RISC-V Based Network on Chip Architecture for Spiking Neuron Processing

- Final Year Project Thesis -

Heshan Dissanayake
Buddhi Perera
Dinindu Thilakarathne

Department of Computer Engineering
University of Peradeniya

Final Year Project (courses CO421 & CO425) report submitted as a requirement of the degree of
B.Sc.Eng. in Computer Engineering

February 2023

Supervisors: Dr. Isuru Nawinne (University of Peradeniya), Dr. Isuru Dasanayake (University of Peradeniya), Dr. Mahanama Wickramasinghe (University of Peradeniya) and Prof. Roshan Ragel (University of Peradeniya)

I would like to dedicate this thesis to my loving parents and “teachers” . . .

Declaration

I/We hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my/our own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgments.

Heshan Dissanayake
Buddhi Perera
Dinindu Thilakarathne
February 2023

Table of contents

Chapter 1

Introduction

Computers that are built until today are mainly developed using the Von Neumann architecture where the inputs are processed and outputs are generated. Fig XX depicts a simple diagram of the Von Neumann architecture. In this Von Neumann architecture, the memory is a separate module and the communication between the processor and memory takes time. With the advancement of information technology in areas such artificial intelligence and machine learning, the computers based on this Von Neumann architecture consumed a large amount of energy. And when comparing these power hungry computers with the biological brain, the biological brain consumes only a very small amount of energy. For example IBM tried to simulate part of the cat's brain on a computer and it consumes 30 MW of power whereas the biological human brain consumes only about 20W of power on average. Due to these reasons, people tried to look into developing brain inspired architectures which are known as neuromorphic architectures. Generally in these architectures there are many processing elements which are similar to the neuron in the biological brain and these processing elements are connected using an interconnected network for the communications similar to the synapses in the biological brain. In this project, we are developing a scalable network-on-chip(NoC) architecture with RISC-V processors as processing nodes and we are implementing this on a FPGA. Then on top of this developed hardware architecture, we are simulating a spiking neuron model known as Izhikevich neuron model and optimizing this hardware to be power efficient by having an event driven messaging architecture.

Chapter 2

Related work

The following manuscript contains the related work that is being done previously.

- A Review on Neuromorphic Architecture Implementation.

A Review on Neuromorphic Architecture Implementation

Heshan Dissanayake
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
e16088@eng.pdn.ac.lk

Buddhi Perera
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
e16276@eng.pdn.ac.lk

Dinindu Thilakarathna
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
e16366@eng.pdn.ac.lk

Dr. Isuru Nawinne
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
isurunawinne@eng.pdn.ac.lk

Dr. Isuru Dasanayake
Department of Electrical & Electronic Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
isurud@ee.pdn.ac.lk

Dr. Mahanama Wickramasinghe
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
mahanamaw@eng.pdn.ac.lk

Prof. Roshan Ragel
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
roshanr@eng.pdn.ac.lk

***Abstract*— Neuromorphic processors are based on an architecture that is inspired by the biological brain that uses neurons and synapses as the basic building blocks. Researchers have made various discoveries with the hope of exploiting the huge parallelism and the energy efficiency of spiking neural dynamics of biological brains. This is a review that tries to state the background and the prior implementations of the neuromorphic architecture that could be developed on top of highly parallelized network on chip architectures with von Neumann processing cores.**

Keywords*— neuromorphic computing, spiking neurons, neuron models, synapse models, network on chip, SpiNNaker, DYNAPs, ODIN, Tinsel, POETS, TrueNorth

I. Introduction

Most of the computing devices developed until today are based on the von Neumann architecture which was defined by John von Neumann in 1945. Fig.1 shows a simple diagram of the von Neumann architecture where the inputs are provided into the central processing unit(CPU) which contains the arithmetic and logic unit(ALU) and the control unit. The data processed by the CPU are sent to outputs and the CPU will communicate with the memory when required. With the development of technologies in areas like artificial intelligence machine learning, people found out that the computers based on the von Neumann architecture are not efficient in power consumption. Due to this reason, many researchers focused on understanding the way that the human brain operates and does the processing.

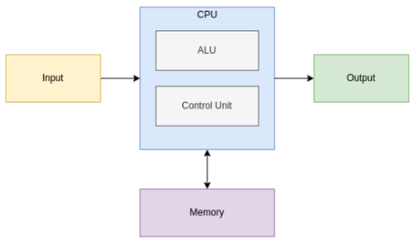


Figure 1: Simple diagram of Von Neumann Architecture

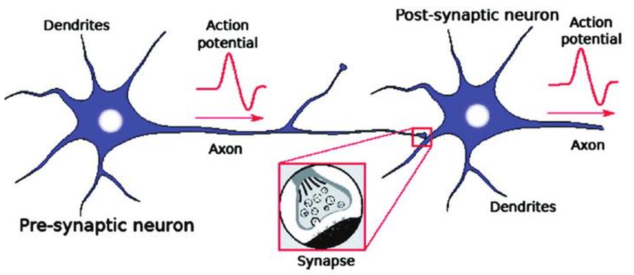


Figure 2: Connection between two neurons

When comparing today's computers with the human brain, there are significant differences. The main difference is the organizational structure of the two. As discussed earlier, today's computers are based on the von Neumann architecture whereas the human brain is based on the neuron synapse structure which is different to the von Neumann architecture. Human brain consists of a vast number of simple biological components known as Neurons. These neurons consist of dendrites, axons and the cell body. Dendrites transfer inputs to the neuron in the form of electrical impulses. Axons are responsible for the transfer output from that neuron to other neurons. Areas where the electrical or chemical signals interchange between neurons are known as the synapses. Fig. 2 shows the connection between 2 neurons and the way the signals are being transferred from one neuron to another.

Another key difference between today's computer and the human brain is the power consumption. On average the human brain consumes about 20W of power [1]. Power consumption of today's computers to simulate brain-like systems is very high. For example, an IBM's supercomputer was used to do simulations the size of a cat's brain and it consumed 2.9MW of power [2]. Processing capability is another key difference when comparing the two. Human brain can perform a vast number of real time computations when compared with von Neumann computers.

Due to these significant differences researchers got motivated to develop architectures that mimicked or inspired by the brain's architecture. These type architectures were known as 'neuromorphic architectures'. In these architectures there are a vast number of simple processing elements which are analogous to the neurons in the brain and these processing elements are densely connected to each other which is similar to the synapses in the brain. The first known effort made to develop a neuromorphic system was done in 1990 by Carver Mead [3]. He tried to develop this system with VLSI and analog components.

Researchers expect performance and efficiency enhancements in computations compared to traditional computing by using the neuromorphic architectures [4]. Low power consumption is the main goal to move towards neuromorphic architectures. Unlike the traditional computers, neuromorphic architectures are event driven. This is the main reason for the low power usage. Parallelism and speed of computations is also another gain in neuromorphic computing. Due to the vast number of processing elements and dense interconnections, these architectures are capable of parallel processing and thereby there is a significant performance enhancement in real time computing. Unlike the von Neumann architecture, in neuromorphic architectures the processing element will contain a memory component which will overcome the von Neumann bottleneck.

The goal of this paper is to discuss the models related to neuromorphic computing architectures, concepts related to network on chip and to review the implementations of network on chip based neuromorphic architectures.

II. Models

A. *Neuron Models*

When designing a neuromorphic system the biological aspect of a neuron is considered in different levels. Basically most of the neuron models have the idea of accumulating charge that is induced by other neurons and emit a charge as the output which affects other neurons. The models that have been implemented so far can be categorized as following.

1) *Biologically plausible:* These models mimic the behaviour of the biological neurons as it is. Hodgkin-Huxley model[5] and Morris-Lecar model[6] are biologically plausible models.

2) *Biologically inspired:* These models implement the behaviour of a biological neuron, but not as extreme as the Biologically plausible way. Fitzhugh-Nagumo model[7], Izhikevich model[8], [9] and Mihalas-Niebur model[10] are biologically inspired models.

3) *Detailed Neuron:* The neuron models that consider the other components such as axons, dendrites and glial cells which are not usually much considered in other neural models.

4) *Integrate and Fire:* These models are a subcategory of Biologically inspired models. Integrate and fire model[11], leaky integrate-and-fire model[12], quadratic integrate-and-fire model[13], exponential integrate-and-fire model[14] and adaptive exponential integrate-and-fire model[15] belongs to this category.

5) *McCulloch-Pitts:* Neuron models that are inspired by the original McCulloch-Pitts neuron [16]. Most of the artificial neuron models are based on this.

B. *Synapse Models*

Synapse models focus on the way that neuron connections are implemented along with the neuron models. These synapse models also can be divided as biologically inspired models which are designed for transmitting spikes and synapse models for artificial neuron models. Most of the time the synapse models stay simple but there can be some complex implementations such as plasticity mechanisms which change the strength of synaptic connections with time.

C. *Network Models*

Network models describe the way neurons and synapses connect and interact with each other.

III. Network on Chip

Network on chip(NoC) is a hardware architecture for communication between the nodes in an integrated circuit. Nodes in NoC can be any module that can send and receive messages such as processing cores, memory controllers, caches. A router based packet switching communication mechanism is used for the communication between the nodes in a NoC and the mechanism is similar to the computer communication networks. Each node is connected to a router via a network interface and connections between routers make the NoC architecture. A simple NoC with 16 nodes arranged in a mesh topology is shown in Fig 3. Network topology, flow control mechanism, routing and arbitration mechanisms are the main considerations when designing NoC[17].

A. *Network Topology*

Network topology describes the arrangement of the nodes in the network on chip. Important measurements in a topology are the bisection bandwidth and the diameter. Bisection bandwidth is the number of links that need to be disconnected in order to divide the network into two equal parts. This is a measure of the path diversity between the nodes and the congestion tolerance. Diameter is the maximum optimal distance between any two pairs of nodes. When selecting a network topology, high bisection bandwidth and low diameter is preferred[18]. Fig 4 shows some of the network topologies.

Chain and ring topologies have low bisection bandwidth leading to less path diversity. Fat tree topology has a hierarchical arrangement and the links get wider towards the root. Mesh and torus topologies are commonly used in NoCs as they have relatively high bisection bandwidth leading to high path diversity, low diameter and ease of implementation. Hypercube and butterfly topologies are high radix topologies. They have high path diversity and low diameter but it is difficult to implement on NoC architectures.

B. *Flow Control Mechanism*

Performance of a network on chip is directly affected by the efficiency of the flow control mechanism. Efficiency of the flow control mechanism can be improved by the use of buffers in the transmission and buffered flow control techniques are used in NoCs. Messages transmitted in the

NoC are divided into packets and the packets are divided into flits. Therefore buffer flow control mechanisms can be further categorized into packet buffer flow control and flit buffer flow control. Flit level buffering is preferred over the packet level buffering due to better storage utilization when buffering.

C. *Packet buffer flow control*

In packet buffer flow control, the buffering takes place at the level of packets. Store and forward flow control and virtual cut through flow control are packet buffer flow control mechanisms. In store and forward flow control, a node will store the incoming packet in the buffer and after ensuring that the entire packet is received, the packet is forward to the next node. In virtual cut through flow control, the node will forward the packet only if the receiving node has enough buffer space to receive the packet. In these packet buffer flow control mechanisms it is required to have large buffers to store the packets.

D. *Flit buffer flow control*

In flit buffer flow control, the buffering takes place at the level of flits. Wormhole flow control and virtual channel flow control are flit buffer flow control mechanisms. In wormhole flow control it is not required to have the space reserved for the entire packet. If the receiving node has enough space to buffer a flit, then the sending node will send a flit. This method of flow control can cause head of the line blocking issues as buffers are first in first out(FIFOs) buffers and the receiving buffer is blocked by the flits to be transmitted to the next node. Fig 5 shows the 3 stage NoC router architecture with wormhole flow control. Virtual channel flow control is a solution head of the line issue in wormhole flow control. In this method a virtual channel is allocated for the transmission. Fig 6 shows a 4 stage NoC router architecture with virtual channel flow control. Flit buffer flow control methods are preferred over packet flow control methods due to better storage utilization in buffers.

