. A more power-efficient solution is to divide the whole into several segments. The point here is that cutting the entire packet into several segments is a process that needs to be evaluated. The appropriate value of the number of segments can be valuable depending on the changes in the test environment and test data. Building the whole modelling in a parameterizable way could make this process much more convenient. As Figure 3.23(a) shows, a header file dedicated to storing relevant parameters is created separately. Before each simulation of this link system, as shown in Figure 3.23(b), the relevant parameters in this file will be predefined to support adjusting the system according to the requirements.



(a) The tree file in project folder.



```
#ifndef VIRTUAL_CHANNEL_PARAMETER_H
#define VIRTUAL_CHANNEL_PARAMETER_H


#define  NUM_PULSE 256
#define  NUM_ARRAY 4
#define  PORT_WIDTH 8
#define  NUM_SEG 16
#define  WIDTH_SEG 5


#endif //VIRTUAL_CHANNEL_PARAMETER_H
```

(b) The content in head file parameter.h .

Figure 3.23: (a) A simplified version of the file tree diagram within this project folder. (b) The listed parameter in the head file parameter.h.

4 Simulation and Results

In this chapter, the power, performance and area of this interconnect system are evaluated with different kinds of application scenarios. A data set from a trained MNIST network is injected to provide the data source for neuron arrays in our system. This helps the testbench to build the simulation environment and verify the functionality of the system. Meanwhile, the real delay of each component and the wire is considered to investigate their effects on the system.

4.1 Simulation with MNIST dataset

In this part, certain data sets from MNIST training results are injected into this on-chip interconnect system to verify the function of our system. In the meantime, different mapping approaches between the MNIST data set and the neuron arrays in this interconnect system are presented. As a result, additional functional criteria can be taken into consideration to evaluate the working conditions of the system. Furthermore, different sets of waveforms for different patterns are illustrated and described. Finally, we studied the system's limitations and determined the throughput range for various application cases.

4.1.1 Mapping to a real MNIST network

Within an MNIST neural network, there exist three types of layers, which are the input layer, the hidden layer as well as the output layer. In our simulation environment, the amount of data is only a quarter of a standard MNIST network. As Figure 4.1 shows, a standard image from an MNIST dataset is seen as a matrix with the size 28x28. While in our use case, only a quarter of the pixels are used to inject into the neural network. In this quarter of the MNIST network, 4 sets of dates encapsulated in separate .txt files are provided, named Spike_in, Spike_h1, Spike_h2 as well as Spike_out. Also, an extra file which stores the time interval between the spikes is provided to offer these data with a uniform timestamp.

Once the appropriate data sources are obtained, the next step is to map this quarter of the MNIST network into our interconnect system and then carry out the simulation for the system. As shown in Figure 4.1, there are only 3 layers of neurons within this network. Our interconnect system, on the other hand, contains four neural arrays. To map this neuron network in our communication link system, the neuron array 0 acts as a virtual input in our testbench. Mapped to the level of reality, this virtual input layer could be regarded as an array of sensors, which read the pixel from the source and

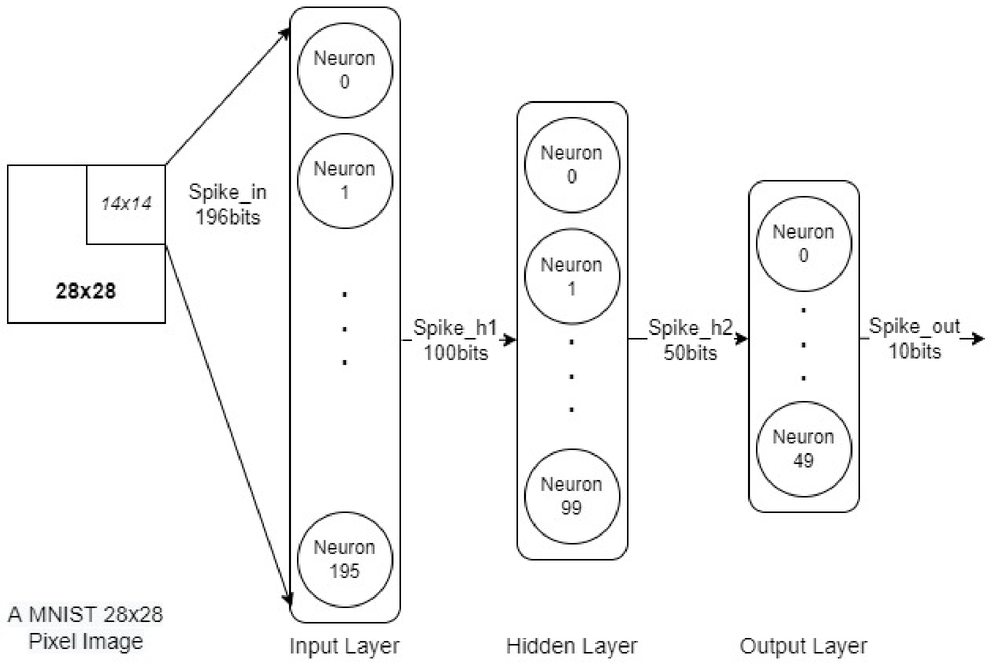


Figure 4.1: The inner structure of a quarter of MNIST Neural Network

transfer the data into our system in the form of spikes. As for the other three neuron arrays, they are in charge of operating as the input layer, the hidden layer, as well as the output layer respectively.

As the design of a neuron array is not involved in this project, after a neuron array receives the signal transmitted through the SerDes link, on the transmitter side, the neuron array will output the packet from the injecting MNIST dataset, instead of the data it received. An interface is created at the output of the neuron array, which allows the data to be injected into the pulse latches (buffers) of each of the controllers. As shown in figure 4.2, these interfaces are represented as red arrows, from the MNIST Dataset to the output of each array. The output bit width of each neuron array is 256 bits, which is larger than the bit width of any layers from the MNIST dataset provided. The bits which are not covered by the data provided by the MNIST dataset are filled with logic 0 by default.

