

# Neural Network Architectures for Integrated Circuits

by

**KHYATI NAGRANI**

**202111031**

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF TECHNOLOGY

in

INFORMATION AND COMMUNICATION TECHNOLOGY

to

**DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY**



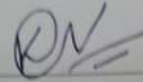
July, 2023

## Declaration

### Declaration

I hereby declare that

- i) the thesis comprises of my original work towards the degree of Master of Technology in Information and Communication Technology at Dhirubhai Ambani Institute of Information and Communication Technology and has not been submitted elsewhere for a degree,
- ii) due acknowledgment has been made in the text to all the reference material used.



---

KHYATI NAGRANI

### Certificate

This is to certify that the thesis work entitled "Neural Network Architectures for Integrated Circuits" has been carried out by KHYATI RAJESH NAGRANI for the degree of Master of Technology in Information and Communication Technology at *Dhirubhai Ambani Institute of Information and Communication Technology* under my supervision.



---

PROF. TAPAS KUMAR MAITI  
Thesis Supervisor

# Acknowledgments

I would like to express my sincere gratitude and appreciation to the following individuals who have played a crucial role in the completion of my work:

First and foremost, I would like to extend my heartfelt thanks to Prof. Tapas Kumar Maiti, my thesis supervisor. His unwavering support, guidance, and expertise have been instrumental in shaping this research. His insightful feedback and constructive criticism have helped me navigate through various challenges, enabling me to reach new heights in my academic journey. I would also like to extend my deepest appreciation to Prof. Ayan Palchaudhuri, currently faculty of IIT-Bhubaneswar, for his valuable inputs and insightful discussions throughout this research endeavor. His vast knowledge and expertise in the field have been truly inspiring, pushing me to explore new avenues and think critically.

I would also like to extend my thanks to my dear friends and PhD scholar, Gulafsha Bhatti. Their friendship, and intellectual discussions have been a source of inspiration and motivation for me.

I am deeply grateful to my parents, Saroj Nagrani and Rajesh Nagrani for their unwavering love, encouragement, and constant belief in my abilities. Their endless support, understanding, and sacrifices have been the foundation of my success, and I am forever indebted to you for the opportunities you have provided me. Furthermore, I would like to express my gratitude to my brother, Hriday Nagrani. His encouragement, motivation, and belief in me have been invaluable throughout this journey.

Lastly, I would like to acknowledge the support and encouragement I received from all my friends and well-wishers who have cheered me on during this academic pursuit. Their friendship, understanding, and motivation have made this journey all the more fulfilling.

To all those mentioned above, as well as to anyone else who has contributed to my work in any way, I offer my heartfelt appreciation. Their guidance, support, and belief in me have been the driving force behind the successful completion of my thesis, and I am truly grateful for their presence in my life.

# Contents

|                                                                                        |            |
|----------------------------------------------------------------------------------------|------------|
| <b>Abstract</b>                                                                        | <b>v</b>   |
| <b>List of Principal Symbols and Acronyms</b>                                          | <b>v</b>   |
| <b>List of Tables</b>                                                                  | <b>vi</b>  |
| <b>List of Figures</b>                                                                 | <b>vii</b> |
| <b>1 Introduction</b>                                                                  | <b>1</b>   |
| 1.1 Neuron and Neural Network . . . . .                                                | 3          |
| 1.2 Artificial Neuron and Artificial Neural Network . . . . .                          | 4          |
| 1.3 Applications of Artificial Neural Network . . . . .                                | 6          |
| 1.4 Chip Design for NNs . . . . .                                                      | 6          |
| 1.5 NNs as Co-Processor . . . . .                                                      | 7          |
| 1.6 Aims and Objectives . . . . .                                                      | 8          |
| <b>2 Review of Neural Network Architectures</b>                                        | <b>10</b>  |
| 2.1 Review of Adder Design . . . . .                                                   | 10         |
| 2.1.1 Comparison of Adders on Basis of Area, Power, and Delay .                        | 10         |
| 2.2 Review of Multiplier Design . . . . .                                              | 12         |
| 2.2.1 Comparative Analysis of Several Multipliers . . . . .                            | 12         |
| 2.3 Review of Activation Function Design . . . . .                                     | 15         |
| 2.3.1 Activation Functions for Neural Networks . . . . .                               | 15         |
| 2.3.2 ReLU, Linear Saturated, and Other Activation Functions:<br>Comparisons . . . . . | 16         |
| 2.3.3 Conclusion for Activation function design . . . . .                              | 17         |
| 2.3.4 Comparison of ReLU and Sigmoid . . . . .                                         | 18         |
| 2.4 Review of Neural Networks Design . . . . .                                         | 20         |
| 2.4.1 Artificial Neural Networks . . . . .                                             | 20         |
| 2.4.2 Energy-Efficient VLSI Architectures . . . . .                                    | 21         |
| 2.4.3 Memory Organization Techniques . . . . .                                         | 21         |

|          |                                                       |           |
|----------|-------------------------------------------------------|-----------|
| 2.4.4    | Algorithmic Innovations . . . . .                     | 21        |
| 2.4.5    | Impact on Real-World Applications . . . . .           | 22        |
| 2.4.6    | Conclusions for Neural Network Design . . . . .       | 22        |
| <b>3</b> | <b>Implementation of Neural Network Architectures</b> | <b>23</b> |
| 3.1      | Implementation of Adder . . . . .                     | 23        |
| 3.2      | Implementation of Multiplier . . . . .                | 24        |
| 3.3      | Implementation of SISO NN Architecture . . . . .      | 26        |
| 3.4      | Implementation of MISO NN Architecture . . . . .      | 27        |
| 3.5      | Implementation of MIMO NN Architecture . . . . .      | 29        |
| <b>4</b> | <b>Neural Networks Layout</b>                         | <b>31</b> |
| 4.1      | Design Flow . . . . .                                 | 31        |
| 4.2      | Design Tools . . . . .                                | 32        |
| 4.3      | Details of Tools Used . . . . .                       | 33        |
| 4.4      | OpenLANE . . . . .                                    | 42        |
| 4.4.1    | OpenLANE Configuration Variables . . . . .            | 43        |
| 4.4.2    | Required Configuration Variables . . . . .            | 43        |
| 4.4.3    | Optional Configuration Variables . . . . .            | 43        |
| 4.4.4    | Details of Configuration Variables . . . . .          | 44        |
| 4.5      | Tape-Out Details . . . . .                            | 45        |
| 4.6      | Tape-Out . . . . .                                    | 46        |
| 4.7      | Comparison with Published Work . . . . .              | 50        |
| <b>5</b> | <b>Conclusion</b>                                     | <b>51</b> |
|          | <b>References</b>                                     | <b>53</b> |

# Abstract

This thesis presents the architecture design and implementation of neural networks (NNs) for integrated circuit design. The architecture consists of adders, multipliers, and rectified linear unit (ReLU) blocks. Three architectures, namely, Single-In Single-Out (SISO), Multiple-In Single-Out (MISO), and Multiple-In Multiple-Out (MIMO) are developed. In neural networks, weight values are necessary and they are supplied from a memory source. The weight values were prepared by training the NNs model on software. Finally, the SISO, MISO, and MIMO neural-networks were taped out. These architectures can be used for intelligent co-processor development.

# List of Tables

|     |                                                      |    |
|-----|------------------------------------------------------|----|
| 2.1 | Performance analysis of various Adders[17] . . . . . | 12 |
| 4.1 | Optional configuration variables . . . . .           | 44 |
| 4.2 | Tape-Out details . . . . .                           | 45 |
| 4.3 | Comparison with existing works . . . . .             | 50 |

# List of Figures

|     |                                                                                      |    |
|-----|--------------------------------------------------------------------------------------|----|
| 1.1 | A biological neuron [7]. . . . .                                                     | 3  |
| 1.2 | A biological neural network (NN) [8]. . . . .                                        | 4  |
| 1.3 | Architecture of three-layer ANN. . . . .                                             | 5  |
| 1.4 | Schematically illustrates the use of a NNs processor as a co-processor.<br>. . . . . | 7  |
| 2.1 | Depicts Sigmoid function and it's derivative [29]. . . . .                           | 19 |
| 3.1 | Depicted the architecture of a carry increment adder (CIA). . . . .                  | 24 |
| 3.2 | Architecture of Multiplier. . . . .                                                  | 25 |
| 3.3 | SISO architecture. . . . .                                                           | 26 |
| 3.4 | Depicted the simulated waveform obtained using SISO architec-<br>ture. . . . .       | 27 |
| 3.5 | Architecture of MISO. . . . .                                                        | 28 |
| 3.6 | Simulated waveform obtained using MISO architecture. . . . .                         | 28 |
| 3.7 | Depicted the developed MIMO Architecture. . . . .                                    | 29 |
| 3.8 | Simulated responses of MIMO architecture. . . . .                                    | 30 |
| 4.1 | Tools used for implementation. . . . .                                               | 33 |
| 4.2 | Workflow of yosys. . . . .                                                           | 35 |
| 4.3 | Tape-out of SISO. . . . .                                                            | 47 |
| 4.4 | Tape-out of MISO. . . . .                                                            | 48 |
| 4.5 | Tape-out of MIMO. . . . .                                                            | 49 |



## CHAPTER 1

# Introduction

Neural networks (NNs) were implemented using both the software and hardware [1], [2]. However, the hardware level implementation of neural network (NN) improves the speed, efficiency, real-time processing, power efficiency, and scalability in comparison to the software level implementation. The improvement details are discussed below:

**Speed and Efficiency:** In comparison to software solutions, NN hardware achieved considerable speed and efficiency benefits [1], [2], [3]. To speed up neural network computations, specialised hardware can be proposed, such as application-specific integrated circuits (ASICs) [3]. Faster inference and training times are achieved by using specialised hardware like field-programmable gate arrays (FPGAs), graphics processing unit (GPU) that take advantage of parallelism and optimise the execution of matrix operations. It may also result in less power usage, increasing its energy efficiency.

**Real-time processing:** Low-latency (Latency refers to the delay or the time it takes for a system to respond to a given input or stimulus. It is the time interval between initiating an action or request and receiving the corresponding response or output) processing is necessary for several applications, including real-time decision-making systems, robots, and autonomous vehicles. NN hardware solutions offer the speed and responsiveness required to satisfy real-time needs [4]. The system may analyse data and produce predictions in real-time without depending on the constraints of software-based systems by outsourcing the computing effort to specialised hardware.

**Power Efficiency:** Neural network calculations are computationally demanding, especially for complicated or large-scale problems. General-purpose processors (CPUs) or graphics processing units (GPUs) are frequently used in software-based systems, which may require more power to run the calculations [5]. In contrast, neural network operations may be extensively optimised for hardware implementations, which reduces power use. Hardware neural network imple-

mentations are ideally suited for use in embedded systems and edge computing settings (Proximity-based data processing, decentralizing from remote clouds.), according to research [6]. In these cases, neural networks are used on hardware with constrained storage and processing power. By implementing neural networks on specialized hardware, the primary processor offloads computational burdens, creating a separation of tasks and freeing up resources for other functions [6].

**Scalability:** Neural network hardware implementations can be designed to effectively scale with the increasing demands of neural network processing [2]. Implementation of NN hardware can be improved and optimized to handle larger models and datasets as the requirements for neural network computations grow. Scalability can be provided via hardware designs like systolic arrays or specialised deep learning accelerators and support for parallel processing, enabling faster training and inference on increasingly complex neural networks. Scalability, in the context of neural network hardware designs, refers to their ability to effectively handle larger and more intricate models and datasets as the demand for neural network processing grows. It encompasses the capacity to manage increased computational loads and accommodate expanding requirements.

One hardware design that supports scalability is the systolic array, commonly utilized for accelerating matrix operations in neural networks. A systolic array comprises a synchronized array of processing elements interconnected in a regular pattern, enabling efficient flow of data through the array. Notably, systolic arrays excel in parallel processing, making them well-suited for performing matrix multiplications, convolutions, and other vital operations in neural networks.

Systolic arrays offer scalability through the parallel execution of operations across multiple processing elements. As neural networks become larger and more complex, systolic arrays can be scaled up by expanding the array's dimensions or increasing the number of processing elements. This allows for distributed computation and optimal resource utilization, enabling faster training and inference on larger neural networks.

## 1.1 Neuron and Neural Network

The fundamental units of an animal's nervous system, including the human nervous system, are called biological neurons. Their job is to process and send information via electrical and chemical signals. They are specialised cells which consist of a cell body, dendrites, and an axon make up each neuron. While the axon sends electrical signals from the cell body to other neurons or muscles, the dendrites receive alerts from other neurons or sensory receptors. Figure 1.1 shows diagram of biological neuron [7]. A neuron produces an electrical impulse known as an action potential when it receives a signal from another neuron or a sensory receptor. The neurotransmitters, molecules that send signals to subsequent neurons in a chain are released due to the action potential moving down the axon. Movement, sensation, perception, learning, and memory are just a few of the numerous brain activities that depend on neurotransmitter-mediated communication between neurons. Neuronal communication dysfunction has been linked to a variety of neurological and psychiatric disorders, such as Alzheimer's disease, Parkinson's disease, and depression.



Figure 1.1: A biological neuron [7].

A biological neural network is called the intricate web of neurons and synapses that makes up the nervous system of living things. Specialised cells called neurons are in charge of sending electrical and chemical signals throughout the body to convey information. The processing and integration of sensory data, the creation of motor orders, and the coordination of body activities are all made possible by the complex networks of neurons that make up the brain. Neuroplasticity is the ability of these networks to adapt and change throughout time in response to experiences and stimuli.



Figure 1.2: A biological neural network (NN) [8].

The neural network's ability to function depends heavily on the synapses or connections between neurons. The strength and effectiveness of these connections, which allow for the passage of information between neurons, can be altered by various variables, including experience and learning [9]. Figure 1.2 shows picture of a biological neural network. [8] The biological neural network, which underlies the operation of the nervous system as well as the behaviour and cognition of living animals is an immensely intricate and dynamic system.

## 1.2 Artificial Neuron and Artificial Neural Network

The basic components of an artificial neuron includes the following:

- An activation function is defined as an equation that calculates the value of output of the neuron on the basis of the weighted inputs.
- Inputs are numerical values that reflect the data to be processed.
- Weights are numerical values that indicate the strength of the link between the inputs and the neuron.

The appropriate weights are multiplied with the inputs, and the resulting values are summed. The total is then passed through the activation function, which generates the output of the neuron. This output can either be used as the final output of the neural network or shared with other neurons for communication purposes [10]. The activation function has various forms, such as step, sigmoid, ReLU (Rectified Linear Unit), and others. The choice of activation function can significantly affect the performance and behavior of the neural network.

Artificial neural networks are created by joining artificial neurons together. These networks can then be trained using a variety of algorithms to identify pat-

terns in the input data and generate predictions based on those patterns [11]. A machine learning technique called an artificial neural network (ANN) is modelled on the basis of the biological neural networks that make up the human brain [12]. A large number of neurons, arranged in layers, make up an ANN. Each neuron takes in one or multiple input signals, computes, and then sends its output to other neuron in the layer below as mentioned in Figure 1.3.

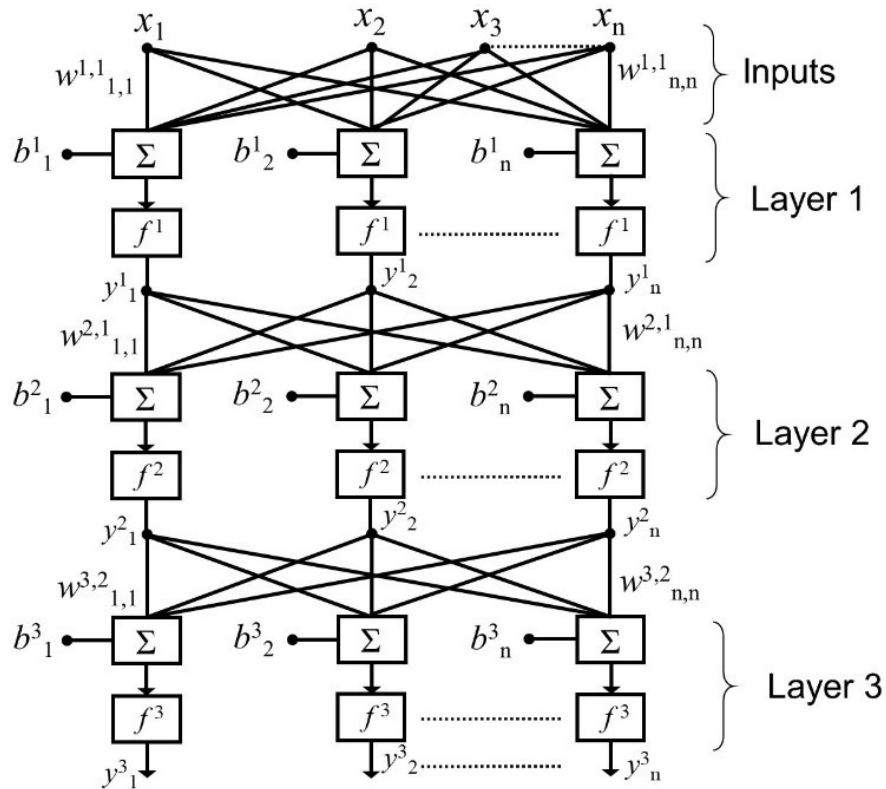


Figure 1.3: Architecture of three-layer ANN.

