**Addis Ababa University**

**College of Technology and Built Environment (CTBE)**

**School of Electrical Engineering**

# Neural Network Based Processing Architecture on Intel Cyclone V SoC 5CSEMA5F31C6 for Amharic Handwriting Recognition

By

Negussu keffeni

Adviser

Mr. Nebyu yonas

Co-adviser

Dr. Wendem Beyene

Presented in Fulfillment of the Requirements for the Degree of Master Science in Microelectronics Engineering

Addis Ababa,
Ethiopia
September 28, 2025

# Addis Ababa University
## College of Technology and Built Environment (CTBE)
## School of Electrical and Computer Engineering

*This is to certify that the thesis prepared by Negussu Keffeni, entitled "Neural Network Based Processing Architecture on Intel Cyclone V SoC 5CSEMA5F31C6 The Amharic Handwriting Recognition", submitted in partial fulfilment of the criteria for the Master of Science degree in Microelectronics Engineering, adheres to university norms and fulfils established standards of originality and quality.*

Approved by the Board of Examiners.

| | | |
|---|---|---|
| Thesis Advisor | Signature | Date |
| Examiner | Signature | Date |
| Examiner | Signature | Date |

## Declaration of Authorship

- This work was conducted entirely or predominantly during the pursuit of a research degree at this university.
- Any segment of this thesis that has been previously submitted for a degree or any other qualification at this university or any other institution has been clearly stated.
- All published works consulted are explicitly credited.
- All quotations from external works are duly cited with their respective sources. This plan is all my own creation, excluding the cited quotations.
- I have recognized all primary sources of assistance.
- In instances when the idea is founded on collaborative efforts, I have explicitly delineated the contributions of others and my own input.

Signed:

_____

Date:

_____

# Acknowledgment

I wish to convey my profound appreciation to my counselor, Dr. Wondimagegnehu Tsegaye, for their kindness in coming to Ethiopia and teaching the class on Advanced VLSI Design: Analysis and System Perspective, where they have elevated my understanding and inspired me for a bigger aim.

Dr. Wondimagegnehu Tsegaye is oriented toward an open-minded approach to searching for value, commitment, and interest, where he has guided me in a professional way of developing a research title. He was committed to helping me search for a research gap I could possibly tackle and to the development of a narrowed target toward the research goal.

My grateful thanks are also extended to Mr. Nebyu Yonas for his brotherly approach and personal development. He has played a cooperative and committed plan so that I stay on schedule.

I can't conclude my acknowledgment without mentioning Dr. Fitsum for allowing me to use the faculty server resources, Dr. Bisrat for commenting on my document for precise output, and Dr. Fetene for guidance on how to appear professionally.

Finally, I would like to thank my classmates and my senior graduate class friends for sharing their ideas and comments while preparing this paper.

## Abstract

*Due to the physical limitations of continued growth predicted by Moore's Law many researchers are seeking new avenues through hardware/software co-design structures to further increase their computer performance. This Thesis proposes New Hardware Architectures that utilize custom hardware to provide Best performance for Deep Learning Applications. More specifically this case study will test these hardware architectures on a specific application the analysis of Amharic handwriting characters through a created database developed from an Amharic handwriting text source that has been created through both Image Processing and Labelling Techniques to be used to effectively train a Neural Network. In order to Facilitate a Bridge between becoming an effective software prototype and developing fast hardware accelerators a full Verilog implementation of a scalable design optimised for the Cyclone V SoC will be created, using fixed Point Arithmetic, Block RAM (BRAM) and Pipe lined Neuron Design structures; additionally, the Hardware Design of the Custom Chip will require the use of techniques including Model Compression & Quantization to manage the different trade-offs involved between throughput, Latency, Power Consumption and Memory Footprint. In addition to enabling the efficient creation of Amharic Character Recognition Solutions using Custom Hardware Design for Neural Networks, this Hardware Architecture will allow the flexible ability to perform both Training and Inference Tasks on the same piece of Hardware. By giving device Designer's the ability to integrate Machine Learning Frameworks with their FPGA & SoC Customisation efforts, this work is expected to lead to further development of Custom, Energy Efficient Domain-Specific Computing Architectures capable of providing longer-term Performance Improvements that extend beyond the existing limits of Moore's Law.*


***Keywords:*** *Neural networks, Intel Cyclone V SoC 5CSEMA5F31C6, hardware architecture, Amharic handwriting recognition, Moore's Law, model compression*

# Contents

# 1. Introduction

Research and engineers need to consider efficient algorithms, distributed computing techniques and specialized hardware architecture to tackle challenges regarding the following points. The relentless pursuit of Moore's Law, which predicts the scaling up of integrated circuits per die, has revolutionized the field of computing over the years. However, as we approach the limits of scaling in 2D silicon photolithography, we confront the probabilistic nature of electrons and the challenges it poses to device integration. This dissertation aims to answer the current research question. It will present the time spent researching and developing alternative strategies to maintain technological advances in computational capacity due to limitations in technology, and how researchers are able to take advantage of this capacity. There is a wide variety of research paths based on IC (integrated circuits)-based architecture, including different types of exploratory research seeking to shape new paths by studying the components of integrated circuit-based architecture.