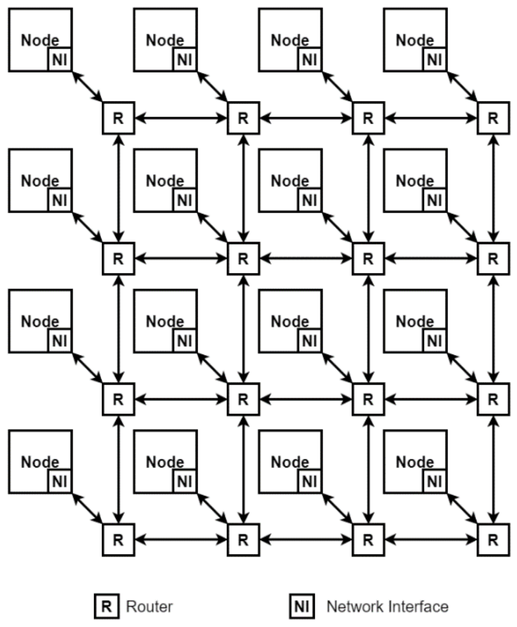


Figure 3: Network of Toroidal Structure

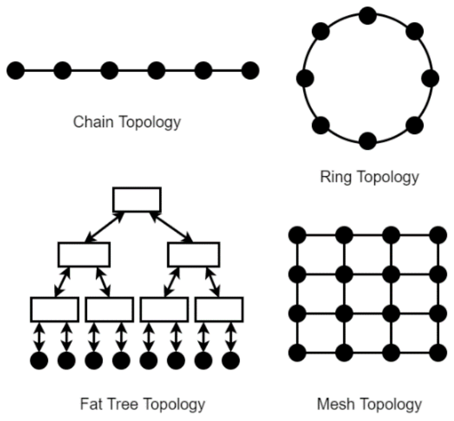


Figure 4: Network of Toroidal Structure

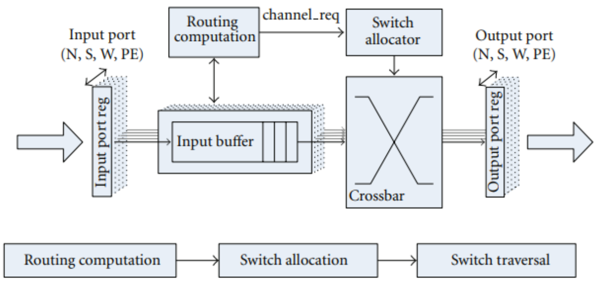


Figure 5: Network of Toroidal Structure

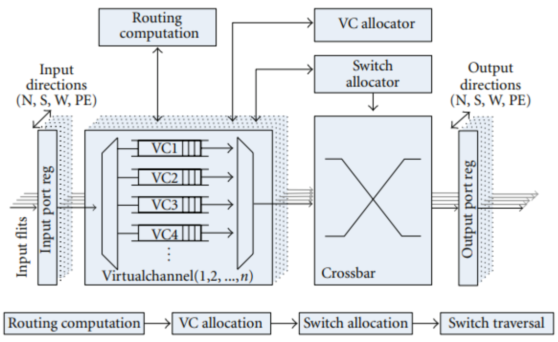


Figure 6: Network of Toroidal Structure

E. *Routing and Arbitration*

Routing and arbitration also affects the performance and the congestion of messages in the NoC. Routing is deciding the path of the packet/flit to be forwarded and arbitration is deciding which packet request to be serviced when there are more than one packet.

Common routing techniques in mesh NoC architectures are dimension ordered routing, obvious routing and adaptive routing. In dimension ordered routing, the 2D mesh is considered as an XY coordinate plane and the routing is done accordingly. This method is also known as X-Y routing and it has fixed paths leading to no path diversity. In obvious routing, the routing algorithm selects an intermediate node at random and the packet is routed to the destination via the selected intermediate node. This method can handle congestion due to the path diversity. Adaptive routing uses the concept of turn models[19] and has high path diversity and has the congestion avoidance capability.

IV. IMPLEMENTATIONS

A. *SpiNNaker System*

Spinnaker [20] is a huge multi core network that is capable of simulating 1 billion neurons. This network consists of 57000 nodes where each node is a processor with 18 ARM968 cores and one internal router to handle the communication between internal core and external network of nodes. The silicon die of the spinnaker node is shown in the Fig. 7. The network of nodes are arranged in a way of toroidal structure (Fig. 8) to make efficient point to point, near neighbour and fixed route communications. This kind of communication is essential for spiking neural systems which rely on event based communications.

The final system will consist of 1,036,800 ARM9 cores and 7 Tera Bytes RAM distributed all over the 57000 nodes. The whole system is built from 1200 PCB boards that consist of 48 nodes. The system is designed to support ethernet networks which have been used to communicate with the software level. They have used PyNN [21] as the framework for network modelling. rely on event based communications.

Since the system they have proposed is massively parallel the communication between the nodes should be reliable and efficient. They have configured their network in a toroidal structure. Another way of representing this topology is shown in the Fig 9. A single node in this network consists of a system that has 18 compute cores and self-timed NoC to handle the communication inside that node and outside with the huge network. The all cores inside of a single chip(node) share the external SDRAM through the NoC. The internal and external communications are based on a packet switching asynchronous NoC where it uses multicast routing. In spiking neural networks firing of a presynaptic neuron will affect all the connected postsynaptic neurons. In spinnaker there could be up to 1000 neurons that connected to a single neuron. Therefore, most of the time one to many communications is required. That’s the main reason they have used multicast routing in their system. A specific data packet is multicast when a neuron is fired. This packet contains which neuron is fired and when it is fired. This concept is called “Address Event Representation" [22] which comes from the event based asynchronous spikes that are generated by neurons. The packet structure is shown in Fig 10. The size of the packet solely depends on the number of neurons in the system since it carries the neuron address. The address encoding is much more efficient since it only requires log2N number of bits for N number of neurons. In this certain project it only needs 32 bits to represent 4 billion of neurons. When the transmitted packet only has the address of the source neuron and the time, the routing mechanism should know where to send a specific packet by only looking at the address. This process is done by the router using a lookup table. where for each address it looks up for a routing word. In each of those routing words it contains 1 bit for each destination which represents whether it should be transmitted or not. This lookup table will be inefficiently long if some optimizations were not done. In spinnaker they have done two optimizations to reduce the length of the lookup table. One of them is using groups of neurons that have associated inputs and outputs where the neuron in each group shares the same destination and can be represented by a single routing entry [23]. Other optimization is using default routes for unmatched route entries.

Also they have implemented a protocol called emergency routing which is used to transmit packets in an alternative route when the assigned route is congested. A counter for each packet counts the clocks until they get transmitted. When this counter hits threshold the packet will be sent in an emergency route.

The spinnaker system is implemented with globally asynchronous locally synchronous (GALS) method [24]. This means the internal routing will be done synchronously and external routing will be done asynchronously. This is done due to several reasons. One is, since the neuron spikes are event driven and asynchronous it is energy efficient to have off chip communications asynchronous. The second is, it is easy to scale up when separate chips decoupled from each other's clock. The third is, since a neuron will fan-out up to 1000 it is difficult to maintain high bandwidth data buses synchronously.

A single node with a chip will dissipate around 1W and with other components a board with 48 chips will dissipate around 75 W. The full system with 57000 chips will consume around 90kW of power. The system is capable of performing 2;200 MIPS/W. Which takes a considerable place in the Green500 supercomputer rankings.

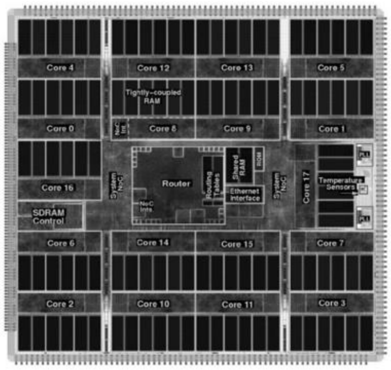


Figure 7: Silicon die of the spinnaker node

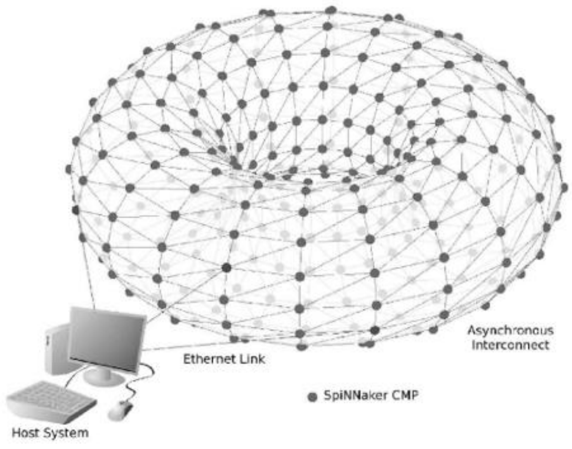


Figure 8: Network of Toroidal Structure

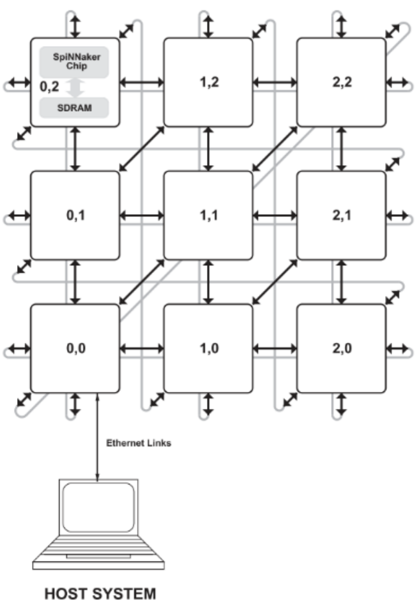


Figure 9: 2D Network structure of SpiNNaker

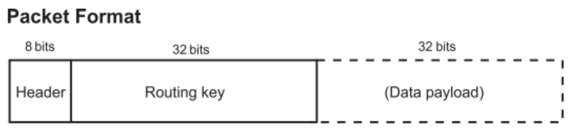


Figure 10: Network packet format

B. *DYNAPs System*

DYNAPs [25] is a system that consists of a multicore neuromorphic processor chip that takes the advantage of analog and digital hybrid circuits to emulate the synapse and neuron behaviour in a spiking neural network. The address event representation [26] communication protocol makes the event-based communication much more efficient. Also the routing between cores in the multicore system is done with a hierarchical router system.

In this paper they have proposed a routing scheme that is much more efficient in memory. The big network of neurons are divided into clusters that consist of groups of neurons. This clustering can make the routing more efficient in memory wise. In the Fig 11 it is shown that the overview of the routing structure. In the left side it is shown the N number of neurons, and in the right side it is shown the clusters that are formed by groups of neurons. But on both sides it represents the same set of N neurons. The nodes that can be seen in the middle are the ones that are used to broadcast a message from a single neuron to a cluster of destination neurons. There will be N/C numbers of nodes where the C stands for number of neurons in a cluster. In a single cluster there are M numbers of neurons that subscribed to a specific neuron. That means when a specific neuron emits a packet with its tag M numbers of neurons in a cluster will accept that incoming tag. In a single cluster there will be up to K numbers of unique tags will be used. In their analysis it is shown that larger clusters and less number of tags leads to low memory usage. But on the other hand, less number of tags reduces the routing flexibility. However, this effect can be overcome by increasing the number of clusters in the system. Also the number of neurons that are subscribed to a specific neuron which is denoted by M, gives the trade that should be done between the point to point communication and the broadcast that happens between a node and a cluster. Therefore it is stated that the optimal choice for minimum memory utilization can be taken by the equation(1). The stands for the ratio between K and C as K/C.

$$M^* = \sqrt{\frac{F \log_2 \alpha N}{\alpha \log_2 \alpha C}} \tag{1}$$

The router system architecture of this paper is proposed as “mixed mode hierarchical mesh routing architecture”. The memory optimized routing scheme that was discussed earlier is implemented in this hierarchical architecture. The clusters of the above scheme are assigned to single cores in the hardware architecture. Also the nodes that broadcast messages to the clusters are implemented by asynchronous routers. Asynchronous Content addressable memory blocks are used to store the tags. Mesh routing schemes require low bandwidth but they have higher latency. On the other hand hierarchical routing schemes have low latency but it has high bandwidth requirement[27]. Therefore in this implementation they have decided to use a mix of both modes. The hierarchy of the routers goes for three levels. The lowest level of the routers R1 handles the local traffic. The events will be sent to the local core of that router or will be transferred to the next level R2 routers. Those events that are sent to the local core will be broadcasted among the all nodes of the local core. These events will be processed with the CAM blocks of all neurons by checking a match between tags. With a successful match the node will accept that certain event. The second level routers R2 have bidirectional channels that can be used to communicate with local cores and with the upper level routers. Unlike other routers, R2 routers span over two levels. This is helpful to handle the complexity of the network. This structure can be seen in the Fig. 12. The level 3 routers R3 give the ability to the system to transmit messages over large distances. A relative 2D mesh also known as XY algorithm is used as the routing strategy in this router level.

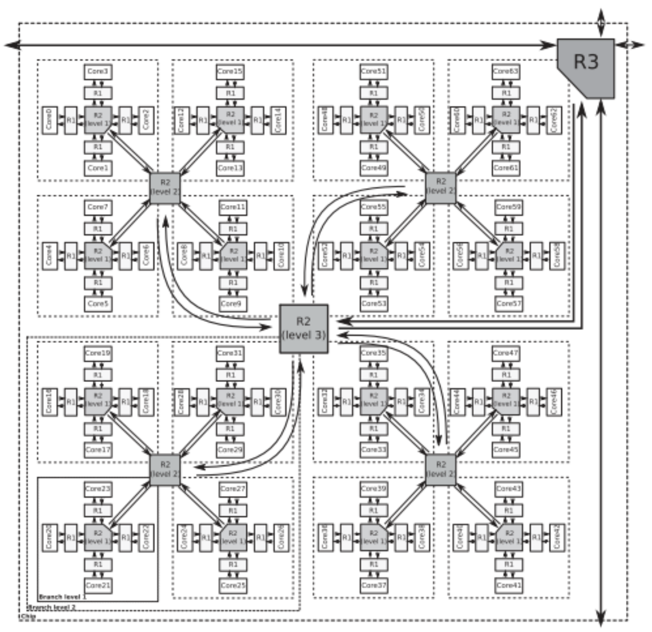


Figure 11: Hierarchical router system

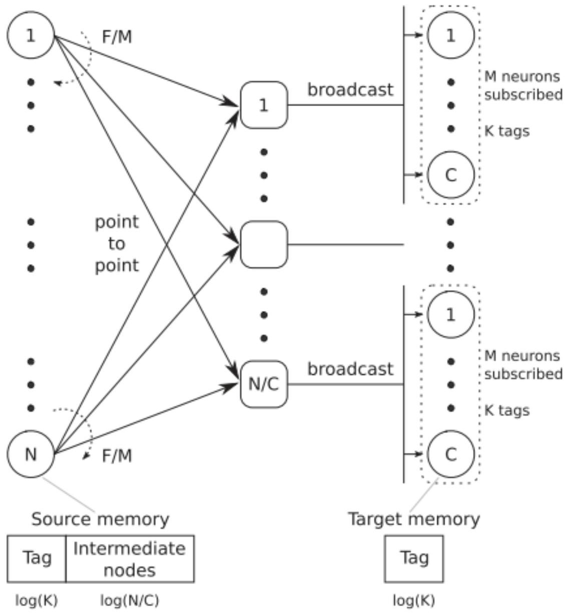


Figure 12: Hierarchical router system

They have fabricated a chip that consists of 4 cores and its hierarchical routers R1, R2 and R3s. Each of those cores can facilitate 256 neuros with 4k fanout. The SRAM and the asynchronous CAM memory cells are distributed over the chip. This chip is fabricated in standard 180nm and the whole chip has an area of 43 mm2. To perform specific tasks, a PCB with 9 of those chips were fabricated. Also a FPGA is added to the system to handle the inter-chip communication.