An ANN typically has three layered structure of neurons which are an input layer, a few to many hidden layers, and lastly the output layer [13]. Raw data is delivered to the input layer, which is processed by the hidden layers. The output layer creates the network's ultimate output, which may be a classification, a prediction, or another type of judgement.

The signal strength of the interconnection between two neurons in an ANN defines how much influence one neuron has on the other since the relations between the neurons in an ANN are often weighted [14]. The backpropagation technique uses weights that are modified during training to reduce the discrepancy between the NN's output and the expected output. Predictive analytics, natural language processing, image and audio recognition, and other fields all make use of ANNs.

## 1.3 Applications of Artificial Neural Network

Neural networks (NN) find applications across diverse fields such as pattern recognition, natural language processing, medical applications, etc., offering various uses as powerful tools. Some notable applications include:

1. **Pattern Recognition:** NNs excel in recognizing patterns in voice, images, and other media. They are extensively employed in facial recognition technology, enabling the identification of faces in images or videos.

2. **Natural Language Processing (NLP):** NNs play a crucial role in NLP applications such as speech recognition, text-to-speech conversion, language translation, and sentiment analysis [15]. They enable the understanding and interpretation of human language.

3. **Medical Applications:** NNs are utilized in predictive modeling to discover patterns in data and generate predictions based on those patterns. For instance, they can leverage a patient's medical history and symptoms to forecast stock prices or consumer behavior.

Overall, NNs possess versatile applications and serve as valuable tools across various industries. Their ability to learn from data and identify trends makes them highly beneficial in a wide range of domains.

## 1.4 Chip Design for NNs

The process of generating ICs that are optimised for neural network (NNs) processing activities are known as NN chip design. By carrying out numerous computations in parallel, these chips are made to speed up the analysis of neural network algorithms.

The development of NNs chips is a complex process that involves both hardware and software elements. The physical layout of the chip involves placing transistors, logic gates, and memory cells. On the other hand, developing the algorithms used to program the device falls under the responsibility of the software components.

Field-programmable gate arrays (FPGAs), which enable testing and architecture modifications before manufacturing, are frequently used to design NN chips. Once the chip's design is finalized, it undergoes a manufacturing process known as photolithography, which involves etching the chip's intricate pattern onto a silicon wafer.

Due to the growing need for quicker and more effective neural network pro-

cessing, the design of NN chips is a crucial field of research. These chips have uses in many different industries, including driverless vehicles, natural language processing, and picture and audio recognition.

## 1.5 NNs as Co-Processor

In some applications, NNs can be a co-processor (see Figure 1.4) to offload computation from the primary processor. This is especially helpful in applications where a real-time reaction is essential or where the central processor's processing capability is constrained.

Applications for processing images and videos are one instance of using NNs as a coprocessor. To increase the speed of the processing of this data, NNs can be trained to recognise patterns in photos and videos. This can help picture recognition, and other computer vision tasks run more quickly and accurately. Applications for natural language processing (NLP) are yet another illustration. The processing of vast amounts of text data, such as language translation, sentiment analysis, and text classification, can be sped up using NNs. This can aid in enhancing the efficiency of NLP tasks by increasing their speed and accuracy.

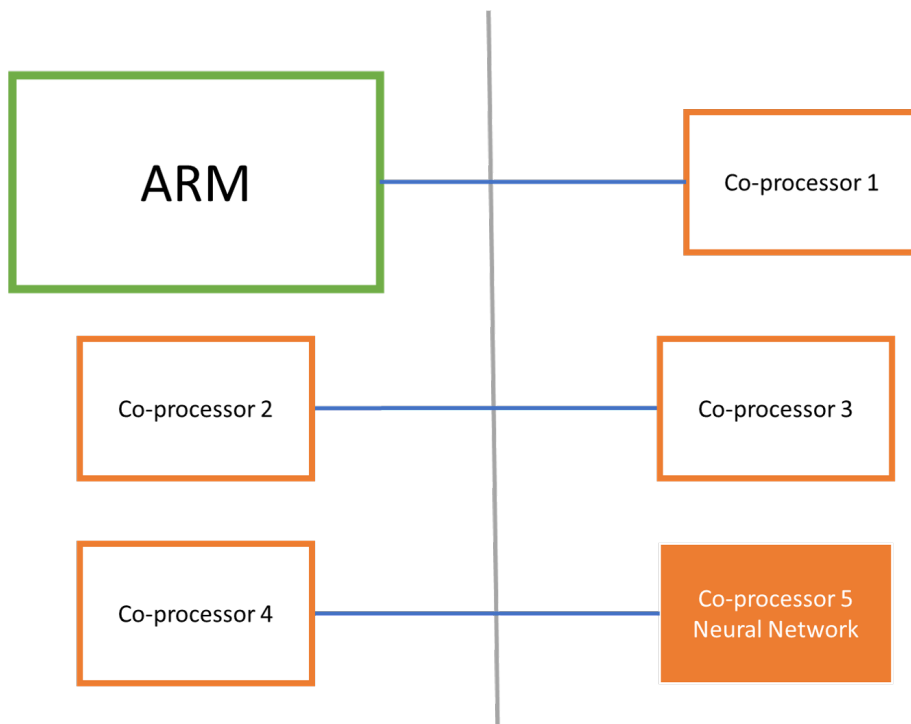


Figure 1.4: Schematically illustrates the use of a NNs processor as a co-processor.

In conclusion, by offloading work off the primary processor, the use of NNs as coprocessors can aid in enhancing the performance of some applications. This

can be especially helpful in applications that demand a quick reaction or a lot of processing power.

## 1.6 Aims and Objectives

In recent years, the field of artificial intelligence has witnessed remarkable advancements, particularly in the realm of NNs [16]. Neural networks, with their ability to learn and adapt, have revolutionized numerous domains, from computer vision to natural language processing [3]. However, their potential is often limited by the constraints of traditional processing architectures. By harnessing the power of neural networks as co-processors, we can unlock unprecedented opportunities for accelerated computations, enhanced pattern recognition, and optimized system design.

The integration of neural networks as co-processors in computing systems presents a paradigm shift in leveraging AI capabilities. By investigating the specific benefits, we aim to provide a comprehensive understanding of why neural networks as co-processors are essential for accelerating computations and advancing system design.

**Enhanced Computational Performance:** The incorporation of neural networks as co-processors enables the offloading of computationally intensive tasks from the primary processor, thereby reducing the overall processing time. Neural networks excel at parallel processing, allowing for the efficient execution of complex algorithms, such as image recognition, speech synthesis, and natural language understanding. By distributing computational workloads across multiple processors, system performance is significantly enhanced, leading to faster results and improved user experience.

**Accelerated Pattern Recognition:** Neural networks possess an innate ability to recognize complex patterns and extract meaningful information from large datasets. By leveraging neural networks as co-processors, we can leverage their parallel processing capabilities to expedite pattern recognition tasks. Whether it is identifying objects in images, analyzing financial data, or detecting anomalies in sensor readings, neural networks can perform these tasks with remarkable speed and accuracy. The integration of neural networks as co-processors offers unparalleled potential for real-time decision-making in applications that require rapid and precise pattern recognition.

**Optimal System Design:** Designing hardware architectures solely based on traditional processing units can be suboptimal for AI-centric applications. Neu-



ral networks have unique architectural requirements and specialized instructions that are distinct from conventional processors. By incorporating neural networks as co-processors, we can design specialized hardware that is tailored to the specific needs of neural network computations. This leads to improved energy efficiency, reduced memory bandwidth requirements, and better utilization of computational resources. Moreover, the flexibility provided by co-processors allows for seamless integration of evolving neural network models, ensuring compatibility with future advancements in AI technology.

In summary, the integration of neural networks as co-processors brings forth a host of advantages that revolutionize computational performance and system design. From enhanced computational speed to accelerated pattern recognition and optimized system architectures, neural networks empower us to overcome the limitations of traditional processing units. By embracing neural networks as co-processors, we unlock a new era of AI-driven applications with improved efficiency, accuracy, and the potential for groundbreaking innovations. Chapter 2 presents a literature survey on ML architecture, examining the various modules employed within these architectures. Chapter 3 focuses on the implementation of the surveyed ML architecture. We describe the ASIC design flow in Chapter 4. Thesis work is summarized in Chapter 5.

## CHAPTER 2

# Review of Neural Network Architectures

Over the past few years, the field of VLSI architecture for neural networks has witnessed significant advancements, fueled by the increasing demand for efficient and powerful hardware implementations of artificial intelligence algorithms. As neural networks become a cornerstone of numerous applications, ranging from image recognition to natural language processing, researchers have focused their efforts on developing novel architectures to address the challenges of processing large-scale neural networks with high computational demands.

This chapter presents a comprehensive literature review that explores various aspects of VLSI architecture for neural networks, including activation functions and modules related to microarchitecture. Through an extensive survey of research papers, we delve into the state-of-the-art techniques and methodologies employed by researchers to enhance the efficiency and performance of neural network hardware.

## 2.1 Review of Adder Design

Numerous VLSI architectures have been proposed over the years to improve the area efficiency of adders, considering their essential role in modern digital systems. This section aims to explore and compare the area characteristics of various adder architectures, shedding light on their strengths and limitations. Following table shows that CIA is better choice the others

### 2.1.1 Comparison of Adders on Basis of Area, Power, and Delay

Standard cells and FPGAs (Field Programmable Gate Arrays) are examples of cell-based design methodologies. These techniques, simultaneously with flexible hardware synthesis, are vital for high productivity in ASIC (Application Specific Integrated Circuit) design. With  $O(n)$  area and  $O(n)$  latency, the RCA (Ripple

Carry Adder) is the simplest but slowest adder, where  $n$  is the operand size in bits.

CSA (Carry Select Adder) has an  $O(\log n)$  latency and an  $O(n)$  area. In an  $n$ -bit adder,  $n/2$  multiplexers are chosen using a carry signal.

The document (particularly for this subsection) studied outlines the adder topology design. The following adder structures are employed in that work: CIA (Carry Increment Adder), CSkA (Carry Skip Adder), CBA (Carry Bypass Adder), and CSA are all examples of ripple adders [17]. One drawback of RCA adder is that its latency increases proportionally with the length of the numbers being added, which limits its efficiency when handling large bit numbers [18].

The performance of the RCA is constrained as the number of bits, increases since the ripple carry adder's latency, is directly proportional to number of bits. In order to establish an impartial testing environment, the simulations have been conducted using a comprehensive input signal pattern that covers every probable transition for all the adders. Its frequencies span from 10 to 500MHz, and its input and output capacitance are each set to 10pf. Adder topologies are assessed based on their ability to withstand challenges related to area, latency, and power consumption. According to the power distribution graph, the carry save adder experience the most power dissipation. CIA and RCA have the lowest power dissipation. CIA, and CSA both have the same minimum latency [19]. In this work, thorough investigations of adder topologies with 120 nm CMOS technology have been done. Area, latency, and power dissipation were used for the comparison. The transistor-level design has taken parasitic layout effects into account. The investigations include performance analysis, simulation results, and comparisons. CLA and CIA, which are apt for high-performance circuits, provide the optimum balance between area, delay, and power dissipation, according to the findings of the analysis.

### **Conclusion for adder design**

The RCA, CSkA, and CBA with the fewest gates, and the greatest delay are the most basic adder topologies appropriate for low-power applications [20]. From the table given below it can concluded that, Carry Increment Adder achieves better performance in terms of area and delay compared to that of other adder topologies [17].

After a thorough research, it has been determined that the CIA demonstrates excellent performance in terms of area efficiency. Consequently, the CIA has been selected as the preferred choice for this particular work.

|   | Design                 | Area(LUTs) | Area(Slices) | Delay(ns) |
|---|------------------------|------------|--------------|-----------|
| 1 | Ripple Carry Adder     | 8          | 5            | 2.191     |
| 2 | Carry Skip Adder       | 8          | 6            | 2.267     |
| 3 | Carry Increment Adder  | 8          | 5            | 1.907     |
| 4 | Carry Look Ahead Adder | 10         | 5            | 2.266     |
| 5 | Carry Save Adder       | 13         | 9            | 1.433     |
| 6 | Carry Select Adder     | 8          | 5            | 2.588     |
| 7 | Carry Bypass Adder     | 12         | 6            | 3.160     |

Table 2.1: Performance analysis of various Adders[17] .

## 2.2 Review of Multiplier Design

Subsequent subsection describes the extensive research conducted on multiplier architecture. These findings are summarized and analyzed in section below.

### 2.2.1 Comparative Analysis of Several Multipliers

First document discusses the development of a generic HDL code for a fast Baugh Wooley multiplier. The conventional Baugh Wooley multiplier architecture is modified by replacing the ripple carry adder in the final stage with a Linear Carry Select Adder. The post-synthesis results of the modified architecture are compared with the conventional Baugh Wooley architecture and the default multiplier architecture generated by a synthesis tool. The comparison is based on various characteristic parameters such as propagation delay, area, and power consumption. The results demonstrate that the modified Baugh Wooley architecture is faster than both the conventional architecture and the default synthesis-generated architecture, particularly for larger operand sizes. This improvement in speed is significant [21] . The paper provides a detailed explanation of the Baugh Wooley multiplication algorithm and outlines the steps to create a generic HDL code for a fast NxN Baugh Wooley multiplier. This approach allows designers to easily generate Baugh Wooley multipliers of any operand size by simply modifying the operand size parameter, thereby reducing design efforts. In conclusion, the modified Baugh Wooley architecture offers superior performance in terms of both delay and power-delay product compared to the conventional architecture and the default synthesis-generated architecture. It presents a valuable solution for digital ASTC designers seeking efficient and fast Baugh Wooley multipliers of different sizes [21] .

The Modified Baugh-Wooley (MBW) multiplier approach is widely employed in digital signal processing and hardware design to achieve high-speed multipli-

cation tasks. It offers several advantages over conventional multiplication methods such as the array multiplier. Research in this field has identified multiple benefits associated with the MBW.

The MBW multiplier approach exhibits lower complexity in implementation compared to alternative methods. It achieves this by reducing the number of partial product terms and overall complexity in the multiplication process. This is accomplished by breaking down incomplete products into smaller sections and employing advanced algorithms to generate the final result. The reduction in complexity has several implications, including decreased hardware requirements and reduced power consumption [42].

To elaborate on the previous statement, the reduction in complexity in the MBW multiplier leads to a decrease in hardware requirements. In traditional multiplication methods like the array multiplier, the number of partial products grows exponentially with the bit size of the operands, resulting in a larger hardware footprint. However, the MBW multiplier breaks down these partial products into smaller chunks, reducing the number of required components. This not only saves physical space but also leads to a more efficient utilization of hardware resources. Furthermore, the decreased complexity in the MBW multiplier results in reduced power consumption. The simplified circuitry and reduced number of partial products contribute to lower power requirements during the multiplication process. This is particularly beneficial in power-constrained applications, such as mobile devices or embedded systems, where minimizing power consumption is crucial to enhance battery life and overall energy efficiency.

The MBW multiplier is built to perform multiplication operations at fast speeds. It takes advantage of effective parallel processing strategies, including carry-lookahead and carry-save adders, to speed up the calculation of the result. The MBW multiplier may lower the multiplication delay and enhance system performance as a whole. Effective use of hardware resources: By lowering the required number of logic gates for the multiplication operation, the MBW multiplier effectively uses hardware resources. In order to optimise the design, symmetry and symmetry-breaking techniques are used. The MBW multiplier provides improved resource allocation and may be more cost-effective in chip space and power consumption by effectively using hardware.