4.1.2 Simulation Waveform

According to the file which stores the timestamp for the testbench, the whole simulation process starts from 0ms to 19.5ms. During the simulation, four neuron arrays keep on extracting data from the MNIST Dataset, which generates thousands of packets. Because of that, it is too ambiguous to illustrate the whole simulation result in one figure. Instead, the waveform that contains the entire simulation process is divided into several segments. By zooming in on the specific segments of the waveform, the working principle and data flow of the whole system can be elaborated in detail.

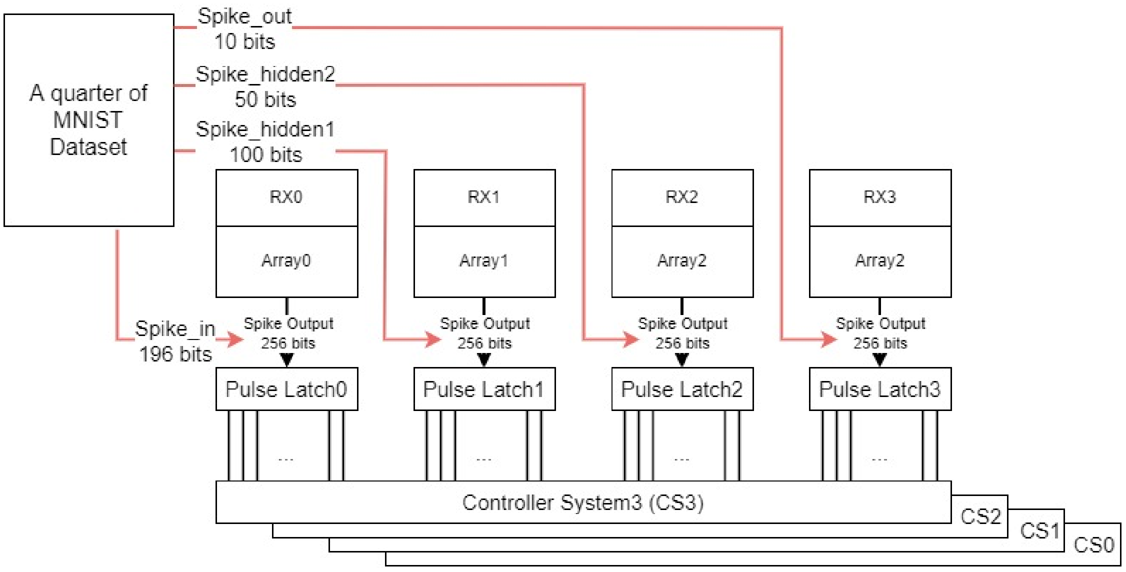


Figure 4.2: The mapping of an MNIST network to this communication system

As shown in figure 4.2, once a neuron array receives a packet from its receiver, it will extract a packet from the MNIST Dataset. This packet will then be processed by the controller and transmitted to its destination through a SerDes link. Figure 4.3 illustrates one transmission between array 0 and array 1 under this methodology. The arrows in red represent the data signal while the arrows in black represent the control signals which are triggered by the data flow. A more detailed explanation with relevant waveform will be provided in the follow-up content.

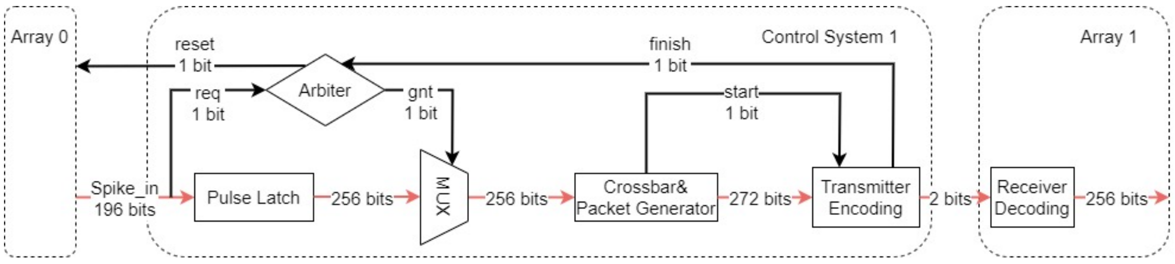


Figure 4.3: The data flow to transmit and receive a packet

The first waveform illustrated in figure 4.4 depicts the system's beginning stage. First, the *global_reset* signal turns from low to high, clearing all pending events or errors and returning the system to a normal state. As there is no spike generated from the neuron array at that time, all the components mentioned in figure 4.3 are in an idle state. Then at 30ns, a spike packet is injected from array 0 into its corresponding Pulse Latch. As shown in figure 4.4, the *pulse_in(0)* signal with 256 bit-width, which represents the output from array 0, changes from 0 to the values read from the *Spike_in*.

In this process of transmitting a packet from array 0 to array 1, only the *Spike_in* from the MNIST Dataset has a specific value while the outputs from the other three are all

0. Because of that, to make the waveform diagram not too cluttered, the outputs from the other three neuron arrays are not included. For the same reason, only the receiver waveform of array 1, *RX_out(1)* is shown in the figure.

Once the arbiter detects a packet in *pulse_in(0)*, it will receive a requirement on the bit corresponding to the array 0, which is the least significant bit in our case. As there is only one requirement among the four arrays, the *gnt* signal from the arbiter turns to '0001', operating the MUX to allow this packet to occupy the subsequent components. In order to facilitate the comparison of the data at the receiving end and the sending end, the data will bypass the Crossbar component directly, which means the segments within this packet will not change their sequences. After the *Packet Generator* component finishes processing the data, it will generate a pulse named *start* to the transmitter. After that, the packet can be transmitted to the destination array through the SerDes link. It could be observed from the waveform that a bunch of pulses are generated in signal *SerDes_ONE* and *SerDes_ZERO* after the rising edge from the signal *start*.

