UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics and Computer Science

Efficient Network On Chip for a RISC-V based neuromorphic processor

Sharon Moolenaar

July 4, 2025

Commitee: dr.ir. S.H. Gerez
dr.ir. B.J. van der Zwaag
A. Yousefzadeh PhD
S. Sohail

Group: CAES Group
Study: (MSc) Embedded Systems

Acknowledgements

I would like to express my deepest appreciation to my supervisor, Amir Yousefzadeh, PhD, for his guidance, support, and insightful feedback throughout this thesis. His knowledge of neuromorphic processors has greatly helped me improve the quality of my work. A special thanks goes to Sameed Sohail, whose support helped me get started and deepen my understanding of the subject. I am also grateful to the committee chair, dr.ir. Sabih Gerez, for his constructive input and feedback. I would like to thank dr.ir. Berend-Jan van der Zwaag for his engagement as an external committee member. Lastly, I'd like to mention Wiebren Wijnstra and Mattias Westerink, who helped me overcome technical challenges regarding the NEORV32 processor.

Abstract

Neuromorphic processors utilize distributed nodes for parallel and energy-efficient computations that are well-suited for neural network workloads. These nodes require a communication mechanism that is both low-overhead and simple, without compromising system performance. This thesis presents the design and evaluation of a power- and area-efficient Network on Chip (NoC) for neuromorphic processors. This Network on Chip (NoC) is designed for optimizing Power, Performance, Area (PPA) trade-offs while managing congestion and data transfer. To achieve these goals, the architecture employs a 2D mesh topology with input First In, First Out (FIFO) buffers, round-robin (RR) scheduling, a ready/valid handshaking protocol, store-and-forward packet switching, and dynamic XY routing. A First In, First Out (FIFO) depth of four and dynamic XY routing were selected to optimize performance under congestion.

Experimental results show that the system achieves a throughput of one packet per clock cycle under non-congested conditions, with a measured propagation delay of four clock cycles. Power analysis reveals that the energy per packet is 296 fJ, and the idle power consumption of one node is $22\mu W$. Optimal performance is achieved with a directional data flow, such as that of a neural network, which is due to the potential for cyclic dependencies introduced by the interaction of routing and flow control.

The NoC interconnects NEORV32 processors optimized for event-driven Artificial Neural Network (ANN) computations. In this experiment, a standardized 4-by-4 NoC is deployed with different mapping strategies of the same ANN. Results show that the NoC contributes only 0.24% to the total area and between 2.9% and 5.2% to overall energy consumption. As the system becomes more parallel, a trade-off emerges between active communication overhead and performance gains.

Contents

List of Figures 6

List of Tables 8

List of Abbreviations 10

1 Introduction 11

2 Background 13

2.1 Neuromorphic computing . . . 13

2.1.1 Spiking Neural Network (SNN) . . . 14

2.2 Network on Chip (NoC) . . . 15

2.2.1 Topology . . . 15

2.2.2 Flow Control . . . 16

2.2.3 Switching . . . 17

2.2.4 Routing . . . 17

2.2.5 Scheduling . . . 19

2.2.6 Mapping . . . 19

2.2.7 Clock distribution . . . 20

2.3 RISC-V . . . 20

2.3.1 NEORV32 . . . 21

3 Related work 22

3.1 SpiNNaker . . . 22

3.1.1 SpiNNaker2 . . . 23

3.2 TrueNorth . . . 24

3.3 Loihi . . . 25

3.4 Seneca . . . 26

3.5 Summary 27

4 Network on Chip Design Choices **28**

4.1 Requirements 28

4.2 Routing 28

4.3 Flow control 29

4.4 Buffer 29

4.5 Arbitration and Scheduling 30

4.6 System-Level Architecture 30

5 NoC Implementation **32**

5.1 Topology 32

5.2 NoC Node 33

5.2.1 Waveform: single packet 34

5.3 NEORV32 34

5.3.1 Waveform: XBUS with FIFO 36

5.4 Packet Structure 37

5.5 Clocking Strategy 37

6 Results **38**

6.1 Experimental Setup 38

6.2 Evaluation of Design Choices 39

6.2.1 Randomized Directional Data 39

6.2.2 FIFO word size 39

6.2.3 Routing algorithm 41

6.3 NoC benchmark 42

6.3.1 Directional data 42

6.3.2 Randomized data 45

6.3.3 Synthesis 46

6.3.4 Power Analysis 47

6.4 Benchmark: MNIST with NEORV32 48

6.4.1 Power, Performance, Area (PPA) breakdown 49

6.4.2 Comparison with other NoCs 52

7 Discussion **53**

7.1 Mapping 53

7.2 Routing algorithm 53

7.3 Head-of-Line (HOL) Blocking 54

7.4 Deadlock 54

8 Conclusion **55**

9 Future Works **57**

References **58**

A Github **62**

B Round-Robin Input Port Scheduler **65**

C XY Routing Algorithm **66**

D Dynamic XY Routing Algorithm **67**

E Inference code **68**

F Randomized Direction Data generation **71**

List of Figures

2.1 A Spiking Neural Network (SNN) neuron [17]. 14
2.2 Common topologies [5] 15
2.3 Standard packet switching methods examples [28] 17
2.4 Turn models of different routing algorithms [23]. 18
2.5 The NEORV32 Processor (Block Diagram) [36] 21

3.1 A SpiNNaker node [29] 22
3.2 SpiNNaker2 Quad-Processing Element (QPE) NoC architecture [26]. 23
3.3 TrueNorth's chip architecture [3] 24
3.4 Four interconnected clusters containing 16 SENECA cores [42]. 26

4.1 Proposed 4x4 NoC architecture. 30

5.1 2x1 connection diagram 32
5.2 A single detailed NoC node 33
5.3 Waveform of a routed packet as simulated in Cadence Xcelium [14]. 34
5.4 XBUS transfers: Write (left)) and Read (right) [36] 35
5.5 Input FIFO communication with the External Bus (XBUS) of NEORV32, simulated in Cadence Xcelium [14]. 36
5.6 Neuron packet 37

6.1 Graphs displaying different metrics versus the FIFO depth. 40
6.2 Heat map of the in-use FIFO occupancy in XY and dynamic XY routing with averages. 41
6.3 Waveform of a congested NoC as simulated in Cadence Xcelium [14]. 43
6.4 Throughput and Number of Stalled Nodes per Insertion Phase 44
6.5 Latency per hop vs Throughput (red dots indicate all nodes stall) 44
6.6 Randomized vs Directional Data Delivered 45

6.7 Deadlock example with randomized data. 46
6.8 Area in μm^2 of one router (total area is 1637.8 μm^2). 46
6.9 Waveform of inserting $100,000$ directional packets into node (3,3) 47
6.10 A fully connected MNIST ANN 48
6.11 ANN mapped onto a 4x4 NoC 49
6.12 Waveform of NEORV32 activity of the mappings in Figure 6.11 50

List of Tables

3.1 Comparison of neuromorphic chip architectures 27

4.1 Design exploration routing algorithm. 29

5.1 XBUS Addresses 35

6.1 Comparison between XY-routing and dynamic XY-routing 42

6.2 Pipeline diagram router. 42

6.3 Power analysis with and without clock gating. 47

6.4 Power and energy analysis of the clock-gated design. 47

6.5 Component Area and Percentage Breakdown of a Middle Node (with Instruction Memory (IMEM) of 32KB, Data Memory (DMEM) of 512KB, and one FIFO of 128 bits) 49

6.6 Energy consumption comparison for different NoC configurations across nodes; red letters indicate the used processes. 51

6.7 Timing and energy per inference of the mapped event-driven ANN (Figure 6.11). Based on a DMEM of 512 KB and an IMEM of 32 KB. Energy values are shown for both Joules report (J) and power-gated (PG) conditions. 51

List of Abbreviations

AI Artificial Intelligence. 11

ANN Artificial Neural Network. 2, 7, 8, 13, 14, 19, 24, 48, 49, 51, 52, 56, 57

ASIC Application-Specific Integrated Circuit. 21

CPU Central Processing Unit. 11, 13, 35, 48, 49

DMEM Data Memory. 8, 21, 34, 35, 49, 51, 56

DVS Dynamic Vision Sensor. 14

EOF End of Frame. 37

FIFO First In, First Out. 2, 6, 8, 23–25, 29–47, 49, 50, 53–57

flit Flow Control Unit. 16, 17, 30, 57

FWFT First-Word Fall-Through. 29, 31, 34

GALS Globally Asynchronous Locally Synchronous. 20, 22, 24–27, 31, 57

GPU Graphical Processing Unit. 11, 13

HOL Head-of-Line. 29, 45, 46, 54, 56

IMEM Instruction Memory. 8, 21, 34, 49, 51, 56

ISA Instruction Set Architecture. 20, 34

LIF Leaky Integrate-and-Fire. 14

LRU Least Recently Used. 19, 30

LSM Liquid State Machine. 54

MAC multiply-accumulate. 34

MC Multicast. 23, 24

NN Nearest-Neighbour. 23, 24

NoC Network on Chip. 2, 4, 6–8, 11–13, 15–17, 19–35, 37–40, 42–57, 62

NoP Neighbors-on-Path. 18, 19, 28, 29

P2P Point-to-Point. 23

PE Processing Element. 11, 12, 15, 16, 19–21, 24, 26, 28, 32, 34–36, 52, 53, 55, 56, 70

PPA Power, Performance, Area. 2, 11, 12, 28, 31, 38, 39, 53–55

QPE Quad-Processing Element. 6, 23, 24

RAM Random Access Memory. 35

RISC Reduced Instruction Set Computing. 20

RR Round Robin. 2, 19, 30, 31, 33, 43, 46, 55

SNN Spiking Neural Network. 6, 12–14, 19, 20, 22, 24, 48, 53, 57

SoC System on Chip. 11

SRAM Static Random Access Memory. 35, 49

STDP Spike Timing-Dependent Plasticity. 14

VCs Virtual Vhannels. 29, 54

XBUS External Bus. 6, 8, 21, 35–37, 56

1 Introduction

Neuromorphic computing is an emerging field that combines concepts from neuroscience and computer engineering to create systems that mimic the human brain's functionality. Our brain can process information in a rapid, parallel, and energy-efficient manner [1]. This approach addresses the growing demand for faster and more efficient methods to process the large amounts of data generated today, particularly in fields such as Artificial Intelligence (AI). Current Central Processing Units (CPUs) and Graphical Processing Units (GPUs) use the Von Neumann architecture, which separates memory and processing units. This approach can lead to bottlenecks in data transfer. In contrast, neuromorphic systems integrate memory and processing within distributed nodes, enabling more decentralized processing.

The following text introduces concepts that may be unfamiliar to the reader. These are mentioned to provide context for the research questions and are explained in more detail in Chapter 2. Some key features of neuromorphic processors are: neuron cores, memory architecture, communication, scalability, and learning algorithms. This research is part of an open-source many-core digital neuromorphic processor called SparkRV, focusing on communication and scalability. The communication has to account for scalability. A technique suitable for this implementation is a Network on Chip (NoC). When comparing an NoC, shared bus, segmented bus, and point-to-point communication in the context of a System on Chip (SoC), the NoC is more scalable due to the structured and efficient communication framework, which can handle increased system complexity and bandwidth requirements as the number of Processing Elements (PEs) increases [10].

A considerable amount of research has already been done on neuromorphic processors and their NoCs. Some examples include IBM's TrueNorth [3], Intel's Loihi [15], SpiNNaker [19], Seneca [42], and others. These NoCs have numerous features to account for data transfer, congestion, packetization, compression, and other related functions. In this master's thesis, the focus will be on designing an open-source, simplistic NoC that accounts for backpressure and packetization without becoming a bottleneck. As it is open-source, the code for this project can be found on GitHub; more information is provided in Appendix A. The NoC is designed by answering the following research question:

"How can the Network on Chip (NoC) in a neuromorphic processor be optimized for efficient Power, Performance, Area (PPA) trade-offs while handling congestion and optimizing data transfer mechanisms?"

To answer the main question, four sub-questions are defined:

1. Which routing methodology is best suited to handle congestion in an efficient Network on Chip (NoC) for a neuromorphic processor?
2. How should packetization in a low-overhead Network on Chip for a neuromorphic processor be optimized?

3. What types of Processing Elements (PEs) will be interconnected using the Network on Chip (NoC) architecture?
4. How much of the total PPA of neuromorphic processors is contributed by the NoC?

The thesis is structured as follows.

- Chapter 2 - Background: The background information needed before designing a Network on Chip is described in this chapter. It will first describe a neuromorphic processor and Spiking Neural Network (SNN). Then, an overview of design choices for an NoC is defined. Finally, the RISC-V implementation is reported.
- Chapter 3 - Related Work: Analyzes multiple neuromorphic processors with a focus on their NoCs.
- Chapter 4 - Network on Chip Design Choices: The design choices for the implemented NoC are described and clarified.
- Chapter 5 - Network on Chip Implementation: This chapter gives a detailed look at the implementation of the NoC.
- Chapter 6 - Results: Presents an experimental setup to test the NoC with and without a PE. This section also performs an application and illustrates the Power, Performance, Area (PPA) trade-offs.
- Chapter 7 - Discussion: The results and limitations are discussed.
- Chapter 8 - Conclusion: Summarizes the design, implementation, and results, as well as answering the research questions.
- Chapter 9 - Future Works: Outlines potential future research.

2 Background

This chapter explains the background knowledge required before starting NoC development. First, the concept of neuromorphic computing is described, followed by an introduction to the fundamentals of a NoC architecture. Finally, RISC-V is explained, along with its utility for neuromorphic processors.

2.1 Neuromorphic computing

Neuromorphic computing is an approach to computing that draws inspiration from the human brain. Unlike traditional computers that follow a sequential processing architecture, neuromorphic systems process information in a rapid, parallel, and energy-efficient manner [1]. To fully utilize the brain-inspired architecture, applications must be designed with the same principles. Many neuromorphic processors utilize a sparse event-driven Artificial Neural Network (ANN) inspired by the human brain, known as a Spiking Neural Network (SNN).

Traditional CPUs and GPUs, based on the Von Neumann architecture, are not optimized for the sparse, event-driven operations of the SNN. To efficiently train and use an SNN, dedicated hardware is required. Examples of neuromorphic hardware are: SpiNNaker [19], TrueNorth [3], and Liohi [15]. More details about these processors are in Chapter 3. These chips process information with minimal power consumption, while handling a large-scale SNN.

For these neuromorphic processors to effectively handle the enormous computational demands of a large-scale SNN, scalability is critical. This involves designing systems that allow multiple chips to be seamlessly connected, enabling the management of increasing workloads. Communication between chips (interchip) and within a chip (intrachip) is a vital aspect of this scalability [1].

2.1.1 Spiking Neural Network (SNN)

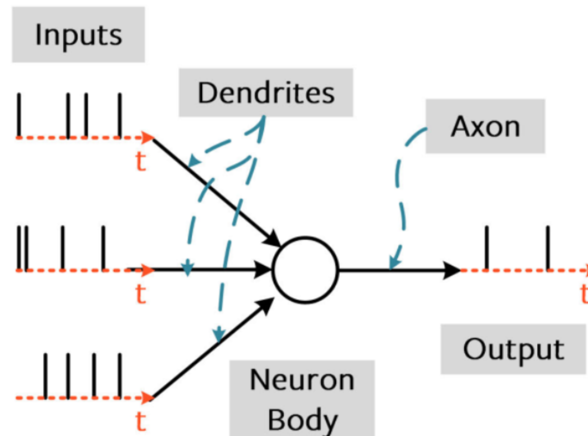


Figure 2.1: A SNN neuron [17].

The SNN works with sparse continuous data, unlike the ANN, which uses discrete data. Events or spikes are generated when a neuron reaches a certain threshold, also known as its membrane potential, mimicking the behavior of biological neurons [17]. Before making an SNN, a few design choices have to be made. These choices can be separated into encoding/decoding algorithms, architecture, and training methods. The encoding and decoding of events can be accomplished in three ways: rate coding, latency coding, and delta modulation. An architectural choice is the way the neurons are modeled. There are multiple ways an SNN can be trained: Spike timing-dependent plasticity (STDP), ANN-to-SNN-conversion, or surrogate gradient descent as explained below.

Rate coding [22] converts input intensity into a corresponding firing rate. This encoding is robust to noise, but it is inefficient in terms of energy consumption and latency. Time-to-first-spike coding [21] converts the input intensity to spike timing; earlier spikes are stronger. This method is more efficient, but less robust to noise.

Neurons can be modeled in multiple ways, one of the most commonly used is the Leaky Integrate-and-Fire (LIF) neuron models [11]. This is a simplistic, biologically inspired neuron that sums the weighted inputs like an artificial neuron, but gradually loses charge (leakage). The Hodgkin–Huxley [25] model is biologically plausible, but not computationally efficient. The Izhikevich model [27] combines the biological plausibility of the Hodgkin–Huxley model, but with the computational efficiency of the LIF model.

The training of a SNN can be separated into unsupervised and supervised learning. An unsupervised learning technique is Spike Timing-Dependent Plasticity (STDP), where the synaptic strength depends on the order and firing time of a pair of connected neurons [9]. The ANN-to-SNN-conversion (shadow learning) is a good option with non-time-varying data [39]. It is a simplistic approach, but it may introduce conversion errors and inefficient spike dynamics. Surrogate gradient descent [35] allows an SNN to be trained as an ANN, while preserving the spike-based computations. It propagates errors to earlier layers, regardless of whether spiking occurs, but the weight is only updated when spiking does occur.

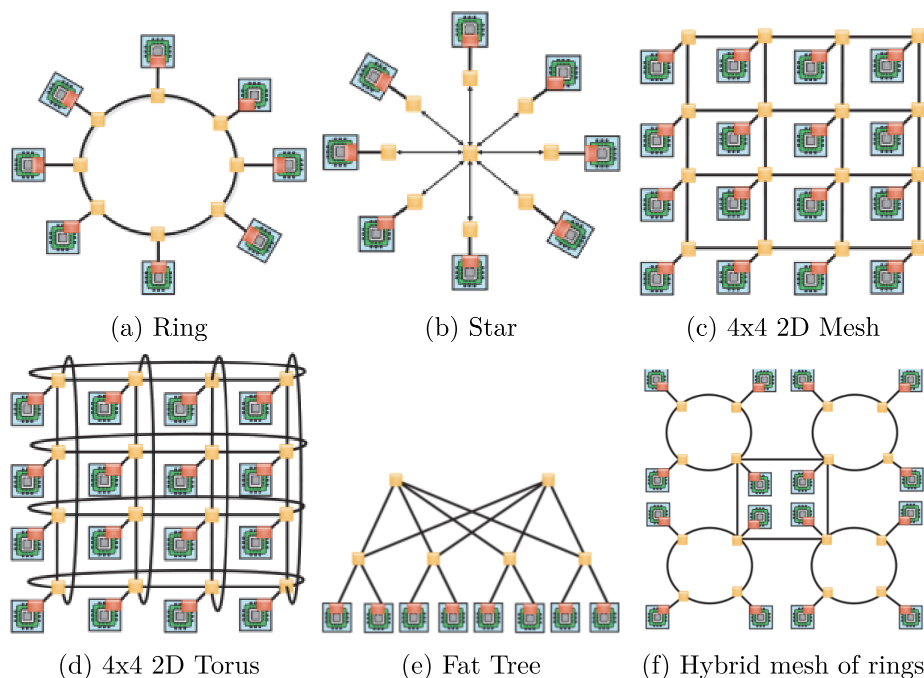
SNN exploits time-varying data such as data from a Dynamic Vision Sensor (DVS). This data is captured with a DVS camera that reports changes in brightness and is otherwise silent. Some of the most used datasets are: IBM's hand gesture recognition [6], MNIST-DVS, and Poker-DVS [41].

2.2 Network on Chip (NoC)

A Network on Chip (NoC) is a mechanism to exchange data between Processing Elements (PEs) on a chip, creating a clear path for data to flow from element to element. To explain and compare various design choices, this section discusses the concepts and background information on NoCs. Each subsection describes a different design choice for a NoC, including topology, flow control, switching, routing, scheduling, mapping, and clock distribution.

2.2.1 Topology

The first step of designing an NoC is to choose a topology; this topology affects other aspects of the network, such as routing and flow control. The most common topologies are Ring, Star, 2D mesh, 2D torus, Fat Tree, and a hybrid of these topologies [30], they are displayed in Figure 2.2. The topologies can be compared using different metrics; these are split into traffic-independent and traffic-dependent metrics [28]. Traffic-independent metrics are the number of links to each node (degree), the maximum distance between any two nodes (diameter), and the scalability. Traffic-dependent metrics are measured from source to destination, and include the number of links traversed (hop count) and path diversity.



(a) Ring (b) Star (c) 4x4 2D Mesh

(d) 4x4 2D Torus (e) Fat Tree (f) Hybrid mesh of rings

Figure 2.2: Common topologies [5]

The **ring topology** shown in Figure 2.2a, in this topology, each node has two neighbors in a circle. So, the degree is two, and the diameter is half of the number of nodes. It is not easily scalable and lacks significant path diversity. The diameter increases with the number of nodes; therefore, this topology is not suitable for on-chip use.