C. ODIN System

ODIN [28] is a system that focuses on implementing a Spiking neural network solution for embedded end devices that can be used in IOT implementations. This approach is meant to make the end IOT devices capable of running spiking neural network tasks by itself without an aid form cloud or any other external computation devices. The implementation consists of a RISC-V based system on chip(SoC). The design architecture uses the SPI communication protocol and the RISC-V core to offload the spiking neural network task to the ODIN. The architecture of the ODIN spiking neural network implementation is shown in Fig 13.

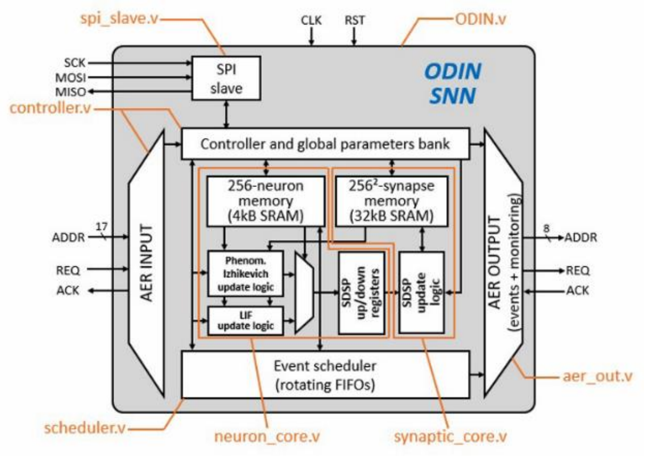


Figure 13: The architecture of the ODIN spiking neural network implementation

ODIN stands for Online-learning Digital spiking Neuromorphic processor. Synaptic and neuronal computations can be carried out with the dedicated hardware inside the core. A single core is capable of managing 256 neuronal and 64k all to all synaptic connections. Other than the SoC that communicates with the ODIN through SPI, it can communicate with other external modules by means of Address Event representation (AER). Two separate hardware modules for input and output handle those communications. The system supports two neuron models. One is the leaky integrate and fire model. And the other is a custom model that was inspired by the Izhikevich model. The controller is the unit that administers all the other units. It is a Moore Finite State Machine, where the outputs will be decided by the current state. This FSM has 13 states to define the functionality. The neuron and synapse memory are stored in SRAM, where 4kB reserved for 256 neurons and 32kB for 64000 of synapses. The generated spikes are transmitted as packets. The packets contain the index of the neuron that generates the spike, number of spikes and the inter spike intervals (ISI). To represent the index of the neuron 8 bits were used in the packet. For both spike number and ISI only 3 bits are reserved and the whole data packet is 14 bits.

The neuron core is the one that is responsible for maintaining the state of the neurons. It consists of the sub modules to compute the LIF and Izhikevich updates as required. The neuron core has a 4kB of SRAM to store neuron data which is 256 16-byte wide data words. The memory can be accessed by a 8 bit neuron address and 4 bit address to locate byte form 16 byte word.

The synaptic core maintains the levels of the synapses. This module gives and receives information with the neuron core. Also, the synaptic weights will be used to do the relevant calculations. When forming the memory structure, the synaptic address is formed by concatenating the post and presynaptic neuron addresses. 8 bits from the presynaptic neuron address and 5 most significant bits of the postsynaptic neuron address form a 13-bit address to locate 4byte words in the memory. The remaining 3 least significant bits address each byte of the 4-byte word.

This is synthesized in the Xilinx PYNQ Z2 board. It was required to remove some parts in the original design in order to synthesize it on the PYNQ board. The utilization of the Xilinx PYNQ Z2 was low where only 15% of LUTs and 11% of BRAM was used.

D. Tinsel System

Tinsel is an FPGA optimized RISC-V core that supports hyperthreading. This system was mainly developed to reduce the memory bottleneck when implementing spiking neural networks. This system is highly modularized and highly scalable. Several Tinsel cores and a few other units are in a single tile. And these tiles are arranged in an NoC (Network on Chip). This NoC design supports synchronous messaging protocol as well as event-based asynchronous messaging protocol. There are several FGAs containing this kind of structure. The interconnections between FPGAs utilize the existing IO (Input Output) ports in the FPGAs. This RISC-V core is based on the RV32IMF instruction set. This core consists of a 6-stage pipeline structure. The stages are scheduled, fetch, decode, execute, writeback, and resume (Fig. 14). In this pipeline structure at most one instruction is present in every stage so the pipeline consists of 6 instructions at a time. This pipeline is designed in such a way that pipeline hazards like data hazards and control hazards are minimal. Threads are scheduled in the scheduling stage. If the instructions are delayed in the execution stage they get suspended and resumed again in the resume stage. This implementation uses two queues to store ready to execute instructions in order to reduce the delays in writeback stages because the writeback instructions can simultaneously issue from execute and resume stages.

All the tinsel cores do not support Floating point operations. There is only one Floating point unit inside a tile. A Tile is a small scalable unit in this system. In the default setup, this tile consists of 4 tinsel cores with 16 threads each and an FPU (Floating Point Unit), a MailBox, and a data cache. This mailbox is connected to other tiles with an NoC (Network on Chip). To utilize the off-chip RAM (Random Access Memory) and to reduce the traffic on the NoC each tile connected to the RAM separately without going through the NoC. So, the data memory space is divided among the tiles without overlapping.

The NoC structure is a grid layout and every router is connected to the mailbox of a router. The creators have used separate on-chip networks for message-passing and off-chip memory access. This is implemented this way to reduce the complexity of the NoC. They have ensured there are no deadlocks in asynchronous message passing. This was achieved by ensuring that the threads are always ready to receive and never block on send operation. The routers are dimension-ordered and connected to a 2D mesh network. This network supports bidirectional half rate FIFOS. A low-performance 8-bit bus connects all the Tinsel cores and this network gives tinsel a virtual UART. This bus provides a non-blocking function to get and put bytes to the core. The USB JTAG is connected with the host server to give the debug capability to the whole system (Fig. 15).

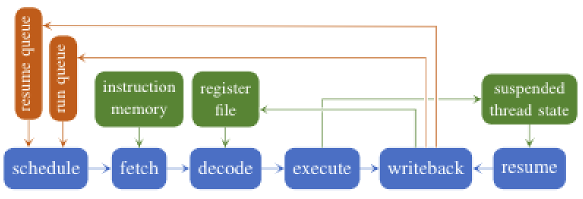


Figure14: 6-stage pipeline of Tinsel core

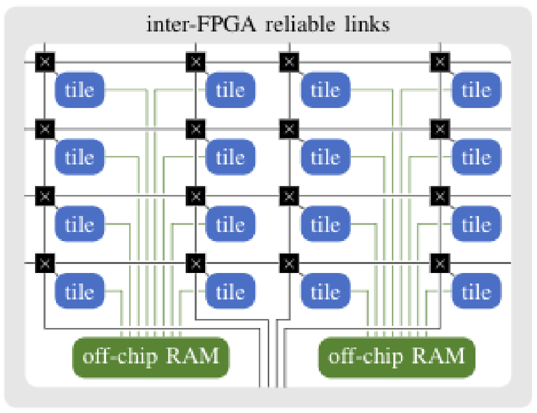


Figure 15: NoC of the Tinsel system

They have compared their FPGA cluster with Xeon cluster. This performance is relative to Blogel performance on a single 12-threaded Xeon machine which consumes around 100W when busy. But the FPGA cluster only consumed around 50W of power when busy.

E. POETS System

The POETS system is a parallel cluster architecture made for Spiking Neural Networks. This system mainly provides the software layer and the hardware extensions to simulate Spiking Neural Networks based on Tinsel hardware. Applications running on the POETS should be converted to Graphs first. The vertices represent the computational units and the edges represent the communication links connected among them. Those communication links are the links sending or receiving messages from other cores. This graph is cut into several segments and put them in tiles then the edges show the communication links between them and estimate how to handle those messages in the NoC (Fig. 16). The POETS compiler is the one responsible for the above-mentioned task. The only things which have to store in off chip memory are the Synaptic Weights and the Neuron Parameters.

The POETS system supports two Spiking Neural Network Models and they are Leaky Integrate and fire model and the Izhikevich model. Table 1 contains the parameters used for testing the system.

This project provides hardware support for an event-driven parallel programming model. POETS system contains an x86 computer system and that system connects to several FPGAs running tinsel cores. The Fig. 17 describes the structure of the POETS box.

The researchers were able to map 50 to 500,000 neurons to a POETS box. And they were able to map about 4 million neurons in 8 boxes. Of these 4 million, about 800,000 neurons are inhibitory neurons. Speed is the most important parameter that is evaluated in this platform.

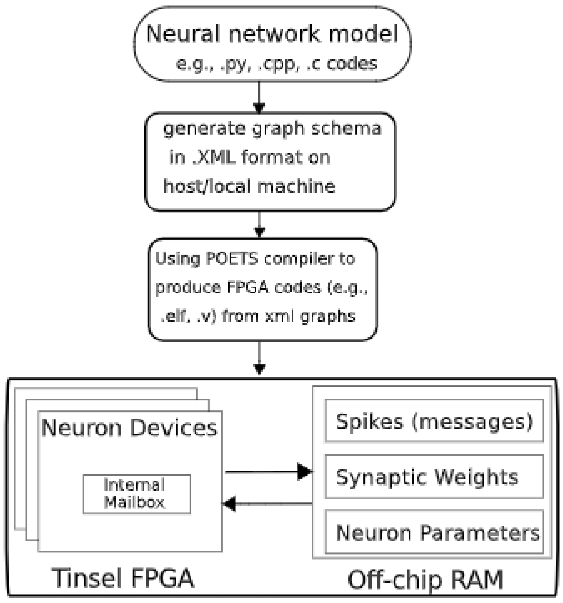

Neural network model
e.g., .py, .cpp, .c codes
generate graph schema in .XML format on host/local machine
Using POETS compiler to produce FPGA codes (e.g., .elf, .v) from xml graphs
Neuron Devices
Internal Mailbox
Tinsel FPGA
Spikes (messages)
Synaptic Weights
Neuron Parameters
Off-chip RAM
Figure 16: Model conversion procedure if the POETS box

TABLE I. PARAMETERS USED FOR TESTING THE SYSTEM

Sub System	Parameter	Default Value
Core	Threads per core	16
Core	Bytes per inst mem	16384
Core	Cores per FPU	4
Core	Cores per DCache	4
Core	Cores per Mailbox	4
Cache	DCaches per DRAM	7
Cache	Bytes per Beat	32
Cache	Beats per Line	1
Cache	DCache Sets per Thread	4
Cache	DCache num ways	8
NoC	Mailbox mesh X length	4
NoC	Mailbox mesh Y length	4
NoC	Bytes per flit	16
NoC	Max flits per msg	4
MailBox	Msg slots per thread	16

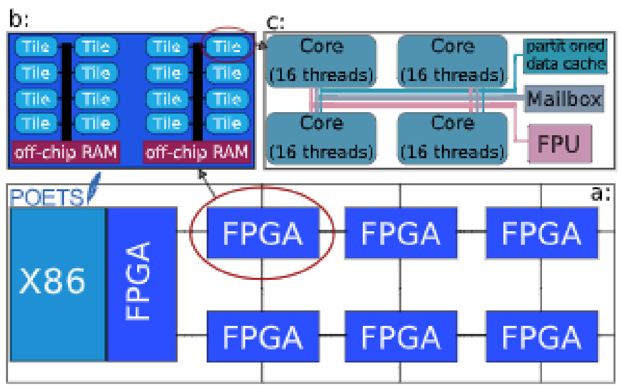


Figure 17: Structure of the POETS box

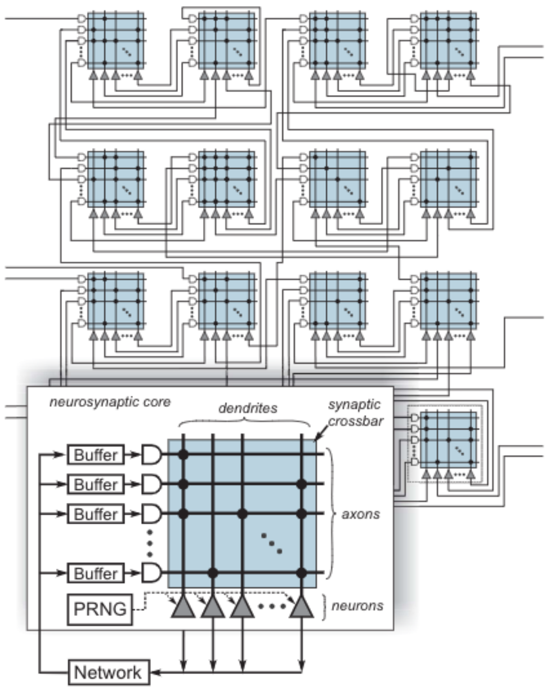


Figure 18: Structure of the TrueNorth System

F. *TrueNorth System*

TrueNorth is a network of cores interconnected in a brain-inspired method. This core is a lightweight neurosynaptic structure. Each of these cores contains memory (“synapses"), processors (“neurons”), and communication (“axons”) in close proximity. This NoC has two types of connections.