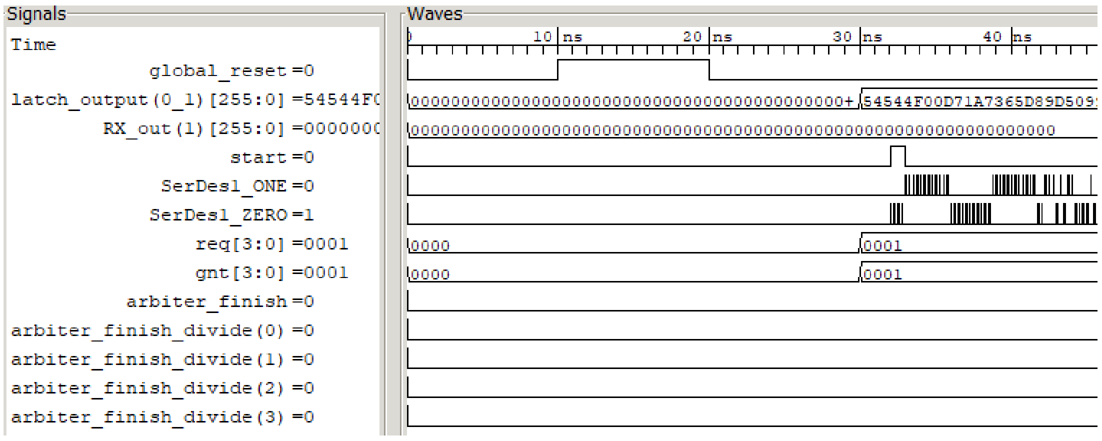


Figure 4.4: The waveform when transmitting a packet

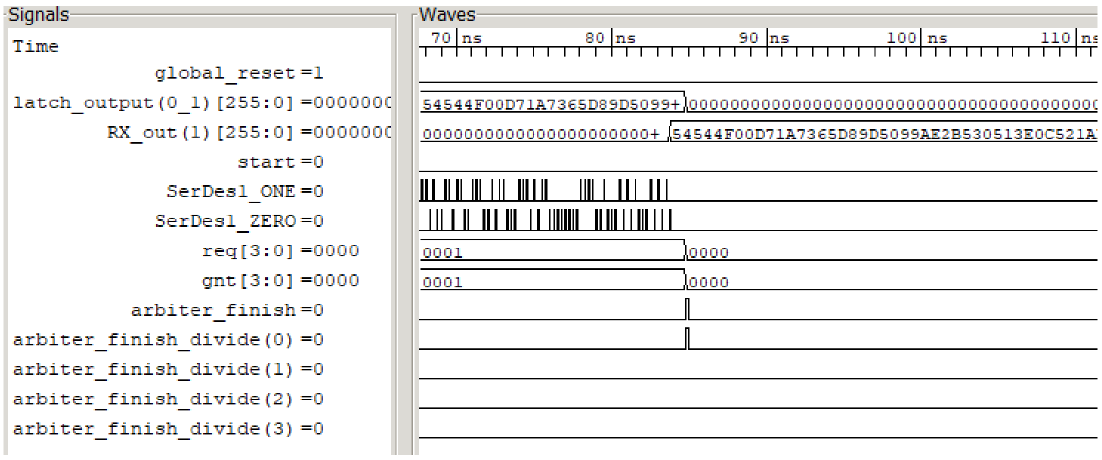

Signals
Waves
Time
70 ns
80 ns
90 ns
100 ns
110 ns
global_reset =1
latch_output(0_1)[255:0] =0000000
RX_out(1)[255:0] =0000000
start =0
SerDes1_ONE =0
SerDes1_ZERO =0
req[3:0] =0000
gnt[3:0] =0000
arbiter_finish =0
arbiter_finish_divide(0) =0
arbiter_finish_divide(1) =0
arbiter_finish_divide(2) =0
arbiter_finish_divide(3) =0
54544F00D71A7365D89D5099+
0000000000000000000000000000000000000000
0000000000000000000000+
54544F00D71A7365D89D5099AE2B530513E0C521A
0001
0000
0001
0000

Figure 4.5: The waveform when receiving a packet

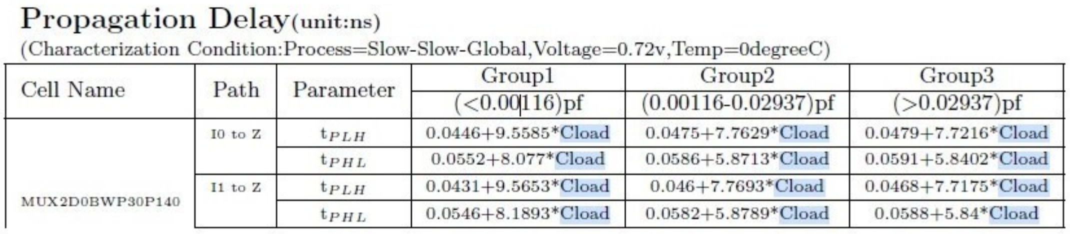
Figure 4.5 depicts a successful packet reception in neuron array 1. At around 85ns, the arbiter receives a finish signal from the transmitter, claiming that the packet has been received successfully. The final decoding result can be found in signal *RX_out(1)*, which is fully equal to the packet sent from *pulse_in(0)*. Also, once the arbiter receives the finish signal, it will send a reset signal to the pulse latch corresponding to that array, which is represented as the least significant bit as well. This signal helps the pulse latch to clear the packet that has already been transmitted so that it can receive new pulses from the array.

4.1.3 Timing and Throughput

Throughput is an essential measure for evaluating the performance of this interconnect system. Based on the waveform produced during the simulation, this interconnect system works properly under the previous MNIST use case. However, this is only a functional simulation which focuses on logic function rather than timing. In the previous analysis, all the gates used in this system are assumed to be ideal, which means that the actual delay of each gate is ignored. As a result, the delays and latencies of our system may not be accurately reflected in the simulation results.

In order to objectively evaluate the throughput of the system, the first step needs to do is to introduce a real physical delay for each gate. Under normal circumstances, this step could be achieved during the synthesis. Synthesis is the process of converting RTL into a gate-level netlist. There is a step in the synthesis process to optimize the netlist based on the designer's delay, area, and other limitations. However, this on-chip interconnect system is developed in SystemC system-level modelling and it is hard to introduce the standard delay file SDF for timing simulation (post-simulation).