Figure 2.2b shows the **star topology**, here all nodes are connected to one central node. The problem with this topology is that all traffic must pass through this node, making it prone to congestion. The degree of the center node is the number of nodes minus one, the diameter is two, and the scalability and path diversity are poor. The maximum hop count is two, which is the main advantage of this design.

One of the most popular on-chip designs is the **2D mesh topology** shown in Figure 2.2c. This topology is great for scalability and path diversity. The degree is varied; the corners have a degree of two, the edges a degree of three, and the inner nodes a degree of four. This variation in degree can increase the diameter when the mesh becomes bigger. The average hop count is $\frac{2k}{3}$ for even k-by-k meshes, and odd meshes, the average hop count is $2(\frac{k}{3} - \frac{1}{3k})$. 3D meshes are an active research area and are not considered due to the increased complexity of implementation.

A **2D Torus topology** is similar to the 2D mesh topology; the difference is how the edges are handled. With a torus, the edges on the opposite side are connected as seen in Figure 2.2d. This wrap-around makes the degree four for all nodes, which reduces the diameter in comparison to a 2D mesh. The average hop count is also improved; for even k-by-k meshes, $\frac{2k}{4}$, and for odd meshes, the average hop count is $2(\frac{k}{4} - \frac{1}{4k})$. There is a high path diversity, but the on-chip scalability is more complex than with a 2D mesh. Another disadvantage of the torus is the increased latency due to the long wraparound wires.

The **Fat Tree topology** is shown in Figure 2.2e. In fat trees, the degree and diameter are not fixed, making it harder to design and scale. There is path diversity, but more routers than PE have to be initiated. Making it worse for on-chip design.

Finally, a **Hybrid topology** can be created, as shown in Figure 2.2f, where the hybrid 2D mesh of the ring topology is portrayed. With a hybrid topology, any of the previous topologies can be combined to create connections tailored to the specific application. For the hybrid mesh of rings, the degree is either 2 or 4, depending on the node's location within the mesh. The diameter is the ring size plus the size of the mesh. There is more path diversity than the ring topology, but less than the mesh topology, because it has fewer interconnecting wires. Lastly, the flexibility of the hybrid system makes it challenging to design a scalable on-chip network.

To make the following sections more specific, the 2D mesh topology is chosen because it:

- is easily scalable for on-chip networks,
- supports both inter- and intra-chip communication,
- offers good path diversity, which helps reduce congestion,
- maintains an acceptable average hop count, and
- has one router per PE.

2.2.2 Flow Control

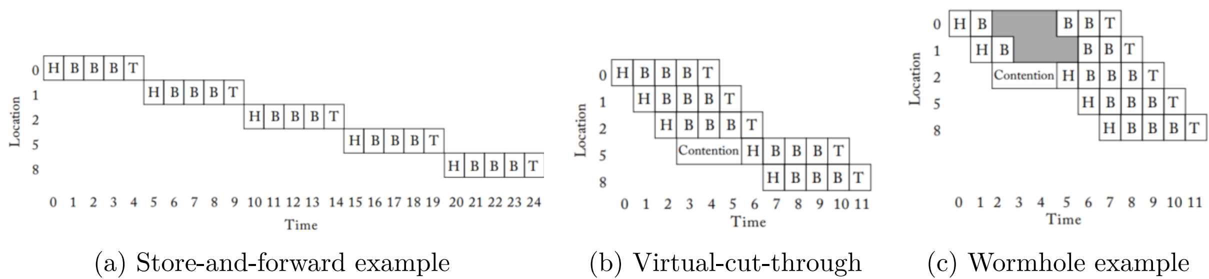
An NoC transmits data in a sequence of packets or, optionally, Flow Control Units (flits). A packet can consist of multiple flits, and a message can be made up of one or more packets. When used, a packet includes a head flit with destination address, a body flit, and a tail flit that indicates the end of a packet. This segmentation allows larger packets to traverse the network with a fine-grained resource allocation. Larger packets mean that more data can be routed with relatively low header overhead.

Flow control manages the transmission of data packets and stalls the packet propagation when the buffers are full. ACK/NACK, credit-based, or on–off flow control are the most used protocols [2]. ACK/NACK is a handshaking protocol between two routers or a router and a PE. When the sender wants to put data on the link, a valid signal is sent. When the receiver is ready, the ready signal is also sent; otherwise, the data is held by the sender. In credit-based data flow, the credit refers to the number of buffers available at the next hop. This credit is increased when a

packet leaves the buffer and decreased when a packet is added to the buffer. On/off signaling is a signal between routers that indicates whether they can receive new data. This signal is turned off when the number of available buffers drops below a threshold.

2.2.3 Switching

The switching technique in an NoC refers to connecting the input port of a router to an output port of a nearby router. The most common methods are circuit switching and packet switching [4]. With circuit switching, a path is established between the source and destination before the data is transferred, and this path remains reserved until the end of the communication. It offers low latency when data is transferred in batches, but it wastes paths when no data is transferred, resulting in scalability issues. With packet switching, packets are transferred one at a time with individual pathing. This will result in delays, but it is scalable and supports multiple traffic flows simultaneously.



(a) Store-and-forward example (b) Virtual-cut-through (c) Wormhole example

Figure 2.3: Standard packet switching methods examples [28]

Packet switching has three standard methods [1]: store-and-forward, virtual-cut-through, and wormhole. With store-and-forward, the router buffers the entire packet before sending it to the next router. An example with flits is shown in Figure 2.3a, an entire packet, all flits, must be routed before the next packet can be processed.

Virtual-cut-through [31] will not wait for the entire packet but will send the flits as soon as the header is received and the next router has space. This creates a virtual path in the NoC to send all flits through. The packet can be blocked due to a busy output channel and will be stored in the intermediate node. This switching technique is a hybrid between circuit switching and store-and-forward switching. Figure 2.3b shows an example of this switching technique, where the packet is delayed at node 2 before reaching node 5. This means that the packet is stored entirely in node 2 before continuing. Wormhole switching [28] also works with a virtual path in the NoC. The difference is that when the header is blocked, the flits are stored at their current location, which means that the buffer size can be smaller than the packet size. This technique is illustrated in Figure 2.3c. Between nodes 1 and 2, congestion is detected, and the flits are stored at their current location before continuing.

2.2.4 Routing

The 2D mesh topology is chosen in Subsection 2.2.1; this means only the routing algorithm of this topology is considered. The routing algorithm influences the network latency, throughput, and hotspot avoidance [28]. There are three types of routing algorithms: deterministic, oblivious, and adaptive pathing algorithms [8]. With deterministic routing, a certain path will be followed, and there is no path diversity. Any message from A to B will follow the same route. Oblivious routing can change the path, but without regard for congestion along the nodes. Network

congestion is considered in an adaptive routing algorithm by changing direction in order to avoid congestion.

A routing algorithm must be deadlock-free; a deadlock occurs when packets in a network are indefinitely blocked due to a cyclic dependency on other nodes, preventing any of them from progressing. The algorithm also has to be live-lock-free: a live-lock happens when packets continuously move through the network without making progress toward their destination, often due to misdirected routing or overemphasis on avoiding congestion. To improve latency, it is also best to follow the minimal path, which is the shortest possible route between a source and destination in terms of hops, ensuring efficient data transmission under ideal conditions.

A widely used on-chip routing algorithm is **XY-routing** [46], this deterministic algorithm first looks at the X direction toward the destination, and then via the Y direction. The advantage of this algorithm is the simplicity and low overhead. Due to the predetermined path, the algorithm is deadlock-free and live-lock-free, and it also takes the minimal path in a 2D mesh. This algorithm disables path diversity and is unable to route around network congestion, so it does not balance the load in the network. This means it is prone to congestion and hotspot nodes.

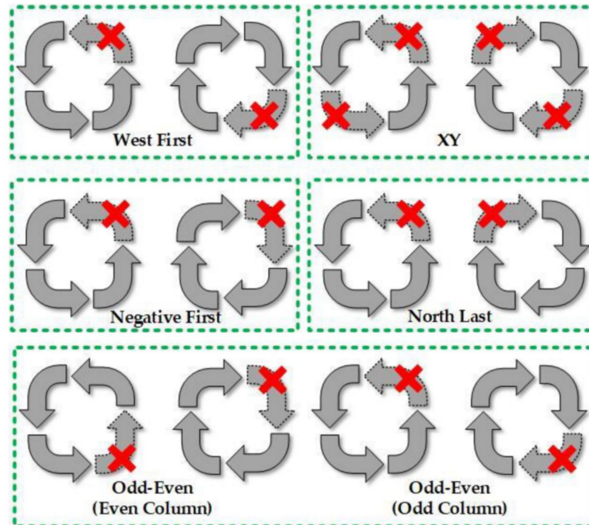


Figure 2.4: Turn models of different routing algorithms [23].

Odd-Even routing [23] is an adaptive routing algorithm based on a turn model. Figure 2.4 shows different turn models with turns of 90°. To avoid deadlocks and live-locks, a turn-based model must disallow at least one turning direction. With the odd-even routing, the north-west and south-west turns are prohibited in even columns (X is even), and the east-north and north-west turns are forbidden in the odd columns (X is odd). This routing algorithm cannot guarantee the minimal path because specific directions are blocked. It has a lot of path diversity and can avoid congested areas when it is not on the same X or Y dimension as the destination.

Dynamic XY Routing [32] is an adaptive form of XY routing. This algorithm will first examine the X-direction; if congestion is detected in that direction, it will then investigate the Y-direction. Because the choice is to move in the direction of the destination in either the X or Y dimension, it will always follow the minimal path, making it deadlock-free and live-lock-free. It has some path diversity, but not in all directions, and it is aware of network congestion among its neighbors.

Neighbors-on-Path (NoP) Wormhole routing [7] is a fully adaptive algorithm with wormhole-based switching. This algorithm considers not only the buffer availability of the

neighbors, but also the buffer availability along the potential path. This algorithm aims to route the message along the least congested path. When multiple paths are available, it makes a choice based on the odd-even algorithm. The steps are as follows: (1) compute the NoP nodes for each candidate output channel, (2) look at the buffer availability of the NoP nodes, (3) assign a score based on availability, and finally (4) select the channel with the highest score. This algorithm is deadlock-free and live-lock-free because of the underlying odd-even algorithm. It has a lot of path diversity and global congestion awareness. The disadvantages are the hardware complexity, a 5-8% increase in area, and it works worse with uniform data than deterministic XY-routing.

2.2.5 Scheduling

In the router, multiple inputs can be received simultaneously. To determine which packet will be processed first, a scheduling policy must be used. The most common strategies are Round Robin (RR), Least Recently Used (LRU), and fixed priority [1]. Round Robin uses a fixed ring of input ports, where the priority pointer rotates around the ring to select the next port based on the last served. The LRU method changes the port priority based on the last time the port was used, so ports that are used more frequently receive the lowest priority. The fixed priority is a list that the router will process every time there is more than one packet ready. This method always serves the highest-priority port first, creating an unbalanced priority.

2.2.6 Mapping

Mapping is important in an NoC, because it affects routing latency and throughput. The mapping problem is NP-hard [40], and application-specific. In this report, the focus will be on neuromorphic processors, so the application focused on is the ANN/SNN. Neuromorphic systems are usually mapped in two phases: (1) partitioning and (2) placing [1]. Partitioning is the process of clustering neurons to put in the PE. Neurons that are heavily connected are put in the same cluster. The next step is to place the clusters on a PE; hereby, it would be best if the clusters with the most communication are placed close to each other.

Mapping can be divided into static or dynamic mapping. Static mapping means that the program is mapped onto the NoC before it is executed. Dynamic mapping is an online mapping strategy, allowing the placement of tasks to change during execution time. Due to the simplistic nature of this NoC, the static mapping strategy will be investigated, which can be divided into exact mapping and search-based mapping. Exact mapping is based on mathematical programming, while search-based methods are either heuristic-based or deterministic. Search-based methods involve exploring multiple possible configurations to find the optimal solution. Heuristic methods use approximations to find a solution, while deterministic methods use fixed rules to find a solution.

SNEAP (Spiking NEural network mAPping toolchain) [33] is designed to map large-scale SNNs to neuromorphic platforms with an NoC. First, the behavior and network of the SNN is extracted into an undirected graph. Then, the graph is partitioned into multiple parts depending on the neuromorphic processor capabilities. A multi-level partitioning algorithm is used to reduce inter-partition spike communication. The partitions are then mapped to the NoC to minimize the average hop count of the spikes, thereby reducing communication latency and energy consumption. This heuristic-based static mapping algorithm optimizes energy consumption and latency associated with spike communication.

NeuMap [45] is designed to map an SNN to multicore neuromorphic processors. It first calculates the spike firing rates of all neurons to obtain communication patterns. It then uses a multilevel graph partitioning algorithm that partitions each subnetwork, which consists of multiple adjacent layers, into multiple clusters while satisfying hardware resource constraints. Finally, it employs a metaheuristic algorithm, a higher-level optimization technique, to find the best cluster-to-core mapping scheme within the smaller search space. NeuMap focuses on optimizing latency and energy consumption, but it does not always find the optimal mapping.

2.2.7 Clock distribution

For a large-scale NoC, clock distribution is a problem. In a completely synchronous system, the clock signal must be distributed throughout a clock tree [24]. As the network scales, the clock tree will need to cover a significantly larger area. This can introduce clock skew and increase power and area overhead. Clock skewness refers to the variability in the arrival time of the clock signal, which can introduce timing delays. With increasingly large systems, clock tree distribution becomes impossible due to overhead and design difficulty. This is why completely synchronous NoCs are not scalable. To address this problem, the large clock tree can be discarded in favor of asynchronous communication with local clock trees, which is called Globally Asynchronous Locally Synchronous (GALS) [37].

In GALS, individual processes are clocked synchronously, with asynchronous connectivity. These clocks can differ in frequency and do not have phase-alignment requirements. In an NoC, the individual process can be defined as a core or a chip. However, asynchronous communication between different clock domains introduces the risk of metastability, where data transfers can cause unpredictable delays or errors if not properly synchronized.

2.3 RISC-V

RISC-V [44] is an open-source Instruction Set Architecture (ISA) based on Reduced Instruction Set Computing (RISC), designed with a focus on simplicity, modularity, and extensibility. RISC-V is open and free, unlike proprietary ISAs, which are gaining popularity among researchers and companies for developing custom processors without licensing restrictions. Modularity refers to the design of a base integer instruction set (I) of 32-, 64-, or 128-bits with optional extensions, such as integer multiplication/division (M), atomic operations (A), and floating-point support (F, D).

RISC-V's open nature makes it easy to extend the instruction set with custom instructions. Neuromorphic processors can utilize this to optimize spike-based computations. Additionally, RISC-V's efficient design supports power-efficient implementations using lightweight cores, and many such cores have already been developed.

This project is a subpart of an open-source many-core digital neuromorphic processor (SparkRV). SparkRV features multiple PEs, comprising a general-purpose controller-RISC-V and domain-specific AI accelerators, connected to the NoC, ensuring reliable connectivity. The RISC-V core chosen for this project is NEORV32, an open-source RISC-V-compatible processor system. Consequently, the NoC should be optimized with the NEORV32 in mind.

2.3.1 NEORV32

NEORV32 [36] is a compact open-source RISC-V soft-core processor designed for embedded applications. While a soft-core processor cannot match Application-Specific Integrated Circuits (ASICs) in performance or efficiency, it does offer flexibility in the design phase. In the NoC design, NEORV32's lightweight, FPGA-optimized design makes it suitable as a PE, for management and other complex routines, such as neuron and address calculations.

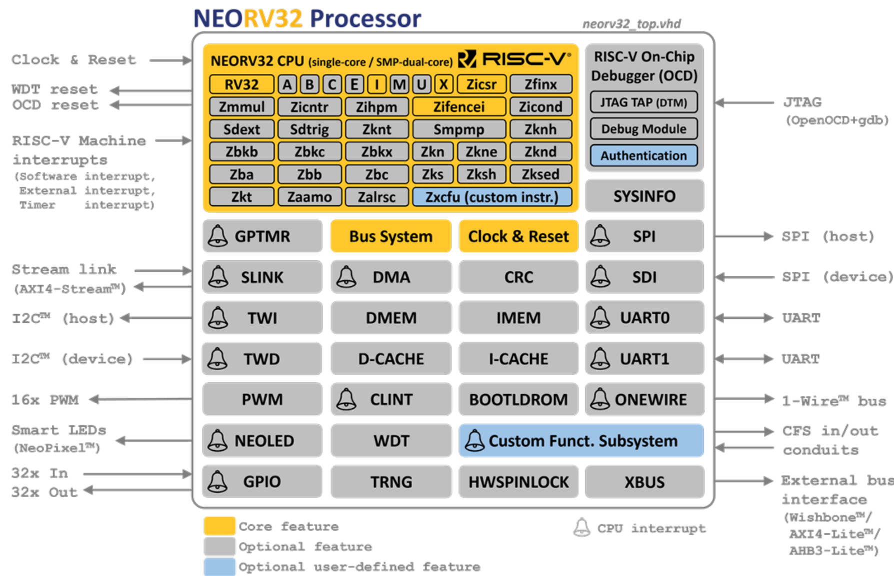


Figure 2.5: The NEORV32 Processor (Block Diagram) [36]

The NEORV32 uses a 32-bit base integer instruction set (I), called RV32I. This base can be extended with configurable instruction sets and extensions. An overview of the core and its optional components is shown in Figure 2.5. The supported instructions are depicted in the NEORV32 CPU section of the figure, while the gray blocks below represent optional extensions. In this project, several extensions are particularly relevant: Data Memory (DMEM), Instruction Memory (IMEM), and External Bus (XBUS). The DMEM can be used to store neural network weights, and the IMEM can store the application code. The XBUS interface enables communication between the NEORV32 and external modules, making it suitable for connecting to the NoC.

3 Related work

For a detailed analysis of neuromorphic processors, some relevant NoCs are described. This section will outline the various features, including topology, flow control, switching, scheduling, routing, mapping, clock distribution, and additional functionalities, that they utilize.

3.1 SpiNNaker

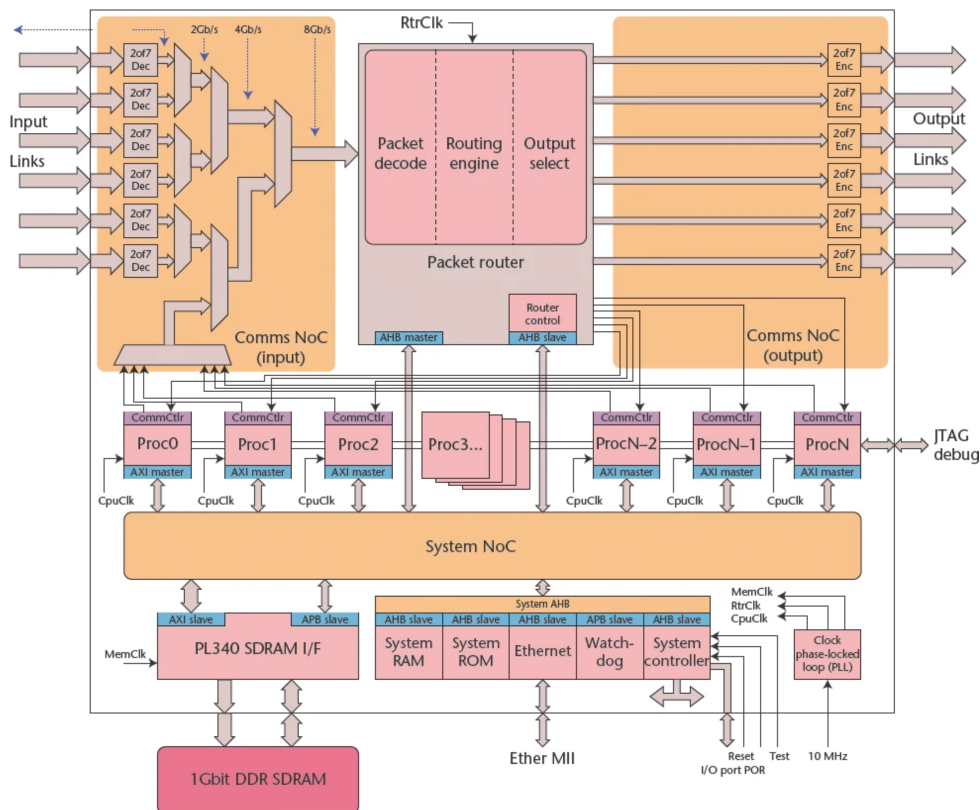


Figure 3.1: A SpiNNaker node [29]

The Spiking Neural Network Architecture (SpiNNaker) [19], developed by the University of Manchester, is a massively parallel multicore system optimized for a real-time SNN. It is designed to be a flexible and scalable neuromorphic processor that meets the high computation and communication tasks of a neural network. To accommodate this highly scalable processor, core-to-core GALS is used. Figure 3.1 shows a block diagram of one SpiNNaker node [18]. The input and output have six signals, which is due to the 2D triangular mesh. This topology is a variation of the 2D mesh that uses 6 neighboring nodes instead of 4 to improve communication bandwidth and reduce latency.