Due to the MBW multiplier's great scalability, expanding it to support multipliers with parallel processing or larger word sizes is simple. It is apt for a wide range of applications, including image processing, digital signal processing, and communication systems, due to its scalability.

The Baugh-Wooley multiplier, Modified Baugh-Wooley (MBW) multiplier, and Wallace multiplier are compared in terms of area, depending on the implementation details and the technology used. Because the MBW multiplier is an advance over the Baugh-Wooley method but does not significantly increase hardware complexity, the Baugh-Wooley multiplier and the MBW multiplier often have similar area requirements [22]. Both algorithms follow a similar methodology and can achieve similar area efficiency. In contrast, the Wallace multiplier often needs more space than the Baugh-Wooley and MBW multipliers. The Wallace multiplier is renowned for its high-speed multiplication, but it does it by employing a tree-like structure with many partial products and complicated hardware. Less complicated hardware: The Baugh Wooley multiplier can be modified to lower the hardware requirements. Lower power consumption may also be achieved through simplified technology and increased effectiveness. This is crucial for battery-operated and mobile devices, where power efficiency is a major issue. The MBW multiplier may be improved such that it is more readily scalable and can multiply values with wider bit widths effectively. Applications that need high-precision computations or work with big data sets might benefit from this flexibility. It is possible to create a modified Baugh Wooley multiplier that works with a variety of number representations, such as signed numbers and floating-point formats. The multiplier may be utilised in a larger variety of applications thanks to its adaptability.

When considering area efficiency as the primary concern in multiplier design, the Modified Baugh Wooley (MBW) multiplier typically outperforms the Wallace multiplier. The area efficiency of a multiplier refers to how effectively it utilizes hardware resources to perform the multiplication operation.

The MBW multiplier is designed based on the concept of Modified Booth Encoding (MBE). It leverages the properties of MBE to reduce the number of partial product terms generated during multiplication. This reduction in partial product terms leads to a decrease in the overall hardware requirements, resulting in improved area efficiency.

In terms of area efficiency, the MBW multiplier offers advantages over the Wallace multiplier due to its ability to reduce the number of partial product terms and minimize the associated hardware requirements. This reduction in hardware translates to a more compact and efficient design. Hence, the MBW multiplier is often more area-efficient when compared to the Wallace multiplier if area efficiency is the primary consideration.

## 2.3 Review of Activation Function Design

Activation functions play a crucial role in shaping the behavior and expressive power of neural networks. In this section, we explore a wide range of papers that investigate different activation functions and their impact on VLSI architectures. From traditional functions such as sigmoid and tanh to more recent advances like rectified linear units (ReLU). We analyze their properties, computational complexity, and suitability for hardware implementations [23].

### 2.3.1 Activation Functions for Neural Networks

With the emergence of various deep learning (DL) architectures, NNs have proven to be effective in addressing complex real-world challenges across different sectors. Activation functions (AFs) play a crucial role in these NNs architectures, facilitating computations between hidden layers and output layers to achieve state-of-the-art results. This work aims to provide insights into the current usage of activation functions in deep learning applications and highlights recent advancements in this field. The unique contribution of the work lies in its comprehensive compilation of the majority of activation functions utilized in NNs, as well as its analysis of their current utilization trends in real-world deep learning implementations compared to cutting-edge research findings. This compilation aims to facilitate the selection of the most suitable activation function for specific applications that are ready for deployment. It stands out as the first article to compare the practical trends in activation function usage with the findings from existing literature in deep learning research. This topicality is significant considering that most research publications on activation functions primarily focus on similar efforts and outcomes.

This section provides an overview of the activation functions currently employed in deep learning applications and highlights recent trends in their utilization in real-world deep learning deployments, contrasting them with cutting-edge research findings. This study identified several important activation functions used in deep learning applications, including variations of the original ReLU (Rectified Linear Unit). Nwankpa and colleagues also provided a concise overview of deep learning and activation mechanisms [24]. Activation functions play a crucial role in neural networks by computing the weighted sum of inputs and biases to determine if a neuron should fire. The Softmax function is commonly used in multi-class models to provide probabilities for each class, with the highest probability assigned to the target class. To mitigate overfitting in neural networks, the

rectified linear unit (ReLU) activation function has been chosen.

The researchers affirm that their findings align with previous research in the field, emphasizing that activation functions regulate neural network outputs and can be linear or non-linear, often referred to as transfer functions. They suggest that future studies should explore these cutting-edge functions on successful architectures using standardized datasets to evaluate potential performance improvements [25]. Non-linear modules in deep learning methods transform initial inputs into more abstract representations, often involving complex learned functions. This study offers a comprehensive overview of activation functions used in deep learning and highlights current trends based on unpublished cutting-edge research findings.

Activation functions have the ability to enhance pattern learning in data, enabling automated feature detection. This justifies their utilization in hidden layers of neural networks and their applicability across different domains. Looking ahead, compounded activation functions hold promise for future research in this area, as activation functions have evolved over time.

### **2.3.2 ReLU, Linear Saturated, and Other Activation Functions: Comparisons**

Stursa *et.al.*, compared two types of linear activation functions used in feedforward neural networks for approximating nonlinear systems: symmetric linear saturated function and rectifier linear unit (ReLU) function. The authors aim to evaluate their performance, convergence speed, and computational demands [26]. The experiments are conducted using one hidden layer with varying numbers of neurons, and strict criteria are applied, including the use of the Levenberg-Marquardt algorithm for training and the Nguyen-Widrow algorithm for initialization. Three benchmark nonlinear systems are selected for approximation, and training data is generated by applying a colored input signal, and computing the corresponding output. The comparison is based on the convergence speed for a fixed error function value and the performance over a constant number of epochs. The results show that both the activation functions exhibit similar performance and convergence speed, with only small differences observed. Although the symmetric linear saturated activation function yields a lower median error function value across different numbers of neurons, the ReLU function proves to be capable of effectively modeling nonlinear systems as well. They also discussed the complexity of training neural networks with linear activation functions [26]. It concludes that the ReLU function, due to its comparable performance and po-



tential computational complexity decrease, can be a suitable choice for nonlinear system modeling in process control and automation applications.

Another study compared the effects of different activation functions (AFs) in Artificial Neural Networks (ANNs) on regression and classification tasks using various datasets. A total of 11 AFs, including 10 commonly used in the literature and a newly proposed Square function, were evaluated. Three different ANN architectures were employed for each dataset. The study examined the success rates in test data and the duration of training for each AF [27]. The results showed that the choice of AF significantly impacted the performance of ANNs across different datasets and architectures. ReLU emerged as the most successful AF for general purposes. In image datasets, the Square function performed better than other AFs when combined with ReLU in architectures involving convolutional layers. Furthermore, the study highlighted the importance of batch normalization before applying AFs. By keeping the inputs to AFs within certain bounds, batch normalization eliminated the limitations of both limited and unlimited AFs. It was found that batch normalization positively affected the results, especially for unlimited AFs.

Overall, the findings emphasize the significance of selecting appropriate AFs in ANNs, as they can greatly influence regression and classification performance, particularly in different types of datasets. ReLU is generally recommended as a reliable choice, while the Square function shows promise for image datasets when used alongside ReLU. The use of batch normalization is also beneficial for improving results, particularly with unlimited AFs [27].

### **2.3.3 Conclusion for Activation function design**

ReLU has the ability to create sparse activations, which means it can set certain neuron outputs to zero, making the data representation more effective. ReLU can also experience a condition known as dying ReLU, in which certain neurons remain permanently inactive (outputting zero) due to this issue.

The sigmoid function is denoted by the formula  $f(x) = 1/(1 + e(-x))$ , where  $x$  represents the function's input. The input is mapped to the range  $[0, 1]$  via a smooth, S-shaped curve. In binary classification situations, where the objective is to forecast probabilities between 0 and 1, sigmoid activation is frequently utilised in the output layer. The input is condensed to a narrow range, making it appropriate for modelling probabilities.

However, sigmoid experiences the Vanishing gradient problem, particularly in deep networks, where the gradients can shrink to extremely tiny values and cause

training to converge slowly. When inputs are far from zero, sigmoid activations can also result in the "gradient saturation" issue, which slows learning by bringing gradients near to zero. In binary classification situations, where the objective is to forecast probabilities between 0 and 1, sigmoid activation is frequently utilised in the output layer. It reduces the input's range, which makes it appropriate for modelling probabilities.

## 2.3.4 Comparison of ReLU and Sigmoid

### 2.3.4.1 Advantages of ReLU over Sigmoid:

Although Rectified Linear Unit (ReLU) and Sigmoid are both frequently used activation functions in neural networks, they each offer unique qualities and benefits. ReLU has the following benefits over sigmoid:

1. ReLU is a simpler function than the sigmoid function since it just requires a threshold operation and no exponential computations, making it more efficient. ReLU is more efficient to calculate thanks to its simplicity, especially when working with large-scale neural networks [24].

2. For inputs that are noticeably positive or negative, the gradient becomes extremely modest since sigmoid activation saturates at the function's extreme ends. The vanishing gradient problem, which results in sluggish convergence and challenges while training deep neural networks, might be brought on by this saturation. ReLU, on the other hand, is linear for positive inputs and does not experience saturation. The gradient can flow more freely as a result, hastening convergence.

3. The Rectified Linear Unit (ReLU) activation function exhibits sparsity, where it selectively activates a subset of neurons while keeping others dormant. This characteristic enables a more efficient representation of data within a neural network. By reducing unnecessary calculations, sparse activation decreases the overall complexity of the network and helps mitigate overfitting by promoting generalization.

4. Avoiding the exploding gradient problem: Because of the sigmoid function's constrained range, it is possible for gradients to explode during backpropagation, leading to unpredictable results. Since ReLU lacks an upper constraint, gradients can spread more readily without leading to instability.

5. ReLU has been claimed to be more physiologically realistic as an activation function, more closely approximating the behaviour of actual neurons. Instead of a sigmoid-shaped activation, the action potential of biological neurons is simply a thresholding operation, akin to ReLU.

6. Due to its simplicity, computational effectiveness, and capacity to overcome some of sigmoid activation's drawbacks, ReLU is a popular option in many applications. But sigmoid activation still has benefits of its own, including offering a probabilistic interpretation and being helpful for jobs where outputs must be constrained between 0 and 1.

### 2.3.4.2 Vanishing Gradient Problem

When training deep neural networks with specific activation functions, such as the sigmoid function, an issue known as the vanishing gradient problem may arise. The definition of the sigmoid function, sometimes referred to as the logistic function, is:  $f(x) = 1/(1 + \exp(-x))$ . The sigmoid function's derivative is tiny at the function's extreme regions, which leads to the vanishing gradient issue [28]. The sigmoid function's derivative may be computed as follows:  $f'(x) = f(x) * (1 - f(x))$

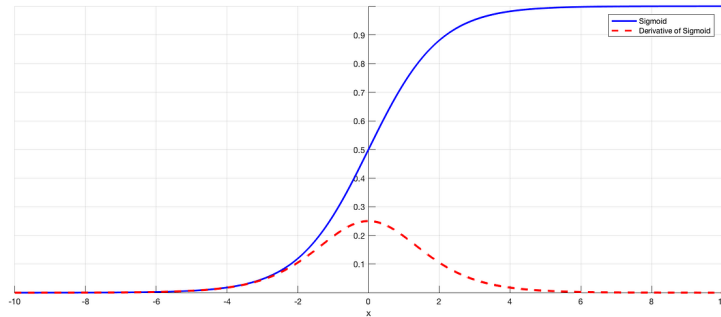


Figure 2.1: Depicts Sigmoid function and its derivative [29].

The sigmoid function approaches either 1 or 0, producing a tiny derivative, as the input goes away from zero in either the positive or negative direction as shown in Figure 2.1 [29]. The modest gradients that are back-propagated during training are doubled at each layer, which might lead to the gradients decreasing exponentially as they go backward through the network [30]. Hence, the early layers of the network experience extremely small gradients, which significantly decelerates convergence or even prevents it altogether.

Deep neural networks with sigmoid activation may find it challenging to learn long-range relationships and recognise complicated patterns in the input due to the vanishing gradient problem [31]. Activation functions that do not have the vanishing gradient problem, such as rectified linear units (ReLU) or its derivatives such as Leaky ReLU and Parametric ReLU, are preferred in practise as a result of this restriction. These alternate activation functions enable for more efficient gra-

dient propagation during back-propagation since they feature non-zero gradients for positive inputs. Consequently, sparse activation functions help address the challenge of vanishing gradients and enable the training of deeper neural networks.

## **2.4 Review of Neural Networks Design**

Recent years have seen substantial advancement in Very Large Scale Integration (VLSI) architecture for Neural Networks (NNs). This section comprehensively summarises current research on VLSI design for NNs, emphasising significant advancements and difficulties encountered. The section discusses many facets of hardware acceleration, energy efficiency, memory organisation, and algorithmic improvements in VLSI design for NNs. Furthermore, it examines future research possibilities and emphasises how these breakthroughs have affected practical applications. Numerous applications, including robotics and autonomous vehicles, as well as natural language processing and image identification, have been transformed by neural networks [32]. However, there is an increasing need for effective hardware designs to expedite the execution of NNs as they become more complicated and demanding in terms of computing needs [33]. Multiple components are integrated on a single chip in VLSI design, which improves performance, energy efficiency, and scalability.

### **2.4.1 Artificial Neural Networks**

#### **CNNs**

Convolutional Neural Networks (CNNs) are commonly employed for image and video processing tasks. Current research efforts have been dedicated to enhancing CNN designs by incorporating techniques such as parallel processing, and weight pruning and hardware architectures like systolic arrays that are used to implement certain algorithms efficiently [34]. These optimization approaches have led to significant advancements in terms of accelerated computation and reduced power consumption in CNN implementations.

#### **RNNs**

Recurrent Neural Networks (RNNs) are beneficial for sequential data processing tasks such as speech recognition and language translation. Recent research has

prioritized the development of hardware acceleration techniques, such as folding, pipelining, and parallel processing, to enable real-time applications while minimizing power consumption [35]. These advancements aim to enhance the efficiency and performance of RNN-based systems in various language-related tasks.

## **SNNs**

Spiking Neural Networks (SNNs) are well-suited for tasks that involve event-driven processing, as they closely emulate the behavior of biological neurons. To enhance the processing capabilities and energy efficiency of SNNs, researchers have introduced innovative VLSI architectures. These include spike-based learning mechanisms and event-driven hardware designs. These novel approaches aim to optimize the functionality of SNNs, enabling more effective processing while conserving energy resources.

### **2.4.2 Energy-Efficient VLSI Architectures**

The battery life of mobile devices and the amount of power used by data centres are both directly impacted by energy efficiency, making it a crucial factor in the design of VLSI for NNs [36]. In order to reduce energy consumption while preserving acceptable accuracy levels, researchers have investigated strategies including approximation computing, voltage scaling, clock gating, low-power digital circuits, and approximate computing.

### **2.4.3 Memory Organization Techniques**

In NN calculations, memory access and data transfer are the main bottlenecks. Researchers have suggested a variety of memory organisation strategies to overcome this, including data reuse, sparsity exploitation, and hierarchical memory architectures [37]. These methods seek to improve data storage efficiency, decrease memory access latency, and boost performance as a whole.

### **2.4.4 Algorithmic Innovations**

Along with hardware improvements, algorithmic advancements have also had a big impact on NN VLSI architecture [38]. To lessen the complexity of NN models without significantly sacrificing accuracy, researchers have investigated meth-

ods including quantization, pruning, and compression. These methods allow for resource-constrained VLSI chips to be implemented effectively.

### **2.4.5 Impact on Real-World Applications**

Numerous real-world applications have been significantly impacted by the developments in NN VLSI architecture. They have aided in the development of voice assistants that can comprehend natural language, real-time object identification in driverless cars [40], picture analysis for medical diagnosis, and many more fields [39]. The breadth of ML applications has been broadened, and user experiences have been improved thanks to the increased efficiency and performance of VLSI-based NN implementations.

### **2.4.6 Conclusions for Neural Network Design**

The deployment of NNs in a wide variety of applications is now possible because of impressive breakthroughs in the field of VLSI design for neural networks. Researchers have significantly improved the field through advances in memory organization, hardware acceleration methods, energy-efficient designs, and algorithmic optimizations. The future of ML will continue to be shaped by more studies in this field, which will encourage the creation of hardware designs for neural networks that are smarter, quicker, and more energy-efficient.