Consequently, We made a delay estimate for all types of gates used in the project. Take a 2-input multiplexer as an example, by searching relevant keywords 'MUX' in the datasheet of TSMC N28HPC Standard Cell Library [24], some data could be found regarding the delay of this type of gate. As shown in figure 4.6, the propagation delay of the gate varies as the working environment changes, while we do not have accurate data on the working environment. To compensate for this, a compromise solution is to add up all the values and take the average. Although the delay obtained in this way will have a large error with the actual one, at least their order of magnitude is at the same level.



Propagation Delay(unit:ns)

(Characterization Condition:Process=Slow-Slow-Global,Voltage=0.72v,Temp=0degreeC)

Cell Name	Path	Parameter	Group1	Group2	Group3
			(<0.00116)pf	(0.00116-0.02937)pf	(>0.02937)pf
MUX2D0BWP30P140	I0 to Z	t_{PLH}	0.0446+9.5585*Cload	0.0475+7.7629*Cload	0.0479+7.7216*Cload
		t_{PHL}	0.0552+8.077*Cload	0.0586+5.8713*Cload	0.0591+5.8402*Cload
	I1 to Z	t_{PLH}	0.0431+9.5653*Cload	0.046+7.7693*Cload	0.0468+7.7175*Cload
		t_{PHL}	0.0546+8.1893*Cload	0.0582+5.8789*Cload	0.0588+5.84*Cload

Figure 4.6: The Delay of a 2-Input MUX Gate [24]

The basic logic of designing the whole system is to design each component individually and connect them together. Because of that, after the delay of each component is clarified, the overall delay simulation of the system is naturally completed. Take the component of the receiver as an example, the inner structure of a receiver is illustrated in figure 3.12 and figure 3.13. It could be judged that each bit needs to be processed by one And Gate and one Latch Gate, which is the critical path of this component. As mentioned in section 3.3, the event-triggering mechanism of the SystemC is very helpful to simulate the propagation delay in this case. The code shown below is a part of the process of a component used to activate its data processing function. The second line of code is the processing method without delay under ideal conditions. As a comparison, the third line of code simulates the delay according to the amount of data that needs to be decoded by the receiver. The variables *LATCH_Delay* and *AND_Delay* are the propagation delays of the corresponding gates estimated from [24]. Furthermore, considering the actual working situation, the delay of each gate is not completely constant when it is working. For a more realistic simulation, a function named *random* is prepared to provide a 10% delay fluctuation, as shown in line 4.

```
1           else if ( count==(16 + 16*num_spike) ){
2               //ev1.notify();
3               ev1.notify((16+num_spike)*(LATCH_Delay+AND_Delay),SC_NS);
4               //ev1.notify(random((16+num_spike)*(LATCH_Delay+AND_Delay)
                   ,0.1),SC_NS);
5               count = 0;
6               num_spike = 0;
7           }
```

After introducing the physical delay to the system, the overall waveform did not change significantly and the system still works properly. This indicates that the current test-bench has not met the throughput limit of this system. Hence, for the MNIST use case, we multiply the numbers in timestamp by the same value, so that the time interval between each of the spike packet decrease at the same rate. When the time interval is reduced to a certain extent (269.47ns on average), it could be found that some packets are lost during the transmission when observing the waveform. Thus, the throughput of our interconnect system under this MNIST use case can be calculated as:

$$Throughput = \frac{4 \times 256}{269.47 \times 10^{-9}} \approx 3.802\ Gbits/s \tag{4.1}$$

4.2 Power and Area

As this on-chip communication system is developed in system-level modelling using SystemC, the code which builds up this system cannot be directly synthesized. As a result, it is quite difficult to acquire accurate data in terms of the power and area of this interconnection system. In order to evaluate the area of this system, what we can do is estimate the gate number of each component based on the code that builds them. Furthermore, for a specific application scenario, the number of gates switching times can be recorded to get a power estimation. The following sections demonstrate the result of the area and power estimations respectively.

- Area

  To estimate the area of this interconnect system, the inner structure of each component needs to be evaluated. For example, the OR Gate Tree group as illustrated in Figure 3.18 needs a bunch of OR Gates to implement its structure. As there are 16 segments and each segment contains 16 bits, the number of OR Gates needed for this component can be calculated:

  $$OR\_Gate = 16 \times (8 + 4 + 2 + 1) = 240 \tag{4.2}$$

  Because of that, the changes in simulation parameters could result in different structures of the relevant component. Again take the OR Gate Tree group in Figure 3.18 as an example, if the number of segments changes from 16 to 8, the number of OR Gates needed is:

  $$OR\_Gate = 8 \times (16 + 8 + 4 + 2 + 1) = 248 \tag{4.3}$$

  In order to ensure that the system structure of the analysis part is consistent with the previous system design part, the following calculations are based on this premise: each neuron array in the system contains 256 neurons, and a 256-bit packet is equalized divided into 16 segments during processing.

  Besides, within this pulse-mode link system, the gate is divided into two categories, one is the normal gate and the other is the pulse-mode gate. Applying the pulse-mode gate to our design is due to the fact that the neurons generate spike signals to carry information within a neural network. However, because of the short duration with the narrow width of a spike, normal logic gates are incapable of handling such high-speed signals. Thus, the pulse-mode gate family are applied in our system, which includes Pulse Latch, Pulse AND as well as Pulse OR gate. A previous contributor has already implemented these pulse-mode gates. The data on the power consumption of these gates is also cited in that contributor's article.