Figure 3.1 shows that two NoCs are available in the SpiNNaker node; a Communications NoC and a System NoC. The Communications NoC support interprocessor communication, both on and off-chip. This is clarified in the diagram, which shows the N-on-chip processors and the external input and output links. The System NoC handles on-chip processor-to-memory and processor-to-peripheral communication, acting as a local shared bus. Incoming packets are buffered between communications with an input FIFO buffer.

SpiNNaker supports four different types of packets: Multicast (MC), Point-to-Point (P2P), Nearest-Neighbour (NN), and fixed-route packets. MC packets are mainly used for neural spike communication. P2P packets enable chip-to-chip communication for machine management, such as system configuration. Nearest-Neighbour packets support the booting of the system and the debugging operations. Fixed-route packets follow a predetermined path and are used for less time-sensitive communication, such as transporting status data to a host. The message passing system is the greatest performance bottleneck, so SpiNNaker was optimized for the short, MC packets. Although this optimization improves performance for spike transfer, it causes overhead when handling other applications, such as loading and controlling neural networks, which is done by sharing the runtime network.

For mapping, SpiNNaker uses a Partitioning and Configuration Manager (PACMAN) [20], which transforms the neural network into binaries for SpiNNaker chip configurations. It uses four steps to achieve this; first, it splits the network into parts that fit on one core. Then it groups the parts able to run with the same application; these groups must fit on one core. These groups are mapped onto the processors, and a routing is calculated. Finally, a binary file is generated of the partitioned and mapped network. Rules can be added to the algorithm based on the neural network; otherwise, it will assign groups on a first-come, first-served basis. This sequential mapping method does not guarantee optimal mapping, but it does reduce the mapping time for large-scale applications.

3.1.1 SpiNNaker2

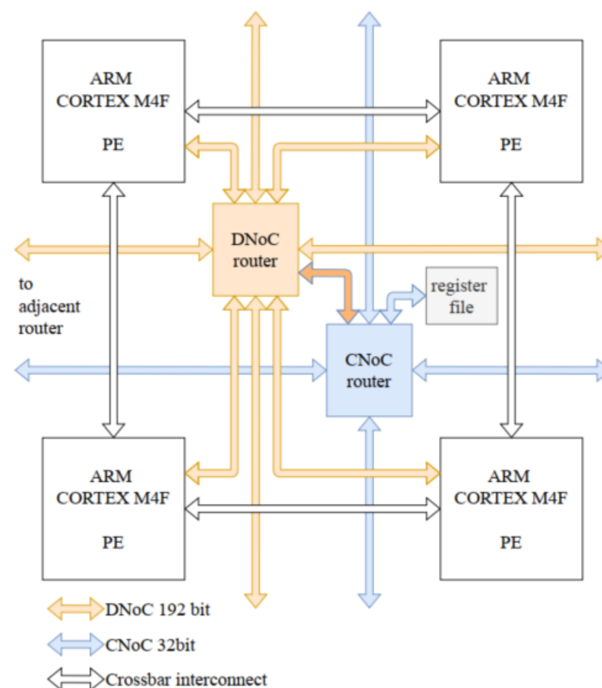


Figure 3.2: SpiNNaker2 QPE NoC architecture [26].

After the first SpiNNaker chip, SpiNNaker2 [34] was created by a collaboration of the Dresden University of Technology and the University of Manchester. This chip is capable of processing more neurons with additional cores while maintaining the same power budget. SpiNNaker2 aims to accelerate both SNNs and hybrids between ANN and SNN. To improve scalability and routing, a 2D mesh topology is used instead of the 2D triangular mesh, allowing a node to have four neighbors instead of six [26]. The system is still core-to-core GALS.

The SpiNNaker2 chip still has two NoCs, now referred to as the data NoC (DNoC) and configuration NoC (CNoC). The DNoC is responsible for data transfer and spike packets. The CNoC has access to all registers during boot-up and configuration, and it also allows data transmission during operation. They use the same packet format for interoperability. The routing in SpiNNaker2 is also different; in addition to using routing tables, it employs deterministic XY-routing. In addition to the four neighboring cores, the NoC can deliver packets to four PEs in a QPE architecture, as shown in Figure 3.2.

The router is responsible for routing three packet types: Multicast (MC), core-to-core, and Nearest-Neighbour packets. To improve throughput, several enhancements are introduced: an improved packet-dropping mechanism, out-of-order issue buffers, larger routing tables for MC packets, and an extended optional packet payload. MC packets are still used for neural events, the core-to-core packets are utilized for machine management, and the Nearest-Neighbour packets are intended for machine boot and debugging. Between routers, they use asynchronous FIFOs to transmit packets synchronously into the router clock domain.

The mapping differs from the first SpiNNaker implementation. The SNN is interpreted into an application graph to show the spike flow among neuron populations. Each population is split into a subpopulation to fit on a PE; these are interpreted in a machine graph. From the machine graph, a routing table is generated, and finally, C-files are generated to be loaded on the SpiNNaker2 chip.

3.2 TrueNorth

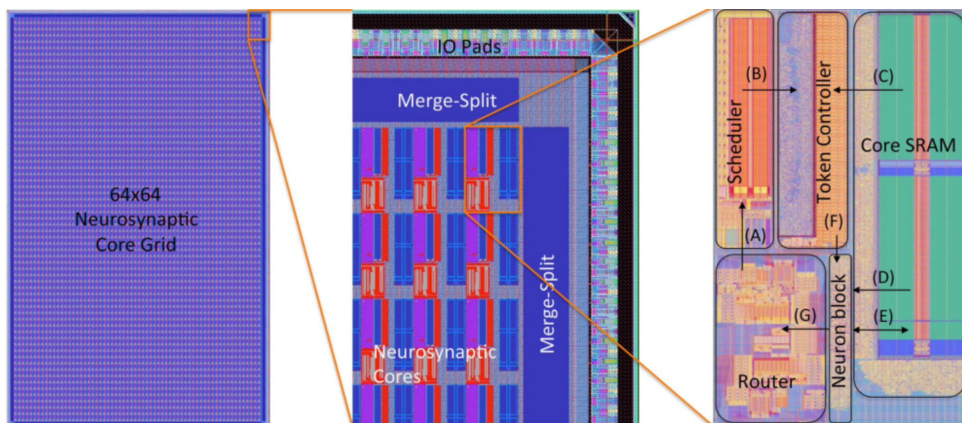


Figure 3.3: TrueNorth's chip architecture [3]

Figure 3.3 shows IBM's neuromorphic processor TrueNorth [3], designed to be parallel, event-driven, low-power, and scalable, using neurosynaptic cores. These cores are connected through a 2D mesh NoC, which enables scalability at both the core and chip levels. A mixed asynchronous–synchronous approach is used; the communication and control circuits are asynchronous, while computations are synchronous (core-to-core GALS).

The router is connected to its core and the east, west, north, and south neighbors via a FIFO. Each spike packet contains several debug flags and destination information: a relative X and Y destination address, a destination index, and a destination tick at which the spike is to be integrated. The packets are routed using a deterministic XY-routing algorithm, and the router operates on a first-come, first-served basis. When a packet must wait on a resource, it is stored in a FIFO buffer. If the buffer becomes full, it can no longer accept new packets, causing the previous node to stall. This mechanism, known as backpressure, is used to manage congestion and prevent packet or spike loss. To mitigate traffic congestion, the router has buffering on the input and output channels. When the packet reaches its destination, the scheduler queues it until the specified destination tick, after which it is sent to the token controller. The token controller controls the input and output flow to the neurosynaptic cores.

With mapping, the problem to solve is to minimize spike communication power; this can be formulated as a wire-length minimization problem. The aim is to map logically connected cores to physically adjacent cores on the same chip. For efficient placement, four algorithms are used. The Multilevel Partitioning-Driven Algorithm recursively partitions the placement area to minimize crossing nets. The Analytical Constraint Generation Algorithm enhances partitioning with cell-distribution constraints. The Hierarchical Quadratic Placement Algorithm combines top-down quadratic partitioning with bottom-up clustering. Finally, the Quadric-Based Force-Directed Analytical Algorithm balances wire length minimization with density constraints.

3.3 Loihi

Loihi [15] is Intel's neuromorphic processor. The topology is a 2D mesh, and the system works with an asynchronous barrier synchronization mechanism. This barrier between cores ensures that all messages are sent and received before proceeding to the next step. Packets are transferred using an asynchronous handshake protocol, where the sender signals a request when a packet is sent, and the receiver acknowledges upon receiving the packet. This GALS implementation is in place rather than using a global clock.

The NoC supports communication between cores through core management messages, spike messages, and barrier messages for time synchronization. These messages are routed using a dimension-order routing algorithm, a deterministic algorithm that follows a specific order; examples include XY-routing (first X then Y) or YX-routing (first Y then X). The NoC only supports packet-switching unicast messages, but the core can enable multicasting by iterating over a list of destinations and sending one spike per destination. Loihi uses two independent router networks to improve bandwidth and prevent deadlock. The cores will send their spike messages alternating between the two networks.

A neural network is assigned to Loihi by using a directed multigraph structure consisting of neurons and synapses. These neurons are first grouped into populations, then the system defines the source and destination populations for communication before mapping them onto the chip's neural cores. This approach is not always optimal, but the hierarchical structure does simplify mapping, improve efficiency, and ensure scalability.

3.4 Seneca

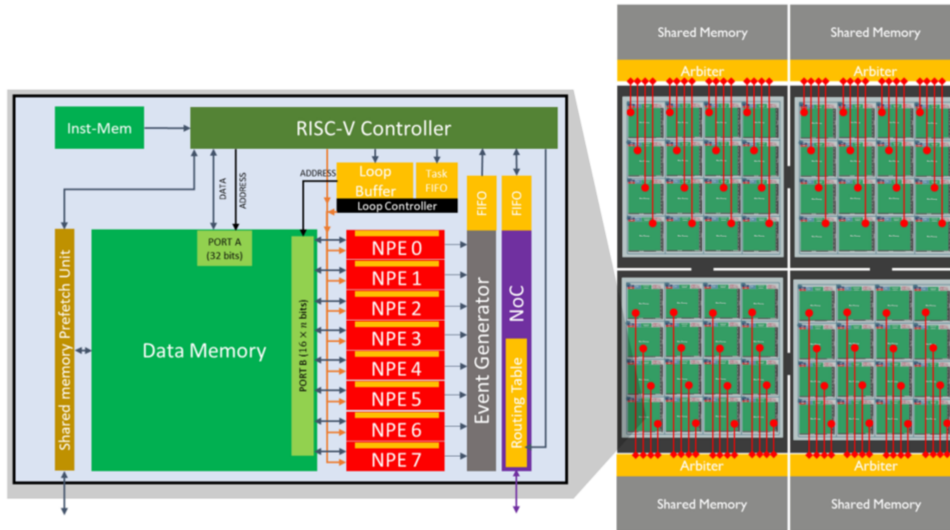


Figure 3.4: Four interconnected clusters containing 16 SENECA cores [42].

Seneca [42], developed by IMEC, is a flexible, efficient neuromorphic processor using a hierarchical control system. This controlling system uses a flexible controller (RISC-V) and an optimized controller (Loop Buffer). The architecture is illustrated in Figure 3.4. The topology is a 2D mesh, and core-to-core GALS is implemented to ensure scalability.

Communication between cores is done using an NoC with a minimal footprint. This NoC supports multicasting for data transfer via a register-based routing table that contains filters to define input and output links. The packets transferred are compressed, allowing for variable-length packets to be supported. Looking at the area of one core, it is calculated that this simplistic NoC implementation occupies 2.1% of the total area. These results are based on an implementation using 8 PEs, 22 KB of register-based memory, 2 MB of Data Memory, and 128 KB of instruction memory, with a 500 MHz clock frequency.

To map a fully connected network, multiple techniques are used. This flexibility allows neural architecture search (NAS) approaches to co-optimize algorithm accuracy and hardware performance (HW-NAS) on the Seneca. Some mapping techniques mentioned include floating-point and binary spikes, spike grouping, and quantization.

3.5 Summary

Table 3.1 show the summary of the techniques used in different NoC designed for neuromorphic processors. The most used design is a 2D mesh with a deterministic routing algorithm, and core-to-core GALS. How the packets traverse through the NoC is different in each design.

	Topology	**Routing**	**Flow control**	**Switching**	**Scheduling**	**Mapping**	**Clock**
SpiNNaker [19]	2D triangular mesh	Router network tables	Blocked packets get timed-out	Packet-switching	Fixed, different per core	PACMAN	Core-to-core GALS
SpiNNaker2 [34]	2D mesh	XY-routing/ routing tables	Blocked packets get timed-out	Packet-switching	Fixed, different per core	Hierarchical mapping	Core-to-core GALS
TrueNorth [3]	2D mesh	Relative XY-routing	Stall/Go	Store-and-forward	First-come first-serve	Four algorithms to minimize wire-length	Core-to-core GALS
Loihi [15]	2D mesh	Dimension-order routing	REQ/ACK handshake	Packet-switching	Event-driven	Hierarchical population mapping	Core-to-core GALS
Seneca [42]	2D mesh	Register-based routing table		Packet-switching		Spike-grouping	Core-to-core GALS

Table 3.1: Comparison of neuromorphic chip architectures

4 Network on Chip Design Choices

In Section 2.2, the design choices found in literature are described. This chapter explains the choices made for the simplistic NoC. First, requirements are defined based on the main and sub-questions. Then, the key architectural components are selected and justified: routing algorithm, flow control, buffer type, switching method, and scheduling technique.

4.1 Requirements

A few requirements are defined to answer the main research question; “How can the NoC in a neuromorphic processor be optimized for efficient Power, Performance, Area (PPA) trade-offs while handling congestion and optimizing data transfer mechanisms?”.

- Power/Area efficiency: The NoC must deliver all packets while maintaining low area and power overhead. This follows from the efficient PPA trade-offs and is an important part of the research.
- Backpressure: The NoC has to account for congestion in the system, while refusing to drop packets. This means that the right flow control must be chosen with a low area and power overhead.
- Optimized packet transfer: The NoC must be optimized for neural network data. When choosing a routing algorithm and flow control, the types of packets and the characteristics of a congested system must be considered.
- Scalability: To make full use of the NoC, it has to be scalable. This allows larger neural networks, which require a larger number of PEs, to be mapped onto the neuromorphic core.

In Subsection 2.2.1, the topology was selected to make the literature review more specific. Due to the path diversity, scalability, and average hop count, the 2D mesh was chosen as the topology. The 2D mesh is the topology most commonly adopted in current neuromorphic processors, as discussed in Chapter 3, allowing system comparability.

4.2 Routing

XY [46], odd-even [23], dynamic XY [32], and NoP wormhole [7] routing algorithms are described in Subsection 2.2.4, these routing algorithms are deadlock and live-lock free. Most neuromorphic processors utilize a type of XY-routing; this simplistic algorithm will serve as a baseline. To ensure low latency, the minimal path must be followed; the requirements state that the power and area overhead must be low. Another requirement states the need for backpressure, which

can congest the NoC, so a routing algorithm with path diversity and congestion avoidance is also preferred. Table 4.1 compares the routing algorithms based on these criteria. With these requirements, the options for an efficient routing algorithm are XY routing and dynamic XY routing. Dynamic XY-routing would be best as it has more path diversity and local congestion awareness. The expectation is that, due to this path diversity and congestion awareness, the throughput should be higher and the latency lower, while maintaining a low power and area overhead. To ensure that dynamic XY-routing is the right algorithm for this design, some tests are defined and performed in Section 6.2.

Routing algorithm	Power/area overhead	Path diversity	Minimal path	Congestion avoidance
XY [46]	Low	Low	Yes	No
Odd-even [23]	Low/Medium	High	No	Yes, locally
Dynamic XY [32]	Low	Medium	Yes	Yes, locally
NoP wormhole [7]	High	High	No	Yes, globally

Table 4.1: Design exploration routing algorithm.

4.3 Flow control

To handle congestion, a flow control algorithm must be defined. In Subsection 2.2.2, ACK/NACK, credit-based, and on–off flow control are discussed. ACK/NACK flow control works with single-bit valid/ready signals. Credit-based flow control adds a signal that must support the depth of the buffer. On–off flow control adds one signal between routers to indicate when it is ready to receive data based on a threshold. As buffer size increases, credit-based flow control becomes less area-efficient due to the additional signals required. The on-off and ACK/NACK controls have the fewest additional signals, but the on-off control cannot fully utilize the entire buffer. So the ACK/NACK control is best for this design, as it optimizes data transfer with one additional signal between the router and buffer.

4.4 Buffer

The type and size of the buffer are important choices in the design, as they can significantly add to the design's overhead. The buffer type most used in Chapter 3 is the FIFO buffer. The buffer can be placed on either the input or the output of a router, but not both, to avoid extra delay. Usually, it is used on the input of the router to simplify the output arbitration. The downside is that it is prone to Head-of-Line (HOL) blocking [38], meaning that one packet can delay other packets to a noncongested direction. To mitigate this issue, Virtual Vhannels (VCs) can be used, which divides packets into multiple FIFOs on the output direction. This type has a lot of power and area overhead, but it does solve the HOL congestion problem. Neuron data are a bunch of packets to the same destination, so HOL blocking will not be the biggest problem. To minimize the power and area overhead, the FIFO buffer is chosen at the input of the router.

To reduce latency, a First-Word Fall-Through (FWFT) FIFO is used instead of a standard FIFO. This type of FIFO makes the first word of incoming data immediately available at the output without requiring a read enable signal. This behavior helps reduce the overall packet latency without introducing additional control complexity or significant area overhead. To choose the optimal FIFO depth, a trade-off must be made in terms of power, area, latency, and throughput. With a large depth, the FIFO will waste power and area, and a small FIFO risks congestion and a lower throughput. To minimize overhead, the depth of the FIFO must be as low as possible

while maintaining a high throughput. Tests are defined and performed in Section 6.2 to find the optimal FIFO depth.

4.5 Arbitration and Scheduling

Arbitration is how the packet is routed to the input FIFO of the next node. This is the connection between the output of the router and the input of the next router, also known as the switching strategy (Subsection 2.2.3). Circuit switching involves reserving a path and routing a stream of packets along this path, which adds some overhead and blocks the reserved path. Packet switching transfers packets one at a time; this is the type of switching most suited for this design. It has less overhead and supports multiple traffic flows simultaneously.

With packet switching, examples are shown of packets containing multiple flits. With neural network data, flit transfer is the most efficient, as the body contains all neural outputs, with a header and a footer. However, this adds complexity to the design and is beyond the scope of this project. This means that the store-and-forward switching technique is used, and the flit implementation is added as a future work in Chapter 9.

The input and output of the routers are connected, but a schedule must be made when a router receives packets from multiple directions. In Subsection 2.2.5, the most common schedules are shown: Round Robin (RR), Least Recently Used (LRU), and fixed priority. The fixed priority creates an unbalanced priority and serves a specified port first. The LRU has a bit more overhead than RR, as it must store the order of which direction is last used. This means that the most optimized option is the RR scheduling, which has a cyclic priority based on the last port served.

4.6 System-Level Architecture

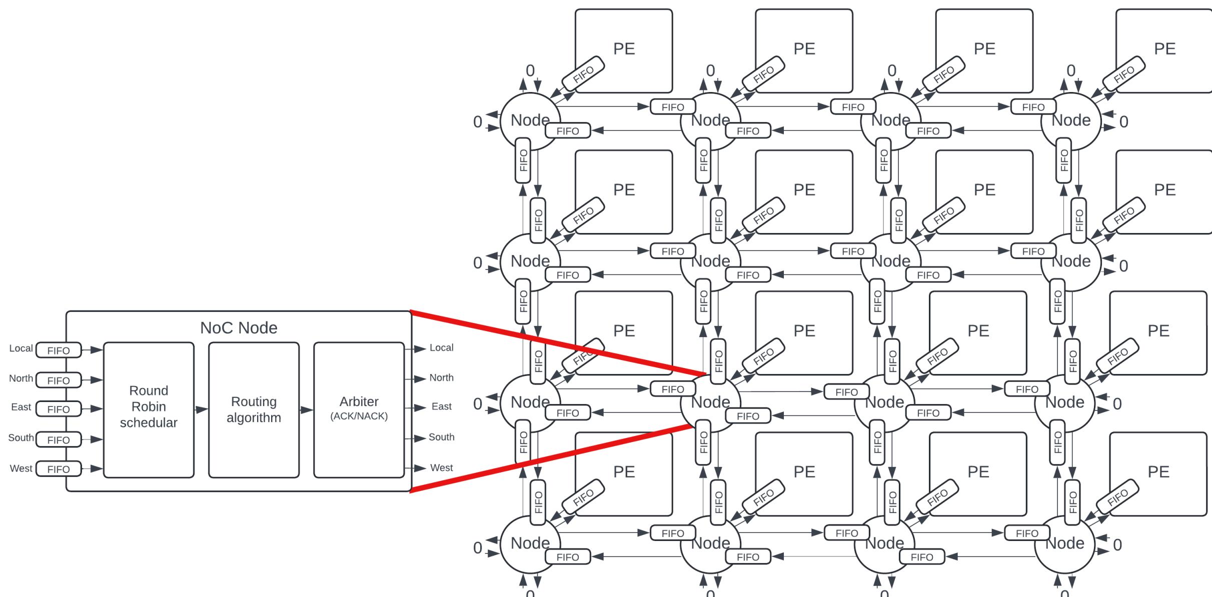


Figure 4.1: Proposed 4x4 NoC architecture.