1) *Gray matter:* short range connections with an intra-core crossbar memory.

2) *White matter:* long range connections through inter-core spike based message-passing networks.

This system is fully configurable in terms of both the “physiology” and “anatomy” of the chip. This TrueNorth neurosynaptic core contains 256 axons, 256x256 synapse crossbar and 256 neurons. Fig.18 shows the structure of the system.

The leaky integrate and fire model has five steps.

1) Synaptic integration
2) Leak integration
3) Threshold
4) Spike firing
5) Reset

Equation (2) describes the synaptic integration, equation (3) describes the leaky integration and (4) describes the threshold, fire and reset mathematical model of the neuron used in the TrueNorth system.

$$V_j(t) = V_j(t-1) + \sum_{i=0}^{N-1} x_i(t)s_i \qquad (2)$$

$$V_j(t) = V_j(t) - \lambda_j \qquad (3)$$

If $V_j(t) > \alpha_j$
Spike
$V_j(t) = R_j$ (4)
Endif

For the jth neuron in tth timestep, the membrane potential $V_j(t)$ is the sum of the membrane potential in the previous timestep $V_j(t-1)$ and the synaptic input. For each of the N synapses, the synaptic input is the sum of the spike input to the synapse $x_i(t)$ at the current timestep, multiplied by the signed synaptic weight s_i. Following integration, the LIF neuron model abstract the leak value j from the membrane potential. With linear leak, this constant is subtracted every timestep, regardless of membrane potential of the synaptic activity. This operation serves as a constant bias on the neural dynamics. Then, the LIF neuron model compares the membrane potential at the current timestep $V_j(t)$ with the neuron threshold j. If the membrane potential is greater than or equal to the threshold voltage, the neuron fires a spike and resets its membrane potential. Most of the time the reset voltage R_j is zero.

When comparing TrueNorth system with above discussed systems it has few key differences. This TrueNorth system core is very basic core. It only supports addition and subtraction and few conditional operations. This system is not a generic system like the above-mentioned systems. This is specifically designed for the Leaky Integrate and Fire (LIF) Model. This model is not using and off-chip RAM like other systems.

TABLE II. COMPARISION OF THE SYSTEMS

Model / Properties	**TrueNorth**	**SpiNNaker**	**DYNAPs**	**ODIN**	**POETS**
Feature Size	2nm	130nm	180nm	-	28nm
Chips	16	48	9	1	48
Power	3.2W	80W	-	-	42.8W
Interconnect	2D mesh-unicast	2D mesh-unicast	Mix-mode 2D mesh	-	2D mesh-unicast
Neuron Model	Configureable LIF	Programmable	LIF	LIF/ Izhikevich	LIF/ Izhikevich
Neurons	16M	768k	9k	256	4M
Synapses	4G	768M	36M	65k	4G

V. CONCLUSION

In this review we have given an overview of the neuromorphic architecture for spiking neural networks. And discussed three types of neuromorphic architecture models. Then we have discussed six implementations of neuromorphic computing platforms which are SpiNNaker, DYNAPs, ODIN, Tinsel, POETS and TrueNorth. Since the network on chip concept plays a major role in this type of highly parallel computer systems, we have discussed some important concepts of NoCs. A comparison between five of the above stated implementations was done to give an overall idea of each system compared to the others. The goal of this paper is to provide readers with some implementations on neuromorphic architectures and give a brief introduction on neuromorphic computing and introduce a few already implemented systems.

REFERENCES

[1] D. Alejandro and V. Jimenez, “Principles of neural science”.

[2] M. Hennecke, W. Frings, W. Homberg, A. Zitz, M. Knobloch, and H. Böttiger, “Comput Sci Res Dev Measuring power consumption on IBM Blue Gene/P”, doi: 10.1007/s00450-011-0192-y.

[3] C. Mead, “Neuromorphic Electronic Systems,” *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1629–1636, 1990, doi: 10.1109/5.58356.

[4] C. D. Schuman *et al.*, “A Survey of Neuromorphic Computing and Neural Networks in Hardware,” May 2017, [Online]. Available: http://arxiv.org/abs/1705.06963

[5] A. L. Hodgkin and A. F. Huxley, “A quantitative description of membrane current and its application to conduction and excitation in nerve,” *J Physiol*, vol. 117, no. 4, p. 500, Aug. 1952, doi: 10.1113/JPHYSIOL.1952.SP004764.

[6] C. Morris and H. Lecar, “Voltage oscillations in the barnacle giant muscle fiber,” *Biophys J*, vol. 35, no. 1, pp. 193–213, Jul. 1981, doi: 10.1016/S0006-3495(81)84782-0.

[7] R. FitzHugh, “Impulses and Physiological States in Theoretical Models of Nerve Membrane,” *Biophys J*, vol. 1, no. 6, pp. 445–466, Jul. 1961, doi: 10.1016/S0006-3495(61)86902-6.

[8] E. M. Izhikevich, “Which model to use for cortical spiking neurons?,” *IEEE Trans Neural Netw*, vol. 15, no. 5, pp. 1063–1070, Sep. 2004, doi: 10.1109/TNN.2004.832719.

[9] E. M. Izhikevich, “Simple model of spiking neurons,” *IEEE Trans Neural Netw*, vol. 14, no. 6, pp. 1569–1572, Nov. 2003, doi: 10.1109/TNN.2003.820440.

[10] Ş. Mihalaş and E. Niebur, “A generalized linear integrate-and-fire neural model produces diverse spiking behaviors,” *Neural Comput*, vol. 21, no. 3, pp. 704–718, 2009.

[11] L. F. Abbott, “Lapicque’s introduction of the integrate-and-fire model neuron (1907),” *Brain Res Bull*, vol. 50, no. 5–6, pp. 303–304, 1999.

[12] R. B. Stein, “A Theoretical Analysis of Neuronal Variability,” *Biophys J*, vol. 5, no. 2, pp. 173–194, Mar. 1965, doi: 10.1016/S0006-3495(65)86709-1.

[13] G. B. Ermentrout and N. Kopell, “Parabolic Bursting in an Excitable System Coupled with a Slow

Oscillation," *http://dx.doi.org/10.1137/0146017*, vol. 46, no. 2, pp. 233–253, Jul. 2006, doi: 10.1137/0146017.

[14] N. Fourcaud-Trocmé, D. Hansel, C. van Vreeswijk, and N. Brunel, "How Spike Generation Mechanisms Determine the Neuronal Response to Fluctuating Inputs," *Journal of Neuroscience*, vol. 23, no. 37, pp. 11628–11640, Dec. 2003, doi: 10.1523/JNEUROSCI.23-37-11628.2003.

[15] R. Brette and W. Gerstner, "Adaptive exponential integrate-and-fire model as an effective description of neuronal activity," *J Neurophysiol*, vol. 94, no. 5, pp. 3637–3642, Nov. 2005, doi: 10.1152/JN.00686.2005.

[16] D. M. Dubois, "Hyperincursive McCulloch and Pitts neurons for designing a computing flip-flop memory," *AIP Conf Proc*, vol. 465, no. 1, p. 3, Mar. 2008, doi: 10.1063/1.58256.

[17] W.-C. Tsai, Y.-C. Lan, Y.-H. Hu, and S.-J. Chen, "Networks on Chips: Structure and Design Methodologies," *Journal of Electrical and Computer Engineering*, vol. 2012, p. 509465, 2012, doi: 10.1155/2012/509465.

[18] S. R. Sarangi, *Advanced Computer Architecture*, 1st edition. McGrawHill.

[19] E. Fusella and A. Cilardo, "Understanding turn models for adaptive routing: The modular approach," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 1477–1480. doi: 10.23919/DATE.2018.8342245.

[20] S. B. Furber *et al.*, "Overview of the SpiNNaker system architecture," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2454–2467, 2013, doi: 10.1109/TC.2012.142.

[21] A. P. Davison *et al.*, "PyNN: A common interface for neuronal network simulators," *Front Neuroinform*, vol. 2, no. JAN, p. 11, Jan. 2009, doi: 10.3389/NEURO.11.011.2008/BIBTEX.

[22] M. A. Sivilotti, "Wiring considerations in analog VLSI systems, with application to field-programmable networks," 1991, doi: 10.7907/STJ4-KH72.

[23] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[24] L. A. Plana *et al.*, "A GALS infrastructure for a massively parallel multiprocessor," *IEEE Design and Test of Computers*, vol. 24, no. 5, pp. 454–463, Sep. 2007, doi: 10.1109/MDT.2007.149.

[25] S. Moradi, N. Qiao, F. Stefanini, and G. Indiveri, "A Scalable Multicore Architecture with Heterogeneous Memory Structures for Dynamic Neuromorphic Asynchronous Processors (DYNAPs)," *IEEE Trans Biomed Circuits Syst*, vol. 12, no. 1, pp. 106–122, 2018, doi: 10.1109/TBCAS.2017.2759700.

[26] K. A. Boahen, "Point-to-point connectivity between neuromorphic chips using Point-to-point connectivity between neuromorphic chips using address events address events," vol. 47, no. 5, pp. 416–434, 2000.

[27] B. V. Benjamin *et al.*, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 699–716, 2014, doi: 10.1109/JPROC.2014.2313565.

[28] G. Urgese, E. Forno, and A. Spitale, "POLITECNICO DI TORINO Interfacing a Neuromorphic Coprocessor with a RISC-V Architecture."

Chapter 3

Methodology

We have divided the methodology and the implementation into 2 sections. The first section is the design, development and testing of the processing element which is the RISC-V core. Then in the second section we have the design and development of the NoC architecture and the simulation of spiking neuron application. Following 2 articles are of the sections.

- RV32IMF five stage pipeline implementation with interrupt and random number generation units.
- RISC-V Based Network on Chip Architecture for Spiking Neuron Processing.

RV32IMF FIVE STAGE PIPELINE IMPLEMENTATION WITH INTERRUPT AND RANDOM NUMBER GENARATION UNITS

Heshan Dissanayake
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
e16276@eng.pdn.ac.lk

Buddhi Perera
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
e16088@eng.pdn.ac.lk

Dinindu Thilakarathna
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
e16366@eng.pdn.ac.lk

Dr. Isuru Nawinne
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
isurunawinne@eng.pdn.ac.lk

Dr. Mahanama Wickramasinghe
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
mahanamaw@eng.pdn.ac.lk

***Abstract*— Five stage pipeline implementation of RV32IMF implementation with additional Interrupt control unit and random number generation unit. This implementation was tested with altera DE2 and DE5 boards.**

***Keywords*— RISCV, RV32IMF, Interrupts, Floating-point, ALU, Cache, Memory**

I. Introduction

In this project we have implemented five stage pipeline CPU according to the RISC V architecture[1], [2]. This architecture is RV32IMF. This means this architecture is based on 32bit data path and this supports Integer operations, multiplication operations and the floating-point operations[3]. We have additionally added an Interrupt controller and Random number generation unit. This CPU was written in Verilog and tested in Altera DE2 and DE5 boards. This code base was synthesized using Quartus Prime FPGA design software. Implementation details were discussed below in detail.

II. MODULES

A. ALU – Arithmatic and Logic Unit

This ALU supports 18 operations. We decided to take the shifting unit inside the ALU and also we have moved the branching unit outside the ALU. According to our design the CPU gets two 32 bit operands as inputs to the ALU. Also, it takes a 5 bit select signal to the ALU to select the operation. And its output is a single 32 bit result. Following tables describes the operation name and there functionalities. Floating point unit was taken from [3]

TABLE 1: INTEGER OPERATIONS

ADD	Addition
SUB	Subtraction
SLL	Shift Left Logical
SLT	Set Larger Than
SLTU	Set Larger Than Unsigned
XOR	XOR Operation
SRL	Shift Right Logical
SRA	Shift Right Arithmetic
OR	OR Operation
AND	AND Operation
MUL	Multiplication
MULH	Return upper 32 bits of result of the Multiplication (signed x signed)
MULHSU	Return upper 32 bits of result of the Multiplication (signed x unsigned)
MULHU	Return upper 32 bits of result of the Multiplication (unsigned x unsigned)
DIV	Division
REM	Signed remainder of integer division
REMU	Unsigned remainder of integer division
FWD	Additional Instruction built to support other instructions (Not included in RV32IM)

TABLE 2: FLOATING POINT OPERATIONS

FADD	Addition of two floating point numbers
FSUB	Subtraction of two floating point numbers

FMUL	Multiplication of two floating point numbers
FDIV	Division of two floating point numbers
FSQRT	Calculate the square root of a floating point number
FEQ	Set the equality of two floating point numbers
FLT	Set Less Than two floating point numbers
FLE	Set Less Than or equal two floating point numbers

TABLE 3: ALU CONTROL SIGNALS

OP Code	**Funct3**	**Funct7**	**operation**	**ALU Select**
0110011	000	0000000	ADD	000000
0110011	000	0100000	SUB	010000
0110011	001	0000000	SLL	000001
0110011	010	0000000	SLT	000010
0110011	011	0000000	SLTU	000011
0110011	100	0000000	XOR	000100
0110011	101	0000000	SRL	000101
0110011	101	0100000	SRA	010101
0110011	110	0000000	OR	000110
0110011	111	0000000	AND	000111
0110011	000	0000001	MUL	001000
0110011	001	0000001	MULH	001001
0110011	010	0000001	MULHSU	001010
0110011	011	0000001	MULHU	001011
0110011	100	0000001	DIV	001100
0110011	101	0000001	REM	001101
0110011	111	0000001	REMU	001111
1010011	-	0000000	FADD	100000
1010011	-	0000100	FSUB	100001
1010011	-	0001000	FMUL	100010
1010011	-	0001100	FDIV	100011
1010011	-	0101100	FSQRT	100111
1010011	010	1010000	FEQ	100110
1010011	001	1010000	FLT	100101
1010011	000	1010000	FLE	100100
Special			FWD	011xxx

B. *Register File*

Register file contains 32 numbers of 32 bit registers. This register file can get two 5 bit addresses as ADDR1 and ADDR2. Then this register file can The 32 bit data corresponding to the previous addresses. The second functionality of the register file is to take a 5 bit WRITE_ADDR and 32 bit WRITE_DATA and write the given data into the register file when the WRITE_EN signal gets activated.