  Table 4.1 and Table 4.2 record the number of logic gates required to build a transmitter and a receiver including both normal and pulse-mode gates. Table 4.3 shows the gate number in a packet generator system. As mentioned before,

Table 4.1: Gate number of the Transmitter

Number of Segment =16	Transmitter_process					
	valid_spike_sel	pulse_generator	one_zero_line	or_gate_tree	3DFF	Total
Pulse Latch	0	0	0	0	0	0
Pulse And	0	0	0	0	0	0
Pulse Or	0	0	0	0	0	0
AND Gate	0	108	16	562	0	686
OR Gate	240	25	8	27	0	300
XOR Gate	0	59	16	18	0	93
NOT Gate	0	3	0	0	0	3
MUX	0	0	480	0	0	480
DFF	0	0	0	0	792	792
Delay Element	0	4	0	0	0	4

Table 4.2: Gate number of the Receiver

Number of Segment =16	Receiver_process			
	do_count	spike_count	ser_to _pal	Total
Pulse Latch	0	0	272	272
Pulse And	0	0	0	0
Pulse Or	272	0	0	272
AND Gate	18	26	8	52
OR Gate	19	5	4	28
XOR Gate	18	16	8	42
NOT Gate	0	2	0	2
MUX	0	14	480	494
DFF	0	0	0	0
Delay Element	0	4	0	4

this packet generator system works as a global controller to operate the data path for physical SerDes links with multiple virtual channels.

From the table shown above, it can be indicated that the number of gates needed for a crossbar is much larger than other components. And the area consumed by the transmitter and receiver is significantly larger than the part where the

Table 4.3: Gate number for Packet Generator System

Number of Segment =16	PG_process					
	spike_detec	arbiter	round-robin	reset_divid	Corssbar	Total
Pulse Latch	0	0	0	0	0	0
Pulse And	0	0	0	0	0	0
Pulse Or	0	0	0	0	0	0
AND Gate	8	12	16	8	65536	65580
OR Gate	4	4	4	4	16	32
XOR Gate	8	8	8	8	0	32
NOT Gate	0	4	0	0	0	4
MUX	8	0	0	0	0	8
DFF	0	0	0	0	0	0
Delay Element	3	1	2	1	0	7

controller removes the crossbar part. In order to display the area consumed by the system more intuitively, the kilo Gate Equivalent (kGE) method is used to define the manufacturing-technology-independent complexity of this circuit. The silicon area of a two-input drive-strength-one NAND gate typically forms the technology-dependent unit area, which is frequently referred to as gate equivalent. According to the complexity of the implementation, various gates are first converted to a number of NAND Gates according to their respective ratios, and then the data from the table 4.1, 4.2, 4.3 are accumulated. The results are shown in Table 4.4.

Table 4.4: The Gate Counting for the System with kGE

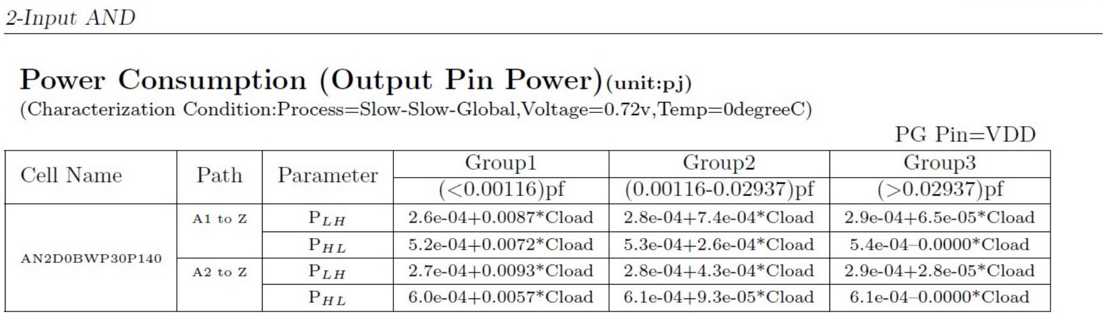
Gate Counting		
Gate	1 array	4 array
Pulse Latch Gate	272	1088
Pulse And Gate	0	0
Pulse Or Gate	272	1088
AND Gate	66318	265272
OR Gate	360	1440
XOR Gate	167	668
NOT Gate	9	36
MUX	982	3928
DFF	792	3168
Delay Element	15	60
kGE(kilo Gate Equivalent)	207.924	831.696

- Power
  The power consumption of this system will be estimated in this subsection. Considering the fact that dynamic power always consumes the majority of overall power usage, the effect of the static power of the circuit is ignored. To calculate the dynamic power consumption, the first thing that needs to be clarified is the energy consumption for each switching of the different gates. In [26], a previous contributor has already implemented the pulse-mode logic gates. Table 4.5 in this thesis displays the energy consumption per switching of the pulse-mode gates family, where the power supply remains at 0.8 volts and the pulse has a period of 200ns. As for the energy consumption of normal logic gates, we refer to the TSMC Standard Cell Library. Each basic logic gate will have a set of relevant data when transferred from high to low and low to high. For example, figure 4.7 shows a group of data under a certain scenario. Since there are multiple sets of data for the same type of gate, our processing method is to accumulate these data and take an average value, and use this average value to represent the power consumption of this gate. Although this is only a rough assumption and not accurate, at least it can ensure that the values we use are in the same order of magnitude as the real values.

  After determining the power consumption of each gate, the next step is to get the switching time of these gates. This problem cannot be simply summarized as

Table 4.5: The Energy Consumption For Pulse-mode Gate Family

Pulse-mode Gate	Energy Consumption(J)
Pulse Latch Gate	3.47E-15
Pulse And Gate	6.17E-15
Pulse Or Gate	6.90E-15
Pulse Delay Gate	5.68E-15



2-Input AND

Power Consumption (Output Pin Power) (unit:pj)
(Characterization Condition:Process=Slow-Slow-Global,Voltage=0.72v,Temp=0degreeC)

PG Pin=VDD

Cell Name	Path	Parameter	Group1	Group2	Group3
			(<0.00116)pf	(0.00116-0.02937)pf	(>0.02937)pf
AN2D0BWP30P140	A1 to Z	P_{LH}	2.6e-04+0.0087*Cload	2.8e-04+7.4e-04*Cload	2.9e-04+6.5e-05*Cload
		P_{HL}	5.2e-04+0.0072*Cload	5.3e-04+2.6e-04*Cload	5.4e-04–0.0000*Cload
	A2 to Z	P_{LH}	2.7e-04+0.0093*Cload	2.8e-04+4.3e-04*Cload	2.9e-04+2.8e-05*Cload
		P_{HL}	6.0e-04+0.0057*Cload	6.1e-04+9.3e-05*Cload	6.1e-04–0.0000*Cload