With the exponential growth and greater sophistication of machine learning applications, there is a high demand for computers designed specifically for machine learning applications (ML). However, hardware architects with expertise in designing new computers to exploit that need are currently in short supply. As demand for machine learning computers continues to outstrip Moore's Law, architects will need to develop new designs of hardware specifically designed to provide machine learning expertise. The current shortage of skilled hardware architects represents a significant problem in accelerating the overall deployment of exciting machine-learning-based applications. The collaboration between hardware architects and machine learning experts will be instrumental in building computers with the scalability and efficiency necessary to meet the demands of the computational requirements of machine learning applications and to solve some of the grand challenges facing society.

Future computing system architecture will primarily consist of Artificial Intelligence (AI). This means AI needs to be incorporated into the hardware and software of various computing systems, and it has become an AI-driven standard for many industries. Because of this, we will now be able to use AI as a way of updating how we build computers, and a key area that researchers are focusing on will be taking the newest ways in machine learning algorithms that are built off Deep Learning technologies for implementing new architectures in computing systems.

This research will include an overview of machine learning, followed by a discussion on various areas of AI and related technologies that have been developed to support the advancement of computing architecturally followed by an overview of all core concepts and principles pertaining to machine learning. Hardware Architecture for Machine Learning Systems - The design of hardware architecture for machine learning systems will provide an overview of the different hardware architecture components of machine learning systems that can be built using concepts of machine learning.

The primary objective of this research is to cover the many major areas related to machine learning, including Data Preparation and Labelling, Feature Extraction, Model Tuning and Model Evaluation. The focus of the majority of this research will be on extracting useful features from raw input data to increase the effectiveness of a machine learning system as well as selecting and evaluating models. Each of the various models of machine learning will be compared against the most commonly used performance metrics, including Accuracy and Precision. Another objective of this research is to explore ways to optimise the parameter and hyperparameter settings of machine learning models to maximise model performance. This research will examine the various ways in which the machine learning algorithm produces results and how these data are visually shown to the viewer. Researchers will have the ability to present a more explicit explanation of how their machine learning model produces outcomes.

Deep Neural Networks (DNNs) are one of the most widely used models of machine learning and are known for their ability to learn complex patterns and relationships, as well as to generate Hierarchical Representations of data. The ability to generate a hierarchical representation creates many complex relationships among the layers of a DNN. DNNs have already had a major impact on many different areas of machine learning due to their best possible performance across the majority of challenging tasks. There is an abundance of large-scale datasets available for DNNs and DNNs will be an essential building block of the next generation of machine learning systems.

This thesis will specifically evaluate DNNs and their design for the purpose of developing new hardware architectures for the specific task of Amharic handwriting recognition. The goal of this thesis is to gain a better understanding of how to optimally design hardware for executing DNN models that can recognise Amharic handwriting. By understanding the underlying hardware architecture,it is the

goal of this thesis to determine the potential use of parallel processing, memory optimization, and the use of specially designed hardware to achieve faster and better results when performing handwriting recognition.

The objective of this Thesis is to merge cutting-edge hardware design and Deep Neural Networks (DNNs) to improve accuracy and efficiency dramatically in terms of image recognition.

To achieve this purpose, a customized solution will be optimally configured as per the requirements of the end-user to use as efficiently as possible all available computing resources while still achieving the maximum degree of benefit associated with computational-based solutions. Therefore, the prototype proposed herein should provide the performance characteristics and low power consumption necessary for field programmable gate arrays (FPGA) to accomplish Amharic handwriting recognition while overcoming the technical challenges associated with the higher level of compute power typically associated with this type of task.

## 1.1. The Evolution of Computing and Its Challenges

As the mid-twentieth century, computing technologies have experienced an extraordinary amount of growth that is primarily attributable to the exponential doubling of transistor density every two years, as predicted by Moore's Law[17]. With the increase of thousands of transistors, we continue to improve performance in a continuous fashion during this digital age. However, the scaling of transistors is approaching fundamental physical limits (heat dissipation, power consumption, and quantum effects), which is creating tremendous barriers to sustaining Moore's Law[1].