## CHAPTER 3

# Implementation of Neural Network Architectures

In this chapter, we discuss the implementation of all the three architectures, including aspects of micro-architecture, SISO, MISO and MIMO. Implementations of adder and multiplier are also illustrated.

### 3.1 Implementation of Adder

Adders are the fundamental building blocks of complicated digital circuits. The layout of this entire adder unit determines the circuit's performance. One of the key performance factors for many digital circuits is the circuit's operating speed, which ultimately depends on the basic adder unit's delay. The delay of the adder circuit has been the subject of extensive investigation. This research study presents an improved version of the carry increment adder (CIA) aimed at enhancing the circuit's delay performance [41]. The enhancement is made by replacing the ripple carry adder (RCA) used in the previous design of the CIA with a carry look adder (CLA). For comparison analysis, simulation research is conducted.

The speedier carry look ahead modules are suggested in this study as a substitute for the significantly slower ripple carry adder in a modified carry increment adder. Although we only implemented the design for two 8-bit binary additions in this study, it might be expanded to a higher-order adder circuit. Fast-evolving biomedical equipment, common wireless and mobile gadgets, and compact, portable devices use high-performance VLSI systems more frequently. Innumerable measures have been taken to decrease the ripple carry adder's latency, area, power usage, and other drawbacks. Architecture of carry increment adder is shown in Figure 3.1.

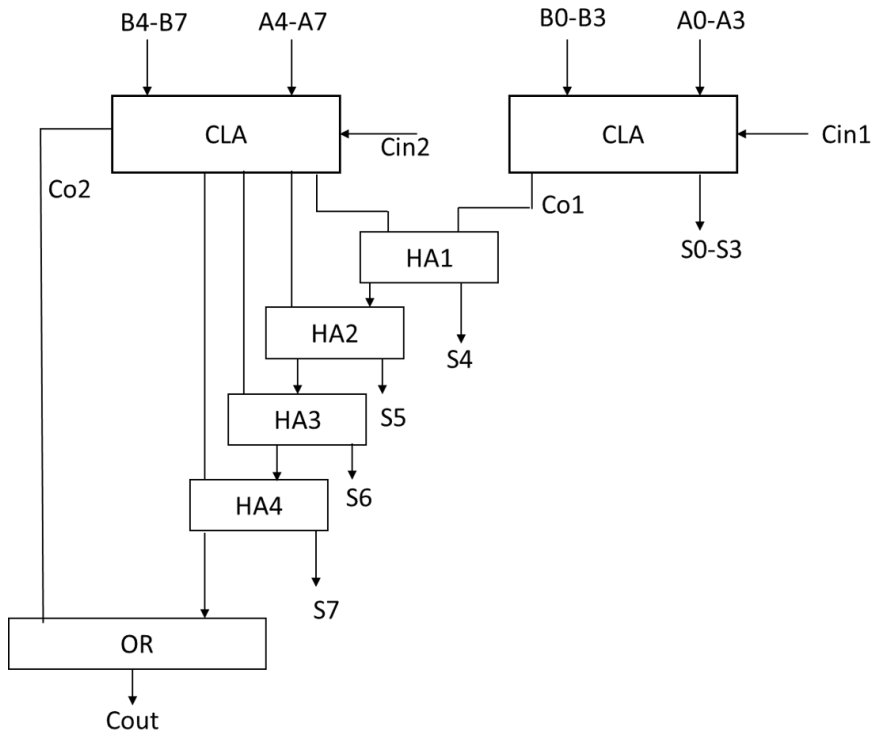


Figure 3.1: Depicted the architecture of a carry increment adder (CIA).

### 3.2 Implementation of Multiplier

The Baugh-Wooley multiplier is a type of parallel multiplier used in digital circuits, specifically designed for signed binary numbers. It is known for its simplicity and efficient implementation in hardware. The modified Baugh-Wooley multiplier is a variation of the original algorithm that aims to improve its performance and reduce the number of partial products generated during the multiplication process. The modified Baugh-Wooley multiplier is particularly well-suited for Very Large Scale Integration (VLSI) architecture, as it can be implemented efficiently in hardware. Architecture of MBW is shown in Figure 3.2. Here’s a brief explanation of the VLSI architecture implementation of the modified Baugh-Wooley multiplier:

**Input and Data Representation:**The modified Baugh-Wooley multiplier takes two signed binary numbers A and B as inputs. These numbers are typically represented using two’s complement notation. The size of the inputs will determine the number of bits in each input and the resulting product. For instance, if the inputs are n-bit numbers, the product will be a 2n-bit number.

**Partitioning the Inputs:** The inputs A and B are partitioned into three segments each: positive, negative, and zero segments. For each input, these segments di-



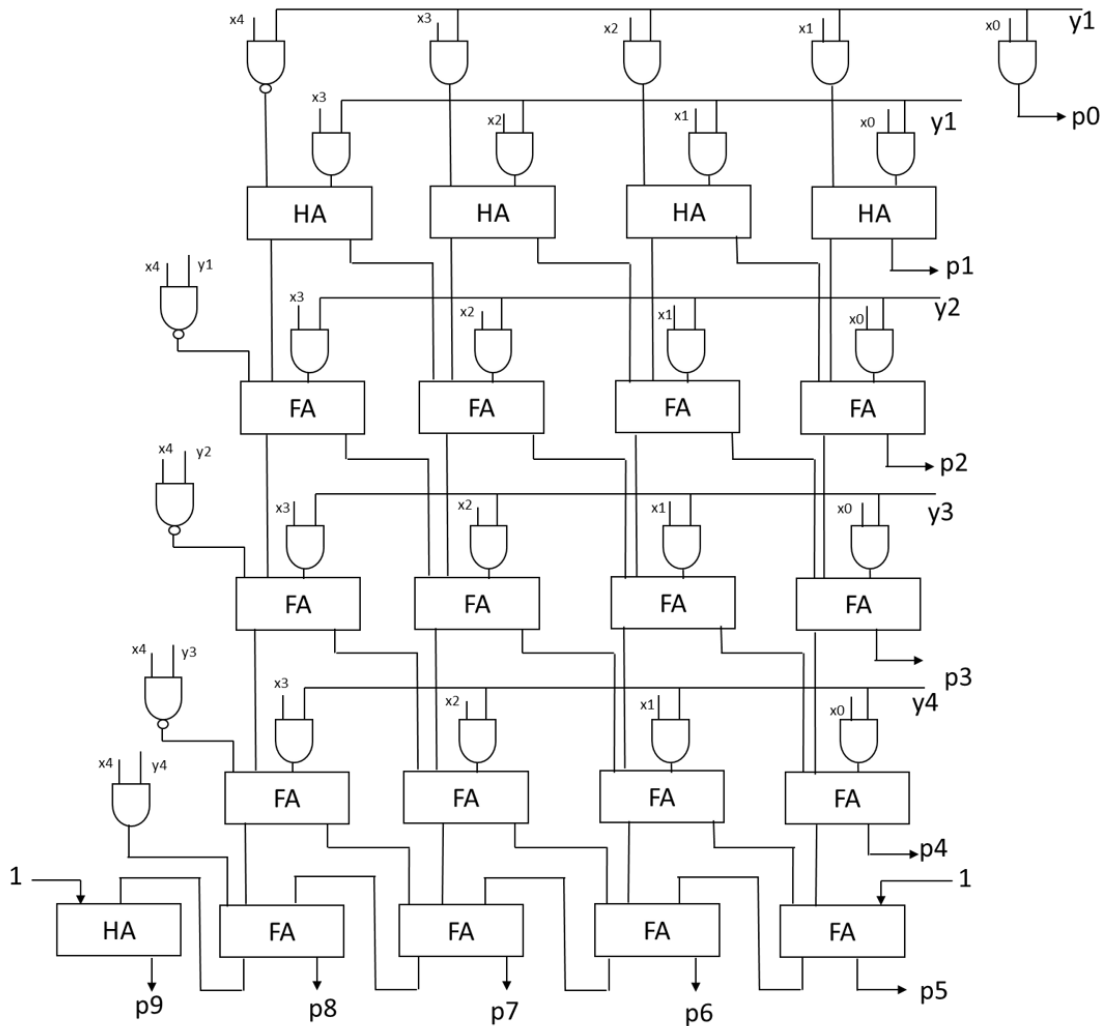


Figure 3.2: Architecture of Multiplier.

vide the bits based on the sign and value of the bit (positive, negative, or zero).  
 Generation of Partial Products: In the traditional Baugh-Wooley algorithm, the number of partial products generated is  $2n^2$  (where  $n$  is the number of bits in each input). The modified Baugh-Wooley algorithm aims to reduce this number for better efficiency. In the modified Baugh-Wooley multiplier, the partial products are generated only for the non-zero segments of both inputs. This reduces the number of partial products significantly compared to the traditional method.  
 Partial Product Reduction: After generating the partial products, the next step is to sum them up to obtain the final product. The partial products are grouped and added together in an efficient manner to produce the final result. The reduction process involves techniques such as carry-save addition, carry-propagate addition, or Wallace tree-based adders to optimize the addition process and minimize the critical path delay.  
 Overflow Handling: Care must be taken to handle overflow

and ensure the correct representation of the product, especially in cases where the product requires more bits than the original input size. Output: The output of the modified Baugh-Wooley multiplier is the product of the two input numbers, represented in signed binary format.

### 3.3 Implementation of SISO NN Architecture

The SISO (Single-Input Single-Output) neural network architecture is designed to take a single-input value and produce a single-output value. Figure 3.3 shows the architecture of a SISO NN. The input is fed into a multiplier, then multiplied by weight value which is predefined in memory, and then the resulting output from the multiplier is then passed to an adder. In the adder, a bias value  $b$  is added to the output of the multiplier. The resulting value from the adder is then forwarded to the output through a transfer function.

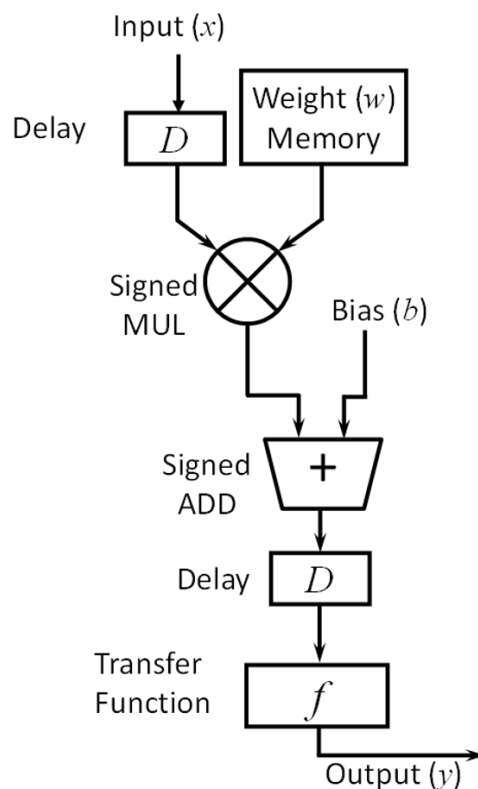


Figure 3.3: SISO architecture.

For details, let's consider an input value, denoted as  $x$ , which is passed through a multiplier to be multiplied by a weight value,  $w$ . The resulting value from the multiplier is  $x * w$ . Next, this value is fed into an adder where a bias, represented

as  $b$ , is added to it. The output of the adder can be expressed as  $((x * w) + b)$ . Subsequently, the resulting value is passed through a transfer function, and if it meets a certain threshold condition, it is forwarded to the output. In summary, the output can be described as  $((x * w) + b)$ . Simulated waveform corresponding to above mentioned SISO architecture is depicted in Figure 3.4. This architecture is particularly useful when sequential processing of data is needed for 1-D signal processing such as gesture recognition, speech recognition, etc.

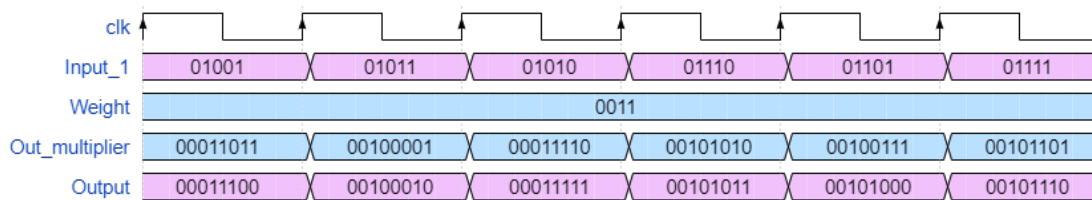


Figure 3.4: Depicted the simulated waveform obtained using SISO architecture.

### 3.4 Implementation of MISO NN Architecture

In this section, the implementation of a multiple-input, single-output (MISO) neural network design at the RTL level is demonstrated. The design takes into account the activation function,  $f$  to determine the signal that will advance. Both serial and parallel architectures can be utilized to construct the MISO architecture. In case of serial architecture, which is comparable to SISO architecture in 1-D data processing, more time is required to process data. However, when processing times are short, a parallel architecture is typically employed, allowing for the simultaneous processing of multiple inputs. The implementation of weights' memory is realized in the form of a Look-Up Table (LUT). The design is implemented at the RTL level using the Verilog hardware description language. Parallel Architecture of MISO is shown in Figure 3.5. Here  $x$ ,  $w$ , and  $b$  represent the input vector, weight matrix, and bias vector, respectively. Subscript value represents input/output number.

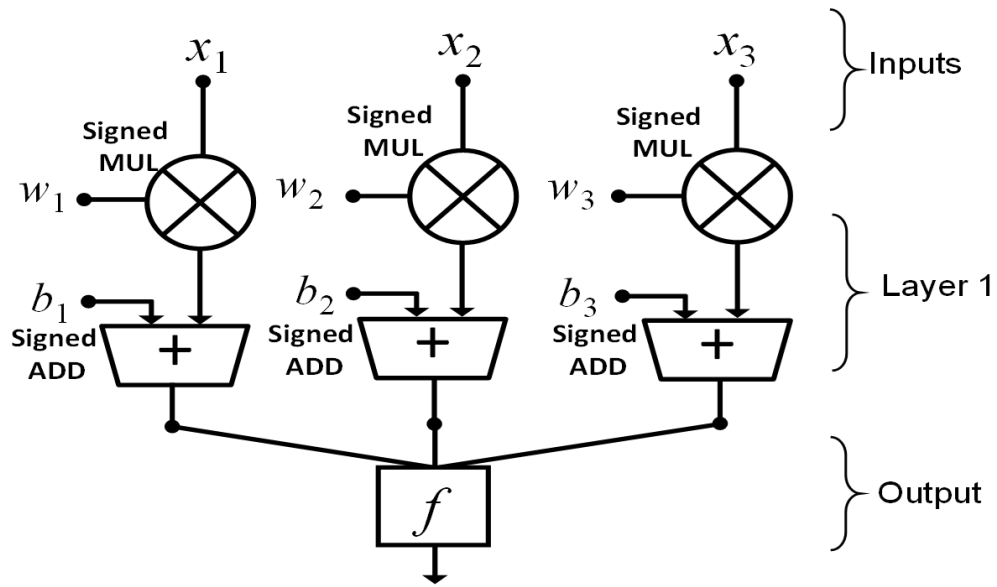


Figure 3.5: Architecture of MISO.

For more details, let's consider the input values  $x_1, x_2$ , and  $x_3$  which are fed to multiplier to get multiplied with the weight values say  $w_1, w_2$ , and  $w_3$ , respectively. Then the outputs of multiplier are  $(x_i * w_i)$ ,  $i$  ranging 1 to  $n$ . It goes to adder where we add a bias  $b$  to it so outputs of adder are  $(x_i * w_i) + b$ . It is then fed to activation function and if it fulfils the threshold condition it moves ahead. Now the highest value which is greater than zero is sent to the output. So output is  $(x_j * w_j) + b$ . as depicted with simulated waveform in Figure 3.6.