32nd register (R31) is reserved for the random number generator. A new random number will generate and available in this register. Any which will store in this register will be replaced by this random number.

31st register (R30) is reserved to store the current programmed counter before going to the Interrupt Service Routine. Any user data which will store in this register will replaced by the PC automatically.

C. *Control Unit*

Control unit is responsible for the generating control signals to alter Datapath and change the functions of functional units like ALU and memory. Following signals are the main control signals of the control unit.

• Immediate select signal	**3 bit**
• Branch select signal	**4 bit**
• ALU select signal	**6 bit**
• Operand 1 & 2 select signal	**1 bit**
• Main memory: write	**3 bits**
• Main memory: read	**4 bits**
• Register file write select	**2 bits**
• Register file write enable	**1 bit**

Immediate select signal [2:0]

This signal is used to control the **immediate select and sign extender unit.** In the RV32IM instruction set there are 6 types of immediate positions that can be seen in the instructions. Therefore the extraction of the immediate value and sign extension should be done according to the corresponding type. Those identified types are shown below.

Type 1 - **LUI** and **AUIPC**

imm[31:12]	rd	0110111

LUI

imm[31:12]	rd	00010111

AUIPC

Type 2 - **JAL**

imm[20|10:1|11|19:12]	rd	1101111

JAL

Type 3 - **JALR, LB ..., ADDI ect…**

imm[11:0]	rs1	000	rd	0000011

LB

Type 4 - **Conditional branching Instructions**

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	0000011

BEQ

Type 5 - **SB, SH, SW**

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011

SB

Type 6 - **SLLI, SRLI, SRAI**

0000000	shamt	rs1	001	rd	0010011

SLLI

When sign extending it should be considered whether the instruction used sign or unsigned immediately. Therefore the MSB of the immediate control signal will be logic 1 if the instruction is an unsigned one. And the lower 3 bits will select the immediate type.

TABLE 4: IMMEDIATE SIGNAL CODES

	Immediate Select [3:0]
TYPE 1	u000
TYPE 2	u001
TYPE 3	u010
TYPE 4	u011
TYPE 5	u100
TYPE 6	u101

u = 1 :unsigned immediate

u = 0 :signed immediate

TABLE 5: IMMEDIATE SIGNAL CODES

JAL	1010
JALR	1010
BEQ	1000
BNE	1001
BLT	1100
BGE	1101
BLTU	1110
BGEU	1111

Branch select signal [3:0]

This is the control signal sent to the branch select module. This module requires a control signal, to specify what kind of branch should be performed. There are 2 unconditional and 6 conditional branch instructions. Therefore we have used 3 bits to specify the type of the branch and one bit (MSB) to enable the branch unit. Therefore all together 4 bits will be used for the branch select signal. The 3 bits that are used for the beach type can be directly taken from the finc3 field of the instruction only for conditional branches. Table 5 shows the corresponding signals.

ALU select signal [5:0]

The ALU signal will define the required ALU operation. There are 18 operations required to be defined, therefore we have used 5 bits for the ALU select signal. Table 3 shows ALU select signals.

Operand 1 and 2 select signals

In the operand 1 of the ALU, it required selecting between PC and register file data 1. Therefore a one bit was used to control this mux.

In the operand 2 of the ALU, it required selecting between immediate and register file data 2. Therefore a one bit was used to control this mux.

TABLE 6: OPARAND 1 SIGNALS

oparand 1 MUX	
1	PC
0	REG_DATA1

TABLE 7: OPARAND 2 SIGNALS

oparand 2 MUX	
1	immediate
0	REG_DATA2

TABLE 8: MAIN MEMORY WRITE SIGNALS

op-code	funct3		Main mem write [2:0]
0100011	000	SB	100
0100011	001	SH	101
0100011	010	SW	110

TABLE 9: MAIN MEMORY READ SIGNALS

op-code	funct3		Main mem read [3:0]
0000011	000	LB	1000
0000011	001	LH	1001
0000011	010	LW	1010
0000011	100	LBU	1100
0000011	101	LHU	1101

Main memory: write signal [2:0]

The MSB of the memory write signal is used to enable the memory module for writing. Furthermore there are three different types of memory writing used in **SB**, **SH** and **SW** instructions. Therefore we have chosen the lower two bits of the signal to specify the three types. Table 8: shows the description of main memory signals.

Main memory: read signal [3:0]

The MSB of the memory read signal is used to enable the memory read in the memory module. Other three bits were used to specify the specific reading method that was required for **LB, LH, LW, LBU, and LHU.** Table 9: shows the description of main memory signals.

Register file write select [1:0]

This signal was used to define what data will be written back to the register file. It requires you to select between three types of write backs.

TABLE 10: REGISTER FILE WRITE SELECT SIGNAL

Reg write select MUX [1:0]	
00	PC+4
01	ALU result
10	MEM_READ

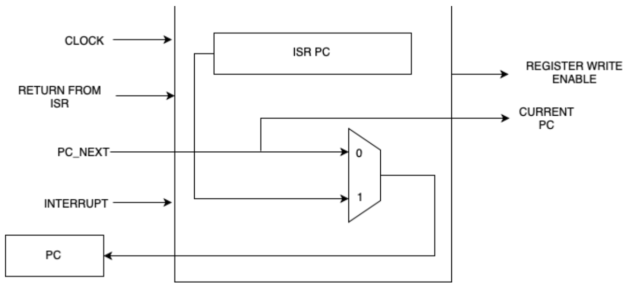


Figure 1: Interrupt Control Unit block diagram

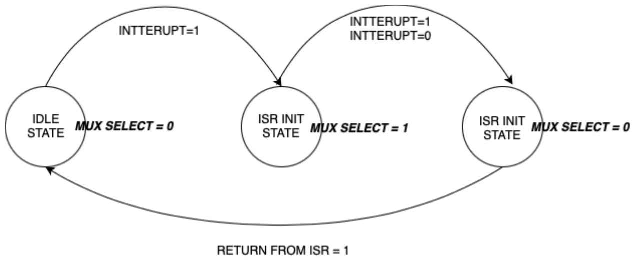


Figure 2: Interrupt Control Unit FSM

Register file write enable

This signal is used to enable the register file to write back

D. Interrupt Control Unit

This interrupt control unit was designed to work with a single interrupt. The ISR location is a fixed value before synthesizing. This controller is based on a three stage Finite State Machine (FSM). This interrupt control unit was placed on the First stage of the five stage pipeline. Figure 1 shows the block diagram of the Interrupt control unit.

How it works

When the interrupt is triggered. The Programme Counter (PC) will jump into a predefined memory location (To the Interrupt service routine) while saving the current PC to the R30 register. Programme have to put

JALR R30

Instruction at the end of the Interrupt Service Routine. Then the Finite state Machine will get reset and get ready for the next interrupt.

Finite State Machine (FSM)

Finite state machine contains three states. When the cpu is in the normal condition the FSM is in the IDLE STATE. The state change will occur on the negative clock edge. When the MUX SELECT signal gets activated the MUX will choose the ISR_PC instead of bypassing the PC_NEXT to the CPU. And when the current state changes to the ISR INIT state the REGISTER WRITE EN signal will be triggered inorder to write the current PC address to the register file. Figure 2: shows the Finite state machine diagram of the Interrupt control Unit.

E. Pseudo Random Number Generator

In some applications, specifically in simulation of spiking neuron models it is required to have a random number seed for the computations. In order to obtain these

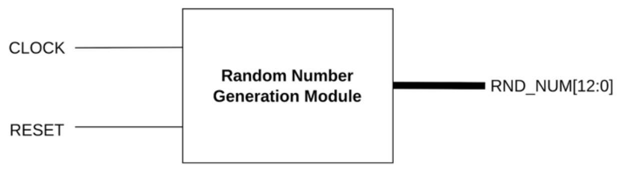


Figure 3: Block Diagram of the Random number generation unit

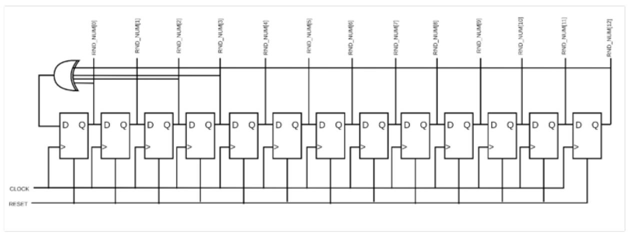


Figure 4: Internal Structure of the Random Number Generation Module

random numbers we created a module to generate a random number by shifting and XORing selected bits in a particular round to compute the number in the next round. Fig 3. depicts the block diagram of the random number generator module.

his module will generate a random number each and every 13 input cock cycles. In the random number generation module there is a 13-bit register. Initially, this register is set to all 1's in when the reset is asserted. Then in each clock cycle, the XOR if 0th, 2nd, 3rd and 12th bits in the register are XORed and the register is left shifted by 1 bit and the XORed value is added to the 0th bit in the register file. Therefore, after 13 clock cycles, a random number generator is generated. Fig 4. shows the internal structure of the module.

When connecting this module with the main CPU, we made sure that the clock input given to the random number generation module is 13 times faster than the clock of the CPU, so a new random number will be available in each clock cycle of the CPU. Also we saved the random number generated in the 32nd register in the register file of the CPU making the 32nd register reserved for the generated random number.

III. INTERGRATION

At this stage the integration of all units and pipeline registers was done. Before the integration, it was tested separately on some crucial parts in the data path. The control logic unit was tested with an automated program that would compare the output of the control unit with a pre-coded CSV file. The CSV file with the control unit outputs was generated from the spreadsheet that was created in the planning stage. More testing details can be found in the testing section.

Since the ALU, instruction memory/cache, data memory/cache, register file was tested separately. Following units were attached separately.

A. Instruction Memory

The instruction memory consists of **1024** bytes. It was designed to support byte addressing. The instruction memory can serve data blocks of **16** bytes (4 words). Since it has byte addressing 28 bits were used to address a block. To load the instructions to the instruction memory a **.bin** file that was generated from the assembler program was used. This **.bin** file has 1024 bytes of instructions data. The instruction block fetching delay was set to 40-time units to simulate the latency of the instruction memory.

B. Instruction Cache

The instruction memory cache consists of 8 blocks. Each block has 4 words. To address a block, a 3-bit index was used. A 2-bit offset is used to fetch the required 32-bit instruction. The remaining 28 bits were used as the tag. Apart from the dirty and a valid bit was assigned per block. The structure is as below.

Dirty	valid	Tag [24:0]	index[2:0]	offset[1:0]	data[127:0]

C. Data Memory

The data memory consists of **256** bytes. It was designed to support byte addressing since some instructions require byte addressing rather than 32-bit word addressing. The data memory is designed to serve 16-byte data blocks to match with the block size of the data memory cache. Since it serves a 16-byte length of blocks 28 bits were used to address each block. The data memory is capable of resetting all data bytes to zero with a positive edge of the reset.

D. Data Cache

The he data memory cache consists of 8 blocks with a size of 16 bytes. Each block has four 32-bit words that can be selected by the 2-bit offset. A 3-bit index was used to address the 8 blocks. Also, a byte offset is used to select the specific byte from the 32-bit word. The structure of the block is as below.

Dirty	valid	Tag [24:0]	index[2:0]	offset[1:0]	byte_offset[1:0]	data[127:0]

Since some instructions such as SB, SH, LB, and LH require byte addressing the cache has additional functionality to mask the required bytes from the 32-bit word in fetching and writing. The additionally added byte_offset is used to select the specific bytes in the 32-bit word.

E. Pipeline Registers

Pipeline registers are the key feature of the pipeline architecture. In this design, there are 5 stages that require 4 sets of pipeline registers. The structure of the pipeline registers throughout the whole data path is as below.