Figure 4.7: The Power Consumption of a 2-Input And Gate

a calculation of how many times each component is called, because the process in a component may be activated multiple times. To get as accurate values as possible, a global variable for each process is created for each of the processes. The process of using this method is shown in the code below. Since these variables are global, their values of them will not be lost every time the corresponding process ends. Instead, every time each process is triggered, the value of its corresponding variable is incremented by one. When the entire simulation ends, these variables are recorded as outputs to facilitate our subsequent calculations.

```
// At the head of a file
int pulse_latch_time = 0;
int pulse_reset_time = 0;

// Within a process
if(flag_reset){
    pulse_reset_time +=1;    //count the reset time

 //At the main body
cout << "Pulse reset time is " << pulse_reset_time <<endl;
cout << "Pulse latch time is " << pulse_latch_time <<endl;
```

To evaluate the power consumption of the system, a specific use case should be provided. On this basis, the switching time can be printed out, multiplied by the power consumption value of a single gate per flip. After that, the total power consumption of this system can be estimated. According to our assumption, the speed of the on-chip link is 2GHz and the time interval between two packets is 5us. Two use cases are defined below to reveal the power consumption in different situations.

- Case 1:
  There are 4 neuron arrays placed on a single chip. Each neuron array sends a packet to another adjacent neuron array for every 5us.
- Case 2:
  There are 4 neuron arrays placed on a single chip. Each neuron array sends a packet to all the other 3 neuron arrays for every 5us.

By applying the method as the code above shows, we could get the number of times each process was activated during the simulation. From table 4.1, 4.2 and table 4.3, the number of gates triggered in each process can be extracted. According to the above two sets of data, the switching time of each gate can be calculated, as shown in the equation 4.4. Also, the power consumption of all kinds of gates used in our system has already been clarified. Thus, the calculation of energy consumption can be processed. The equation 4.5 illustrates how to calculate the power consumption of the system.

$$Number_{gates\_switching} = \sum_{process} (Trigger\_times \times Gates\_in\_process) \tag{4.4}$$

$$E_{gates} = \sum_{gate} (Switching\_times \times Energy\_per\_switching) \tag{4.5}$$

Refer to the data in a relevant article [5], it is assumed that each spike takes 7.36pJ to propagate through the on-chip link. With this data, the total energy of the system can be determined. Then based on equation 4.6, the power of the system can be calculated from the energy consumption during the whole simulation. In this equation, the α refers to the energy to transfer a single spike, which is 7.36 in our case. The β refers to the total number of packets transferred by this link system. And the parameter E_{gates} is obtained from the equation 4.5.

$$P_{system} = \frac{E_{gates} + \alpha \times Packet\_Size \times \beta}{Total\_Time} \tag{4.6}$$

From the Area analysis (part 4.2), it can be concluded that the number of gates consumed by the component *Corssbar* occupies more than 90% of the system.

And, the chip area consumed by the controller part except the crossbar is negligible compared with the transmitter and receiver. Therefore, it is necessary to divide the same use case into two scenarios for discussion. In one scenario, the crossbar works normally, and the segments in a packet will be arranged according to the configuration before being transmitted; in the other case, the crossbar does not work, and the segment selected as "valid" will directly bypass this part.

Table 4.6: Energy Consumption of the Transmitter

	Switching Time	Sum	Total energy
Pulse Latch	776	2.69E-12	1.48E-11
Pulse And	0	0.00E+00	
Pulse Or	0	0.00E+00	
AND Gate	16341	1.10E-11	
OR Gate	978	5.17E-13	
XOR Gate	891	3.45E-13	
NOT Gate	41	2.17E-14	
MUX	341	1.04E-13	
DFF	8	2.10E-14	
Delay Element	55	2.42E-14	

Table 4.7: The Energy Consumption of the Receiver

	Switching Time	Sum	Total energy
Pulse Latch	549	1.90E-12	1.38E-11
Pulse And	0	0.00E+00	
Pulse Or	567	3.91E-12	
AND Gate	6029	4.07E-12	
OR Gate	2058	1.09E-12	
XOR Gate	4662	1.81E-12	
NOT Gate	273	1.45E-13	
MUX	2778	8.49E-13	
DFF	0	0.00E+00	
Delay Element	39	1.71E-14	

Table 4.6 and 4.7 show the energy consumption of a transmitter and a receiver when they transfer a 256-bit packet respectively. And the 4.8 shows the energy consumption for the global controller to allocate the data path for a packet. There are two values in the rows of AND and OR gates in Table 4.8, in which the values in brackets represent the overall energy consumption of the part when the crossbar in the controller is working. From these tables, it could be concluded that when the power consumption of the transmitter and the receiver is quite similar. When the crossbar in the controller is working, the overall energy consumption of this part accounts for 80.4% of the total energy consumption. However, when the crossbar is not working, the energy consumption of the controller is negligible compared to the transmitter and receiver.

After estimating the energy consumption, the next element to consider is the

Table 4.8: The Energy Consumption of the Global Controller

	Switching Time	Sum	Total energy
Pulse Latch	0	0.00E+00	
Pulse And	0	0.00E+00	
Pulse Or	0	0.00E+00	
AND Gate	3573(145745)	2.03E-12 (9.84E-11)	
OR Gate	54(1085)	2.84E-14 (2.99E-14)	2.14E-12
XOR Gate	90	3.49E-14	(9.85E-11)
NOT Gate	44	2.31E-14	
MUX	36	1.09E-14	
DFF	0	0.00E+00	
Delay Element	30	1.33E-14	

time factor. As mentioned in equation 3.1, the time needed to propagate a 256-bit packet through the SerDes link is about $0.5\mu s$. According to [12], the latency of an on-chip communication is in the order of nanoseconds, which is significantly smaller than the results from equation 3.1. Considering the worst case, a pair of a transmitter and a receiver needs to encode and decode three 256-bit packets for every $5\mu s$, and the time required to finish this operation is less than $2\mu s$. Therefore, these two use cases do not break the theoretical timing constraint of the system.