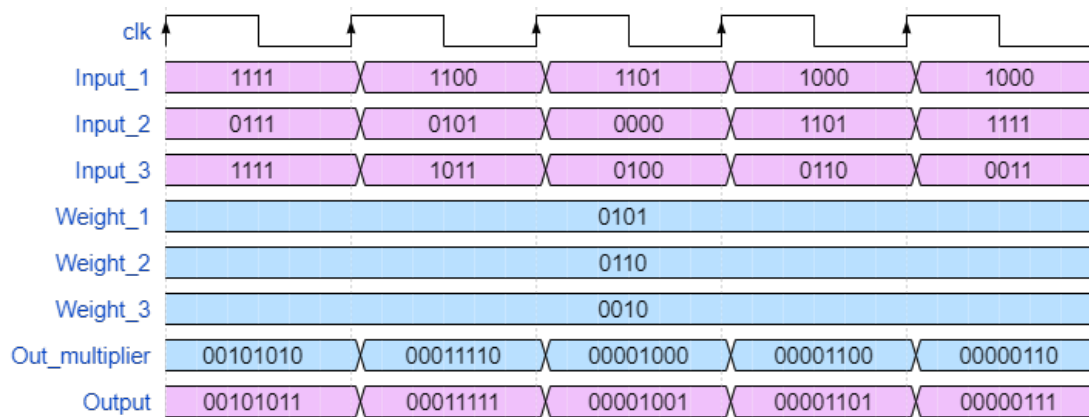


Figure 3.6: Simulated waveform obtained using MISO architecture.

### 3.5 Implementation of MIMO NN Architecture

MIMO (Multiple-Input Multiple-Output) networks can process many input streams simultaneously, such as video or picture frames captured from various angles or sensing modalities. This enables them to increase object identification precision by considering diverse viewpoints or sensor inputs. It can be also used for Natural Language Processing (NLP). To improve language comprehension and production, MIMO networks can handle a variety of information sources, including text, audio, and visual data. For instance, a MIMO network may use textual and contextual information to improve sentiment categorization in sentiment analysis.

Figure 3.7 depicts the architecture design of a MIMO neural network, which has two layers and eight outputs ( $out1, out2$ ) and inputs ( $x_1, x_2, \dots, x_8$ ). Weight memory is made up of input-corresponding weight values. To choose which value needs to be transmitted at the output, weight values and inputs are provided to the ReLU neuron. A value from the data memory will be supplied to the output in accordance with the ReLU value. As previously mentioned, the signed adder and signed multiplier make up the neuron. Other elements of the architecture, such as the activation function and neuron (which include signed multipliers and signed adders as discussed in Chapter 2), are identical to those in the MISO architecture.

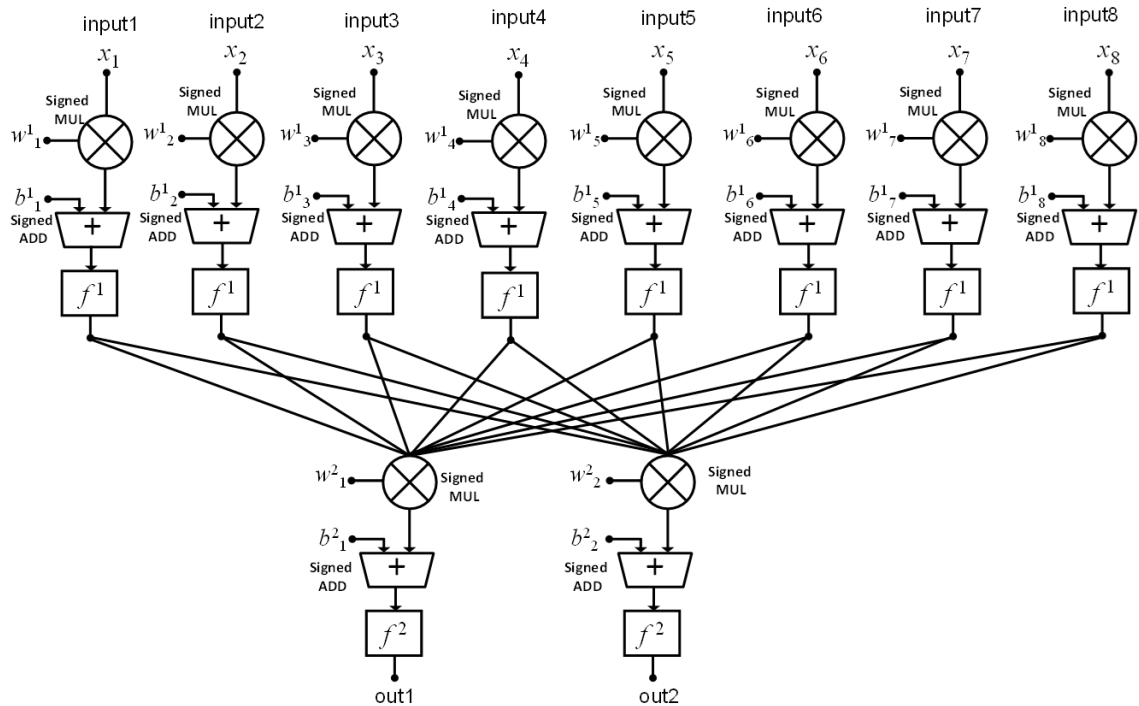


Figure 3.7: Depicted the developed MIMO Architecture.

For details, we considered a scenario where we have input values  $x_1, x_2, x_3, x_4, \dots, x_8$ . These inputs are passed through a multiplier, where they are multiplied with corresponding weight values  $w_1, w_2, w_3, w_4, \dots, w_8$ . The outputs of the multiplier can be represented as  $(x_1 * w_1), (x_2 * w_2), (x_3 * w_3), (x_4 * w_4), \dots, (x_8 * w_8)$ . These outputs are then fed into an adder, where a bias value  $b$  is added to each of them. Consequently, the outputs of the adder can be expressed as  $((x_1 * w_1) + b), ((x_2 * w_2) + b), ((x_3 * w_3) + b), ((x_4 * w_4) + b), \dots, ((x_8 * w_8) + b)$ . The next step involves passing these values through an activation function, where they are checked against a threshold condition. If the condition is met, the values proceed further in the process.

Finally, the two largest values, which are greater than zero, are selected and sent to the output. These values can be denoted as  $((x_j * w_j) + b)$  and  $((x_k * w_k) + b)$ . In summary, the input values are multiplied with corresponding weights, then added with a bias, and passed through an activation function. The two largest values meeting the threshold condition are chosen as the outputs. Figure 3.8 illustrates the simulated waveform generated by MIMO architecture.

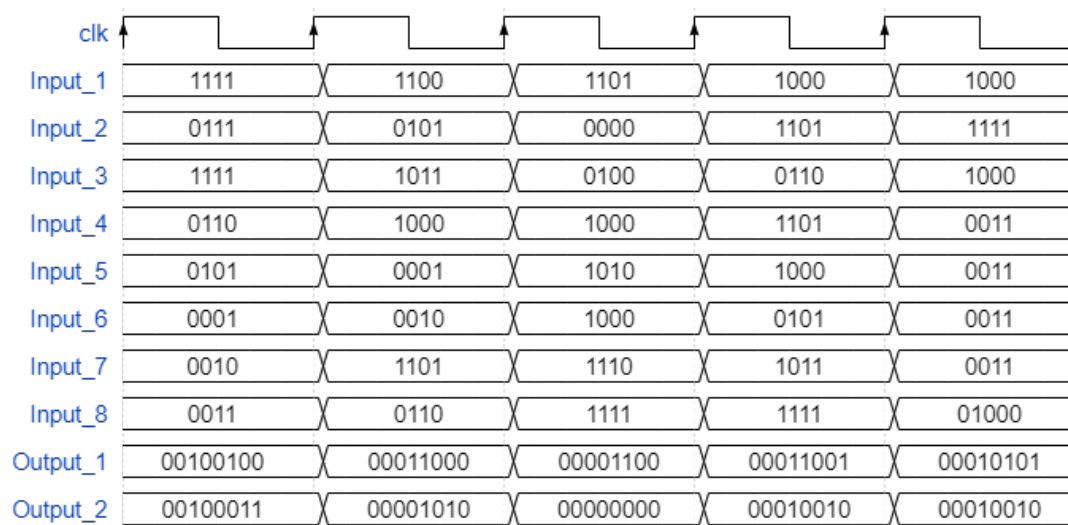


Figure 3.8: Simulated responses of MIMO architecture.

## CHAPTER 4

# Neural Networks Layout

As mentioned earlier, three neural network architectures have been implemented, covering the entire process from RTL (Register Transfer Level) to GDSII (Graphic Data System version II), utilizing open-source tools. The following discussion will delve into the design flow employed for tape-out generation, the tools utilized in this process, the tape-outs for each of the three architectures, and the corresponding parameters associated with them. Before delving into these details, let's first explore the concept of tape-out and the advantages of employing open-source technologies.

In the realm of VLSI (Very Large Scale Integration), the term tape-out refers to the physical design phase of an integrated circuit (IC). It involves the intricate positioning and routing of various components such as interconnects, capacitors, resistors, and transistors onto the surface of the chip. The tape-out design phase plays a critical role in determining the performance, power consumption, and reliability of the chip. It represents a crucial stage in the overall design process of integrated circuits (ICs). The tape-out design process begins with a circuit schematic or netlist, which specifies the logical connections between the components of the circuit. This serves as the initial step for the tape-out design.

Following that, the components are placed on the chip's surface, and the designer routes the interconnects between them to create a physical representation of the circuit.

## 4.1 Design Flow

**RTL Design:** The RTL description of the chip is first created using a hardware description language (HDL), such as Verilog or VHDL. This description provides a higher-level definition of the circuit's behaviour and functioning. The gate-level netlist is created by synthesising the RTL code. In order to develop a gate-level representation of the circuit, synthesis entails translating the RTL description to

a library of standard cells and optimising the design for area, power, and time. The location of the chip's primary components and their relative placements on the silicon die is determined by the floorplan, which is developed in this step. In floorplanning, variables including chip size, cell count and type, power distribution, and I/O positioning are taken into account. Placement: This process establishes the precise location of each chip cell. The positions are optimised to reduce wire length, timing errors, and power usage. In order to identify the best answer, placement algorithms employ strategies like simulated annealing, genetic algorithms, or analytical approaches. Clock Tree Synthesis (CTS): To maintain perfect synchronisation and slight clock skew, CTS requires creating an optimised clock distribution network throughout the chip. To balance the clock tree and adhere to timing specifications, it inserts buffers and buffers. During the routing step, metal rails are used to link the chip's components and nets (interconnections). The wire length is optimised, design rules and limitations are met, and signal route conflicts are resolved. The general routing structure is decided by global routing, while the final connectivity is carried out by detailed routing. Physical Verification: During physical verification, the layout is examined for design rule violations (DRC) to make sure it complies with the foundry-specified production restrictions. It comprises inspections for width, overlap, spacing, and other regulations to guard against potential manufacturing problems. Timing Analysis: To make sure the chip complies with the given timing limitations, timing analysis assesses the timing routes in the design. It takes into account things like the circuit's delays, setup and hold times, and clock frequency. To confirm timing at various design phases, static timing analysis (STA) is used. Power Analysis: Power analysis calculates the chip's power use and indicates potential areas for power optimisation. Examining the switching activity and power distribution within the design assists in lowering power consumption and mitigating thermal concerns. The final GDSII layout file is created once the design has been confirmed to fulfil all necessary requirements. The placements, forms, and layers of each component are all included in the geometric representation of the chip that is contained in the GDSII file.

## 4.2 Design Tools

Synthesis: RTL synthesis is performed by yosys/ABC. OpenSTA: Creates timing reports by doing static timing analysis on the generated netlist. Floorplanning: initfp - Specifies the rows (used for placement) and tracks (used for routing), as



well as the core area for the macro.Places the macro input and output ports using ioplacer. Creates the power distribution network (pdngen). Inserts welltap and decap cells into the floorplan using tapcell. Placement: RePLace performs global placement. Resizer: Resizing is an optional design optimization. OpenDP: Performs detailed placement to legalize the globally placed components. Synthesises the clock distribution network (the clock tree) using TritonCTS. Routing: FastRoute - Creates a guide file for the detailed router by doing global routing performs thorough routing with TritonRoute Performs SPEF extraction with OpenRCX

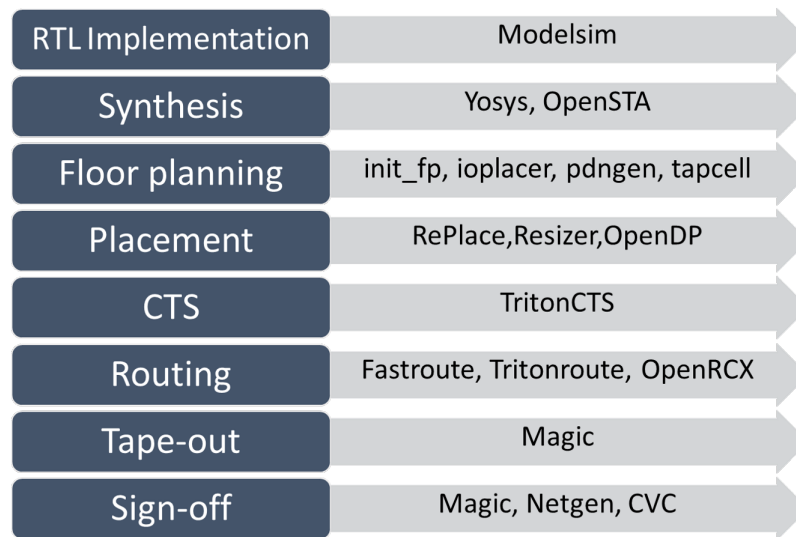


Figure 4.1: Tools used for implementation.

GDSII : The final GDSII layout file is generated by Tapeout Magic based on the routed definition. As an alternative option, KLayout can also generate the final GDSII layout file from the routed definition. The Signoff process includes performing Design Rule Checks (DRC) and Antenna Checks, which are done using Magic and KLayout in conjunction. Netgen is utilized for performing Layout versus Schematic (LVS) Checks. Additionally, Circuit Validity Checks are conducted by CVC. Figure 4.1 illustrates the design flow and corresponding tools used for this work. Detailed discussion on tools will be seen in next sub-section.

### 4.3 Details of Tools Used

OpenROAD: OpenROAD is an all-encompassing digital design pipeline that includes placement, optimisation, and routing tools. For phases in floorplanning, placement, and routing [43].

yosys: Yosys functions as a Verilog HDL synthesis tool, taking a high-level behavioral design description as its input and producing an output in the form of RTL, logical gate, or physical gate level descriptions of the design. Its primary strengths lie in behavioral and RTL synthesis. Yosys offers a wide array of commands, referred to as synthesis passes, enabling a diverse range of synthesis tasks across behavioral, RTL, and logic synthesis domains. The architecture of Yosys is designed for extensibility, making it a suitable foundation for creating custom synthesis tools tailored to specific tasks. Yosys employs two distinct internal formats. The first format is utilized for storing an abstract syntax tree (AST) representation of a Verilog input file. This format is referred to as AST and is generated by the Verilog Frontend component. Subsequently, this AST is processed by a module called the AST Frontend, which transforms it into Yosys' principal internal format known as RTLIL (Register-Transfer-Level Intermediate Language) representation. This conversion involves performing various simplifications within the AST representation prior to generating the RTLIL structure from the simplified AST data. The RTLIL representation serves as the common input and output format for all Yosys passes, offering several advantages over employing distinct formats for various stages: Passes can be reorganized or modified by adding/removing them. Passes can efficiently preserve unchanged design components without format conversions, as Yosys passes maintain the same data structure and execute modifications directly. A uniform interface is maintained across all passes, streamlining comprehension of Yosys source code and facilitating feature augmentation. The RTLIL representation encompasses features beyond a standard netlist: Internal cell library with predefined functional cells for RTL datapath, registers, and logical gate-level elements. Accommodation for multi-bit values employing wire bits and constants for coarse-grained netlist representation. Support for fundamental behavioral constructs like conditional statements and multi-case switches with output update sensitivity. Capability for multi-port memories. However, the use of RTLIL also presents a drawback as it maintains a high-level format across all passes, even when dealing with gate-level synthesis that doesn't require advanced features. For simplicity in passes targeting low-level representations, these passes inspect RTLIL input for utilized features and halt if unsupported high-level constructs are detected. In such cases, a preliminary pass that translates higher-level constructs into lower-level equivalents must be invoked before synthesis. Yosys, an open-source, extensible hardware synthesis tool, caters to designers seeking a universally accessible, vendor-independent synthesis solution. It also appeals to

EDA researchers seeking an open synthesis framework to experiment with complex real-world designs and test algorithms. Yosys is capable of synthesizing a substantial portion of Verilog 2005 and has undergone extensive testing with a diverse set of practical designs, including the OpenRISC 1200 CPU, openMSP430 CPU, OpenCores I2C master, and k68 CPU. Currently, a VHDL frontend for Yosys is under development. Yosys is predominantly coded in C++, incorporating certain elements from the C++11 standard. This section outlines the foundational data structures in Yosys. For clarity, the C++ type names utilized within the Yosys codebase are used throughout this section. Nevertheless, the chapter elucidates the core concepts, serving as a reference for implementing a similar system in any programming language. Figure 4.2 shows working of yosys in depth.