Figure 4.1 shows a 4-by-4 NoC implementation, with the following design choices:

- Topology: The 2D mesh is selected due to its path diversity, scalability, and average hop count. It is also used by existing neuromorphic processors, making the systems comparable.
- Routing Algorithm: The best algorithm for this design is chosen in Section 6.2, where XY and dynamic XY-routing are compared. The theory states that dynamic XY-routing has more path diversity, making it less prone to congestion and maintaining a higher throughput with a lower latency.
- Flow Control: The ACK/NACK control is chosen to be implemented using valid/ready signals because it can be implemented with low area overhead while holding congested packets in the entire buffer.
- Buffer: An input FWFT FIFO is chosen as the buffer because it has a low power and area overhead compared to other buffer types. To maintain this low overhead while maintaining high throughput, tests are performed in Section 6.2 to define the optimal FIFO depth.
- Switching Strategy: Store-and-forward is chosen as a straightforward switching technique that avoids blocking paths and has low area overhead.
- Scheduler: A Round Robin (RR) input scheduler is chosen because it gives a cyclic priority to the input ports based on the last port served while having a low overhead.

The goal of this thesis is to develop a PPA optimized and efficient NoC for a neuromorphic processor. As a result, certain design choices discussed in Chapter 2 are beyond the scope of this project. These design choices, such as GALS clock distribution and mapping, are left for future works in Chapter 9. The current design utilizes a synchronous clock and therefore employs synchronous FIFOs and manual mapping.

5 NoC Implementation

The described NoC is implemented using VHDL and made open-source (Appendix A). This section provides a detailed explanation of the data flow, coding decisions, and the connection to the NEORV32.

5.1 Topology

The 2D mesh topology is configurable through global parameters in a package file. This file consists of constants and types used throughout the VHDL code. The top architecture is called Mesh, which instantiates the cores consisting of NoC nodes and the PEs. The NoC nodes are connected with the data, valid, and ready signals. The data signal carries the packet and has a word length equal to the size of the packet. The valid and ready signals are single-bit flags used to synchronize communication between nodes. As shown in Figure 4.1, when a node is located on the edge of the mesh, the edge ports are connected to zeros. This means that all three signals are set to zero, so the data cannot be routed in that direction, as the valid and ready signals are never active.

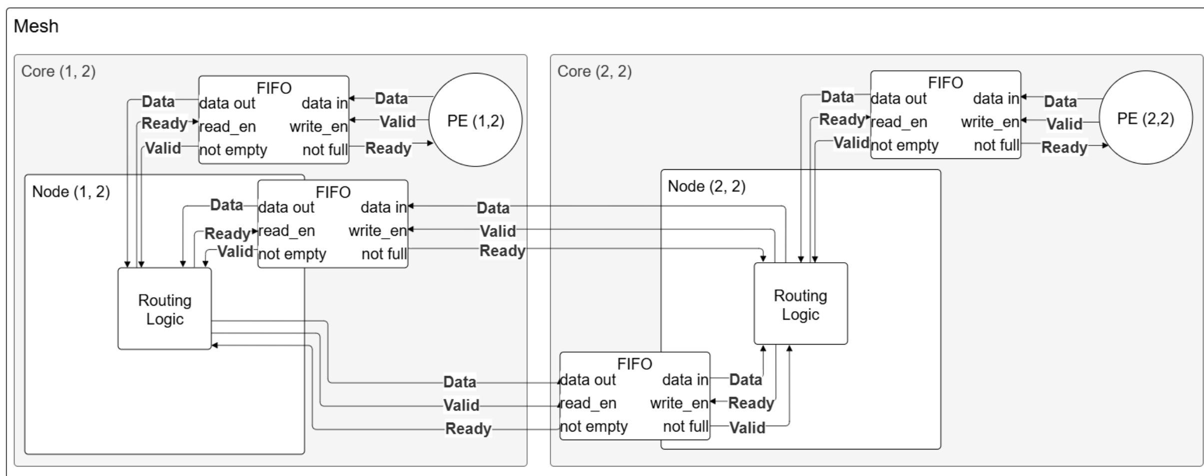


Figure 5.1: 2x1 connection diagram

Figure 5.1 shows a 2-by-1 mesh with a detailed communication connection between the two nodes. The FIFO is located on the input of its related node, which connects the FIFO to the router. The output of the FIFO is connected to the input of the router with three signals: the `data` signal, the `read_en` signal to the `ready` signal of the router, and the FIFO `not_empty` to the `valid` signal. The router output is then connected to the input FIFO of the next node. This router and FIFO are also connected with three signals: the output of the router to the input of the next FIFO, the valid signal of the router to the write enable signal of the next FIFO, and the ready signal of the router is the FIFO not full.

5.2 NoC Node

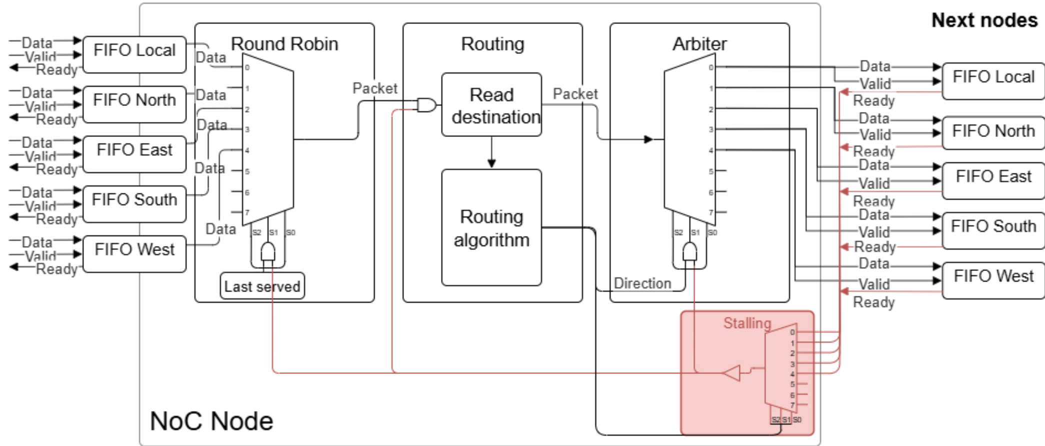


Figure 5.2: A single detailed NoC node

In Figure 5.2, a detailed look inside the NoC node is displayed. This figure shows the scheduling, routing, backpressure, and arbitration of a packet. When a packet arrives in one of the FIFOs, the FIFO ready signal is set high, and the scheduler will know that a valid packet can be received. The packet gets a priority based on the last port served with the RR scheduler (Appendix B). The selected FIFO is read with the ready signal, and the packet is propagated to the router. Inside the router, the X and Y destinations of the packet are determined, and the packet is routed with the routing algorithm. As stated Chapter 4, this is either the deterministic XY routing (Appendix C), or the dynamic XY routing (Appendix D). The routing algorithm provides the direction towards the destination, and the packet is routed accordingly.

When a packet is routed to a direction where the FIFO is full, the whole node is stalled. This stalling can occur in the router using the dynamic routing algorithm or during arbitration when a packet cannot be routed in a particular direction. With dynamic XY routing (Appendix D), a local awareness of the readiness of the next FIFO is displayed. When the X and Y directions towards the destination of a packet are both full, the router is stalled.

5.2.1 Waveform: single packet

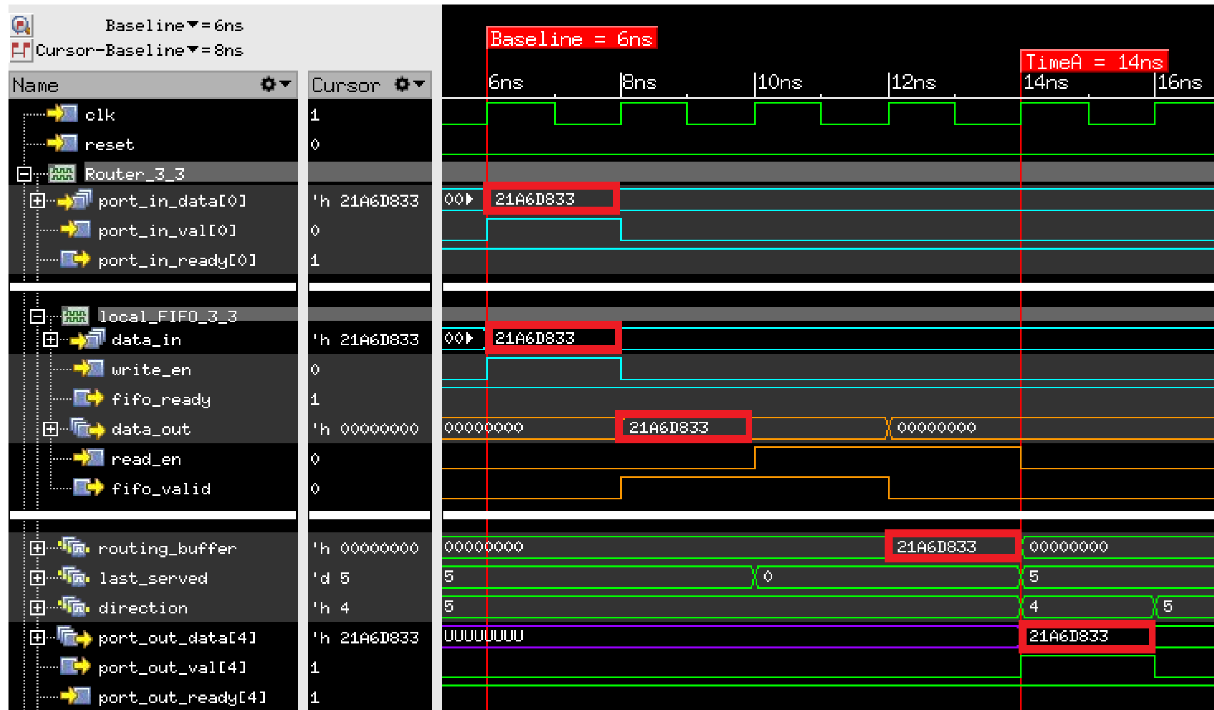


Figure 5.3: Waveform of a routed packet as simulated in Cadence Xcelium [14].

Figure 5.3 shows a single packet being routed through the NoC. At 6 ns, the waveform shows a packet coming from the PE of (2,2). In the same clock cycle, the packet is written to the local FIFO. The blue signals in the waveform are of the same type, but are observed in different parts of the system. The first three blue signals are seen at the mesh interface, while the other three are their counterparts within the FIFO. In the next clock cycle, the packet is written into the memory of the FIFO. This packet is also immediately seen on the output, due to the FWFT FIFO. Once a packet is detected in the memory, the `fifo_ready` is enabled, indicating to the routing logic that data is available in this FIFO. The scheduler detects this valid signal in the next clock cycle. Since no other FIFOs are available, it selects the packet inside the local FIFO (port 0) for processing. The packet is routed through the `routing_buffer` in the next clock cycle in the right direction. In this case, the correct direction is the west (port 4), regardless of the routing algorithm. Determining the routing direction takes one clock cycle, after which the packet is placed on the fourth output of the node, heading west.

5.3 NEORV32

With NoC communication in place, the NEORV32 can be integrated as a PE. The NEORV32 is a simple RISC-V core with optional features that allow customization for specific applications, as described in Section 2.3. For the neural network application, the integer base and integer multiplication/division instructions are selected, resulting in the RV32IM Instruction Set Architecture (ISA). Although multiplication increases the area, it is beneficial for neural networks due to the use of multiply-accumulate (MAC) operations.

To execute the neural network code, an Instruction Memory (IMEM) is required to store the application image and initiate execution at boot. Additionally, the NEORV32 needs a Data

Memory (DMEM) to store the network's weights. While the standard NEORV32 instance uses true Random Access Memory (RAM), this implementation replaces it with Static Random Access Memory (SRAM). The SRAM is initialized using hexadecimal files that contain control data and weights. To make the DMEM adaptable for each node, the NEORV32 instance is modified so that the base name of the SRAM folder can be specified through generics. Finally, to enable communication with the external NoC, the NEORV32 is connected via the External Bus (XBUS) interface.

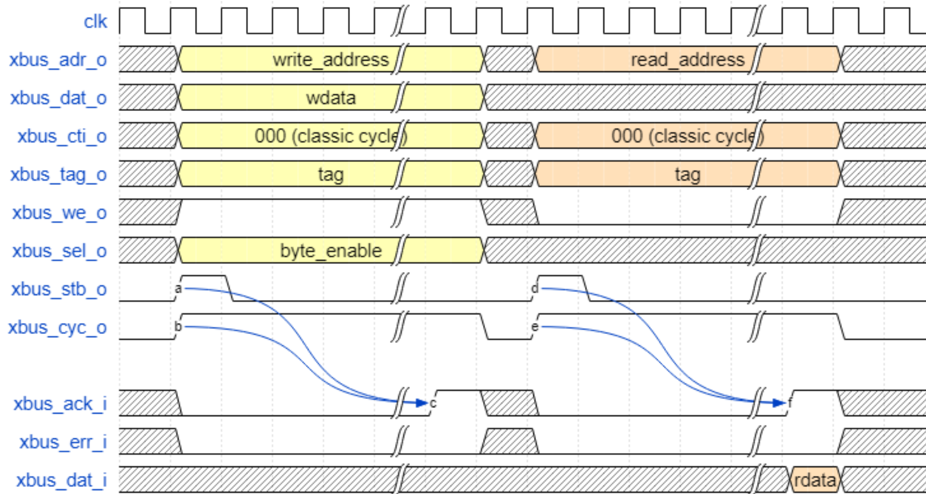


Figure 5.4: XBUS transfers: Write (left)) and Read (right) [36]

The XBUS uses the Wishbone protocol [43] to connect processor-external modules. This bus protocol features a master-slave architecture, where the NEORV32 CPU serves as the master, and the external hardware entities function as the slaves. Figure 5.4 shows the XBUS read and write transfers. The master initiates a read or write transfer to the target address (`xbus_adr_o`) within the non-assigned processor address space, enabling the cycle (`xbus_cyc_o`) and strobe (`xbus_stb_o`) signals. To write to the address, the write enable (`xbus_we_o`) is high, and the data in (`xbus_dat_i`) is written by the slave. To read from the address, the write enable is low, and the content of the address is on the data out (`xbus_dat_o`) signal. The transfer cycle is disabled when the slave generates a cycle-terminating signal, which is either an acknowledgment (`xbus_ack_i`) or an error (`xbus_err_i`) signal.

Communicated data	Address	Data description
INPUT_FIFO_VALID_ADDR	0xF0001100	Valid signal of the input FIFO
INPUT_FIFO_DATA_ADDR	0xF0001000	Data output of the input FIFO
OUTPUT_FIFO_READY_ADDR	0xF0002100	Ready signal of the output FIFO
OUTPUT_FIFO_DATA_ADDR	0xF0002000	Data input of the input FIFO

Table 5.1: XBUS Addresses

Table 5.1 shows the different addresses used in the XBUS communication; these addresses are within the unused address ranges of the NOERV32 processor. The table shows the four types of data communicated between NEORV32 and NoC. Inside the PE instance, NEORV32 is initialized, and slave communication is handled. The slave responds to these addresses and communicates the related data to or from the NEORV32. The NOERV32 CPU will sleep until the valid signal triggers a machine external interrupt, which means the FIFO holds a packet. This valid signal is communicated to the RISC-V, and the packet inside the input FIFO is read when the valid signal is one. After the neural network calculations, the output is sent to the next node. This packet must first be received by the output FIFO; this is done by trying to write the

FIFO until the FIFO is not full. When the output FIFO is written and the acknowledgment is received, the next output packet is written.

5.3.1 Waveform: XBUS with FIFO

Listing 5.4 Simplified code snippet for NEORV32 input FIFO read

```
1 neorv32_cpu_sleep();
2 uint32_t in_val =  *(volatile uint32_t*)INPUT_FIFO_VALID_ADDR;
3 if (in_val == 1) {
4     int32_t value =  *(volatile uint32_t*)INPUT_FIFO_ADDR;
5     ...
6 }
```

In Listing 5.4, a small part of the implemented event-driven neural network C-code is shown; the whole code can be seen in Appendix E. The NEORV32 is put to sleep; this sleep function puts the NEORV32 in low-power mode, waiting for an interrupt. The `fifo_valid` signal is connected to the machine's external interrupt. When this signal is one, the NEORV32 will be interrupted from sleep and continue. After this interrupt, the `fifo_valid` signal is read via the XBUS. When this is one, the packet inside the FIFO is read and processed.

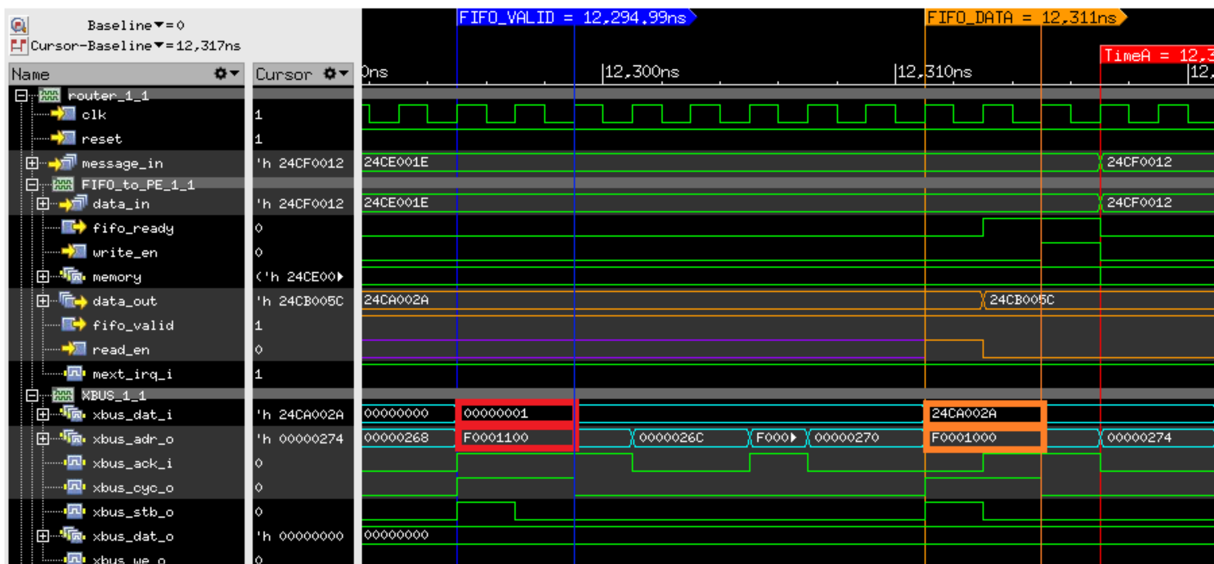


Figure 5.5: Input FIFO communication with the XBUS of NEORV32, simulated in Cadence Xcelium [14].

The waveform in Figure 5.5 shows the communication between the PE input FIFO and the NEORV32 XBUS while the NEORV32 executes the code shown in Listing 5.4. Initially, data is available inside the FIFO, which sets the interrupt signal `mext_irq_i` high. Although this interrupt is not connected to an interrupt routine, it wakes the NEORV32 from sleep. Next, the XBUS initiates communication with the FIFO by attempting to read the `INPUT_FIFO_VALID_ADDR`, as indicated by the red blocks in Figure 5.5. This enables the `xbus_cyc_o` and `xbus_stb_o` signals. When the `xbus_stb_o` is detected, the communication handler will compare the `xbus_adr_o` against the addresses in Table 5.1. At address `0xF001100`, the `fifo_valid` signal is transferred via the `xbus_dat_i`. The handler then acknowledges the transfer, completing the communication cycle in two clock cycles.

The software then checks whether the received data is equal to one. Since it is, a second XBUS transfer is initiated, as shown by the orange blocks in Figure 5.5. A new transfer cycle begins, and the packet is read from the FIFO. This packet is placed on the `xbus_dat_i` signal, and acknowledged by the handler. The packet is then ready for processing by the NEORV32 CPU.