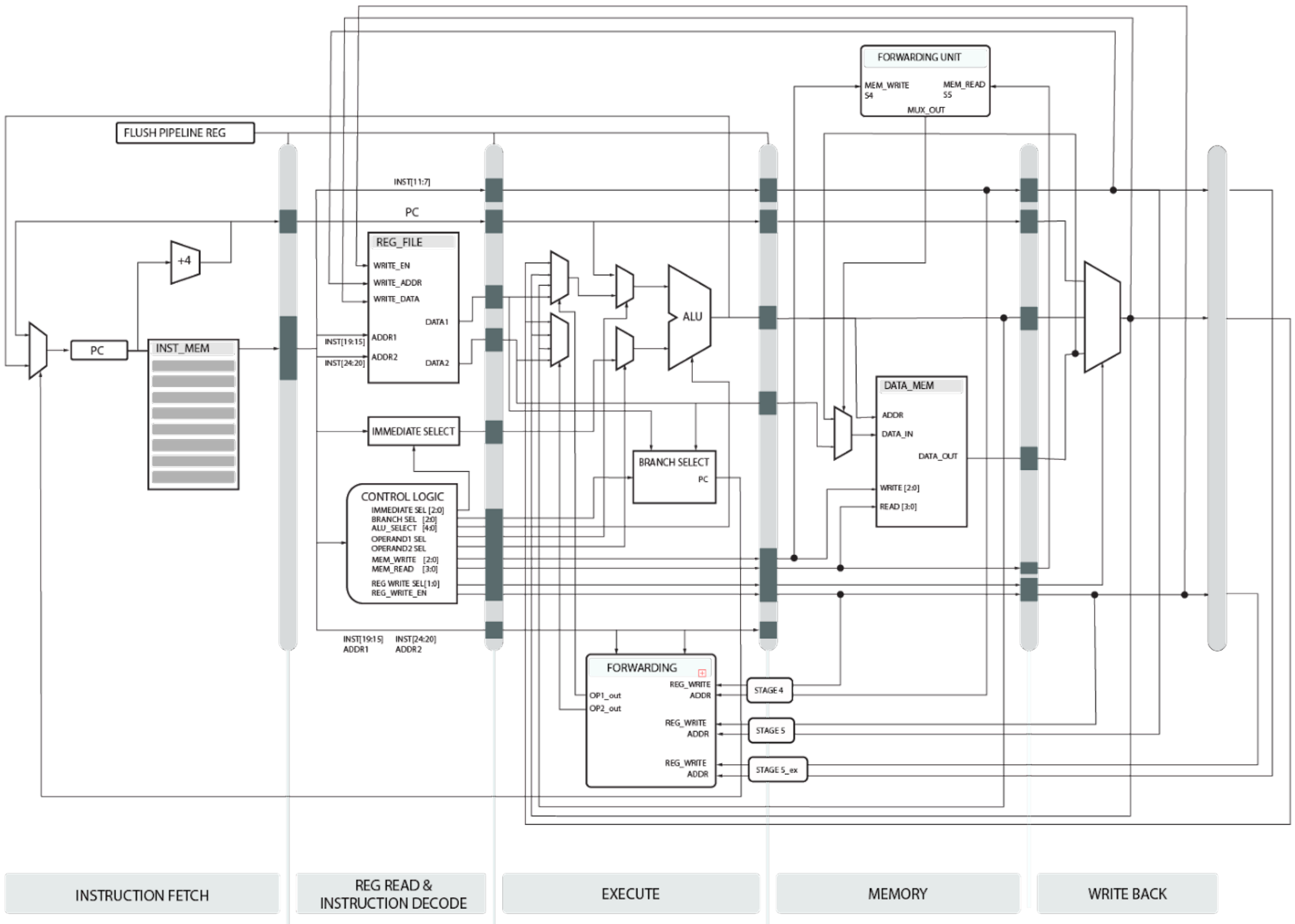


Figure 5: Datapath of the CPU

DESIGN DECISIONS

Moving the shifters inside the ALU

We have moved the shifter inside the ALU rather than having a separate barrel shifting module. We have implemented this way to reduce the complexity of the Design

Design a separate branch selection unit

We have implemented the branching unit as a separate module without implementing it inside the CPU because more than five instructions are using this unit. Like **JAL, LALR, BEQ, BNE, BLT, BGE, BLTU, BGEU**

Implementing the byte selection inside the data cache

The SB, SH, LB, LH kinds of instructions are required to store or load a byte or 2 bytes as their requirement. Therefore it is required to implement a unit that has the ability to mask out the required byte or 2 bytes according to the requirement of the instruction.

In our design, we decided to implement that unit inside of the data memory module. Let's see y what it does. Consider the requirement of the instruction is to load a single byte from a 32-bit word. Then the data cache will fetch the corresponding 32-bit word that contains the relevant byte. Then the additional byte filtering mechanism will extract the required byte and extend its sign if needed(sign will not be extended in LBU, LHU instructions) and serve the byte as output. In the case of two byte loads for LH the process will be the same, the only change will be the size of the mask. For the store scenario, the required byte will be replaced with the 32-bit word without changing the other three bytes.

IV. HAZARD HANDLING

<u>Data Hazard Handling</u>

Data hazards mean the data dependencies between instructions. We have implemented two forwarding units inside stage three and stage 4. There are several methods to handle data hazards. The easy and basic method is to insert bubbles into the pipeline if there are data dependencies. This is an easy approach, but this method decreases the efficiency of the pipeline. The second method is to use forwarding methods. That means taking the ALU results from stages 4 and 5 as operand 1 and operand 2 in the execution stage. The following diagram shows the results that should be forwarded in order to handle the dependency data hazards.

A. Stage 3 (Execution Stage) forwarding unit

This forwarding unit keeps the eye on operand 1 and operand 2 and checks whether there are new values for those lines in Stage 4 or Stage 5. If that is the case the forwarding unit sends the signal to the Hazard Handling Muxes in operand 1 and operand 2 data paths.

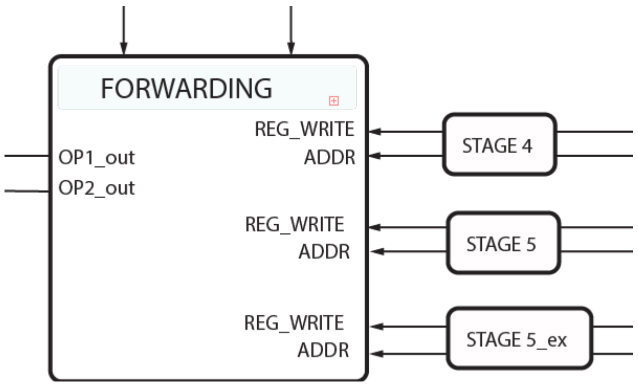


Figure 6: Stage 3 Forwarding Unit

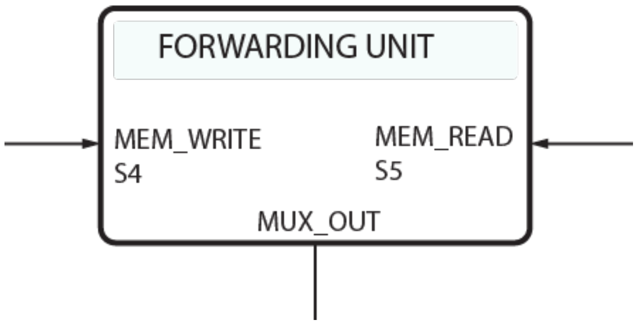


Figure 7: Stage 4 Forwarding Unit

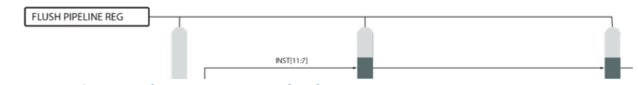


Figure 8: Pipeline register Flushing

Data dependency could happen even between 3 stages apart. To be more clear, data that are in the write back stage are not written to the reg file while the instruction at the decode stage fetches old data from the reg file. Therefore the forwarding unit should forward the data that is currently in the writeback stage at the next clock edge. Since there is no buffer to hold the data that will be omitted from the stage 5 (WB) pipeline register. To hold those data an additional pipeline register set was used.

B. Stage 4 (Main Memory access Stage) forwarding unit

This forwarding unit keeps the eye on operand 1 and operand

```
Lw r3, 8(r0)
sw r3, 8(r0)
```

Control Hazard Handling

Control hazards occur when the instruction flow is diverted because of branching. In order to handle this hazard we have to flush a few pipeline stages. We have connected the first two pipeline registers flushing lines. If the branch control unit decides the current instruction as a branch, then the first two pipeline registers get flushed so the two instructions loaded before the jump instructions are flushed. Fig 8. Shows the flushing lines.

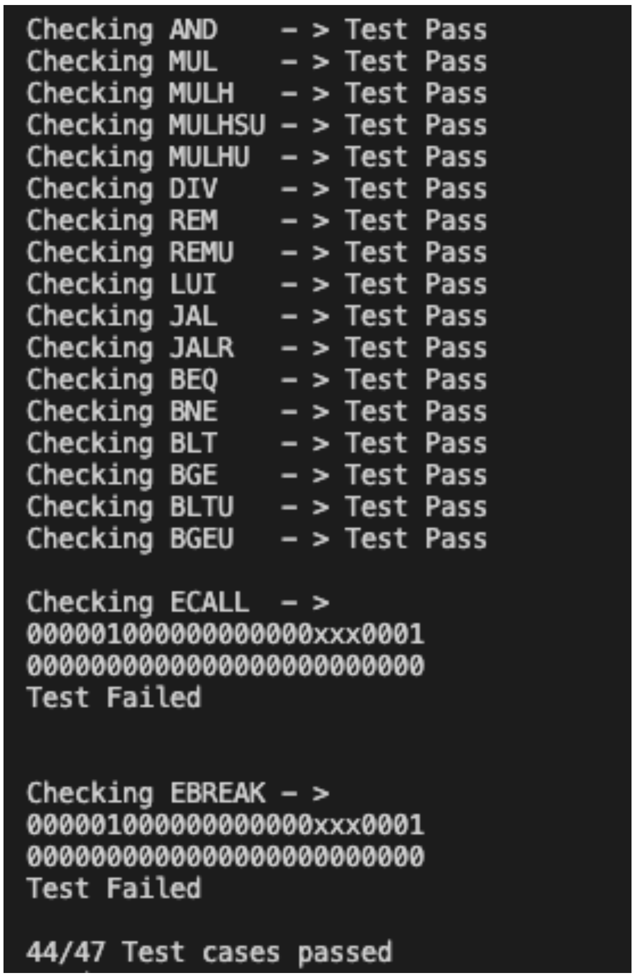
V. TESTING

A. Automated control unit testing

We have implemented the automated testing procedure for the control unit with help of a CSV file. If we want to change or optimize any control unit signals. The testing process is fully automated. So after we have changed the structure or any optimizations in the control unit, the testing code will automatically check for all the supported instructions for the CPU and give us the results and if there are any failures. It also provides which signals are failed. The sample test result is as follow. Fig 9. Illustrate. The test results.

B. Pipeline propagation testing

We have tested the pipeline propagations. With the help of Gtkwave timing diagrams. A sample test result is given below. Fig 10. Shows the testing of data propagation through the pipeline and Fig. 11 shows how the control signals propagate through the pipeline.



```
Checking AND    - > Test Pass
Checking MUL    - > Test Pass
Checking MULH   - > Test Pass
Checking MULHSU - > Test Pass
Checking MULHU  - > Test Pass
Checking DIV    - > Test Pass
Checking REM    - > Test Pass
Checking REMU   - > Test Pass
Checking LUI    - > Test Pass
Checking JAL    - > Test Pass
Checking JALR   - > Test Pass
Checking BEQ    - > Test Pass
Checking BNE    - > Test Pass
Checking BLT    - > Test Pass
Checking BGE    - > Test Pass
Checking BLTU   - > Test Pass
Checking BGEU   - > Test Pass

Checking ECALL  - >
0000010000000000xxx0001
0000000000000000000000000
Test Failed


Checking EBREAK - >
0000010000000000xxx0001
0000000000000000000000000
Test Failed

44/47 Test cases passed
```