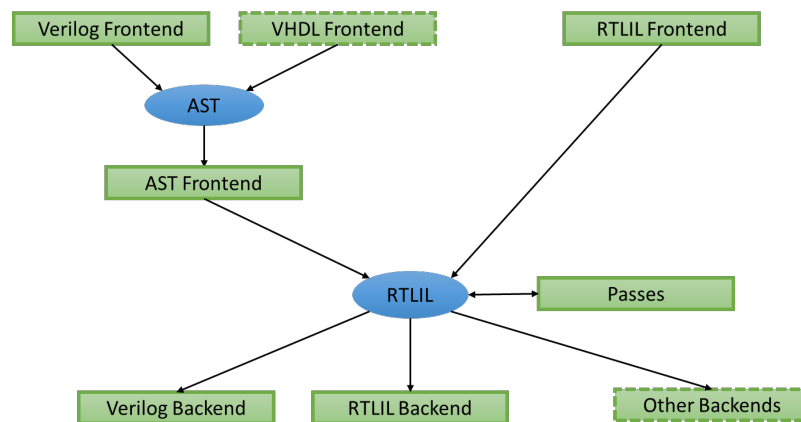


Figure 4.2: Workflow of yosys.

OpenSTA: OpenSTA is an open-source tool utilized for Static Timing Analysis (STA) in the domain of digital chip design and verification. Static Timing Analysis is a pivotal stage in the development and validation of integrated circuits (ICs) that ensures the designed circuit adheres to its specified timing benchmarks. This tool examines the timing characteristics of digital circuits under varying conditions to guarantee their proper operation.

1. Timing Analysis: OpenSTA undertakes an extensive timing analysis by factoring in the delays introduced by diverse components in the design, including gates, interconnects, and wires. It calculates the most critical delay paths within the circuit and verifies their alignment with the predefined timing constraints.
2. Setup and Hold Checks: OpenSTA conducts assessments to identify any violations of setup and hold time parameters, which are critical elements in digital circuits. These checks guarantee the stability of signals within the designated setup and hold time windows of the receiving flip-flops.
3. Clock Tree Analysis: OpenSTA additionally analyzes the distribution network of clocks within the chip, thereby detecting issues related to clock skew

and the synthesis of the clock tree. This step is essential to ensure synchronized operation throughout the chip. 4. Constraint Management: The tool supports the incorporation of various design constraints, encompassing clock frequency, input/output delays, and other specific requirements tailored to the design. 5. Generation of Reports: OpenSTA produces comprehensive timing reports that outline instances of timing violations, pinpoint the paths contributing to these violations, and provide other pertinent details necessary for optimization and debugging. Functioning of OpenSTA: 1. Input: Design Files: OpenSTA takes in the gate-level netlist, a representation of the digital circuit's components and connections. Constraint Files: Design constraints are specified in separate files and include details such as clock frequencies and input/output delays. 2. Pre-processing: The tool conducts pre-processing on the design files to construct an internal data structure reflecting the hierarchy and interconnections within the design. 3. Timing Analysis: OpenSTA simulates the circuit by propagating signal delays through gates, flip-flops, and interconnects. It identifies critical paths with the longest delays and assesses their compliance with timing constraints. 4. Verification of Constraints: The tool compares calculated delays with specified constraints, including setup and hold times. 5. Report Generation: OpenSTA generates detailed timing reports containing insights into timing violations, critical paths, and pertinent statistical data. 6. Optimization: Designers employ the analysis outcomes to make necessary refinements to the design, such as adjusting gate sizes or modifying the clock distribution network, to ensure adherence to timing requirements.

`init_fp`: As the name suggests it initialize floorplan by making list of sites to make rows for die area and core area in micron The die area and core area used to write ROWs can be specified explicitly with the `-die_area` and `-core_area` arguments. It makes tracks place pins around core boundary.

`ioplacer`: `ioplacer` arrange pins along the perimeter of the die using the track grid, with the goal of minimizing the lengths of nets. This pin placement process simultaneously generates metal shapes for each pin while adhering to minimum-area regulations. In cases where cells have not yet been positioned, the calculation of net wirelength takes into account the center of the die area as the assumed position for these unplaced cells.

`pdngen`: This tool is designed to streamline the integration of a power grid into a floorplan. The objective is to define a concise collection of power grid guidelines for the design, encompassing aspects like preferred layers, stripe width, and spacing. Subsequently, the tool automates the generation of the physical metal connections, based on the specified policies.

These grid policies are customizable both within the standard cell region and within regions taken up by macros.

RePLace:RePLace (Regular Placement Engine) is an open-source software tool used in the field of chip design for performing placement optimization. Chip placement is a critical step in the physical design of integrated circuits (ICs) where the positions of various functional blocks, standard cells, and other components are determined on the chip's layout. The goal of RePLace is to find an optimal arrangement of these components to achieve better performance, power efficiency, and manufacturability.

**Key Features and Working of RePLace:**

1. **Global Placement:** RePLace primarily focuses on global placement, which involves finding an initial, coarse-grained arrangement of components on the chip's layout. This placement acts as a starting point for subsequent optimization stages.
2. **Objective Function:** The primary objective of RePLace is to minimize the overall wirelength or interconnect length between components. Minimizing wirelength helps in reducing signal delays and power consumption associated with long interconnects.
3. **Force-Directed Approach:** RePLace uses a force-directed approach, inspired by physics-based simulations, to optimize placement. It models the placement problem as a system of forces acting on each cell to drive them towards an equilibrium position.
4. **Netlist and Constraints:** RePLace takes a netlist as input, which includes information about the connectivity and relationships between different cells in the design. It also considers design constraints such as blockages, pre-placed cells, and user-defined regions.
5. **Hierarchical Approach:** In modern chip designs, hierarchy is common due to the use of macros and IP blocks. RePLace often works hierarchically, optimizing placement at different levels of hierarchy to manage complexity and maintain performance.
6. **Multi-Objective Optimization:** While wirelength minimization is a primary objective, RePLace may also consider other factors like timing, power, and congestion, depending on user-defined constraints and optimization goals.
7. **Iterative Refinement:** RePLace typically performs multiple iterations of the placement refinement process to gradually improve the quality of the placement. Each iteration aims to reduce the overall wirelength and meet the specified constraints.
8. **Output and Integration:** After the placement optimization, RePLace produces a new arrangement of cells and components on the chip layout. This output is then fed into subsequent steps of the chip design flow, such as global routing, detailed placement, and routing stages.
9. **Customization and Tuning:** RePLace often provides various configuration options and parameters that users can customize to achieve

the desired optimization trade-offs, such as focusing more on performance versus power.

**Resizer:** The commands for Gate Resizer are outlined as follows: The resizing process halts when the design area reaches a utilization level of `-max_utilization` percent of the core area. The variable `util` lies within the range of 0 to 100. If the maximum utilization threshold is surpassed, the resizer will terminate and provide an error report.

**Triton CTS:** TritonCTS 2.0 is accessible within the OpenROAD application through the `(clock_tree_synthesis)` command. The provided TCL snippet outlines the method to invoke TritonCTS. This version of TritonCTS, namely 2.0, introduces on-the-fly characterization capabilities, eliminating the need to generate separate characterization data. Nevertheless, users have the flexibility to manage the on-the-fly characterization feature by specifying parameters through the `(configure_cts_characterization)` command. The `(set_wire_rc)` command facilitates the setup of clock routing layers. For TritonCTS to operate, it requires five input files, while producing two output files. The inputs encompass library characterization files, Verilog files with gate-level netlists, placed DEF files with netlists, and a configuration file. On the output side, TritonCTS generates placed DEF files featuring clock buffers, as well as Verilog files integrated with clock buffers. This tool is designed with certain supported features and assumptions, including catering to a single clock source.

**Fastroute:** FastRoute is an open-source software tool used in the field of chip design for global and detailed routing of integrated circuits. It's designed to efficiently and effectively route the interconnects (wires) that connect various components, such as gates, standard cells, and macros, on an integrated circuit layout. FastRoute is particularly useful for designs where high-performance and fast turnaround are crucial. **Key Features and Working of FastRoute:** 1. **Global and Detailed Routing:** FastRoute supports both global routing and detailed routing. Global routing involves determining the approximate path and layer for interconnects, while detailed routing involves placing wires within the specified routing channels. 2. **Multi-Layer Routing:** Modern integrated circuits often have multiple metal layers for routing. FastRoute can handle multi-layer routing, optimizing the use of different layers for routing paths of varying lengths. 3. **Congestion-Driven Routing:** Congestion occurs when there is a high demand for routing resources in a specific area. FastRoute employs congestion-driven routing algorithms to man-

age and avoid congestion, which can help prevent performance bottlenecks. 4. Maze Routing Algorithms: FastRoute employs maze routing algorithms, which involve finding paths through a grid-like structure that represents the available routing resources and obstacles. These algorithms consider obstacles and connectivity requirements. 5. Wirelength Minimization: One of the primary objectives of FastRoute is to minimize the total wirelength or interconnect length. Shorter interconnects lead to reduced signal delays and improved overall circuit performance. 6. Pin Access Optimization: FastRoute optimizes the positions of the pins (connection points) of the components to reduce wirelength and improve routing feasibility. 7. Hierarchical Routing: For larger and more complex designs, FastRoute supports hierarchical routing. This involves routing sub-blocks or regions of the design separately before integrating them into the final routing solution. 8. User-Defined Constraints: Designers can input various constraints, such as minimum/maximum wirelength, preferred routing direction, and area constraints, to guide FastRoute's routing process. 9. Open-Source and Customizability: FastRoute is open-source, allowing designers to modify and customize its algorithms to better suit their specific design requirements. 10. Output Generation: Once the routing is completed, FastRoute generates a routed netlist or a routed layout that can be used in subsequent steps of the chip design flow.

**TritonRoute:** TritonRoute is an open-source software tool used in the field of chip design for detailed routing of integrated circuits. It's designed to efficiently route the metal interconnects (wires) that connect various components, such as gates, standard cells, and macros, on an integrated circuit layout. TritonRoute focuses on providing high-quality routing solutions while considering factors like wirelength, congestion, and design rules.

**Key Features and Working of TritonRoute:**

1. **Detailed Routing:** TritonRoute handles the final stages of routing in the chip design process, where it places wires within the specified routing channels based on the global routing results.
2. **Multi-Layer Routing:** Modern integrated circuits use multiple metal layers for routing. TritonRoute supports routing on multiple metal layers, optimizing the use of different layers for different routing paths.
3. **Congestion Management:** TritonRoute employs sophisticated algorithms to manage congestion, which occurs when routing resources are heavily demanded in certain areas of the layout. By avoiding congestion, TritonRoute ensures that signal integrity and overall performance are maintained.
4. **Design Rule Checking:** TritonRoute adheres to design rules specified by the semiconductor foundries and technology nodes. It ensures that the generated routing solu-

teons comply with manufacturing constraints and guidelines. 5. Layer Assignment: TritonRoute assigns wires to appropriate metal layers based on the design requirements, routing constraints, and the characteristics of the routing paths. 6. Maze Routing Algorithms: Similar to other routing tools, TritonRoute uses maze routing algorithms to navigate through a grid-like structure representing the routing resources and obstacles. These algorithms optimize for wirelength and other design goals. 7. Wirelength Minimization: TritonRoute aims to minimize the total wirelength or interconnect length, which contributes to reduced signal delays and improved circuit performance. 8. Pin Access Optimization: The tool optimizes the positions of pins (connection points) on components to further reduce wirelength and improve routing feasibility. 9. User-Defined Constraints: Designers can input various constraints, such as preferred routing direction, layer-specific routing preferences, and area restrictions, to guide TritonRoute's routing decisions. 10. Hierarchical Routing: TritonRoute can handle hierarchical designs, routing sub-blocks or regions separately before integrating them into a complete routing solution. 11. Open-Source and Customizability: TritonRoute is open-source, allowing users to modify and customize its algorithms to suit their specific design needs. 12. Output Generation: After completing the routing process, TritonRoute generates a detailed routed layout that adheres to the design rules and constraints.

OpnerCX: OpenRCX functions as a Parasitic Extraction (PEX) tool integrated with OpenDB design APIs. It facilitates the extraction of routed designs using the LEF/DEF layout model. The tool calculates both Resistance and Capacitance values for wires, considering factors such as coupling distance to the nearest wire and track density context over/under the target wire. This process also involves cell abstracts. The capacitance and resistance computations rely on equations involving coupling distance, which are interpolated using precise measurements from a calibration file termed the Extraction Rules file. This Extraction Rules file, also known as the RC technology file, is produced for each process node and corner. It is generated through a utility that involves DEF wire pattern generation and regression modeling. OpenRCX subsequently stores the derived resistance, coupling capacitance, and grounded capacitance onto OpenDB objects. These objects have direct pointers connecting them to the corresponding wire and via database elements.



**Magic:** Magic is an open-source software tool used in the field of chip design for layout and circuit design. It is particularly known for its capabilities in layout design and editing, making it a popular choice for creating and manipulating integrated circuit layouts.

**Key Features and Working of Magic:**

1. **Layout Design:** Magic is primarily used for creating and editing layout designs for integrated circuits. It allows designers to visually design the physical layout of components such as gates, standard cells, macros, and other circuit elements.
2. **Customizable Design Rules:** Magic provides the ability to define and enforce custom design rules, which are constraints that ensure the layout adheres to manufacturing and performance specifications.
3. **Hierarchical Layout:** Magic supports hierarchical design, allowing complex circuits to be organized into hierarchical blocks. This is particularly useful for managing large and intricate designs.
4. **Interactive Layout Editing:** Magic provides an interactive graphical interface for editing layout designs. Designers can add, move, and modify components directly on the layout canvas.
5. **Textual and Scripting Interface:** While Magic is known for its graphical interface, it also offers a textual scripting interface that allows users to automate design tasks and perform batch operations.
6. **DRC (Design Rule Checking):** Magic has built-in design rule checking capabilities to ensure that the layout adheres to specified design rules and constraints. This helps identify and fix potential manufacturing issues.
7. **Extraction and Simulation:** Magic can be used in conjunction with other tools to perform extraction of circuit parameters from the layout and even basic simulation tasks.
8. **File Formats:** Magic supports several industry-standard layout file formats, making it compatible with various other tools in the chip design flow.
9. **Community and Customization:** Magic being open-source, it has a community of users and contributors who provide support, share tips, and develop extensions or customizations to enhance its functionality.
10. **Education and Research:** Magic is widely used in academia for teaching chip design concepts and for research purposes. Its open-source nature allows students and researchers to study and modify its source code.
11. **Cross-Platform Compatibility:** Magic is designed to run on various operating systems, making it accessible to users regardless of their preferred platform.

**Netgen:** Netgen is an open-source software tool used in the field of chip design for digital integrated circuits. It primarily focuses on performing Logical and Physical Verification tasks. These tasks are essential to ensure that the designed circuit adheres to design rules, functions correctly, and can be manufactured successfully.

**Key Features and Working of Netgen:**

1. **Logical Equivalence**

Checking: Netgen performs logical equivalence checking between two representations of a circuit, usually comparing a design's RTL (Register Transfer Level) description with its gate-level netlist. It identifies any inconsistencies or mismatches between the RTL and gate-level representations, which helps catch errors introduced during synthesis or optimization.

2. Logical Synthesis Verification: After synthesis, Netgen verifies that the gate-level netlist is functionally equivalent to the RTL description. Any discrepancies between these two levels of representation are flagged as potential issues.

3. Physical Verification: Netgen also performs various types of physical verification checks, ensuring that the layout of the circuit adheres to manufacturing rules and constraints. This includes checks like Design Rule Checking (DRC), which verifies that the layout adheres to the technology-specific manufacturing rules, and Layout vs. Schematic (LVS) checks, which ensure that the layout corresponds accurately to the netlist.

4. Gate-Level Simulation: Netgen can perform gate-level simulations to verify the functionality of a circuit after synthesis and optimization. It compares the simulation results of the gate-level netlist against expected results to identify any functional errors.

5. Hierarchical Verification: Netgen supports hierarchical designs, allowing verification of large and complex circuits by breaking them down into smaller manageable blocks.