5.4 Packet Structure

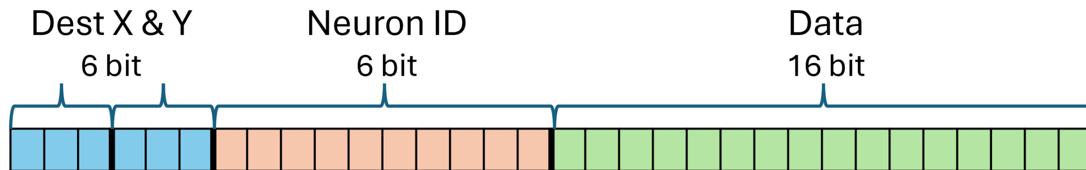


Figure 5.6: Neuron packet

The packet traversing through the NoC is shown in Figure 5.6. The first 6 bits are the destination address of the packet, which is used to route the packet to its intended destination. The second 6 bits are the neuron ID, which is used inside the NEORV32 to select the correct set of weights based on the neuron from which the packet originates. The last 16 bits are the data bits; the output data type is a 16-bit fixed-point number, with the most significant bit serving as the sign bit and the last seven bits representing the fractional part (Q9.7). Packets are not sent if the payload is zero, preventing irrelevant data from being transmitted. The end of a neuron output stream is indicated with an End of Frame (EOF) packet; this control packet has a neuron ID of all ones, and a data payload of all zeros. When the EOF packet is received, the neuron calculation is finished, and the output is sent.

5.5 Clocking Strategy

To optimize for power consumption, the NEORV32 is put to sleep when no packets are processed. Another power optimization technique is to turn off the clock to the NoC node and FIFOs when they are idle. This technique is known as clock gating and is implemented using a clock enable signal and an AND gate in a synchronous system. In the NoC node, the clock enable is based on the valid signal of the input FIFO and some internal routing and stalling signals. The clock must be enabled when an input FIFO has a packet and when a packet is being routed or stalled; otherwise, a packet could be dropped. The clock gating in the FIFOs is based on the empty and write enable signals. The FIFO clock can only be turned off when there is nothing inside the FIFO and nothing is written to the FIFO.

6 Results

This chapter presents the results of the design choices mentioned in Chapter 4. Tests are defined to determine which routing algorithm and which FIFO depth are best suited for this design. When the design is complete, the NoC without the NEORV32 is benchmarked. Then, the NEORV32 with a neural network is added to the design. Finally, the PPA traded-offs of the system are shown.

6.1 Experimental Setup

To benchmark the implemented NoC, tests with standardized metrics and tools must be defined. The main question already mentions Power, Performance, Area (PPA), which is an important part of the results. Latency and throughput are chosen to measure the performance of the NoC. Latency is reported as latency per hop in nanoseconds, and throughput is the number of packets per clock cycle. The NoC implementation operates at a clock frequency of 500 MHz, meaning that one clock cycle takes 2 ns. For a fair comparison, all simulations are carried out using Cadence Xcelium [14]. Area and power measurements are performed with Cadence Genus [12] and Cadence JOULES [13]. The tests are performed with Global Foundries FDX+ 22nm technology with the ultra-low power library and physical conditions of 0.5 VCC and 25°C.

In Chapter 4, two design choices are proposed for testing to see which is best for this design. In this section, the tests are carried out. The first design choice is the FIFO depth; this test will compare multiple FIFO depths to determine which is best in terms of area, latency, throughput, and percentage of packets delivered. The second design choice is the routing algorithm; in this test, a comparison is made between XY routing and dynamic XY routing. The metrics important for this choice are: area, latency, throughput, and hotspot detection.

To obtain the benchmark results, two NoC configurations are defined: one without NEORV32 and one with NEORV32. The one without NEORV32 is used for the following tests:

- Randomized data: shows the metrics when the mapping is not optimal. This test tries to insert a packet into every local input FIFO in a random direction with each increase in the clock cycle. This means that the system becomes congested, allowing the behavior to be studied.
- Randomized directional data is more representative of a well-mapped neural network. This test also attempts to insert a packet at every clock cycle into all nodes. The packets are directional; this means all packets flow towards the north-west.
- Insertion sweep test: The randomized directional data is also tested in more detail with different insertion rates, so instead of inserting a packet into every local input FIFO per clock cycle. The packets are inserted: first, one packet is inserted into one FIFO per clock cycle, then two packets per clock cycle, and so on.

- 100.000 packets into one node: This is used for power analysis. With this data, the energy per packet can be calculated, and the power of an idle node is easily read.

With NEORV32, a fully connected neural network is mapped to the NoC. This test will compare the NoC PPA with the NEORV32 PPA, to answer the sub-question; “How much of the total PPA of neuromorphic processors is contributed by the NoC?”.

6.2 Evaluation of Design Choices

To choose the last two design choices, the randomized directional test is defined and performed. First, the FIFO depth for this design must be chosen. The metrics used to determine this are: latency, throughput, area, and percentage of delivered packets. This test will be performed with both the XY and dynamic XY algorithms. With this FIFO depth, the XY and dynamic XY-routing algorithms are compared in terms of throughput, latency, hotspot detection, and area to make an informed decision about the right trade-off for this algorithm.

6.2.1 Randomized Directional Data

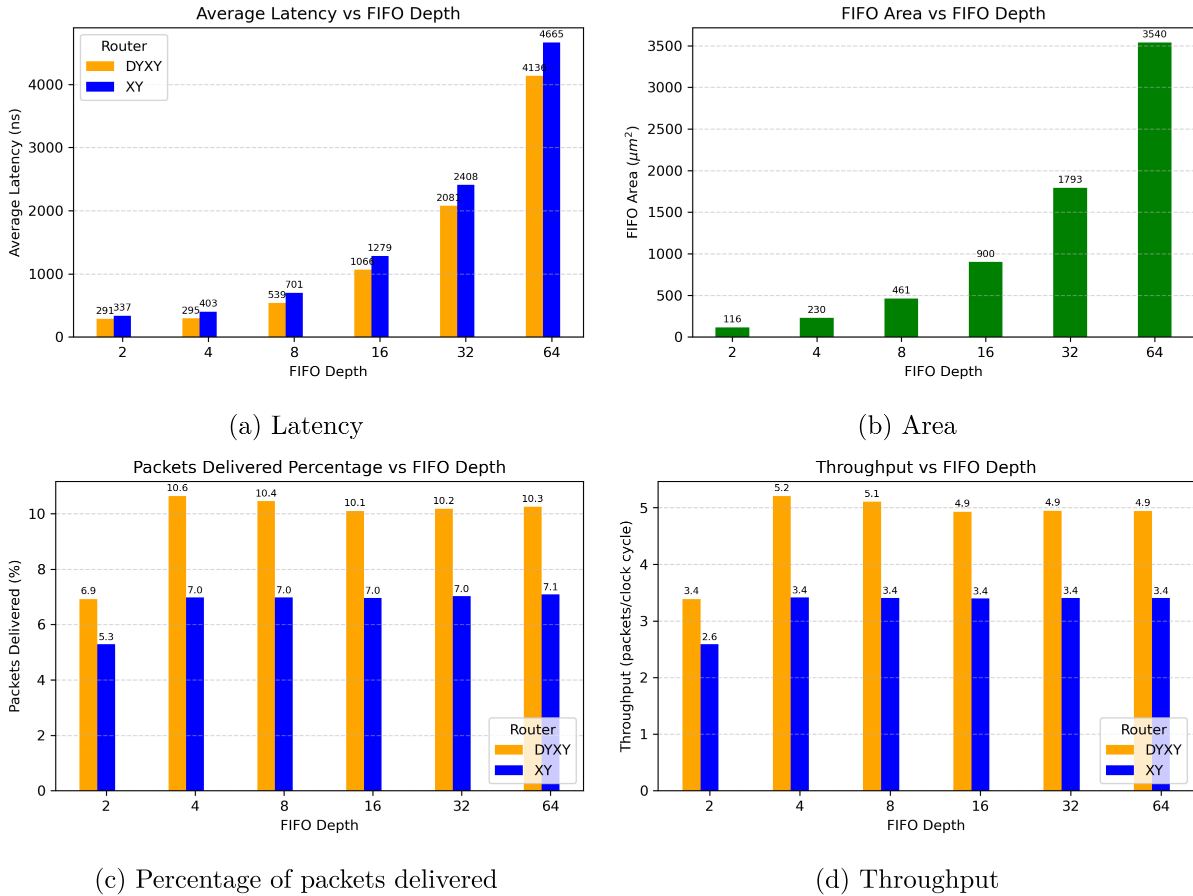
Listing 6.0 Code snipped for generating upstream data.

```
1  # Find valid upstream destinations (strictly above and to the left)
2  possible_destinations = [
3      (x, y)
4      for x in range(source_x)
5      for y in range(source_y)
6  ]
7
8  if not possible_destinations:
9      continue  # No valid destination
10
11 # Choose a random valid destination
12 dest_x, dest_y = random.choice(possible_destinations)
```

The randomized directional test data is generated using the code in Appendix F, a snippet is shown in Listing 6.0. This code generates a data flow to the top left of the NoC. It skips nodes located on the top and/or the left in the NoC. An input file is generated that can be loaded by the testbench. The testbench will attempt to insert 49 packets in an 8-by-8 mesh per clock cycle for a specified number of clock cycles. This test can reject packets when the local FIFO is full. For example, at a particular clock cycle, only one local FIFO is available and the other 48 packets are refused. The amount of packets delivered fluctuates due to this refusal and the difference in routing. The data flows towards the north-west, so only the south and east FIFOs are used.

6.2.2 FIFO word size

To determine the FIFO depth, a trade-off must be made between throughput, latency, area, and the number of delivered packets. This trade-off is tested with the randomized directional data. The expected result is that with a higher FIFO depth, the area, throughput, latency, and the percentage of delivered packets will increase.



(a) Latency

(b) Area

(c) Percentage of packets delivered

(d) Throughput

Figure 6.1: Graphs displaying different metrics versus the FIFO depth.

Figure 6.1a shows the expected increase in latency with a greater FIFO depth, as packets experience longer wait times within the system. The corresponding growth in area as the FIFO depth increases is seen in Figure 6.1b. Figure 6.1c presents the percentage of packets delivered by the congested NoC, where 4.9 million packets are attempted to be inserted. While no packets are dropped, not all are accepted into the system due to flow control. Even in the best-performing case, only 10.6% of the packets are successfully delivered, meaning that 89.4% of the attempted packets are rejected due to congestion. This graph is closely related to the throughput graph in Figure 6.1d, where throughput is calculated as the number of packets delivered divided by the duration in clock cycles. The data shows that throughput is affected by a very low FIFO depth and the routing algorithm. From a FIFO depth of 4, the throughput remains consistent, indicating that a larger FIFO depth does not increase throughput. In this test, a FIFO depth of 4 achieves the highest efficiency, delivering the maximum number of packets in the shortest duration.

The graphs in Figure 6.1 show that the optimal depth of a FIFO is four packets. At this depth, latency and area overhead remain low, while throughput is maximized. Increasing the FIFO depth beyond four does not lead to further improvements in throughput; instead, throughput peaks at this point, delivering the highest number of packets in the shortest time. This suggests that a depth of 4 offers the best trade-off between area and performance, making it the most efficient choice for the system.

6.2.3 Routing algorithm

With the FIFO depth tests, both routing algorithms were used to conclude the best depth for this design. This test will use the FIFO depth of 4 to determine which algorithm is more suitable: XY-routing or dynamic XY-routing. The same randomized directional test is used, with the metrics: throughput, latency, area, and hotspot detection.

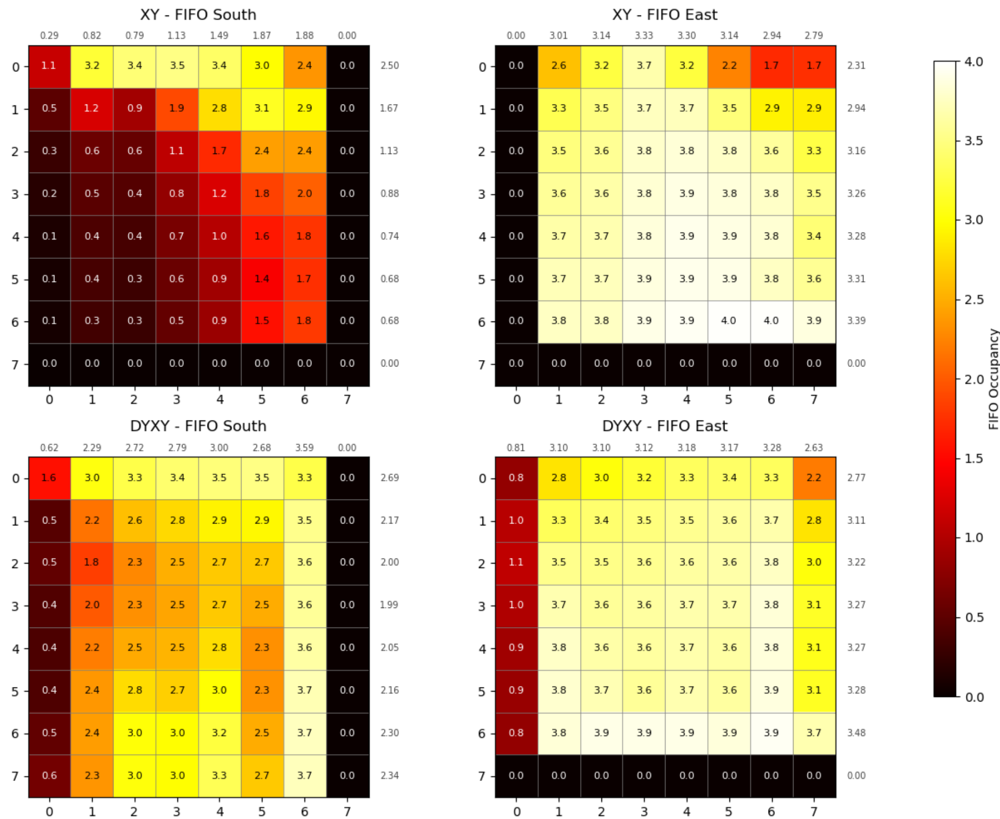


Figure 6.2: Heat map of the in-use FIFO occupancy in XY and dynamic XY routing with averages.

Figure 6.2 shows heat maps of the average occupancy in the used FIFOs for each algorithm. This test uses the south and east FIFOs, because packets are sent to lower X and Y coordinates in the north-west direction. Within the heat maps, brighter nodes indicate higher average occupancy levels. The gray text along the sides represents the average occupancy per row (left) and per column (top). In both routing algorithms, the east FIFO shows the highest occupancy. This is because both algorithms prefer to route packets along the X direction first, making greater use of the east FIFO. The south FIFO occupancies differ the most, because the dynamic XY algorithm routes a packet towards Y when X is full. This behavior increases path diversity in a congested system, distributing traffic more evenly and resulting in more balanced occupancy across the multiple FIFOs.

Table 6.1 reports the throughput, latency, and area of each routing algorithm. The throughput is reported as the number of packets handled per clock cycle in a fully congested system. The throughput of the dynamic XY-routing is almost two packets per clock cycle higher than that of the XY algorithm. This performance improvement is due to the path diversity seen in the heatmaps. This path diversity also lowers the latency per hop for the dynamic XY algorithm.

Metric	XY	Dynamic XY
Throughput ($\frac{packets}{clock_cycle}$)	3.4	5.2
Latency per Hop (ns)	92.1	64.9
Router Area (μm^2)	486	488

Table 6.1: Comparison between XY-routing and dynamic XY-routing

When synthesizing the implemented algorithms, the dynamic XY algorithm introduces only a minimal increase in area. Considering the trade-offs between performance and area, the dynamic XY routing algorithm is selected, as it offers superior performance with negligible additional area cost.

6.3 NoC benchmark

With the last two design choices in place, the NoC without NEORV32 can be benchmarked. The benchmark begins with the same test used in the design choice section, using randomized directional data to simulate structured traffic. Then, an insertion sweep test is performed using the same directional data. This test increases the number of packets inserted each clock cycle from 1 to 48 to analyze the behavior in different congestion states. Finally, the system is tested with fully randomized data to simulate unstructured traffic patterns and assess performance in less predictable scenarios.

6.3.1 Directional data

In Table 6.1, throughput, latency per hop, and area of the dynamic XY router are displayed. These metrics are based on a fully congested system, so the latency per hop also includes the time in the input FIFO and the stall time.

Stages FIFO	**1**	**2**	**3**	**4**
Receive input	P1	P2	P3	P4
Write input to memory		P1	P2	P3
Put memory data on the output			P1	P2
Stages Router	**1**	**2**	**3**	**4**
Read enable non-empty FIFO and read output			P1	P2
Extract X & Y destinations, and route to right direction				P1

Table 6.2: Pipeline diagram router.

A pipeline diagram of the implemented NoC node is shown in Table 6.2. The diagram illustrates the stages a packet undergoes in a non-congested system, which is also depicted in Subsection 5.2.1. This node takes four clock cycles for a packet to go from the input FIFO to the next node's FIFO. One clock cycle takes 2 ns, due to the 500 MHz clock frequency, so the latency per hop in an empty NoC is 8 ns. The propagation delay is four clock cycles, and the throughput is 1 packet per clock cycle after this initial propagation.

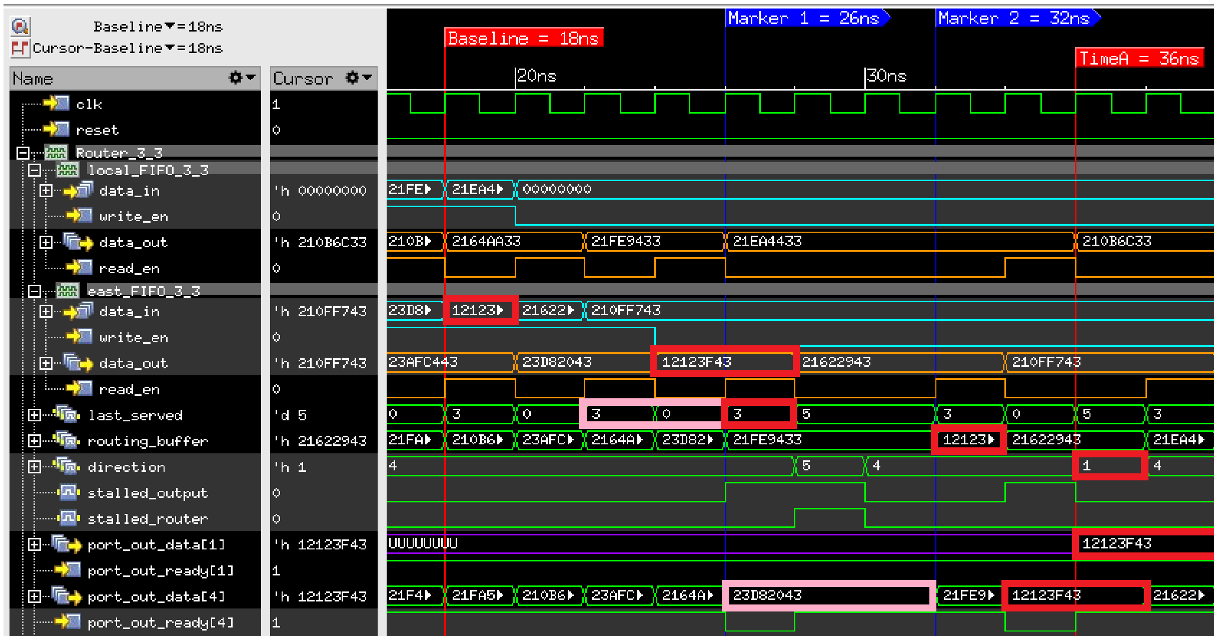


Figure 6.3: Waveform of a congested NoC as simulated in Cadence Xcelium [14].

Figure 6.3 illustrates the behavior of a congested NoC. This waveform shows node (3, 3), where a packet (highlighted in red) arrives from the east direction at 18 ns. This packet is read 4 clock cycles after it is available, at 26 ns. The packet must first be available in memory, which takes 2 clock cycles. In the next clock cycle, the previous packet from the east FIFO (port 3) is served, and in the cycle afterwards, the local FIFO (port 0) is chosen by the Round Robin (RR) scheduler (indicated in pink). After the packet is read, the node stalls because the pink packet on the output cannot be routed. At 32 ns, the packet can finally be routed after being stalled for 3 clock cycles. The west direction (port 4) is chosen because of a preference for the X direction, and its ready signal is high. However, in the next clock cycle, this direction is no longer ready, and the packet is rerouted to the north direction (port 1). This congested packet took 18 ns or 9 clock cycles to traverse from input to output. Under normal conditions, the latency is 4 clock cycles. In this case, 2 additional cycles were spent waiting for the FIFO to be selected by the scheduler, and 2 more cycles were lost due to stalling caused by output congestion. The last cycle is lost because the packet was rerouted.