Figure 9: Control unit testing

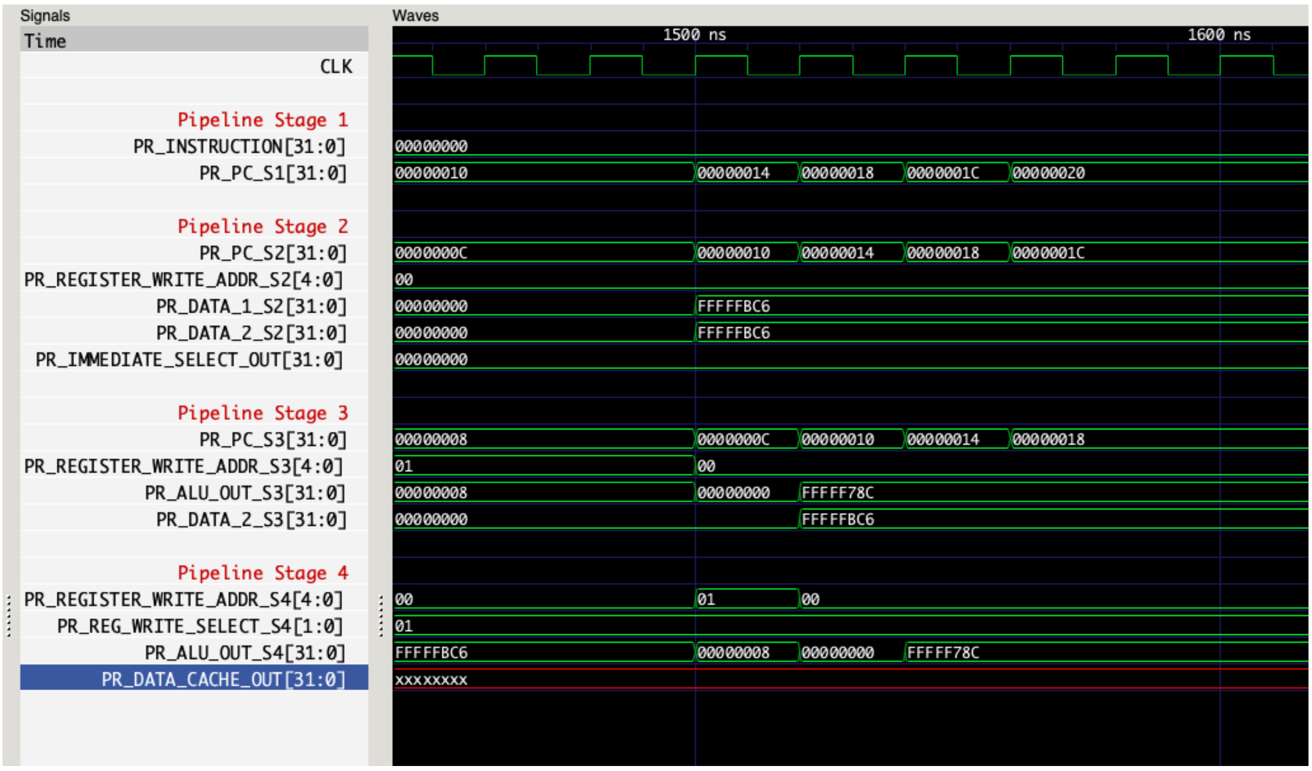


Figure 10: Data propagation through the pipeline

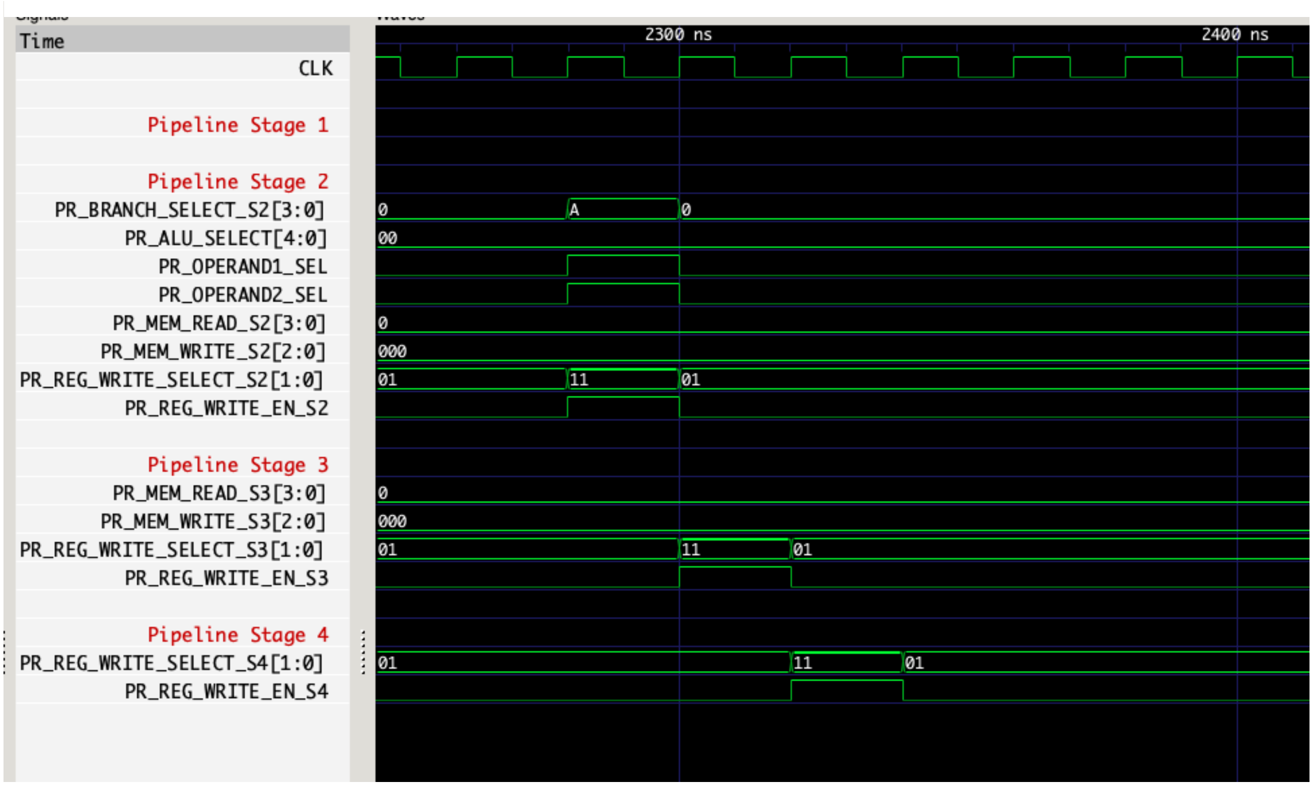


Figure 11: Control signal propagation through the pipeline

VI. CONCLUSION

In conclusion we have build and tested an RV32IMF CPU with additional pseudo random number generation unit and Interrupt unit. This design was done with Verilog and the testing were done with Gtkwave and Quartus Prime software in combination with Altera DE2 and DE5 boards. Loads of programs were Compiled and tested in this system and there correctness were ensured using various types of tests.

References

[1] A. Waterman and K. A. Asanovi´c, “The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2 Editors,” 2017.

[2] “Computer Organization and Design: The Hardware/Software Interface - David A. Patterson, John L. Hennessy - Google Books.” https://books.google.lk/books/about/Computer_Organization_and_Design.html?id=U6lpXmu7OngC&printsec=frontcover&source=kp_read_button&hl=en&redir_esc=y#v=onepage&q&f=false (accessed Feb. 25, 2023).

[3] “GitHub - nishthaparashar/Floating-Point-ALU-in-Verilog: 32-Bit Algorithms of Floating Point Operations are implemented on Verilog with logic Operations.” https://github.com/nishthaparashar/Floating-Point-ALU-in-Verilog (accessed Feb. 25, 2023).

RISC-V Based Network on Chip Architecture for Spiking Neuron Processing

Heshan Dissanayake
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
e16088@eng.pdn.ac.lk

Buddhi Perera
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
e16276@eng.pdn.ac.lk

Dinindu Thilakarathna
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
e16366@eng.pdn.ac.lk

Dr. Isuru Nawinne
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
isurunawinne@eng.pdn.ac.lk

Dr. Isuru Dasanayake
Department of Electrical & Electronic Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
isurud@ee.pdn.ac.lk

Dr. Mahanama Wickramasinghe
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
mahanamaw@eng.pdn.ac.lk

Prof. Roshan Ragel
Department of Computer Engineering
Faculty of Engineering, University of Peradeniya
Peradeniya, Sri Lanka
roshanr@eng.pdn.ac.lk

***Abstract*— This article described a new hardware architecture based on RISC-V instruction set architecture and a scalable network-on-chip design with a router design. Izhikevich spiking neural network model was implemented on top of this Hardware architecture.**

***Keywords*— neuromorphic computing, spiking neural networks, Izhikevich model, network on chip, RISC-V, RV32IMF**

I. Introduction

Most of the computing devices developed until today are based on the von Neumann architecture which was defined by John von Neumann in 1945. With the development of technologies in areas like artificial intelligence and machine learning, computers that are based on the von Neumann architecture are not efficient in power consumption. But when considering the biological brain, it is capable of handling these complex learning and inferencing techniques with a very low power consumption. This is mainly due to the organizational structure and the event-driven nature of the brain. When considering the organizational structure of the brain, is made out of simple biological components known as neurons and these neurons communicate with each other using areas called synapses. The computer architectures that are developed and inspired by these biological brain-like architectures are known as neuromorphic architectures[1]. Generally, in these neuromorphic architectures, there are many simple processing components which are similar to the neurons in the brain and these simple processing elements are connected using a dense network that is similar to the synapses in the brain.

In this research, we have designed and developed a scalable network on chip (NoC) architecture with RISC-V[2] based processing elements and it is implemented on an FPGA. Then on top of the hardware implementation, we are simulating a spiking neural network[3] application by having an event-driven messaging mechanism. Then we are optimizing the NoC hardware architecture for the spiking neural network applications.

II. Methodology

The methodology section is divided into 2 main sections namely the NoC hardware architecture implementation and the software for simulation of the spiking neural network application.

A. *Network on Chip (Hardware architecture)*

Network on chip (NoC) is a hardware architecture for communication between the nodes in an integrated circuit[4]–[6]. Nodes in NoC can be any module that can send and receive messages such as processing cores, memory controllers, and caches.

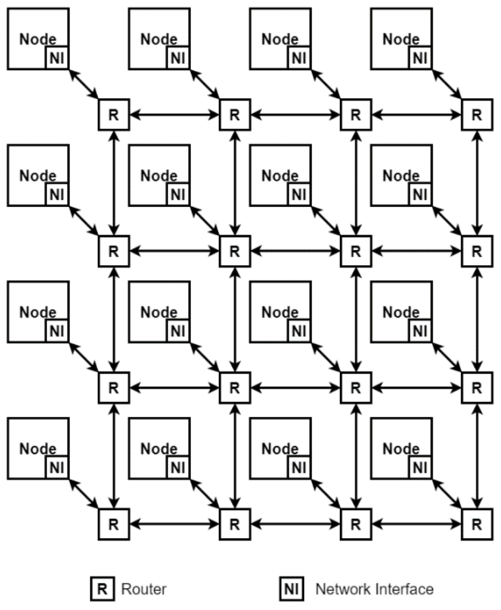


Figure 1: Proposed Network on Chip Design

A router-based packet-switching communication mechanism is used for the communication between the nodes in an NoC and the mechanism is similar to the computer communication networks. Each node is connected to a router via a network interface and connections between routers make the NoC architecture. A simple NoC with 16 nodes arranged in a mesh topology is shown in Figure 1. In the following sections, the main components of the NoC are explained in detail.

1) *Nodes and the Network Interface*

A router-based packet-switching communication mechanism is used for the communication between the nodes in an NoC and the mechanism is similar to the computer communication networks. Each node is connected to a router via a network interface and connections between routers make the NoC architecture. A simple NoC with 16 nodes arranged in a mesh topology is shown in Fig 1. In the following sections, the main components of the NoC are explained in detail.

When using these processing elements the nodes in the network required an additional feature which is the network interface. The network interface was designed to communicate with the routers in the NoC. This was connected to the RV32IMF processor at the fourth stage in the pipeline. This network interface will be connected to stage 3 and stage 4 forwarding units to remove data hazards. This interface works mostly like the main memory. We have added two additional instructions to the RV32IMF instruction set, Those instructions are mentioned below,

a) LWNET - Load word from the network interface

Reading a data word from the network interface is a two-cycle process. Each cycle will be similar to load word instruction. We have to get the address of the sender CPU and the data word separately. If we call the LWNET instruction with the address as 0 then the top element of the receive FIFO will pop out and put the sender's address to the pipeline, save it to the destination register, and store the data word to a special register. Then we have to call LWNET again with address 1 then the previous data word will be put into the pipeline and saved to the destination register.

b) SWNET - Send word from the network interface

This works mostly like the store word instruction. We can mention the address of the CPU we want to send the information to and we can specify the source register which contains the data word. Then the data will be added to the send FIFO at the network interface.

2) *Construction of the network interface*

The network interface mainly consists of two FIFOs, they send FIFO and receive FIFO. Both of these FIFOs have 32 slots and the slot size is 64 bits. This stores a 32-bit address and 32-bit data word. Network interface was connected to the status register to update the status of the two FIFOs. And they receive a buffer threshold signal that will be connected to the interrupt control unit to invoke the special ISR.

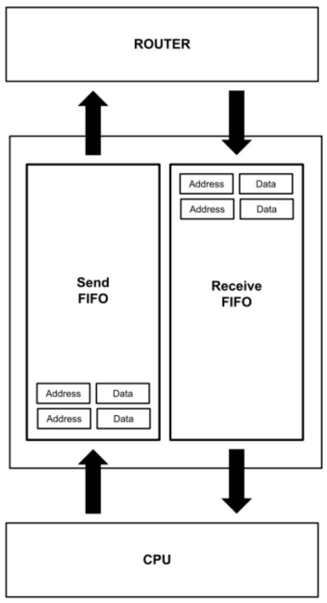


Figure 2: Block diagram of the Network Interface

3) *Inner workings of the network interface*

When the CPU wants to send a data packet. That data packet will be added to the send FIFO. When the send FIFO gets full, the full signal will be set in the status register. When the FIFOs count falls below the preset threshold. The send full signal will be de-asserted from the status register.

When the received FIFO is filled above the preset threshold an interrupt will be generated and the CPU will move into a special ISR (Interrupt Service Routine). Then the CPU will consume the whole received FIFO.

4) *Router*

Routers are required in NoCs to facilitate the data transfer from one node to another node through the network. To translate data, these data are arranged into packets and these packets are communicated over the network. In our communication protocol, the packets are further divided into small parts called flits. Each flit is 8 bits wide and there are three types of those flits. They are,

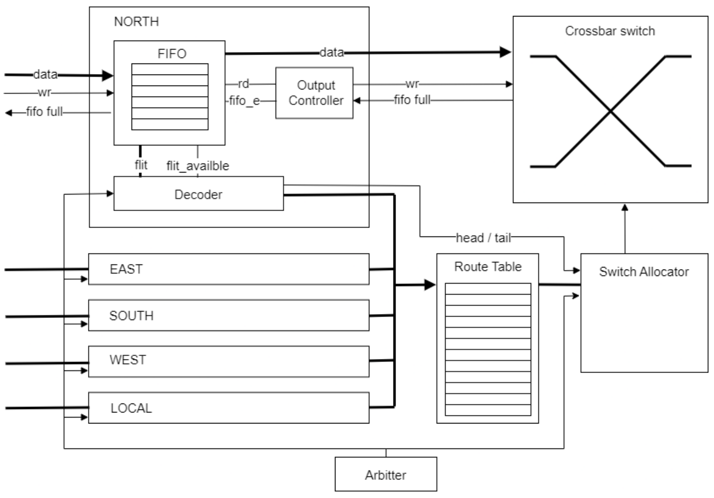


Figure 3: Router Block Diagram

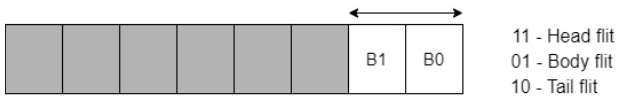


Figure 4: Packet Structure

1. Head flits: These are the flits that will store the routing information such as destination, and the number of flits in the packet, and are used to compute the route of the packet.
2. Body flits: These contain the contents of the packet and are divided into many. For a single packet, there might be many body flits.
3. Tail Flits: These are the flits that represent the end of a packet.