Concurrently, the demands for software have significantly increased in complexity with the rise of AI and ML, which has created bottlenecks for traditional Von Neumann architectures due to the large volume of data being processed by modern applications. As a result, the challenges associated with these technological advances require new computing paradigms that allow for the execution of high-performance computing while also being energy-efficient and scalable.

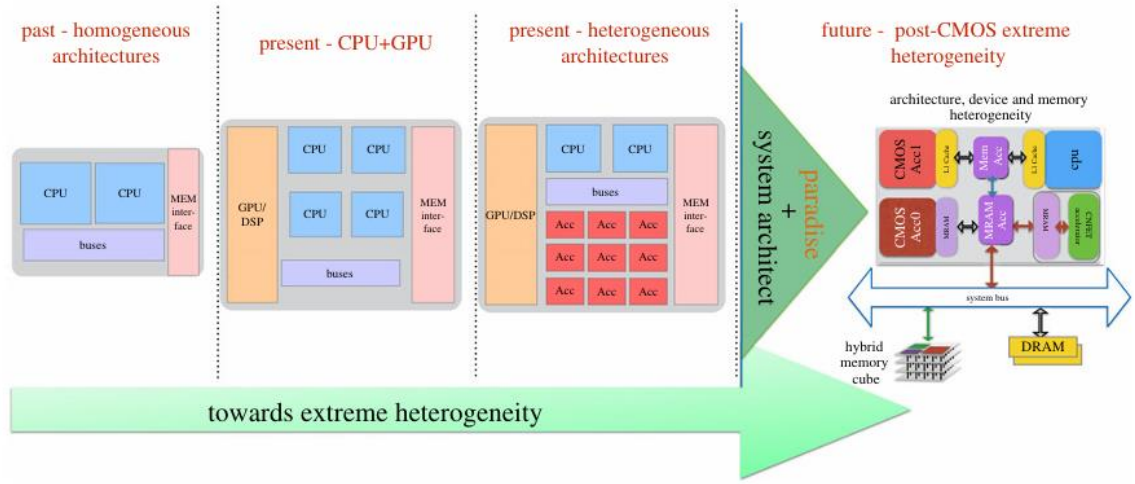


Figure 1 The near-term consequence to the end of classical technology scaling is expected to be heterogeneity and architecture specialization.

## 1.2. Emergence of Machine Learning and Deep Learning

The use of Machine Learning (ML) and how it is utilized in Computer Vision, Natural Language Processing (NLP), Speech Recognition, and other fields are many. The most notable of ML techniques has been Deep Learning (DL), which employs Deep Neural Networks to perform hierarchical feature extraction of raw data [20]. CNNS and RNNs have broken through barriers in image and sequence processing, respectively, while the newest advancements will further expand the capabilities of AI [21].

While these advances represent a significant step forward, the computational intensity involved in training and inference for DL and its associated cost (i.e., the high cost of floating-point operations, memory bandwidth requirement, and overhead associated with the parallelization of significant amounts of data) have posed significant challenges to the efficient utilisation of these systems. As such, this has led to the design of both hardware accelerators and algorithms aimed at decreasing the computational complexity of processing data [22].

## 1.3. Toward Hardware Acceleration

Due to the inefficient nature of traditional CPU-based systems, as well as the increase in the use of traditional CPUs in conjunction with high-performance GPUs, specialized hardware solutions (accelerators) have evolved to enhance the performance of deep learning. Graphics Processing Units (GPUs) are one of these accelerators and have fostered rapid progress in deep learning due to their ability to process thousands of concurrent tasks. Nevertheless, the power demands of GPUs are forcing researchers to look for alternative solutions for deep learning applications. Consequently, FPGA and ASIC hardware options are now available to fill this need[23].

Many researchers are shifting to FPGA (Field Programmable Gate Array) solutions for machine learning due to the various advantages they offer [24]. FPGAs are flexible and can implement many calculations in parallel. Furthermore, they offer the option of fine-tuning the data path used for a specific machine learning model (Neural Network) - allowing hardware designers to choose what type of arithmetic operation to perform (Quantized or Floating point). FPGAs also have a capability of being power-efficient while maintaining high performance and high-level efficiency.

## 1.4. Amharic Handwriting Recognition

Handwriting Recognition remains a challenging area of research within Pattern Recognition for a number of reasons. While there has been much previous research done in terms of papers produced on languages such as English, Chinese, and Arabic, there has not been nearly the same amount of research focus on under-studied/under-researched less-resourced languages. (e.g., Amharic is the official working language). Examples include Ethiopia, which is one of the most heavily spoken languages in Africa, and Amharic has a unique character (the Ge'ez script) with over 200 different symbols; this presents difficulties with handwriting recognition [26].