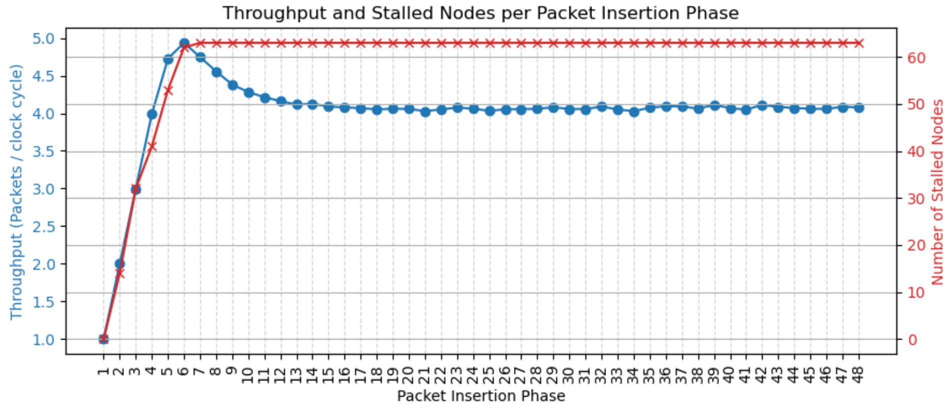


Figure 6.4: Throughput and Number of Stalled Nodes per Insertion Phase

To analyze the NoC with different levels of congestion, the directional insertion sweep test is performed. This test first inserts one packet per clock cycle and gradually increases to a maximum of 48 packet insertions per clock cycle. Between each insertion level, there is an idle period to ensure the NoC is empty before inserting new packets.

Figure 6.4 illustrates the impact the insertion phase has on the throughput (blue on the right) and number of stalled nodes (red on the left). In the first four phases, the number of packets attempted to be inserted and the number of packets delivered per clock cycle are the same, indicating that the system handles all packets optimally. Beyond this point, the system begins to insert more packets per clock cycle than the FIFO depth allows, resulting in fewer packets being accepted. The number of stalled nodes increases until insertion phase 7, when all nodes stall at least once. Before this phase, throughput peaks at 4.9 packets per clock cycle at insertion phase 6. This peak in throughput indicates an optimal workload; before this, the FIFOs are underutilized, and after this point, the system becomes congested.

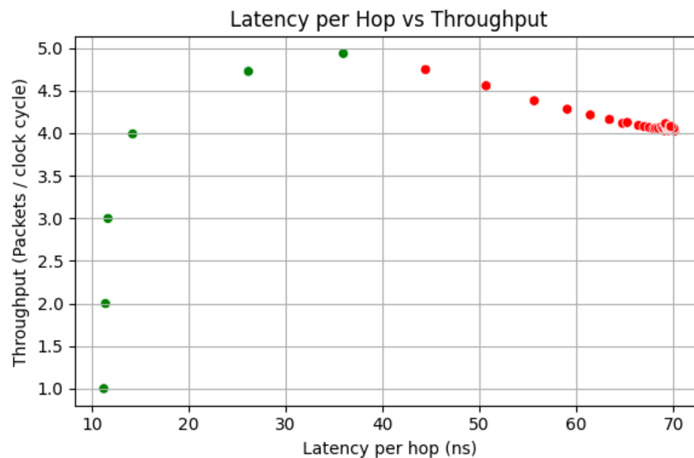


Figure 6.5: Latency per hop vs Throughput (red dots indicate all nodes stall)

Figure 6.5 shows the throughput versus the latency per hop of each packet insertion phase, where the red dots indicate all nodes are stalled at some point in the insertion phase. In the first four phases, the insertion phase and throughput are the same, and the average latency ranges from 10 to 14 ns. The latency is never exactly 8 ns per hop, due to stalls. As the insertion phase increases, the number of stalled nodes also increases, meaning the FIFOs become full. The peak

at 4.9 packets per cycle is also evident in this graph, after which the throughput drops, and the latency continues to increase. Most insertion cycles exhibit a latency of approximately 65 to 70 ns and a throughput of around 4.1 packets per clock cycle.

The stalling of all nodes is caused by the Head-of-Line (HOL) blocking of packets. As the system becomes congested, the FIFOs fill up, limiting the available buffer space. Because data can only be routed in two possible directions, packets often get blocked by a full FIFO in one direction, even if the other direction remains uncongested. This behavior is also illustrated in the congestion waveform shown in Figure 6.3, where a single blocked packet delays other packets, including those destined for non-congested directions. This HOL blocking will not become a problem, as the system is optimized for the sparse data of an event-driven neural network. It is a performance limitation and will be discussed in Chapter 7.

6.3.2 Randomized data

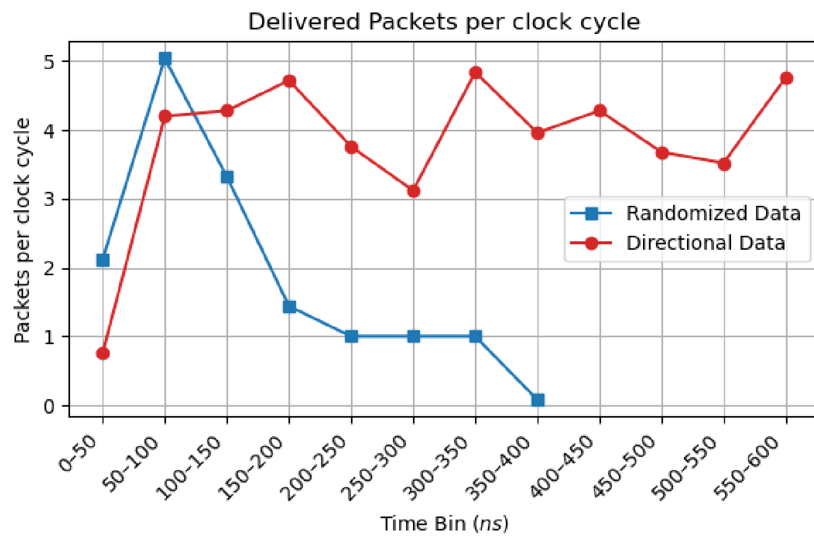


Figure 6.6: Randomized vs Directional Data Delivered

Figure 6.6 analyzes the delivered packets per clock cycle for directional and randomized data. In both cases, packets are inserted every clock cycle. For randomized data, 64 packets are attempted per cycle, whereas for directional data, 48 packets are attempted per cycle. This difference explains why the number of packets delivered in the first two bins is slightly higher for randomized data. After these initial bins, the delivery rate for randomized data decreases, eventually stalling at 1 packet per clock cycle. Ultimately, no packets are delivered, indicating that the system is fully stalled. In contrast, for directional data, throughput remains between 3 and 5 packets per clock cycle even after the system becomes congested.

This stalling of the NoC is caused by one or more deadlocks in the system. With randomized data, packets can flow in all directions, increasing the likelihood of circular dependencies. The deadlock arises from the interaction between the routing algorithm and the flow control mechanism. While the dynamic XY-routing algorithm is itself deadlock-free, problems occur when a cyclic dependency forms in the flow control.

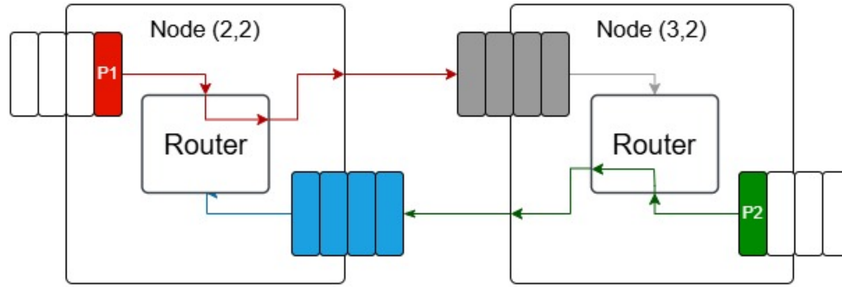


Figure 6.7: Deadlock example with randomized data.

Figure 6.7 illustrates a deadlock scenario seen in testing. In this example, only the relevant FIFOs are shown for clarity. A red packet, P1, gets into the west FIFO of node (2, 2) and is destined for node (3, 2). To reach its destination, it must pass through a full FIFO, which causes it to stall. Simultaneously, a green packet, P2, enters node (3, 2) from the east and is destined for node (2, 2). Packet P2 also needs to be routed to a full FIFO at node (2,2), causing it to stall as well. This Head-of-Line (HOL) blocking leads to a cyclic dependency, as each packet is waiting on a buffer blocked by the other. As a result, the NoC enters a deadlock state.

6.3.3 Synthesis

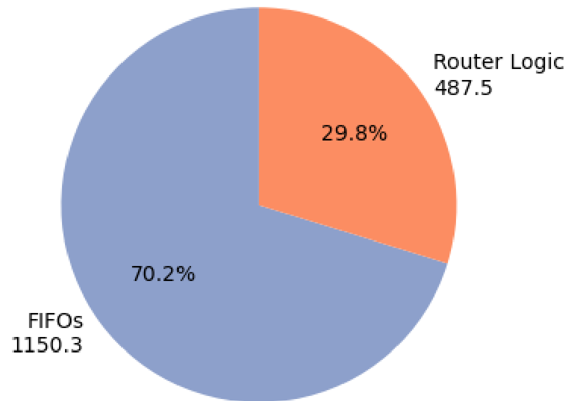


Figure 6.8: Area in μm^2 of one router (total area is 1637.8 μm^2).

The NoC is synthesized with Cadence Genus [12] using GlobalFoundries FDX+ 22nm technology. The synthesis is done with a clock frequency of 500 MHz and meets the timing constraints. The area of a middle node is shown in Figure 6.8. The total area of this node is 1637.8 μm^2, with the majority occupied by the FIFOs. The routing logic includes the Round Robin scheduler, the dynamic XY-routing algorithm, and the arbiter.

6.3.4 Power Analysis

Figure 6.9: Waveform of inserting 100, 000 directional packets into node (3,3)

Power analysis is done in Cadence Joules [13], by inserting 100, 000 packets into node (3,3). This test is shown in Figure 6.9. The waveform shows one insertion burst in (3,3), and all the output nodes are to the north-west of (3,3). With this test, an estimate of the energy per packet can be made. This is calculated using $E_{\text{per_packet}} = P \times t_{\text{one_packet}}$, where the time per packet is 2 ns, corresponding to a throughput of one packet per clock cycle.

	Idle Power (μW)	**Energy per packet** (fJ)
No clock gating	77	306
Router & FIFO clock gated	22	296

Table 6.3: Power analysis with and without clock gating.

Table 6.3 compares the NoC power consumption with and without clock gating. This table shows the energy consumption for the active node (3,3) and the idle power of node (5,5) that is not used in this simulation. The clock gating reduces the idle power consumption by 71.4% and the energy consumption by 3.3%. This reduction in energy consumption is due to the two clock-gated idle FIFO in node (3,3).

	Idle Power (μW)	**Energy per packet** (fJ)
Routing logic	2	184
FIFO	20	112
Total	22	296

Table 6.4: Power and energy analysis of the clock-gated design.

To further analyze the clock-gated NoC, Table 6.4 breaks down the idle power and energy usage per packet into two key components: the routing logic and the FIFOs. Idle power is dominated by the five FIFOs, with each consuming 4 μW, totaling 20 μW. This is twice the idle power consumption of the routing logic, which draws 2 μW. In contrast, energy consumption during active operation is dominated by the routing logic. From the energy per packet data, routing one packet costs 296 fJ, of which 184 fJ is attributed to the routing logic and 112 fJ to the FIFOs. This energy consumption is caused by the switching activity, resulting from tasks such as arbitration, scheduling, and direction selection.

6.4 Benchmark: MNIST with NEORV32

The application used to benchmark the performance of the NoC with NEORV32 is an event-driven ANN. The ANN is easier to train outside the neuromorphic processor than an SNN, and the accuracy is maintained. Furthermore, the packet information is the same; a destination, neuron ID, and neuron content must be sent. The ANN is event-driven, because neurons will not send zero packets. The data used for benchmarking is the MNIST dataset [16]. This dataset is used to benchmark multiple neuromorphic processors, such as IBM's TrueNorth [3] and Intel's Loihi [15]. These neuromorphic processors use the MNIST dataset for digit classification using a Spiking Neural Network (SNN).

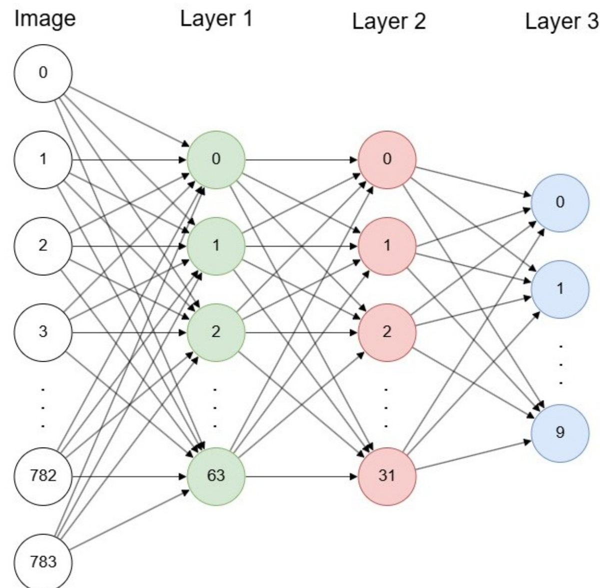
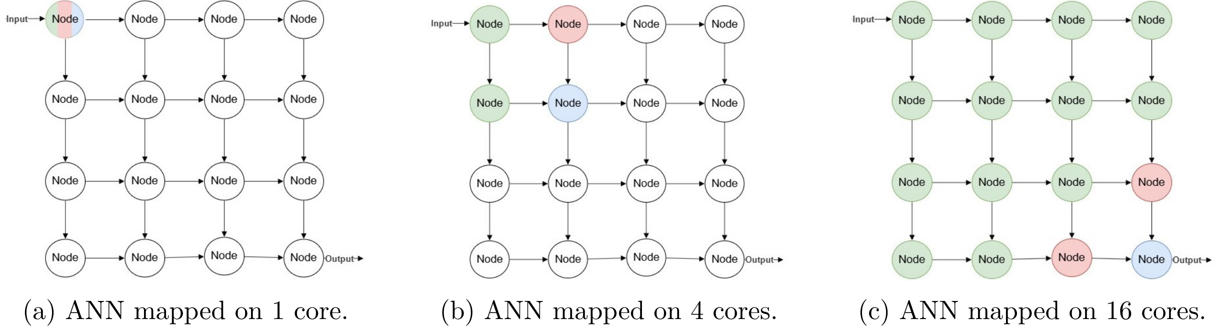


Figure 6.10: A fully connected MNIST ANN

The MNIST dataset contains 28×28 pixel grayscale images of handwritten digits, with each image representing one of the ten digits from 0 to 9. A fully connected neural network designed for this dataset is illustrated in Figure 6.10. In this network, the input layer receives the flattened 28×28 image (resulting in 784 input features). The first hidden layer consists of 64 neurons, followed by a second hidden layer with 32 neurons. Finally, the output layer has 10 neurons, each corresponding to one of the digit classes.

The described neural network is trained on a CPU in Python. The trained weights are converted to 16-bit fixed-point, with 1 sign bit, 8 integer bits, and 7 fractional bits (Q9.7). These weights are separated with manual mapping. The mapping is based on splitting neurons across multiple nodes, ensuring they fit within the data memory. To map the weights to the data memory, a folder is created for each node in the NoC. This folder contains the hexadecimal image file with initial data and the mapped weights. After the data memory files are generated, the input image is converted to the 16-bit format, and the Python code calculates the output based on an example image. This output is compared to the output of the NoC to verify the simulation.

To compare different mappings of the same neural network, three implementations are used (Figure 6.11). The first implementation involves mapping the entire neural network to a single core, as shown in Figure 6.11a. Then, the neural network is mapped to a 4-core. The first layer is split into two nodes, and the second and third layers each have their own core, as shown in Figure 6.11b. The final mapping in Figure 6.11c maps the first layer to 13 cores, the second



(a) ANN mapped on 1 core. (b) ANN mapped on 4 cores. (c) ANN mapped on 16 cores.

Figure 6.11: ANN mapped onto a 4x4 NoC

layer to two cores, and the final layer to one core. These tests are done on a 4-by-4 NoC, with a DMEM size of 512 KB and an IMEM size of 32 KB. This means the IMEM has one SRAM block, and the DMEM has 16 of them.

6.4.1 Power, Performance, Area (PPA) breakdown

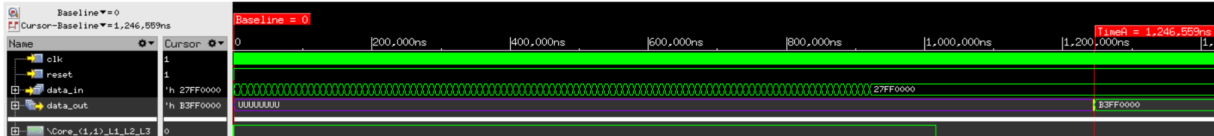
The standardized NoC is synthesized, and power analysis is done per mapping in Figure 6.11. The time taken and energy consumption are compared for the different mappings. Additionally, the node and NEORV32 are compared per node in the 4-by-4 mesh. In this comparison, the idle cores can be compared to the active cores.

Component	Area (μm^2)	Percentage (%)
pe_inst	692,858	99.76
local_fifo_to_pe	233	0.03
neorv32_inst	692,457	99.70
cpu_inst	6,363	0.92
dmem_inst (512KB)	646,135	93.03
imem_inst (32KB)	40,360	5.81
node_inst	1,670	0.24
five fifos	1,170	0.17
routing_logic	500	0.07
Total	694,528	100.0

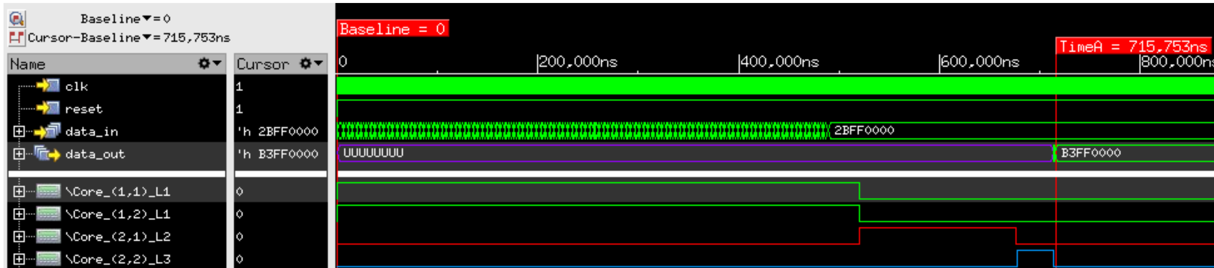
Table 6.5: Component Area and Percentage Breakdown of a Middle Node (with IMEM of 32KB, DMEM of 512KB, and one FIFO of 128 bits)

The synthesis of the standardized NoC implementation used for the event-driven neural network is shown in Table 6.5. The DMEM size is 512 KB, the IMEM is 32 KB, and a FIFO uses 128-bit memories. The NEORV32 CPU consists of all processes except the memory. The router area is approximately the same as the results in Figure 6.8, and is only 0.24% of the whole node. The area is dominated by the SRAMs, occupying 98.84% of the area. One SRAM block is initiated for the IMEM, and 16 SRAMs are used for the DMEM.

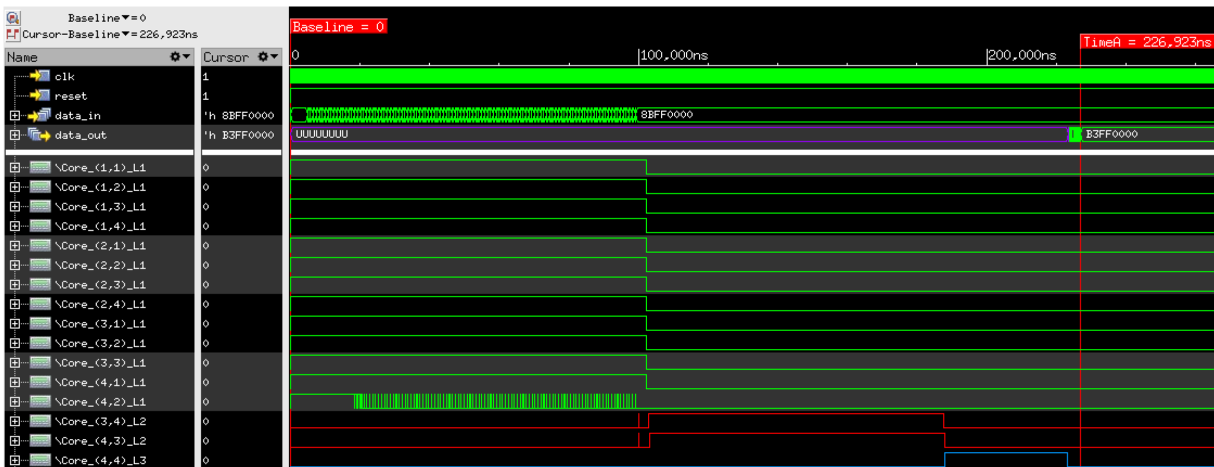
Figure 6.12 displays the waveforms of the three neural network mappings shown in Figure 6.11. These waveforms indicate the time a NEORV32 instance is active per core. The color and name show the layer to which a node is mapped. The active time is derived from the time the interrupt is active, which means a packet can be received from the FIFO by the NEORV32. The timing is an estimation that doesn't consider the output sending phase in the multicore implementation, as well as the calculation phase in the single-core implementation.