6. Netlist and Layout Formats: Netgen supports various industry-standard netlist and layout formats, making it compatible with other tools in the chip design flow.

7. Integration with EDA Flows: Netgen is typically used as part of larger electronic design automation (EDA) flows, working in conjunction with other tools such as synthesis, place-and-route, and simulation tools.

8. Scripting and Automation: Netgen provides scripting interfaces that allow users to automate verification tasks and perform batch operations.

9. Open-Source and Community Support: Being open-source, Netgen has a community of users and contributors who share information, provide support, and contribute to its development.

## 4.4 OpenLANE

There are several EDA tools available to generate GDSII file for a Verilog design. Most commonly used are paid. But recent trends have encouraged chip design by introducing open source tools [44]. R Timothy from efabless has compared 3 such open source tools that facilitate RTL to GDSII flow named Qflow, CloudVSoC and OpenLANE. M. Shalan Research talks about OpenLANE EDA tool we used to generate Tape-Outs for this work.

The combination of the tools described earlier, along with custom scripts for design exploration and optimization, forms the automated RTL to GDSII flow called OpenLANE [45]. OpenLANE provides a range of specialized scripts specifically designed for design exploration and optimization purposes [46]. The flow encompasses all the steps involved in ASIC implementation, starting from RTL (Register Transfer Level) and continuing all the way to GDSII (Graphic Data System Version II) [47]. It supports both variants A and B of the sky130PDK and includes instructions for incorporating support for additional PDKs, including proprietary ones [48]. It takes the source file and configuration file as input. The source file is Verilog code, and the configuration file is written in JSON using particular setting values for particular configuration variables.

#### **4.4.1 OpenLANE Configuration Variables**

There are two types of configuration variables for configuration files. The first one is required configuration variables, and a second and much larger set of configuration variables is optional. They are used to optimize design accordingly [49]. All these variables take a particular value or string as input, and further process is completed accordingly. All the configuration variables are case sensitive and are to be written as shown.

#### **4.4.2 Required Configuration Variables**

These configuration variables are necessary to mention. They are directly used in design flow.

1. DESIGN\_NAME : Here, the name of the .v file is mentioned
2. VERILOG\_FILES : The path of the source file is mentioned here without white spaces. This is required to access the file,
3. CLOCK\_PERIOD : Clock period is defined herein nano-seconds.
4. CLOCK\_NET : The name of the net input to the root clock buffer is mentioned for the process of CTS.
5. CLOCK\_PORT : The name of the design's clock port is specified here, and this information is used during STA.

#### **4.4.3 Optional Configuration Variables**

These configuration variables have a particular default values in case we don't define one it uses that particular value. In case we need to change the values from

default they need to be defined. Some commonly used configuration variables, their default values and their functionality is listed in following table. Then we will discuss them in depth later on.

|   | Configuration Variable | Default Value | Functionality                        |
|---|------------------------|---------------|--------------------------------------|
| 1 | PDK                    | sky130        | Specifies the process design kit     |
| 2 | SYNTH_CLOCK_TRANSITION | 0.15          | Specifies a value for the clock slew |
| 3 | SYNTH_STRATEGY         | AREA0         | Trade-off between area and delay     |
| 4 | IO_PCT                 | 0.2           | % of clock_period used in I/O        |
| 5 | FP_CORE_UTIL           | 50%           | Core utilization %                   |
| 6 | DIE_AREA               | unset         | Specifes co-ordinates x0 y0 x1 y1    |
| 7 | FP_IO_MODE             | 1             | Mode of random I/O placement         |

Table 4.1: Optional configuration variables.

#### 4.4.4 Details of Configuration Variables

STD\_CELL\_LIBRARY is a configuration variable that mentions the standard cell library to be used under the specified PDK. Default value for this variable is Sky130\_fd\_sc\_hd.

Next is STD\_CELL\_LIBRARY\_OPT which specifies the standard cell library to be used during resizer optimizations. Default value for same is STD\_CELL\_LIBRARY. SYNTH\_STRATEGY is also an important configuration variable which plays an important role in optimizing trade-off between area and delay. It strategies for logic synthesis. It's possible values are DELAYAREA 0-40-3. AREADELAY mentions optimization target of the synthesis strategy and the numbers value 0-4/0-3 is an index. Default value is AREA0.

IO\_PCT mentions the percentage of the clock period used in the I/O delay. It ranges between 0 to 1. Default value for the same is 0.2

STA\_REPORT\_POWER is variables that enables generation of power report in STA. It's default value is 1.

FP\_CORE\_UTIL mentions core utilization percentage. Core utilization percentage defines the area occupied by standard cells, macros, and blockages. Default value for the same is 50 %.

FP\_ASPECT\_RATIO mentions core's aspect ratio. Aspect ratio is calculated by height/width. Default value of this configuration variable is 1.

DIE\_AREA as the name suggests it specifies die area which will be used during floor planning when another variable known as FP\_SIZING is set to absolute. Specified as a 4-corner rectangle "x0 y0 x1 y1". These are coordinates for chip design. Generally (x0,y0) is (0,0). Although this variable comes under optional

configuration variable it is necessary to mention this variable as its default value is not set. Unit used for the measurement is  $\mu\text{m}$ .

`CORE_AREA` Die area minus margins gives us core area. As mentioned in core utilization this area is where standard cells and macros will be placed. This will also be used in case `FP_sizing` is absolute.

`FP_IO_MODE` decides mode of placement that whether I/O placement is random and equidistant(1) or on matching mode(0).

`PL_TARGET_DENSITY` mentions desired placement density of cells. It shows how distant cells are inside core area. The value ranges from 0 to 1 where 0 means widely spread and 1 means dense. Default value is calculated by:

`CORE_UTILL+10+5(5(GP_CELL_PADDING)100)`.

Sample value: 0.55. `CTS_TARGET_SKEW` specifies target clock skew. Its defined in picoseconds and it's default value is 200ps

`RUN_CTS` enables CTS in case it's value is 1, which is also it's default value.

These and several more configuration variables make a configuration file which is given to OpenLANE as input with verilog and PDK files to generate Tape-out.

Note that one major issue with OpenLANE is timing closure. While several flow tools have timing awareness, the RTL synthesizer(Yoosys) used in OpenLANE does not support timing-driven synthesis and does not accept standard timing constraints in SDC format. This makes timing closure a real challenge in OpenLANE. Moreover, automatic,post-routing timing closure is not a possibility. In future designs can be optimized based on the placement information but not on the routing information due to lack of incremental routing support. OpenLANE has above mentioned constraints but we have used OpenLANE just to verify our concepts as it's an opensource tool.

## 4.5 Tape-Out Details

Table shows Tape-out details for all three architectures generated.

|   | Design | Die-Area               | Target Density | Cell count | Critical Path delay |
|---|--------|------------------------|----------------|------------|---------------------|
| 1 | SISO   | 829.5 $\mu\text{m}^2$  | 0.75           | 15         | 1.06 ns             |
| 2 | MISO   | 4003.8 $\mu\text{m}^2$ | 0.55           | 83         | 2.4ns               |
| 3 | MIMO   | 8693.3 $\mu\text{m}^2$ | 0.65           | 101        | 4 ns                |

Table 4.2: Tape-out details.

## 4.6 Tape-Out

SISO Architecture is the simplest of the three architectures. Figure 4.2 shows the tape-out generated for SISO architecture. Figure 4.3 shows tape-out generated for MISO architecture. Figure 4.4 shows tape-out generated for MIMO architecture.

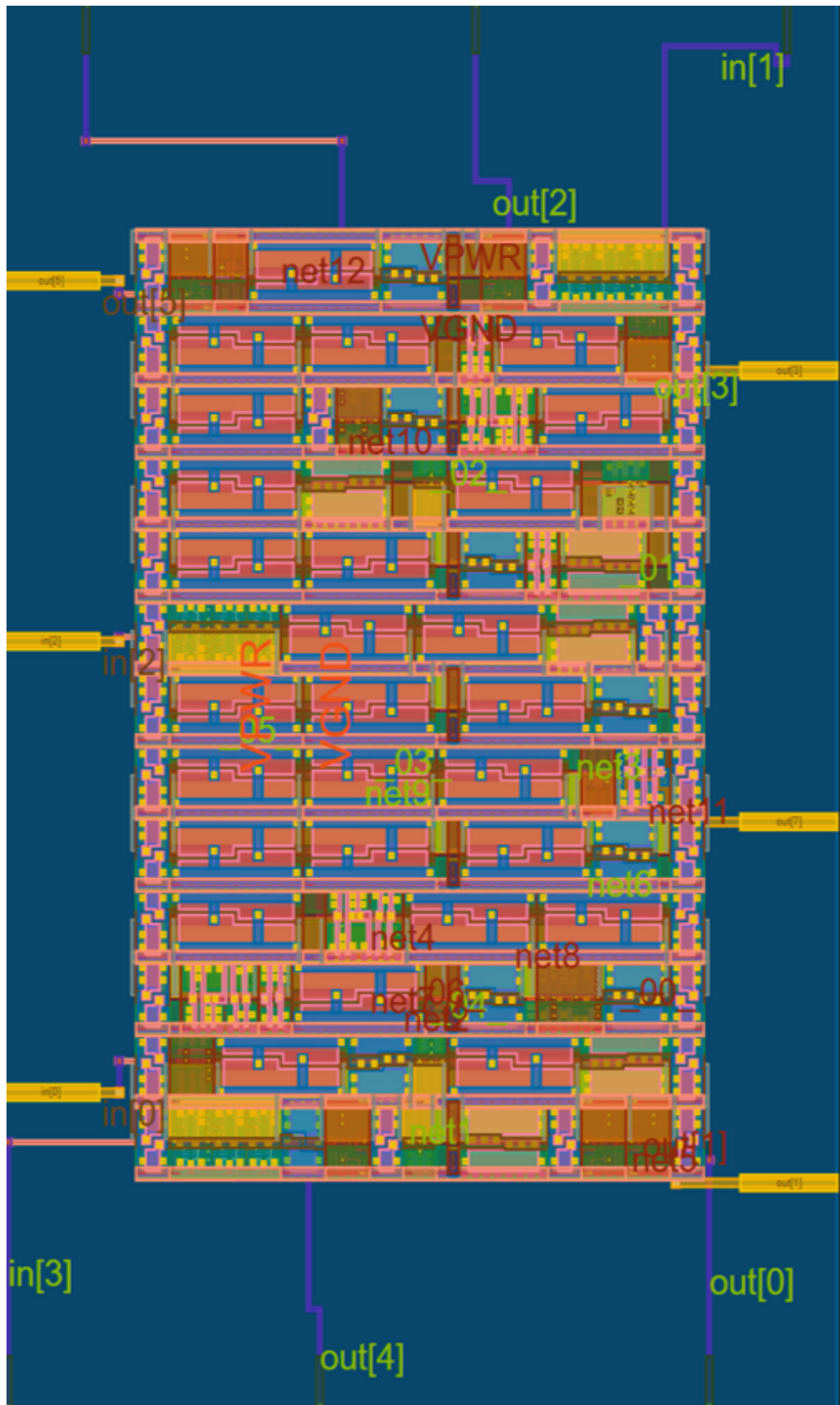


Figure 4.3: Tape-out of SISO.

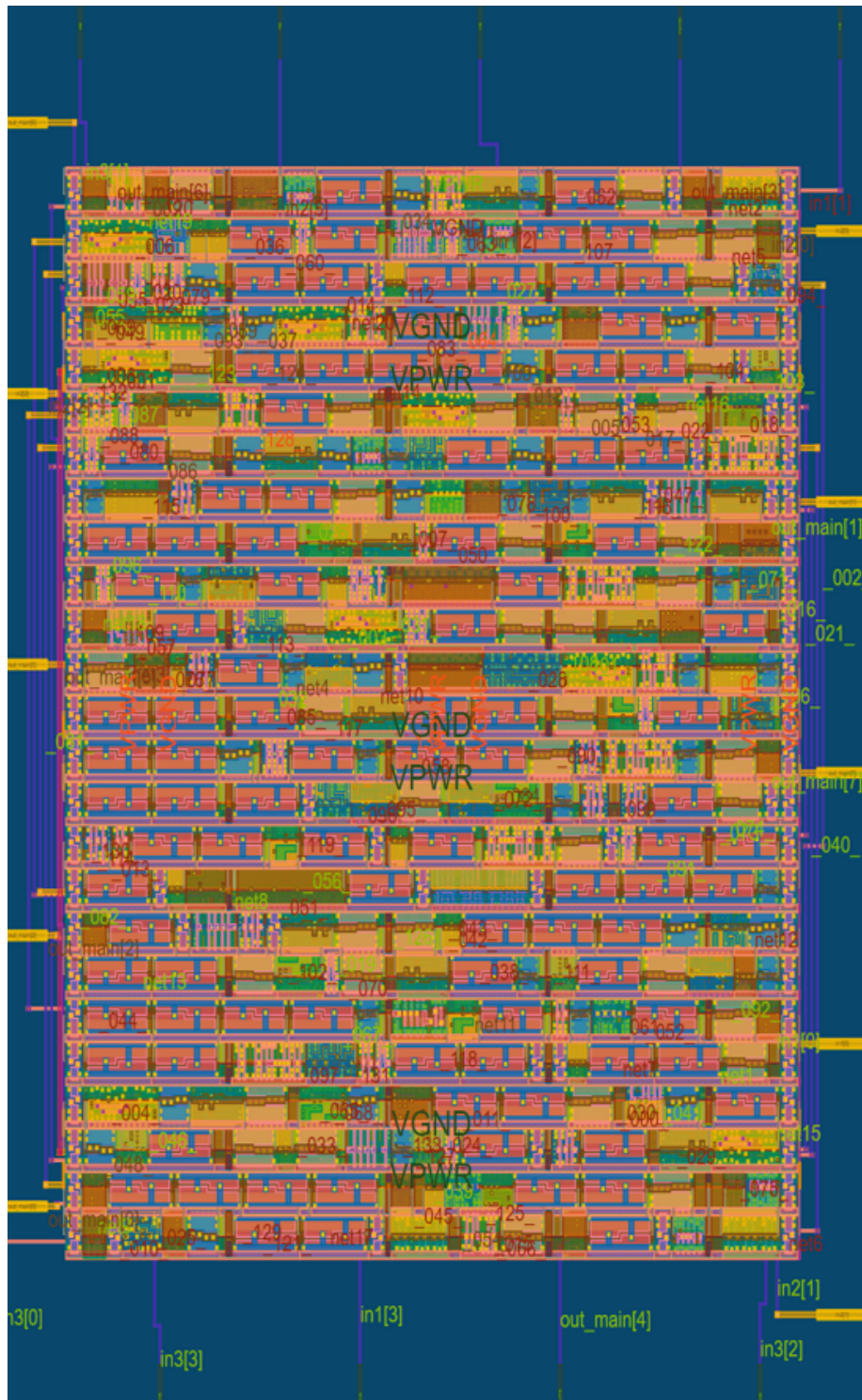


Figure 4.4: Tape-out of MISO.





## 4.7 Comparison with Published Work

Table 4.3 compares a few existing work on NN and this work. Aim of this work was to reduce area. As discussed previously with configuration variables there is trade-off between delay and area, and there also we have optimized area. Literature survey also shows that area is also considered. Since OpenLANE offers PDK(Process design kit) of 130nm technology only, we can't reduce the technology still area of our design is smaller than previous works hence area wise it is optimized.

|   | Reference | Architecture           | Technology | Core-Area              | Learning  |
|---|-----------|------------------------|------------|------------------------|-----------|
| 1 | [50]      | Crossbar               | 45 nm      | 4.2mm <sup>2</sup>     | On-chip   |
| 2 | [51]      | Crossbar               | 45 nm      | 4.2mm <sup>2</sup>     | Off- chip |
| 3 | [52]      | Crossbar               | 35 nm      | –                      | Off- chip |
| 4 | [53]      | 2- Layer Grid and ring | 65 nm      | 3.1mm <sup>2</sup>     | On-chip   |
| 5 | [54]      | 2- Layer Grid and ring | 65 nm      | 1.8mm <sup>2</sup>     | On-chip   |
| 6 | [55]      | SIMD,MAC,MEM           | 40 nm      | 2.4mm <sup>2</sup>     | –         |
| 7 | This Work | SISO                   | 130 nm     | 829.5 um <sup>2</sup>  | Off- chip |
| 8 | This Work | MISO                   | 130 nm     | 4003.8 um <sup>2</sup> | Off- chip |
| 9 | This Work | MIMO                   | 130 nm     | 8693.3 um <sup>2</sup> | Off- chip |

Table 4.3: Comparison with existing works.