In the flit structure, the leading two bits will mention the type of that flit.

a) Router FIFO

In the routing mechanism, the route computation must be performed when a head flit is received. To do this first it is necessary to buffer that incoming flit. That's where the router buffer takes place. This buffer can hold multiple flits at once, which will give the ability to the predecessor router to transfer multiple packets at once. A FIFO is used for this purpose because of the first in first out nature.

The main purpose of the buffer is to buffer data issue packets as requested in first out fashion. To receive data, the FIFO buffer is connected to the previous router channel through 3 data lines, Which are data(4-bit input), write_en(1-bit input), and fifo_full(1-bit output). The data flow will happen through the data line with 4 bits at a time. Since a single flit is 8-bit wide, two reads from the data line are required to transfer a flit. The write_en line will be generated from the router to notify the FIFO to buffer the incoming data. Once the FIFO is full fifo_full signal will be asserted to notify the transmitting router to stop the transmission.

The FIFO has a total of 16 bytes of space to store the incoming data. This data space can be addressed by a 4-bit address which can address each 16 bytes. There are 3 different pointers in the FIFO to govern the internal mechanism. They are write_pointer, read_pointer, and the flit_poiner. The write pointer is the one that keeps track of the next possible location in the buffer to store the incoming data packet. This pointer will increment at every incoming data packet and will be stalled when the FIFO is full. The read pointer is the one that keeps track of the next location of the FIFO that can be read. This pointer will be incremented at every read request and will be stalled if FIFO is empty. These FIFO full and empty signals will be created by comparing the FIFO read and write pointers. Read and write pointers are common for FIFOs, but in our implementation, we need a mechanism to scan each incoming data to detect head flits and proceed with route computation. In that case, the flit pointer will keep track of the latest flit that is available in the FIFO that can proceed for route computation. Since it requires 2 data receptions to complete a flit, the flit pointer will be incremented byte-wise. In the flit structure, the leading two bits will distinguish whether this flit is a head, body, or tail. Whenever the current flit is detected as a head, the incrementation of the flit pointer must be suspended until the route computation is completed. These flit-type detections will be done by the decoder unit.

When issuing packets for the next router, the data out port of the FIFO will reveal the flit that is at the location of the read pointer. The read_en signal which is an input to the FIFO will make the read pointer incrementable to move on to the next reading location. The fifo_empty signal generated by the FIFO is used to notify the output controller about the data availability of the FIFO.

b) Decoder

This module is responsible for decoding the incoming data flit. It has to detect the type of data flit. This module outputs a signal as head_flit, that will notify that the current flit is a head flit. This signal is useful for the FIFO to stall the flit pointer until the route computation is over. The switch allocator will use the head_flit signal to allocate a free channel in the switch. Also, the tail_flit signal will be used to deallocate that previously allocated channel in the switch.

In the case of head flit detection, now we have to proceed to the route computation. For that, the decoder will decode the flit and will separate the destination data from the flit. To find the relevant channel that is required for that specific destination, it should be found in the routing table. For this, we have to send the destination to the routing table. Since there is only one routing table and 5(North, south, east, west, and local) decoders per router, we have to arbit between each of those decoders. Each decoder will have an id and the arbiter will arbit within those ids one by one. Whenever the relevant Id is called by the arbiter, the destination data will be put into a data bus that will connect all the decoders and the routing table.

c) Arbiter

To manage multiple input channels and a single routing table for a network switch, an arbiter is necessary. The arbiter is responsible for selecting a channel at a particular time, which can then use the routing table to determine the appropriate destination for the current flit. Since the switch allocator needs to be aware of the selected channel at a given moment, the arbiter's identification is sent to the switch allocator. This process ensures that the switch can effectively route data packets to their intended destinations by coordinating the input channels and the routing table.

d) Routing Table

The routing table is a lookup table that has entries that specify which output channel should be taken for each possible destination. By using this it is possible to find which output channel should be taken by an incoming head flit.

e) Output Controller

The output controller lies between the crossbar switch and the FIFO. It will regulate the output flow of the data in the router. The grant signal from the switch allocator is considered to start and stop the data issuing for the next router. The data flow is controlled according to the data availability of the current router and the buffer availability of the next router.

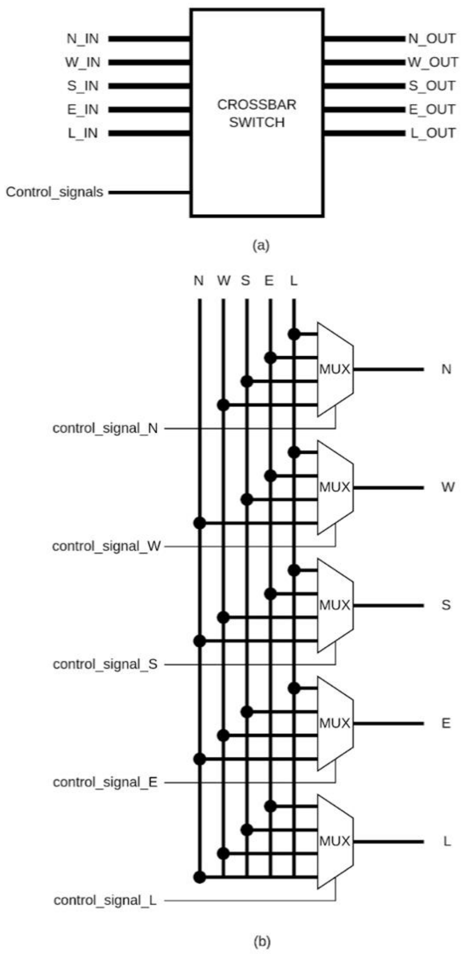


Figure 5: Crossbar switch design

f) Crossbar Switch

The crossbar switch[7] is an important component in the router. The purpose of the crossbar is to connect an input port of the router to the relevant output port according to the control signals provided. There are 5 input ports, 5 output ports, and 5 control signals.

Since there are 5 input ports and 5 output ports, a 5x5 crossbar switch was used. To build this crossbar switch, 5 4x1 multiplexers were used for each output port. For a particular multiplexer, the input ports were the 4 input ports except for the input port corresponding to the output port connected to that multiplexer. This exclusion of the input port was done because the data incoming from a direction should not be forwarded in the same direction.

Each control signal is connected to each of the multiplexers and is a 2-bit signal. According to the control signals the multiplexer connects accordingly. Figure 05 depicts the block diagram and the schematic view of the crossbar switch.

g) Switch Allocator

A switch allocator is used to allocate output ports until the data packet from an input port is transmitted. This component will generate the control signal that drives the crossbar switch and also it will generate an enable signal to all 5 output controllers. Inputs to the switch allocator are the requests to allocate the output channel to the input channel. The request consists of 2 bits, 1 from the input arbiter and 1 from the router computation stage. Therefore altogether there are 20 request signals connected to the input of the switch allocator.

B. Software for Simulation of the Spiking Neural Network Application

An instruction set was made to simulate one neuron according to the Izhikevich model[8], [9]. We optimized the assembly instructions set to run the model with around 25 instructions. We have modeled the assembly code with python and numpy before writing the code in assembly. We have organized the Register values to the corresponding points in the int array size 32. Izhikevich model consists of the following two equations and one if condition.

$$v' = 0.04v^2 + 5v + 140 - u + I \qquad (1)$$

$$u' = a(bv - u) \qquad (2)$$

$$if\ v\ \geq 30mV, then \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \qquad (3)$$

the following two equations and one if condition.

```
# ================== start ==================

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

r = [0 for i in range(11)]
print(r)
r[0] = 0.03                    #a
r[1] = 0.2                     #b
r[2] = -65                     #c
r[3] = 8            #d
r[4] = -65                     #v
r[5] = r[1]*r[4]               # u = b*v
r[6] = 0.04                    #need for eqn
r[7] = 5                       #need for eqn
r[8] = 140                     #need for eqn

I = 1
data = np.zeros((200))

count = 0
while True:
  count = count + 1
  I = 5*np.random.normal()

  r[9] = r[4] * r[4]  # v**2
  r[9] = r[9] * r[6]  # 0.04*(v**2)

  r[10] = r[7]*r[4]  # 5*v
  r[9] = r[9] + r[10]            # 0.04*(v**2) + 5*v
  r[9] = r[9] + r[8]  # 0.04*(v**2) + 5*v + 140
  r[9] = r[9] - r[5]  # 0.04*(v**2) + 5*v + 140 - u
  r[9] = r[9] + I     # 0.04*(v**2) + 5*v + 140 - u + I
  r[4] = r[4] + r[9]  # v = v + 0.04*(v**2) + 5*v + 140 - u + I

  r[9] = r[1] * r[4]  # b*v
  r[9] = r[9] - r[5]  # b*v - u
  r[9] = r[0]*r[9]    # a*(b*v - u)
  r[5] = r[5] + r[9]  # u = u + a*(b*v - u)

  print(r[4])
  data = np.roll(data,shift =-1,axis=0)

  if r[4]>30:
    r[4] = r[2]                # v = c
    r[5] = r[5] + r[3]         # u = u + d
    print('spiked')
    data[-1] = r[4]
    plt.cla()
    plt.plot(data)
    plt.pause(0.00005)
# ================== end ==================
```

Then the above code was converted to the following assembly code and generated the RV32 machine code using our assembler.

```
; ================== start ==================

     ; for comparisons
     addi r12, r12, 0
     addi r13, r13, 1

     ; r5 = r1 * r4
     fmul r5, r1, r4

     Loop:
     fmul r9 , r4, r4
     fmul r9 , r9, r6
     fmul r10, r7, r4
     fadd r9, r9, r10
     fadd r9, r9, r8
     fsub r9, r9, r5
     fadd r9, r9, r31
     fadd r4, r4, r9

     fmul r9, r1, r4
     fsub r9, r9, r5
     fmul r9, r0, r9
     fadd r5, r5, r9

     ; r4 - voltage  r11 - 30
     flt r10, r4, r11
     beq r10, r13, notspiked
     addi r4, r2, 0
     fadd r5, r5, r3
     addi r15, r15, 1

     notspiked:
     ; Loops:
     jal r14, Loop
; ================== end ==================
```

III. Results

We have tested the Izhikevich model inside one core. The assembly code was optimized to run in 25 instructions.

Performance

Core Clock	=	5 MHz
Instruction per Iteration	=	5,000,000 / 25
	=	200 K

One iteration simulates 1ms

Iterations per neuron	=	1000
Total Neuron	=	200K / 1000
	=	200

According to the above calculation, we can simulate 200 neurons inside a single core without using a network chip. When assuming a clock rate of 5 MHz

IV. Future works

In the future, we are planning to test the Network on Chip and implement the Izhikevich model in multiple cores. Then we planned to implement a software tool to map an existing spiking neural network to our SOC. Finally, we are planning to benchmark our system with an existing artificial neural network and analyze the performance and power requirements.

References

[1] C. D. Schuman *et al.*, “A Survey of Neuromorphic Computing and Neural Networks in Hardware,” May 2017, [Online]. Available: http://arxiv.org/abs/1705.06963

[2] A. Waterman and K. A. Asanovi´c, “The RISC-V Instruction Set Manual Volume I: User-Level ISA Document Version 2.2 Editors,” 2017.

[3] L. P. Maguire, T. M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin, “Challenges for large-scale implementations of spiking neural networks on FPGAs,” *Neurocomputing*, vol. 71, no. 1–3, pp. 13–29, Dec. 2007, doi: 10.1016/j.neucom.2006.11.029.

[4] W.-C. Tsai, Y.-C. Lan, Y.-H. Hu, and S.-J. Chen, “Networks on Chips: Structure and Design Methodologies,” *Journal of Electrical and Computer Engineering*, vol. 2012, p. 509465, 2012, doi: 10.1155/2012/509465.

[5] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Elsevier, 2004.

[6] S. R. Sarangi, *Advanced Computer Architecture*, 1st edition. McGrawHill.

[7] V. tiwari and K. khare, “Efficient Configurable Crossbar Switch Design For Noc,” *INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH*, vol. 8, no. 11, 2019, Accessed: Feb. 20, 2023. [Online]. Available: www.ijstr.org

[8] E. M. Izhikevich, “Simple model of spiking neurons,” *IEEE Trans Neural Netw*, vol. 14, no. 6, pp. 1569–1572, Nov. 2003, doi: 10.1109/TNN.2003.820440.

[9] E. M. Izhikevich, “Which model to use for cortical spiking neurons?,” *IEEE Trans Neural Netw*, vol. 15, no. 5, pp. 1063–1070, Sep. 2004, doi: 10.1109/TNN.2004.832719.