(a) 1 core



(b) 4 cores



(c) 16 cores

Figure 6.12: Waveform of NEORV32 activity of the mappings in Figure 6.11

Table 6.6 shows the energy consumption (in nanojoules) of all 16 cores in the 4-by-4 mesh of the three mappings. The red letters show which NoC nodes and NEORV32 instances are used. The idle power is different per corner, edge, and middle node. This difference arises from the number of FIFOs connected, as JOULES removes the FIFOs that are always connected to 0. The energy consumption of the NEORV32 instances is related to the estimated active time shown in Figure 6.12.

The single core mapping waveform in Figure 6.12a shows that only core (1, 1) is used; the other NEORV32 instances are in sleep mode. This behavior is reflected in Table 6.6, where only the (1, 1) core shows significant energy usage. The NoC nodes have a fixed energy consumption based on their location: corner, edge, or middle. Even nodes that route the output packets still consume this fixed energy, because the energy per packet is low, and the output packets are sparse. However, node (1, 1) is an exception. It consumes more energy because it remains active (stalled) for an extended period. When a router is stalled, its clock continues to run, resulting in increased energy consumption. This happens because the NEORV32 is slower than the NoC node, resulting in the router busy waiting for a longer time.

The four core mapping also shows this correlation between processing time and energy consumption. Figure 6.12c shows that core (1, 1) and (1, 2) are the most active cores, as

they process the first layer. In Table 6.6, these cores also consume the most energy. The routers that consume more energy are also the ones that are busy waiting for an extended period.

Node	**Figure 6.11a: 1 core (nJ)**		**Figure 6.11b: 4 cores (nJ)**		**Figure 6.11c: 16 cores (nJ)**	
	Node	NEORV32	Node	NEORV32	Node	NEORV32
(1, 1)	45	4481	26	2029	13	339
(1, 2)	25	210	14	1697	5	336
(1, 3)	25	210	14	122	7	338
(1, 4)	20	210	11	122	4	339
(2, 1)	25	210	41	626	6	340
(2, 2)	29	210	37	235	8	337
(2, 3)	29	210	16	122	14	345
(2, 4)	25	210	14	122	14	345
(3, 1)	25	210	14	122	8	337
(3, 2)	29	210	16	122	15	441
(3, 3)	29	210	16	122	14	349
(3, 4)	25	210	14	122	10	311
(4, 1)	20	210	11	122	4	340
(4, 2)	25	210	14	122	13	310
(4, 3)	25	210	14	122	11	310
(4, 4)	20	210	11	122	12	162

Table 6.6: Energy consumption comparison for different NoC configurations across nodes; red letters indicate the used processes.

Mapping	**Time (μs)**	**Active core time (μs)**	**Node Energy (μJ)**		**NEORV32 Energy (μJ)**		**Total Energy (μJ)**	
			J	**PG**	**J**	**PG**	**J**	**PG**
1 core (Figure 6.11a)	1,322	1,017	0.42	0.19	7.63	4.48	8.04	4.67
4 cores (Figure 6.11b)	715	1,232	0.28	0.17	6.06	4.59	6.33	4.76
16 cores (Figure 6.11c)	227	1,456	0.16	0.16	5.28	5.28	5.44	5.44

Table 6.7: Timing and energy per inference of the mapped event-driven ANN (Figure 6.11). Based on a DMEM of 512 KB and an IMEM of 32 KB. Energy values are shown for both Joules report (J) and power-gated (PG) conditions.

Table 6.7 shows the timing and total energy for the mappings shown in Figure 6.11. Two types of timing are reported: the total simulation time and the total time the NEORV32 instances are active. The energy is separated into two columns: the total Joules (J) result and a power-gated (PG) result. This power-gated result is the sum of the red processes (used processes) in Table 6.6. This power gating is not implemented, but it does show the active energy consumption per implementation.

As the number of cores increases, the computation of the ANN is distributed more effectively across the system, leading to a significant reduction in inference latency. Compared to sequential

execution on a single core, the 4-core mapping achieves a 45.9% speedup, while the 16-core mapping delivers an 83.1% speedup. However, the total active core time increases as the number of cores increases. This increase is due to the communication overhead.

With the increased communication overhead, one might expect a rise in NoC node energy, due to a higher volume of packet communication across the network. However, the results show the opposite: total energy consumption decreases as the number of cores increases. This reduction in energy is largely due to the decrease in inference time, which minimizes overall energy usage throughout the system. Another factor is the low energy cost per packet. Even though more packets are transmitted when additional cores are used, the data remains sparse. Therefore, the added communication overhead does not significantly impact the overall energy consumption, as the NoC contributes between 2.9% and 5.2% of the total power, depending on the mapping of the ANN.

The overall energy consumption of the NEORV32 instances also indicates that timing is a significant factor in the energy calculation. The Joules report indicates that the energy consumption for the NEORV32 cores decreases as the system becomes more parallel; however, the power-gated result shows the opposite. The Joules report and Table 6.6 indicate that the idle NEORV32 instances still consume a significant amount of energy. When these idle energy values are excluded (as in the power-gated scenario), the true communication overhead of using multiple NEORV32 cores becomes more visible. In other words, although parallelism reduces execution time, it introduces additional energy costs from NoC traffic, which are otherwise masked by the idle energy consumption of the non-power-gated NEORV32 cores.

6.4.2 Comparison with other NoCs

Intel's Loihi [15] reports the energy per tile hop as 3.0 pJ in the east and west directions, and 4.0 pJ in the north and south directions. The corresponding latencies per tile hop are 4.1 ns (east/west) and 6.5 ns (north/south). In comparison, the energy per packet in this implementation is shown in Table 6.4 to be 296 fJ (0.3 pJ). This NoC uses 10× less energy to deliver a packet than Loihi. However, as shown in Table 6.2, the latency without congestion is 8 ns, indicating that while this implementation is more energy-efficient, it is also slower.

Seneca [42] reports the area of the NoC relative to the entire core. This core includes 8 PEs, 22 kB of register-based memory, 2 Mb of data memory, and 128 kB of instruction memory, operating at 500 MHz in the GF-22 nm FDX technology node. The reported NoC area is 12.1 $k\mu m^2$. In comparison, the NoC area per core in this design is $1,670$ μm^2 (1.7 $k\mu m^2$), significantly smaller due to the simplistic architecture. Seneca's reported total energy per instruction for the NoC is 2 pJ, twice as efficient as Loihi. This design further reduces energy use, delivering a packet with 5× less energy than Seneca.

7 Discussion

In this thesis, an efficient Network on Chip (NoC) is designed for a neuromorphic processor. The NoC is designed for neural network data with backpressure, optimized packet flow, low power and area overhead, and scalability. The optimized packet flow is achieved by implementing the dynamic XY routing algorithm, which reduces latency while increasing throughput through path diversity. The optimal FIFO size was found to be 4, striking a balance between buffering capacity and area cost. For running the MNIST neural network, multiple NEORV32 instances are integrated with the NoC. The NEORV32 PE was chosen by the requirements of the overarching project called SparkRV. The NoC is open-source and available on GitHub; more information can be found in Appendix A. The GitHub repository includes two implementations of the NoC: one standalone version without the NEORV32, and one integrated with NEORV32 (without SRAM). The NoC can interface with any PE by connecting to two FIFOs—one for input and one for output.

7.1 Mapping

Mapping a neural network to the NoC requires a trade-off between core size and the number of cores. Dividing a network across multiple cores can reduce latency by enabling parallel processing; however, this parallelism introduces increased communication overhead. Subsection 6.4.1 shows the Power, Performance, Area (PPA) of three neural network mappings with an increase in core utilization. These results show that latency can be reduced, but at the cost of higher energy consumption, primarily due to the communication overhead introduced by the NEORV32 cores. The energy consumption correlates closely with the active duration of each process: cores that are actively computing or busy-waiting for long periods consume significantly more energy.

7.2 Routing algorithm

In this design, the dynamic XY routing algorithm is chosen because it resulted in an increase in throughput and a decrease in latency. This adaptive algorithm introduces potential issues for temporally coded SNNs. SNNs that employ time-to-first-spike or other latency-based coding schemes rely heavily on the precise timing and ordering of spikes. With XY-routing, packet delivery is predictable, ensuring that temporal relationships are preserved. In contrast, dynamic routing may cause packets to take different paths through the network, resulting in varying arrival times. This variation can disrupt temporal coding and affect the correctness of time-sensitive neural computations.

7.3 Head-of-Line (HOL) Blocking

With the congestion tests, HOL blocking was identified as a key issue, leading to lower throughput, increased latency, and, in the case of randomized data, deadlock. Although performance is negatively affected by HOL blocking, the area and power consumption remain favorable to alternatives. As briefly discussed in Section 4.4, this limitation is caused by the input FIFO design, which restricts packet routing under congestion. A potential solution to HOL blocking is the use of Virtual Vhannels (VCs). VCs mitigate HOL blocking by allowing multiple FIFOs on the same output direction, enabling parallel packet progression. However, implementing VCs introduces significant area and power overhead, which goes against the design goals of this NoC. This network is optimized for Power, Performance, Area (PPA) trade-offs, with a focus on low area and low power. Given that the target applications involve sparse, event-driven data, area and power efficiency are prioritized over peak throughput.

7.4 Deadlock

Deadlock is a concern in scenarios with random or recurring connectivity patterns. The routing algorithm with the flow control can form a cyclic dependency as shown in Subsection 6.3.2. Potential strategies to mitigate deadlocks include:

- A turn-based model routing, such as odd-even routing, to break routing cycles.
- The use of VCs, allowing multiple data streams to share the same output port without interfering with each other.
- Packet dropping or timeout mechanisms, which can be used to break deadlocks at the cost of data loss or retransmission complexity.

These solutions come with significant overhead in terms of area, logic complexity, and energy consumption. Given the design goal of minimizing area and power, such design choices are avoided. As a result, this NoC is best suited for event-driven, feedforward neural networks, where communication is sparse and predictable. More complex networks, such as Liquid State Machine (LSM) with random recurrent connectivity, are not well-suited to this architecture. These types of networks have been successfully mapped onto platforms like Loihi and SpiNNaker, which provide more robust support for dynamic routing and recurrent architectures.

8 Conclusion

This thesis presented the design and evaluation of a power- and area-efficient Network on Chip (NoC) architecture for a neuromorphic processor by answering the question:

"How can the Network on Chip (NoC) in a neuromorphic processor be optimized for efficient Power, Performance, Area (PPA) trade-offs while handling congestion and optimizing data transfer mechanisms?"

In contrast to current NoCs, which often includes complex features for efficient data processing, this NoC was kept simplistic to reduce overhead without becoming a performance bottleneck. The design requirements are defined to achieve efficient Power, Performance, Area (PPA) trade-offs, support congestion handling through backpressure, ensure optimized packet transfer, and allow scalability.

To meet these goals, several design choices were made:

- A 2D mesh topology was selected for scalability and path diversity, which helps reduce congestion and maintain a reasonable hop count.
- A ready and valid handshaking protocol was implemented for its simplicity and efficient use of buffer space.
- Store-and-forward packet switching was selected to minimize complexity.
- For routing, the sub-question (1) *"Which routing methodology is best suited to handle congestion in an efficient Network on Chip (NoC) for a neuromorphic processor?"* is answered. The XY and dynamic XY algorithms are compared, and the results indicate that the dynamic XY algorithm has improved throughput and reduced latency, which is implemented in this design.
- Multiple inputs arriving at the same time are managed using Round Robin (RR) scheduling to maintain fairness.
- An input First In, First Out (FIFO) buffer with a depth of four packets was determined to be optimal, striking a balance between area and throughput.

These design choices answer the sub-question (2) *"How should packetization in a low-overhead Network on Chip for a neuromorphic processor be optimized?"*. The use of store-and-forward switching with valid/ready flow control and fixed-size packets enables efficient and low-overhead data transfer.

The NoC is benchmarked without a PE connection to assess its performance in a fully congested state and determine if the NoC becomes a bottleneck in the neuromorphic processor. In this benchmark, a propagation delay of 4 clock cycles is observed, after which the throughput is

one packet per clock cycle in a non-congested system. As the NoC becomes more congested, the throughput suffers from HOL blocking. This results in a decrease in throughput and an increase in latency. With data flowing in all directions, a cyclic dependency is observed due to the combination of routing and backpressure flow control. This deadlock can be prevented with a well-designed mapping strategy that allows data to flow in one direction. When synthesizing the NoC, it is found that 70.2% of the area is occupied by the five FIFOs. The power analysis gives an energy per packet of 296 fJ, which is 10 times better than Loihi and 5 times more efficient than Seneca. The idle power is reduced by 71.4% by clock gating, and is dominated by the FIFOs.

(3) The PE interconnected using the Network on Chip (NoC) architecture is the NEORV32 RISC-V processor with an event-driven ANN, in accordance with the SparkRV project's requirements. The NEORV32's customizability allowed integration of DMEM, IMEM, and the XBUS for executing the neural network and communicating via the NoC through two FIFOs. The event-driven, fully connected MNIST ANN is implemented on a standardized NoC. This 4-by-4 NoC has a DMEM size of 512 KB and an IMEM size of 32 KB. With this NoC, three mappings are compared against each other to evaluate the performance and power consumption. By mapping an ANN to more cores, the execution time reduces with an 83.1% speed up when comparing a single core to a 16-core mapping. The energy consumption in the NoC node is also reduced due to the reduction in timing. The energy consumption of the active NEORV32 cores was found to be increasing, showing a communication overhead as the system becomes more parallel. To answer the last sub-question (4) The area of the NoC is $695k\mu m^2$, which is only 0.24% of the total area. The NoC contribution to the power is between 2.9%-5.2% of the total power, depending on the mapping of the ANN. The performance of the NoC for a neural network is shown not to be the bottleneck.

9 Future Works

This work presents a NoC optimized for low energy and area overhead in neuromorphic systems. Some design choices were outside the scope of this project due to timing constraints. The design choices that could improve the NoC are:

- Implementation of Globally Asynchronous Locally Synchronous (GALS), instead of synchronous FIFOs, asynchronous FIFOs can be used as a barrier between asynchronous and synchronous processes. GALS will improve on-chip scalability, as synchronous systems must distribute the clock signal through a clock tree, which can become impossible to design with increasingly large systems. To enable the scalability of large networks, GALS must be implemented.
- To increase scalability, relative routing can be implemented. The packet currently uses 6 bits to address the destination in the NoC. These 6 bits are 3 bits for the X destination and 3 bits for the Y destination, which means the current system can have a maximum mesh of 8-by-8. To address larger networks, relative routing can be added, which indicates the distance from the destination. An example, it needs +2 hops in the X direction and -2 hops in the Y direction.
- A different switching technique can be added using flits. For neural network data flit, data transfer is optimal as it can send multiple pieces of data within a single packet using a shared header and footer flit. Since a flit is a fragment of a packet, this approach can reduce the number of wires between cores, thereby improving scalability. However, this flit communication adds an overhead to handle such data in the routing logic.
- The manual mapped implementation can be suboptimal for this kind of communication. To improve performance, an automated mapping algorithm can be developed to optimize data communication within the NoC.

While the event-driven ANN proved to be sparse and followed the same packet structure, an Spiking Neural Network (SNN) can be implemented on the NoC to test the spiking packet data.

Bibliography

[1] Abderazek Ben Abdallah and Khanh N. Dang. *Neuromorphic Computing Principles and Organization*. Springer Cham, 2022. ISBN: 978-3-030-92525-3. DOI: 10.1007/978-3-030-92525-3. URL: https://doi.org/10.1007/978-3-030-92525-3.

[2] Babak Aghaei et al. “Network adapter architectures in network on chip: comprehensive literature review”. In: *Cluster Computing* 23.1 (Mar. 2020), pp. 321–346. ISSN: 1573-7543. DOI: 10.1007/s10586-019-02924-2. URL: https://doi.org/10.1007/s10586-019-02924-2.

[3] Filipp Akopyan et al. “TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.10 (2015), pp. 1537–1557. DOI: 10.1109/TCAD.2015.2474396.

[4] Isiaka A. Alimi et al. *Network-on-Chip*. Rijeka: IntechOpen, Apr. 2022. ISBN: 978-1-83968-158-5. DOI: 10.5772/intechopen.91110. URL: https://doi.org/10.5772/intechopen.91110.

[5] Isiaka A. Alimi et al. “Network-on-Chip Topologies: Potentials, Technical Challenges, Recent Advances and Research Direction”. In: *Network-on-Chip*. Ed. by Isiaka A. Alimi et al. Rijeka: IntechOpen, 2021. Chap. 3. DOI: 10.5772/intechopen.97262. URL: https://doi.org/10.5772/intechopen.97262.

[6] Arnon Amir et al. “A Low Power, Fully Event-Based Gesture Recognition System”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Accessed: 15-11-2024. 2017, pp. 7388–7397. DOI: 10.1109/CVPR.2017.781.

[7] Giuseppe Ascia et al. “Implementation and Analysis of a New Selection Strategy for Adaptive Routing in Networks-on-Chip”. In: *IEEE Transactions on Computers* 57.6 (2008), pp. 809–820. DOI: 10.1109/TC.2008.38.

[8] Asma Benmessaoud Gabis and Mouloud Koudil. “NoC routing protocols – objective-based classification”. In: *Journal of Systems Architecture* 66-67 (2016), pp. 14–32. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2016.04.011. URL: https://www.sciencedirect.com/science/article/pii/S1383762116300303.

[9] G. Q. Bi and M. M. Poo. “Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type”. In: *The Journal of Neuroscience* 18.24 (1998), pp. 10464–10472. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.18-24-10464.1998. URL: https://doi.org/10.1523/JNEUROSCI.18-24-10464.1998.

[10] Evgeny Bolotin et al. “Cost considerations in network on chip”. In: *Integration* 38.1 (2004), pp. 19–42. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2004.03.006. URL: https://www.sciencedirect.com/science/article/pii/S0167926004000343.

[11] Nicolas Brunel and Mark C. W. van Rossum. “Quantitative investigations of electrical nerve excitation treated as polarization”. In: *Biological Cybernetics* 97.5 (2007), pp. 341–349. ISSN: 1432-0770. DOI: 10.1007/s00422-007-0189-6. URL: https://doi.org/10.1007/s00422-007-0189-6.

[12] Cadence Design Systems, Inc. *Genus Synthesis Solution*. Cadence Design Systems, Inc. 2025. URL: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html.

[13] Cadence Design Systems, Inc. *Joules RTL Power Solution*. Cadence Design Systems, Inc. 2025. URL: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/power-analysis/joules-rtl-power-solution.html.

[14] Cadence Design Systems, Inc. *Xcelium Parallel Logic Simulation*. Cadence Design Systems. San Jose, CA, USA, 2024. URL: https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html.

[15] Mike Davies et al. “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning”. In: *IEEE Micro* 38.1 (2018), pp. 82–99. DOI: 10.1109/MM.2018.112130359.

[16] Li Deng. “The mnist database of handwritten digit images for machine learning research”. In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[17] Jason K. Eshraghian et al. “Training Spiking Neural Networks Using Lessons From Deep Learning”. In: *Proceedings of the IEEE* 111.9 (2023), pp. 1016–1054. DOI: 10.1109/JPROC.2023.3308088.

[18] Steve B. Furber et al. “Overview of the SpiNNaker System Architecture”. In: *IEEE Transactions on Computers* 62.12 (2013), pp. 2454–2467. DOI: 10.1109/TC.2012.142.

[19] Steve B. Furber et al. “The SpiNNaker Project”. In: *Proceedings of the IEEE* 102.5 (2014), pp. 652–665. DOI: 10.1109/JPROC.2014.2304638.

[20] Francesco Galluppi et al. “A hierachical configuration system for a massively parallel neural hardware platform”. In: *CF ’12 - Proceedings of the ACM Computing Frontiers Conference/CF - Proc. ACM Comput. Front. Conf.* ACM Computing Frontiers Conference, CF ’12 ; Conference date: 01-07-2012. United States: Association for Computing Machinery, 2012, pp. 183–192. ISBN: 9781450312158. DOI: 10.1145/2212908.2212934.

[21] Julian Göltz et al. *Fast and energy-efficient neuromorphic deep learning with first-spike times*. 2021. DOI: https://doi.org/10.1038/s42256-021-00388-x. arXiv: 1912.11443 [cs.NE]. URL: https://arxiv.org/abs/1912.11443.

[22] Wenzhe Guo et al. “Neural Coding in Spiking Neural Networks: A Comparative Study for Robust Neuromorphic Systems”. In: *Frontiers in Neuroscience* Volume 15 - 2021 (2021). ISSN: 1662-453X. DOI: 10.3389/fnins.2021.638474. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2021.638474.