Table 4.9 displays the energy consumption and the power of this on-chip communication link system. In the same manner as in Table 4.8, the data in brackets represents the estimation of energy and power when the crossbar is working. According to this table, power obviously changes with different use cases. As more packets are transmitted and travel through the SerDes link, the energy as well as power increases.

Table 4.9: The Energy consumption and Power of the System

	α	β	Total Energy(J)	Total Power(mW)
Use case 1	7.36E-12	8	1.633E-8 (5.084E-8)	6.519(20.295)
Use case 2	7.36E-12	24	9.786E-8 (2.926E-7)	19.578(58.544)

5 Conclusion and Future Work

5.1 Conclusion

Within a neuromorphic computing system, a high-speed interconnect link system is needed to transfer the packets between the neurons. Moreover, as the spiking neural networks are event-driven, the communication link system generally excludes the clock signal and associated blocks.

A customized high-speed on-chip interconnect system in self-timed burst mode asynchronous logic is proposed in this thesis. This on-chip interconnect system is basically created from a SerDes link, which eliminates the clock signal and uses the burst-mode protocol to transform the packet from parallel to a two-wire serial format. To capture the spikes from the neuron arrays, a customized pulse latch gate designed by a previous contributor is needed. Each neuron is assigned a SerDes link with its matching transmitter and receiver. The resources of these SerDes links are uniformly scheduled by their controllers. As the neural array generates spikes at random, it is very common to cause a collision within a SerDes link. To address this issue, the arbitration and virtual channel working mechanism are introduced into the controller. To further improve the throughput and decrease power consumption, a packet is split into multiple segments before being transmitted, and the segments that do not contain valid information will be filtered out. Furthermore, the system remains adaptable and parameterizable throughout the design process, allowing it to be configured with different use cases.

The system is designed in SystemC, which is quite difficult to be synthesized. In order to evaluate the power, area and throughput performance, we use the datasheet of the TSMC28nm library [24] to estimate the area, power consumption and delay of each gate. To verify the system's functionality, we map a neural network from a quarter MNIST training result into our interconnect system. In order to better simulate real working scenarios, a 10% bandwidth to the propagation delay of each gate is added during the simulation. With the shrink of the time interval between the spike packets, the throughput limit of this interconnect system is reached. In this MNIST testing case, this interconnect system's throughput is around 3.802 *Gbits/s*, which meets the demands of this application scenario. The power of the system varies greatly with the change in the use case. Considering the worst case, when all neuron arrays generate packets at the same time and send them to all other neuron arrays, the power consumption of the system is 58.544mW, most of which is consumed by the crossbar.

5.2 Future Work

Although the functionality of our interconnect is verified, this project still has deficiencies to be improved in the future. Some of the possibilities where further work might be optimized are as follows:

- Synthesis of the Circuit
  In this project, SystemC is used to design the system in high-level modelling. Because of that, the synthesis of the circuit is missed. The core components of this system can be rewritten in a synthesizable language such as SystemVerilog in the future so that more accurate data on power consumption, area and throughput can be obtained.
- Hierarchical Routing of Wires
  As shown in figure 3.1, the outputs of the pulse latches are directly transmitted to the control system through thousands of wires. This not only occupies a large amount of on-chip space but also causes crosstalk interference between signal lines, affecting data transmission in other signal lines. In the future, we could consider using the hierarchical wiring method to optimize this part, or perform an additional serial-to-parallel operation at the expense of a part of the transmission rate.
- More Flexible Control Of Data Flow
  According to the current design of this interconnect system, the flow of data is defined before the system starts to operate. That is to say, we cannot temporarily change the flow of data between neuron arrays when the program is running halfway. This limits the flexibility of the system. For example, the mapping of MNIST dataset is illustrated in figure 4.2. However, the output of the hidden layer and the output layer only occupies a small part of a neuron array. If the direction of data flow between neuron arrays is variable, the mapping strategy can be shown in Figure 5.1. With the crossbar, the *spike_in* and *spike_out* can share the same neuron array, while the other two *Spike_hidden* spikes could use another neuron array. This can significantly reduce the area and power consumption. Furthermore, a more flexible control makes it feasible to map with a recurrent neuron network, which allows output from some nodes to affect subsequent input to the same nodes.

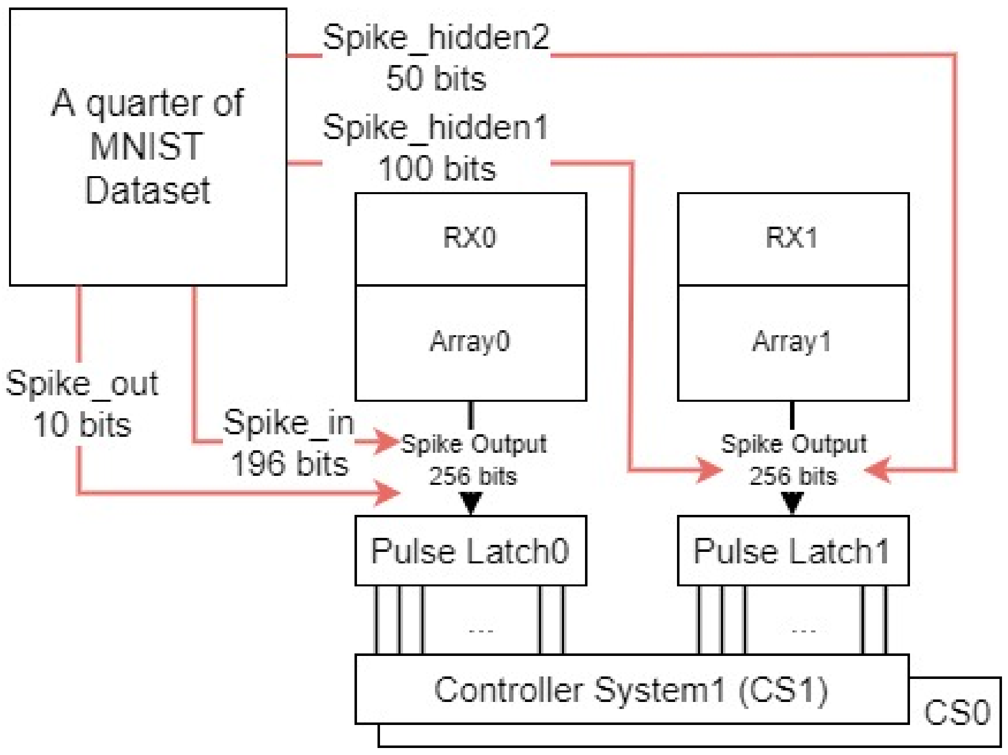


Figure 5.1: A more flexible mapping strategy

Bibliography

[1] Filipp Akopyan and Sawada. 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*, 34(10):1537–1557, 2015.