## CHAPTER 5

# Conclusion

The architectural design and implementation of neural networks (NNs) for integrated circuit design have been successfully accomplished. The architecture encompasses fundamental components such as adders, multipliers, and rectified linear units (ReLUs).

Three distinct architectures have been developed: Single-In Single-Out (SISO), Multiple-In Single-Out (MISO), and Multiple-In Multiple-Out (MIMO). Weight values are an essential requirement in NNs and are acquired from a manually created memory. These weight values have been generated through software-based training of the NNs model.

Subsequently, layout of the SISO, MISO, and MIMO neural networks were taped out using Sky130 open-source process node PDK which is a 180nm-130nm hybrid technology originally developed internally by Cypress Semiconductor. The respective layout areas for the SISO, MISO, and MIMO architectures are 829.5  $\mu\text{m}^2$ , 4003.8  $\mu\text{m}^2$ , and 8693.3  $\mu\text{m}^2$ , respectively. As the complexity in functions of the architecture increases, so does the area. Total physical cells increase approximately six times for MIMO in comparison to SISO, and total core area is approximately 10 times more for MIMO in comparison to SISO.

## **Publications From This Thesis**

K. Nagrani and T. K. Maiti, "Neural Network Architectures for Integrated Circuits," in *International Symposium on Devices, Circuits and Systems (ISDCS 2023)*, pp. 1-4, Japan, May 2023, doi: 10.1109/ISDCS58735.2023.10153560.

## References

- [1] G. M. Iodice, *TinyML Cookbook: Combine Artificial Intelligence and Ultra-Low-Power Embedded Devices*, Packt Publishing, 1st Edition, 2022.
- [2] A. D. Thakare and S. U. Bhandari, *Artificial Intelligence Applications and Reconfigurable Architectures*, Scrivener Publishing LLC, 1st Edition, 2023.
- [3] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, A. M. Umar, O. U. Linus, H. Arshad, A. A. Kazaure, U. Gana, and M. U. Kiru, "Comprehensive review of artificial neural network applications to pattern recognition," *IEEE Access*, vol. 7, pp. 158820–158846, 2019.
- [4] D. Srinivasan, M. Choy, and R. Cheu, "Neural networks for real-time traffic signal control," *IEEE Transactions on Intelligent Transportation Systems*, vol. 7, no. 3, pp.261–272, 2006.
- [5] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a message-driven parallel application to GPU-accelerated clusters," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, pp. 1–9, 2008.
- [6] A. Shakarami, M. Ghobaei-Arani, and A. Shahidinejad, "A survey on the computation offloading approaches in mobile edge computing: A machine learning-based perspective," *Computer Networks*, vol. 182, pp. 107496, 2020.
- [7] Biological neuron, Ref: <https://www.upgrad.com/blog/biological-neural-network/>
- [8] News medical, Ref: <https://www.news-medical.net/news/20230116/Researchers-develop-an-artificial-neuron-closely-mimicking-the-characteristics-of-a-biological-neuron.aspx>
- [9] J.-W. Lin, "Artificial neural network related to biological neuron network: a review," *Advanced Studies in Medical Sciences*, vol. 5, no. 1, pp. 55–62, 2017.
- [10] K. Wang, Y. Zhu, and C.-Z. Chen, "Perceptron algorithm and its Verilog design," in *China Semiconductor Technology International Conference (CSTIC)*, pp. 1–3, 2020.

- [11] A. Manoharan, G. Muralidhar, and B. J. Kailath, "A novel method to implement STDP learning rule in Verilog," *In IEEE Region-10 Symposium (TEN-SYMP)*, pp. 1779–1782, 2020.
- [12] H. Yu, H. Chalamalasetty, and M. Swaminathan, "Modeling of voltage-controlled oscillators including I/O behavior using augmented neural networks," *IEEE Access*, vol. 7, pp. 38973–38982, 2019.
- [13] O. Awodele and O. Jegede, "Neural networks and its application in engineering," *Sci IT*, pp. 83–95, 2009.
- [14] S. Himavathi, D. Anitha, and A. Muthuramalingam, "Feedforward neural network implementation in FPGA using layer multiplexing for effective resource utilization," *IEEE Transactions on Neural Networks*, vol. 18, no. 3, pp. 880–888, 2007.
- [15] W. Wang and J. Gang, "Application of convolutional neural network in natural language processing," *in International Conference on Information Systems and Computer Aided Education (ICISCAE)*, pp. 64–70, 2018.
- [16] V. Dunjko and H. J. Briegel, "Machine learning and artificial intelligence in the quantum domain: a review of recent progress," *Reports on Progress in Physics*, vol. 81, no. 7, pp. 074001, jun 2018.
- [17] B. Harish, K. Sivani, and M. Rukmini, "Design and performance comparison among various types of adder topologies," *in 3rd International Conference on Computing Methodologies and Communication (ICCMC)*, pp. 725–730, 2019.
- [18] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders," *in IEEE 8th Symposium on Computer Arithmetic (ARITH)*, pp. 49–56, 1987.
- [19] K. A. K. Maurya, Y. R. Lakshmana, K. B. Sindhuri, and N. U. Kumar, "Design and implementation of 32-bit adders using various full-adders," *in Innovations in Power and Advanced Computing Technologies (i-PACT)*, pp. 1–6, 2017.
- [20] R. Zimmermann, Binary adder architectures for cell-based VLSI and their synthesis. Citeseer, 1998.
- [21] A. Mukherjee and A. Asati, "Generic modified Baugh Wooley multiplier," *in International Conference on Circuits, Power and Computing Technologies (IC-CPCT)*, pp. 746–751, 2013.

- [22] C. Y. Lee, L. H. Hiung, S. W. Lee, and N. H. Hamid, "A performance comparison study on multiplier designs," in *International Conference on Intelligent and Advanced Systems*, pp. 1–6, 2010.
- [23] A. D. Rasamoelina, F. Adjailia, and P. Sinčák, "A review of activation function for artificial neural network," in *IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMII)*, pp. 281–286, 2020.
- [24] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *arXiv preprint*, arXiv:1811.03378, 2018.
- [25] A. Holzinger, B. Malle, A. Saranti, and B. Pfeifer, "Towards multi-modal causability with graph neural networks enabling information fusion for explainable AI," *Information Fusion*, vol. 71, pp. 28–37, 2021.
- [26] D. Stursa and P. Dolezel, "Comparison of ReLu and linear saturated activation functions in neural network for universal approximation," in *22nd International Conference on Process Control (PC19)*, pp. 146–151, 2019.
- [27] C. Bircanoglu and N. Arica, "A comparison of activation functions in artificial neural networks," in *26th signal processing and communications applications conference (SIU)*, pp. 1–4, 2018.
- [28] Z. Hu, J. Zhang, and Y. Ge, "Handling vanishing gradient problem using artificial derivative," *IEEE Access*, vol. 9, pp. 22371–22377, 2021.
- [29] Towards Science, Ref: <https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e>
- [30] H. H. Tan and K. H. Lim, "Vanishing gradient mitigation with deep learning neural network optimization," in *7th International Conference on Smart Computing Communications (ICSCC)*, pp. 1–4, 2019.
- [31] T. Szandała, "Review and comparison of commonly used activation functions for deep neural networks," *Bio-inspired neurocomputing*, pp. 203–224, 2021.
- [32] J. Patel, H. Advani, S. Paul, and T. K. Maiti, "VLSI implementation of neural network based emergent behavior model for robot control," in *International Conference on Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, pp. 197–200, 2022.

- [33] L. Ranganath, D. J. Kumar, and P. S. N. Reddy, "Design of MAC unit in artificial neural network architecture using Verilog-HDL," in *International Conference on Signal Processing, Communication, Power and Embedded System (SCOPEs)*, pp. 607–612, 2016.
- [34] L. Ruthotto and E. Haber, "Deep neural networks motivated by partial differential equations," *Journal of Mathematical Imaging and Vision*, vol. 62, pp. 352–364, 2020.
- [35] K. M. Tarwani and S. Edem, "Survey on recurrent neural network in natural language processing," *Int. J. Eng. Trends Technol*, vol. 48, no. 6, pp. 301–304, 2017.
- [36] N. Chhedaiya and V. Moyal, "Implementation of back propagation algorithm in Verilog," *Int. J. Comput. Technol. Appl.*, vol. 3, no. 1, pp. 340–343, 2012.
- [37] M. Kang, S. Lim, S. Gonugondla, and N. R. Shanbhag, "An in-memory VLSI architecture for convolutional neural networks," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 3, pp. 494–505, 2018.
- [38] P. J. A. Evert, R. V. Amudhan, and P. S. S. Paul, "Implementation of neural network based controller using Verilog," in *2011 International Conference on Signal Processing, Communication, Computing and Networking Technologies*, pp. 353–357, 2011.
- [39] P. Treleaven, M. Pacheco, and M. Vellasco, "International exploration describes how the natural massive parallelism of the brain inspires novel VLSI designs," *IEEE Micro.*, vol. 9, no. 6, 1989.
- [40] I. E. Ebong and P. Mazumder, "CMOS and memristor-based neural network design for position detection," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 2050–2060, 2011.
- [41] N. Varshney and G. Arya, "Design and execution of enhanced carry increment adder using han-carlson and kogge-stone adder technique : Han-carlson and kogge-stone adder is used to increase speed of adder circuitry," in *3rd International conference on Electronics, Communication and Aerospace Technology (ICECA)*, pp. 1163–1170, 2019.
- [42] N. Parvatham and S. Gopalakrishnan, "A novel architecture for an efficient implementation of image compression using 2D-DWT," in *3rd International Conference on Intelligent Systems Modelling and Simulation*, pp. 374–378, 2012.



- [43] T. Ajayi and D. Blaauw, "Openroad: Toward a self-driving, open-source digital layout implementation tool chain," in *Proceedings of Government Microcircuit Applications and Critical Technology Conference*, pp. 1105-1110, 2019.
- [44] S. Hesham, M. Shalan, M. W. El-Kharashi, and M. Dessouky, "Digital ASIC implementation of RISC-V: OpenLANE and commercial approaches in comparison," in *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pp. 498-502, 2021.
- [45] M. Chupilko, A. Kamkin, and S. Smolov, "Survey of open-source flows for digital hardware design," in *Ivannikov Memorial Workshop (IVMEM)*, pp. 11-16, 2021.
- [46] C. Lück, D. Sánchez Lopera, S. Wenzek, and W. Ecker, "Industrial experience with open-source EDA tools," in *Proceedings of the ACM/IEEE Workshop on Machine Learning for CAD*, pp. 143-143, 2022.
- [47] D. Zezin, "Modern open source IC design tools for electronics engineer education," in *International Conference on Information Technologies in Engineering Education (Inforino)*, pp. 1-4, 2022.
- [48] A. Ghazy and M. Shalan, "OpenLANE: The open-source digital asic implementation flow," in *Proceeding Workshop on Open-Source EDA Technol.(WOSET)*, pp. 1-5, 2020.
- [49] M. Shalan and T. Edwards, "Building OpenLANE: a 130nm openroad-based tapeout-proven flow," in *Proceedings of the 39th International Conference on Computer-Aided Design*, pp. 1-6, 2020.
- [50] J.-S. Seo, B. Brezzo, Y. Liu, B. D. Parker, S. K. Esser, R. K. Montoye, B. Rajendran, J. A. Tierno, L. Chang, D. S. Modha, et al., "A 45nm cmos neuromorphic chip with a scalable architecture for learning in networks of spiking neurons," in *IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1-4, 2011.
- [51] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm," in *IEEE custom integrated circuits conference (CICC)*, pp. 1-4, 2011.
- [52] S. Shapero, C. Rozell, and P. Hasler, "Configurable hardware integrate and fire neurons for sparse approximation," *Neural Networks*, vol. 45, pp.134-143, 2013.

- [53] D. Blaauw, D. Sylvester, P. Dutta, Y. Lee, I. Lee, S. Bang, Y. Kim, G. Kim, P. Pannuto, Y.-S. Kuo, et al., "IoT design space challenges: Circuits and systems," in *2014 Symposium on VLSI Technology (VLSI-Technology): Digest of Technical Papers*, pp. 1–2, 2014.
- [54] J. K. Kim, P. Knag, T. Chen, and Z. Zhang, "A 640m pixel/s 3.65 mW sparse event-driven neuromorphic object recognition processor with on-chip learning," in *Symposium on VLSI Circuits (VLSI Circuits)*, pp. C50–C51, 2015.
- [55] B. Moons and M. Verhelst, "A 0.3–2.6 tops/w precision-scalable processor for real-time large-scale convnets," in *IEEE Symposium on VLSI Circuits (VLSI-Circuits)*, pp. 1–2, 2016.

# Appendix: Verilog Code

## 1. SISO main

```
module siso_main(input [3:0] in, output [7:0] out_main);

    wire [3:0] weight;
    wire [7:0] bias = 8'b00000001;
    wire [3:0] mem=4'b0011;
    wire [7:0] out_temp;
    wire cin=1'b0;
    wire cout;

    weight_mem w1(mem,weight); // Adds Weight Value to memory
    mul4b M1(out_temp,weight,in); // Multiplies input and weight
    add8b A1(cout,out,out_temp,bias,cin); // Adds output of multiplier to bias.
    relu R1(out_main,out); // Output of adder goes through ReLU.

endmodule
```

## 2. MISO main

```
module miso_M(in1,in2,in3,out_main);

    input [3:0] in1,in2,in3;
    output [7:0] out_main;

    wire [3:0] i1 = 4'b0000;
    wire [3:0] i2 = 4'b0001;
    wire [3:0] i3 = 4'b0010;
    wire [3:0] w_val1,w_val2,w_val3;
    wire [7:0] out_n1,out_n2,out_n3;
```

```

wire [7:0] out_temp;

mem m1(i1,w_val1);
neuron n1(out_n1,in1,w_val1);

mem m2(i2,w_val2);
neuron n2(out_n2,in2,w_val2);

mem m3(i3,w_val3);
neuron n3(out_n3,in3,w_val3);

relu R1(out_temp,out_n1,out_n2,out_n3);

assign out_main = out_temp;

endmodule

```

### 3. MIMO main

```

module mimo_p(clk,in1,in2,in3,in4,in5,in6,in7,in8,out_main1,out_main2);

```

```

input clk;
input [3:0] in1,in2,in3,in4,in5,in6,in7,in8;
output [7:0] out_main1,out_main2;

```

```

wire [3:0] i1 = 4'b0000;
wire [3:0] i2 = 4'b0001;
wire [3:0] i3 = 4'b0010;
wire [3:0] i4 = 4'b0011;
wire [3:0] i5 = 4'b0100;
wire [3:0] i6 = 4'b0101;
wire [3:0] i7 = 4'b0110;
wire [3:0] i8 = 4'b0111;
wire [3:0] i9 = 4'b1000;
wire [3:0] w_val1, w_val2, w_val3, w_val4, w_val5,w_val6,w_val7,w_val8;
wire [7:0] out_m1,out_m2,out_m3,out_m4,out_m5,out_m6,out_m7,out_m8;
wire [7:0] out_n1,out_n2,out_n3,out_n4,out_n5,out_n6,out_n7,out_n8;

```

```

wire [7:0] out_1,out_2;
wire [7:0] out_temp1,out_temp2;
wire [7:0] bias= 8'b00000011 ;

mem m1(i1,w_val1);
neuron n1 (clk,in1,w_val1,out_n1);
mem m2(i2,w_val2);
neuron n2 (clk,in2,w_val2,out_n2);

mem m3(i3,w_val3);
neuron n3 (clk,in3,w_val3,out_n3);

mem m4(i4,w_val4);
neuron n4 (clk,in4,w_val4,out_n4);

mem m5(i5,w_val5);
neuron n5 (clk,in5,w_val5,out_n5);

mem m6(i6,w_val6);
neuron n6 (clk,in6,w_val6,out_n6);

mem m7(i7,w_val7);
neuron n7 (clk,in7,w_val7,out_n7);

mem m8(i8,w_val8);
neuron n8 (clk,in8,w_val8,out_n8);

relu R1(out_1,out_2,out_n1,out_n2,out_n3,out_n4,out_n5,out_n6,out_n7,out_n8);

assign out_main1 = out_1; assign out_main2 = out_2;

endmodule

```