[23] Garrett Hentges and Mostafa Abdelrehim. “Odd-Even Flexible Router for High Performance Network-on-Chips”. In: *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*. 2021, pp. 1817–1820. DOI: 10.1109/CSCI54926.2021.00343.

[24] Jake Hertz. *What is Clock Skew? Understanding Clock Skew and Clock Distribution Networks*. Oct. 2022. URL: https://www.allaboutcircuits.com/technical-articles/what-is-clock-skew-understanding-clock-skew-and-clock-distribution-networks/.

[25] A.L. Hodgkin and A.F. Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *Bulletin of Mathematical Biology* 52.1 (1990), pp. 25–71. ISSN: 0092-8240. DOI: https://doi.org/10.1016/S0092-8240(05)80004-7. URL: https://www.sciencedirect.com/science/article/pii/S0092824005800047.

[26] Sebastian Höppner et al. "The SpiNNaker 2 Processing Element Architecture for Hybrid Digital Neuromorphic Computing". In: (2022). DOI: 10.48550/arXiv.2103.08392. arXiv: 2103.08392 [cs.AR]. URL: https://doi.org/10.48550/arXiv.2103.08392.

[27] E.M. Izhikevich. "Simple model of spiking neurons". In: *IEEE Transactions on Neural Networks* 14.6 (2003), pp. 1569–1572. DOI: 10.1109/TNN.2003.820440.

[28] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. *On-Chip Networks: Second Edition*. 2nd. Morgan & Claypool Publishers, 2017. ISBN: 1627059148.

[29] Xin Jin et al. " Modeling Spiking Neural Networks on SpiNNaker ". In: *Computing in Science & Engineering* 12.05 (Sept. 2010), pp. 91–97. ISSN: 1558-366X. DOI: 10.1109/MCSE.2010.112. URL: https://doi.ieeecomputersociety.org/10.1109/MCSE.2010.112.

[30] Alakesh Kalita et al. "A topology for network-on-chip". In: *2016 International Conference on Information Communication and Embedded Systems (ICICES)*. 2016, pp. 1–7. DOI: 10.1109/ICICES.2016.7518838.

[31] Parviz Kermani and Leonard Kleinrock. "Virtual cut-through: A new computer communication switching technique". In: *Computer Networks (1976)* 3.4 (1979), pp. 267–286. ISSN: 0376-5075. DOI: https://doi.org/10.1016/0376-5075(79)90032-1. URL: https://www.sciencedirect.com/science/article/pii/0376507579900321.

[32] Ming Li, Qing-An Zeng, and Wen-Ben Jone. "DyXY - a proximity congestion-aware deadlock-free dynamic routing method for network on chip". In: *2006 43rd ACM/IEEE Design Automation Conference*. 2006, pp. 849–852. DOI: 10.1109/DAC.2006.229242.

[33] Shiming Li et al. "SNEAP: A Fast and Efficient Toolchain for Mapping Large-Scale Spiking Neural Network onto NoC-based Neuromorphic Platform". In: *Proceedings of the 2020 on Great Lakes Symposium on VLSI*. GLSVLSI '20. Virtual Event, China: Association for Computing Machinery, 2020, pp. 9–14. ISBN: 9781450379441. DOI: 10.1145/3386263.3406900. URL: https://doi.org/10.1145/3386263.3406900.

[34] Christian Mayr, Sebastian Hoeppner, and Steve Furber. *SpiNNaker 2: A 10 Million Core Processor System for Brain Simulation and Machine Learning*. 2019. DOI: 10.48550/arXiv.1911.02385. arXiv: 1911.02385 [cs.ET]. URL: https://doi.org/10.48550/arXiv.1911.02385.

[35] Emre O. Neftci, Hesham Mostafa, and Friedemann Zenke. "Surrogate Gradient Learning in Spiking Neural Networks: Bringing the Power of Gradient-Based Optimization to Spiking Neural Networks". In: *IEEE Signal Processing Magazine* 36.6 (2019), pp. 51–63. DOI: 10.1109/MSP.2019.2931595.

[36] Stephan Nolting and All The Awesome Contributors. *The NEORV32 RISC-V Processor*. Accessed: 15-11-2024. Apr. 2023. DOI: 10.5281/zenodo.13872735. URL: https://github.com/stnolting/neorv32.

[37] Luis A. Plana et al. "A GALS Infrastructure for a Massively Parallel Multiprocessor". In: *IEEE Design & Test of Computers* 24.5 (2007), pp. 454–463. DOI: 10.1109/MDT.2007.149.

[38] Anjana Ramachandran and M. Vinodhini. "An Optimized Buffer Architecture for Network on Chip Router". In: *Proceedings of International Conference on Data, Electronics and Computing*. Singapore: Springer Nature Singapore, 2024, pp. 149–161. ISBN: 978-981-97-6489-1.

[39] Bodo Rueckauer et al. “Conversion of Continuous-Valued Deep Networks to Efficient Event-Driven Networks for Image Classification”. In: *Frontiers in Neuroscience* 11 (2017), p. 682. ISSN: 1662-4548. DOI: 10.3389/fnins.2017.00682. URL: https://doi.org/10.3389/fnins.2017.00682.

[40] Pradip Kumar Sahu and Santanu Chattopadhyay. “A survey on application mapping strategies for Network-on-Chip design”. In: *Journal of Systems Architecture* 59.1 (2013), pp. 60–76. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2012.10.004. URL: https://www.sciencedirect.com/science/article/pii/S1383762112000902.

[41] Teresa Serrano-Gotarredona and Bernabé Linares-Barranco. “Poker-DVS and MNIST-DVS. Their History, How They Were Made, and Other Details”. In: *Frontiers in Neuroscience* 9 (2015), p. 481. ISSN: 1662-4548. DOI: 10.3389/fnins.2015.00481. URL: https://doi.org/10.3389/fnins.2015.00481.

[42] Guangzhi Tang et al. “SENECA: building a fully digital neuromorphic processor, design trade-offs and challenges”. In: *Frontiers in Neuroscience* 17 (2023). ISSN: 1662-453X. DOI: 10.3389/fnins.2023.1187252. URL: https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2023.1187252.

[43] The FOSSi Foundation. *Wishbone Interconnect Documentation*. Oct. 2019. URL: https://wishbone-interconnect.readthedocs.io/en/latest/index.html.

[44] Andrew Waterman et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1*. Tech. rep. UCB/EECS-2016-118. EECS Department, University of California, Berkeley, May 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html.

[45] Chao Xiao, Jihua Chen, and Lei Wang. “Optimal Mapping of Spiking Neural Network to Neuromorphic Hardware for Edge-AI”. In: *Sensors* 22.19 (2022). ISSN: 1424-8220. DOI: 10.3390/s22197248. URL: https://www.mdpi.com/1424-8220/22/19/7248.

[46] Wang Zhang et al. “Comparison Research between XY and Odd-Even Routing Algorithm of a 2-Dimension 3X3 Mesh Topology Network-on-Chip”. In: *2009 WRI Global Congress on Intelligent Systems*. Vol. 3. 2009, pp. 329–333. DOI: 10.1109/GCIS.2009.110.

A Github

The code is available open-source on GitHub:

`https://github.com/sharonmoo/efficient_noc_neuromorphic_processor.git`

To gain access to the full code, contact your supervisor. The SparkRV will use the described NoC with NEORV32. The README.md of the GitHub is displayed:

Efficient Network on Chip (NoC) for a neurmorphic processor

This repository consists of an efficient network on chip implementation described in a master thesis project. In this repository only the NoC without NEORV32 is shown. A Processing Element connection to the NEORV32 is shown in PE_with_NEORV.vhd, this file can be used as inspiration to connect the NoC to the NEORV32 project v1.10.5. In the NEORV32 instance a closed-source SRAM is used. The synthesis libraries are also removed from this projet due to an NDA.

Structure

In the folder RTL the following files can be found:

1. 2DMesh.vhd: Makes the 2D mesh in the system and connects signals between routers. For each Node in the mesh a router and PE instance is made.
2. definitions.vhd: The global variables of the system
3. FIFO_cg.vhd: A clock gated FIFO instance.
4. FIFO.vhd: A FIFO instance.
5. PE_with_NEORV.vhd: This PE initiates the NEORV32 instance with a FIFO. This FIFO is connected to the XBUS of the NEORV32.
6. PE.vhd: This PE does nothing to the data, but does initiate a FIFO. This FIFO is needed when connecting to the XBUS of the NEORV32. The FIFOs can be used to connect other RISC-V instances.
7. Three routers are made with the same processes:
    - First five FIFOs are initiated at every input direction (local/N/E/S/W).

- *valid_port_process*: A process which determines the valid ports based on the node potition.
- *fifo_management*: The Round-Robin scheduler is in this process.
- *routing_logic*: The routing logic, this is either a dynamic XY algorithm or a deterministic XY algorithm.
- *output_logic*: The output arbiter, which puts the routed packet to the right direction.

1. router_dyxy_cg.vhd: The clock gated dynamic XY implementation.
2. router_dyxy.vhd: The dynamic XY implementation.
3. router_xy.vhd: The XY routing implementation.

In the `sim` directory all the simulation files are added for Cadence:

1. file_lists: This directory contains all file lists used for different implementations, such as simulation of every routing algorithm or a clock gated implementation. And for each of these simulations a synthesis file is also added.
2. input_cg.tcl: Simulation tcl file of the clock gated implementation.
3. input.tcl: Simulation tcl file.

In the `testbenches` directory are all files related to testing the system:

1. all_avg_latency_hop.py: makes plots comparing XY and DYXY routing. Both of these implementations have to be simulated with run_sim.sh before this Python script can be ran.
2. fifo_occupancy.py: Makes a CSV file in "output_files/binned_fifo_{router_name}.csv", this is done after the simulation. This CSV file contains a binned FIFO occupancy for each FIFO in the Mesh.
3. gen_grid_data_single.py: Generates directional data in an "output_files/input_file.txt" file, used by the testbench to insert data. In this file a specified "-n number" of packets are generated for a single node: node (3,3).
4. gen_grid_data_sweep.py: Generates directional data in an an "output_files/input_file.txt" file, used by the testbench to insert data. In this file a specified "-n number" of clock cycles are generated for each insertion cycle. Here first one packet per clock cycles is inserted, then two packets per clock cycle etc. With a gap between packet insertion phase of the same number of clock cycles.
5. gen_grid_data.py: Generates directional data in an an "output_files/input_file.txt" file, used by the testbench to insert data. In this file a specified "-n number" of clock cycles attempts to insert a packet into all valid nodes.
6. process_output.py: Processes the packets inserted packets and maps them to the delivered packets in a CSV file with some additional information like latency, source, destination etc. This is also ran in run_sim.sh/
7. requirements.txt: The Python requirements for the virtual enviroment.

8. run_python.sh: Runs process_output.py, and fifo_occupancy.py for an implementation, xy, dyxy, dyxy_cg.

9. stalling_cores.py.sh: receive the timing of when a core stalls.

10. sweep_data.py: can be ran after a sweep test to receive plots about the tests.

11. tb_2DMesh.vhd: The testbench of the NoC. Here the input_file.txt is read and put on the input of the NoC each clock cycle. The testbench will attempt to insert a packet every clock cycle that is specified and write to an in_timing.txt file the actual time a packet is inserted and to missed_packets.txt the missed packets. Between testing the FIFO count is read for each FIFO in a fifo_count_file.txt file. Finally it will write the output to output_file.txt.

Running the NoC testbench

The testbench tb_2DMesh.vhd has been run in Cadence Xcelium. With the TCL files and file lists indide the sim folder. To run for example a sweep test:

1. generate the input_file.txt by running

```
python gen_grid_data_sweep.py -n 1000
```

2. Then run the prefered implementation with the right file list in Candence Xcelium.

3. Run the post simulation processing by running the following with a name like xy, dyxy or dyxy_cg:

```
run_python.sh dyxy_cg
```

4. Get graphs about the sweep data with

```
python sweep_data.py
```

B Round-Robin Input Port Scheduler

Algorithm 1 Round-Robin Input Port Scheduler

```
 1: for i = 0 to 4 do                        ▷ Round-robin based on the last server port
 2:     current_port ← (last_served + i + 1) mod 5
 3:     if FIFO_valid[current_port] = 1 then                  ▷ Read the selected FIFO
 4:         if current_port = 0 then
 5:             Enable LOCAL FIFO read
 6:         else if current_port = 1 then
 7:             Enable NORTH FIFO read
 8:         else if current_port = 2 then
 9:             Enable SOUTH FIFO read
10:         else if current_port = 3 then
11:             Enable EAST FIFO read
12:         else if current_port = 4 then
13:             Enable WEST FIFO read
14:         end if
15:         last_served ← current_port
16:         exit loop
17:     else
18:         last_served ← last_served
19:     end if
20: end for
```

C XY Routing Algorithm

Algorithm 2 XY Routing Algorithm

```
1: if current position = destination then
2:     select LOCAL
3: else if destination X > current X then
4:     select EAST
5: else if destination X < current X then
6:     select WEST
7: else if destination Y > current Y then
8:     select SOUTH
9: else if destination Y < current Y then
10:    select NORTH
11: end if
```

D Dynamic XY Routing Algorithm

Algorithm 3 DyXY Routing Algorithm (Pseudocode)

```
1: if current position = destination and LOCAL port is ready then
2:     select LOCAL
3: else                                  ▷ General case: try X direction first
4:     if destination X > current X and EAST port is ready then
5:         select EAST
6:     else if destination X < current X and WEST port is ready then
7:         select WEST
8:     else if destination Y > current Y and SOUTH port is ready then
9:         select SOUTH
10:    else if destination Y < current Y and NORTH port is ready then
11:        select NORTH
12:    else
13:        STALL
14:    end if
15: end if
```

E Inference code

```c
#include <neorv32.h>

// XBUS Base Address
#define XBUS_BASE_ADDR 0xF0000000
#define INPUT_FIFO_ADDR  (XBUS_BASE_ADDR + 0x1000)
#define OUTPUT_FIFO_ADDR (XBUS_BASE_ADDR + 0x2000)

#define INPUT_FIFO_VALID_ADDR  (XBUS_BASE_ADDR + 0x1100)
#define OUTPUT_FIFO_READY_ADDR (XBUS_BASE_ADDR + 0x2100)

#define STARTING_ADDR 0x80000400
#define MAX_DESTINATIONS 4

typedef struct {
    uint32_t x;
    uint32_t y;
} Coord;

typedef struct {
    int32_t input_size;
    int32_t output_size;
    int32_t X_coord;
    int32_t Y_coord;
    // The following fields exist only if input_size != 0
    int32_t neuron_loc;
    int32_t layer_id;
    int32_t weights[];
} Params;

// Processes the data and separate the source
static inline uint32_t process_output_data(uint32_t data, uint32_t neuron_id, uint32_t
↪  X_coord, uint32_t Y_coord, uint32_t dest_x, uint32_t dest_y) {
  neuron_id = (neuron_id & 0x03FF);
  data = (data & 0x0000FFFF);
  return (dest_x << 29) | (dest_y << 26) | (neuron_id << 16) | data;
}

int main()) {
  neorv32_rte_setup();
  neorv32_cpu_csr_set(CSR_MIE, (1 << CSR_MIE_MEIE));
  volatile Params *params = (volatile Params *)STARTING_ADDR;

  uint32_t input_size = params->input_size;
  uint32_t output_size = params->output_size;
  uint32_t X_coord = params->X_coord;
  uint32_t Y_coord = params->Y_coord;
```

```
46
47    if (input_size != 0) {
48      uint32_t weights_size = input_size * output_size;
49
50      volatile int32_t *weights = params->weights;
51      volatile int32_t *output = weights + weights_size;
52
53      uint32_t layer_id = params->layer_id;
54      uint32_t neuron_loc = params->neuron_loc;
55
56      uint32_t num_input_cores;
57      Coord dest_coords[MAX_DESTINATIONS];
58      int dest_count = 0;
59
60      switch (layer_id) {
61        case 1:
62          dest_coords[0] = (Coord){3, 4}; // 2,1
63          dest_coords[1] = (Coord){4, 3};
64          dest_count = 2; // 1
65          num_input_cores = 1;
66          break;
67        case 2:
68          dest_coords[0] = (Coord){4, 4}; //2,2
69          dest_count = 1;
70          num_input_cores = 13; // 2
71          break;
72        case 3:
73          dest_coords[0] = (Coord){5, 4};
74          dest_count = 1;
75          num_input_cores = 2; // 1
76          break;
77        default:
78          dest_coords[0] = (Coord){5, 4};
79          dest_count = 1;
80          num_input_cores = 1;
81          break;
82      }
83
84      // Initialize output buffer
85      for (int i = 0; i < output_size; i++) {
86        output[i] = 0;
87      }
88      int32_t neuron_id = 0, payload = 0, number=0;
89      uint8_t end = 0;
90      // Stream process input and compute NN output on-the-fly
91      do {
92        neorv32_cpu_sleep();
93        uint32_t in_val =  *(volatile uint32_t*)INPUT_FIFO_VALID_ADDR;
94        if (in_val == 1) {
95          int32_t value = *(volatile uint32_t*)INPUT_FIFO_ADDR;
96          payload = value & 0x0000FFFF;
97          neuron_id = (value >> 16) & 0x03FF;
98
99          if (neuron_id != 1023) {
100           for (int i = 0; i < output_size; i++) {
101             output[i] += payload * weights[neuron_id * output_size + i];
102           }
103         } else {
```

```
104          number++;
105          if (number == num_input_cores) {
106            end = 1;
107          };
108        }
109      }
110    } while (!end);
111
112    for (int i = 0; i < output_size; i++) {
113      output[i] = output[i] >> 7;
114      if (output[i] <= 0 && layer_id != 3) {
115        output[i] = 0;
116      } else {
117        for (int j = 0; j < dest_count; j++) {
118          do {
119            uint32_t data = process_output_data(output[i], (neuron_loc+i), X_coord,
             ↪  Y_coord, dest_coords[j].x, dest_coords[j].y);
120            xbus_write_fifo(data);
121          } while (*(volatile uint32_t*)OUTPUT_FIFO_READY_ADDR != 1) ;
122        }
123      }
124    }
125
126    // EOF
127    neuron_id = 1023;
128    uint32_t data;
129    for (int i = 0; i < dest_count; i++) {
130      do {
131        data = process_output_data(0, neuron_id, X_coord, Y_coord, dest_coords[i].x,
           ↪  dest_coords[i].y);
132        *(volatile uint32_t*)OUTPUT_FIFO_ADDR = data;
133      } while (*(volatile uint32_t*)OUTPUT_FIFO_READY_ADDR != 1);
134    }
135  }
136
137  neorv32_cpu_sleep();
138  return 0;
139 }
```

Listing E.0: The main.c code running on the NEORV32 PE

F Randomized Direction Data generation

```python
import random
import argparse
import os

def generate_random_grid(output_file, grid_size=4, data_bits=16, num_grids=10, max_entries=6):
    hex_chars_per_value = data_bits // 4  # Number of hex characters per data value
    grids = []

    for _ in range(num_grids):
        time_ns = 2
        grid_line = [["00000000" for _ in range(grid_size)] for _ in range(grid_size)]
        data_entries = grid_size*grid_size

        used_positions = set()

        for _ in range(data_entries):
            while True:
                source_x, source_y = random.randint(0, grid_size - 1), random.randint(0, grid_size - 1)
                if (source_x, source_y) not in used_positions:
                    used_positions.add((source_x, source_y))
                    break

            # Find valid upstream destinations (strictly above and to the left)
            possible_destinations = [
                (x, y)
                for x in range(source_x)
                for y in range(source_y)
            ]

            if not possible_destinations:
                continue  # No valid destination

            # Choose a random valid destination
            dest_x, dest_y = random.choice(possible_destinations)

            data = "".join(random.choices("0123456789ABCDEF", k=hex_chars_per_value))
            formatted_data = f"{dest_x:01X}{dest_y:01X}{data}{source_x:01X}{source_y:01X}"
            grid_line[source_x][source_y] = formatted_data

        # Format the output
        flattened_grid = "".join(f"{cell:>32}" for row in grid_line for cell in row)
        grids.append((time_ns, flattened_grid))

```

```
44     # Sort grids by time_ns
45     grids.sort()
46
47     with open(output_file, "w") as f:
48         for time_ns, flattened_grid in grids:
49             f.write(f"{time_ns} ns{flattened_grid}\n")
50
51 parser = argparse.ArgumentParser(description="Process command-line arguments.")
52 parser.add_argument("-n", "--number", required=True, help="Specify the number of lines
   ↪  to be created.")
53 num_grids = int(parser.parse_args().number)
54 print(num_grids)
55
56 # Define output file and parameters
57 output_file = "output_files/input_file.txt"
58 grid_size = 8
59 data_bits = 16  # 16-bit data values (4 hex characters)
60
61 # Generate the randomized grid file
62 os.makedirs(os.path.dirname(output_file), exist_ok=True)
63 generate_random_grid(output_file, grid_size, data_bits, num_grids)
64 print(f"Randomized grid file '{output_file}' has been generated.")
```

Listing F.0: Generation of the randomized directional grid data.