[2] Luca Benini and Giovanni De Micheli. Networks on chips: A new soc paradigm. *computer*, 35(1):70–78, 2002.

[3] Forrest Brewer, David McCarthy, and Merritt Miller. Automated timing constraint generation for pulse gate circuits. 2021.

[4] Snaider Carrillo, Jim Harkin, Liam McDaid, Sandeep Pande, Seamus Cawley, Brian McGinley, and Fearghal Morgan. Advancing interconnect density for spiking neural network hardware implementations using traffic-aware adaptive network-on-chip routers. *Neural Networks*, 33:42–57, 2012.

[5] Zengguang Cheng, Carlos Ríos, Wolfram H. P. Pernice, C. David Wright, and Harish Bhaskaran. On-chip photonic synapse. *Science Advances*, 3(9):e1700160, 2017.

[6] PS Churchland and TJ Sejnowski. The computational brain mit press. *Cambridge, Massachusetts*, 1992.

[7] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[8] Jose Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection networks*. Morgan Kaufmann, 2003.

[9] M.R. Greenstreet and Jihong Ren. Surfing interconnect. In *12th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC'06)*, pages 9 pp.–106, 2006.

[10] Jim Harkin, Fearghal Morgan, Liam McDaid, Steve Hall, Brian McGinley, and Seamus Cawley. A reconfigurable and biologically inspired paradigm for computation using network-on-chip and spiking neural networks. *International Journal of Reconfigurable Computing*, 2009, 2009.

[11] Scott Hauck, Katherine Compton, Ken Eguro, Mark Holland, Shawn Phillips, and Akshay Sharma. Totem: domain-specific reconfigurable logic. 01 2023.

[12] Rajib Kar, Vikas Maheshwari, Ashis Kumar Mal, and A.K. Bhattacharjee. Delay analysis on-chip vlsi interconnect using gamma distribution function. *International Journal of Computer Applications*, 1, 02 2010.

[13] Santosh Kulkarni, Sishaj P. Simon, and K. Sundareswaran. A spiking neural network (snn) forecast engine for short-term electrical load forecasting. *Applied Soft Computing*, 13(8):3628–3635, 2013.

[14] Shih-Chii Liu, Tobi Delbruck, Giacomo Indiveri, Adrian Whatley, and Rodney Douglas. *Event-based neuromorphic systems*. John Wiley & Sons, 2014.

[15] Misha Mahowald. Vlsi analogs of neuronal visual processing: a synthesis of form and function. 1992.

[16] Carver Mead. Adaptive retina. In *Analog VLSI implementation of neural systems*, pages 239–246. Springer, 1989.

[17] Paul A. Merolla, John V. Arthur, Rodrigo Alvarez-Icaza, Andrew S. Cassidy, Jun Sawada, Filipp Akopyan, Bryan L. Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, Bernard Brezzo, Ivan Vo, Steven K. Esser, Rathinakumar Appuswamy, Brian Taba, Arnon Amir, Myron D. Flickner, William P. Risk, Rajit Manohar, and Dharmendra S. Modha. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673, 2014.

[18] Johannes Schemmel, Johannes Fieres, and Karlheinz Meier. Wafer-scale integration of analog neural networks. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 431–438. IEEE, 2008.

[19] Dimitrios Serpanos and Tilman Wolf. Chapter 4 - interconnects and switching fabrics. In Dimitrios Serpanos and Tilman Wolf, editors, *Architecture of Network Systems*, The Morgan Kaufmann Series in Computer Architecture and Design, pages 35–61. Morgan Kaufmann, Boston, 2011.

[20] Rafael Serrano-Gotarredona, Matthias Oster, Patrick Lichtsteiner, Alejandro Linares-Barranco, Rafael Paz-Vicente, Francisco Gómez-Rodríguez, Luis Camuñas-Mesa, Raphael Berner, Manuel Rivas-Pérez, Tobi Delbruck, et al. Caviar: A 45k neuron, 5m synapse, 12g connects/s aer hardware sensory–processing–learning–actuating system for high-speed visual object recognition and tracking. *IEEE Transactions on Neural networks*, 20(9):1417–1438, 2009.

[21] Jan Stuijt, Manolis Sifalakis, Amirreza Yousefzadeh, and Federico Corradi. μbrain: An event-driven and fully synthesizable architecture for spiking neural networks. *Frontiers in Neuroscience*, 15, 05 2021.

[22] I. Sutherland and S. Fairbanks. Gasp: a minimal fifo control. In *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001*, pages 46–53, 2001.

[23] Ted H. Szymanski, Martin Saint-Laurent, Victor Tyan, Albert Au, and Boonchuay Supmonchai. Field-programmable logic devices with optical input–output. *Appl. Opt.*, 39(5):721–732, Feb 2000.

[24] Taiwan Semiconductor Manufacturing Company Ltd. *TSMC N28HPC Standard Cell Library*, January 2016. Version 110c.

[25] T. Theocharides, G. Link, N. Vijaykrishnan, M.J. Invin, and V. Srikantam. A generic reconfigurable neural network architecture as a network on chip. In *IEEE International SOC Conference, 2004. Proceedings.*, pages 191–194, 2004.

[26] Fang Yang. Designing asynchronous gate library with new system level trade-offs. 08 2021.

[27] Jilin Zhang, Jinsong Wei, and Hong Chen. An address event representation circuits design with rotation priority against pulse collision. In *2019 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–3, 2019.