The primary difficulties associated with handwriting recognition for Amharic include a very large number of character types, many different styles used by the various writers of Amharic, and the no availability of sufficient labelled data sets necessary to train the required models. Significant investment in developing advanced Machine Learning algorithms capable of learning the

discriminative features, but at the same time are sufficiently efficient in terms of computing resources required to deploy in real-life scenarios.

## 1.5. Research Problem and Objectives

The use of deep neural networks to recognize handwriting has yielded positive results; however, the substantial computational resources needed to operate these systems in an efficient manner make deployment extremely difficult, especially when previously identified difficulties surrounding deployment of deep neural networks to Amharic script are taken into consideration, such as: high dimensionality of input images, large output space of characters, and real-time inference requirements (i.e., the need to perform an inference in near real-time) [27]. Further complicating this situation is the limits imposed by Moore's law combined with the increasing need for high-performance, low-power computing solutions, which necessitates the creation of new processing architectures that are inspired by biological cognitions (brains) as well as next-era paradigms (quantum computing, brain-inspired processing) [8]. Quantum computing will provide additional tools to assist in our ability to tackle and solve combinatorial problems; however, digital computing will still be critical to such applications as pattern recognition, data mining, and real-time inference.

This paper integrates and provides a bridge between these perspectives via the development of a power-efficient processing architecture using recent developments in neural network algorithms for implementation on Field Programmable Gate Arrays (FPGAs) to implement a power-efficient Amharic handwriting recognition system. There are three objectives of this paper:

1. To characterize the computational demands of using deep neural networks for Amharic handwriting recognition.
2. To design and implement optimized FPGA-based systems that will accelerate the process of inferring through quantization and parallelization methods.
3. To evaluate the trade-offs between accuracy and resource utilization, thereby contributing to the broader effort of developing architectures that balance performance and efficiency in the post-Moore's law era.

## 1.6. Contributions and Research Questions

This thesis provides insight into efficient machine learning, as well as how to support under-resourced languages in the digital age. Solutions developed in this thesis using an efficient hardware architecture provide a means to enable commercial and practical applications of Amharic handwriting recognition. A primary goal of digitizing Amharic manuscripts is to preserve cultural heritage; by creating an efficient Amharic OCR capability for mobile education applications, e-Governance initiatives can be enhanced, and access to educational resources in Ethiopia and Amharic-speaking countries increased. Additionally, the co-design methodology detailed in this thesis provides an avenue for building hardware accelerators for complex script languages and subsequently provides an opportunity for reducing the computational burden and energy requirements for business applications that are language-specific. Furthermore, since the majority of AI hardware relies on CPU/GPU technology to perform AI inference tasks, the FPGA techniques explored in this thesis directly address the resulting challenges created by the end of Moore's Law and demonstrate how to leverage the capabilities of reconfigurable hardware to achieve power-efficient high-performance implementations of AI tasks. Developments in FPGA [Field Programmable Gate Array] technology have been shifting toward a focus on lower power consumptions when performing AI [Artificial Intelligence] inference operations, meaning the potential savings in power usage by FPGA's when performing AI inference operations could be as high as 60% less than that consumed by traditional CPU [Central Processing Unit] and GPU [Graphics Processing Unit] technology when performing the same types of AI inference operations in the year 2023 [22] . The thesis provides the result of the research for projecting those decreases by evaluating data collected during the conduct of this thesis. Also, this thesis is focused on educating engineers about how to create more innovative AI hardware designs through the development of methodologies that are accessible to a wider audience.

The contributions of this thesis are at the intersection of hardware design and machine learning and include:

- Provides a thorough review of the computational requirements of deep neural networks for Amharic handwriting recognition;

- Provides FPGA-based design techniques for achieving accelerated inference, including architectures for data quantization and architecture optimizations; and,
- Evaluates the relative accuracies, latencies, and efficiency of hardware resources of each of these three approaches.

These contributions are framed by the following research questions:

- What are the computational requirements of deep neural networks used to accomplish Amharic handwriting recognition?
- What architectural optimizations are available on the FPGA platform to increase the speed of the inference process without introducing significant error into the process?
- How do FPGA-based implementations compare to traditional CPU and GPU implementations with regard to performance and efficiency?

## 1.7. **Thesis Structure**

This thesis is laid out in the following chapters: Chapter 2 reviews the literature surrounding the existing research and advancements in both deep learning, handwritten character recognition, and the hardware accelerators that have been developed for those applications. Chapter 3 details how the methodology to derive the implementation strategy for the neural networks created in Chapter 2 was developed, as well as how to evaluate the various implementations of those networks that were developed using the framework defined in Chapter 2.

The fourth chapter contains the results of the experiment and a performance analysis of the trial runs conducted. In the last chapter, the findings, contributions and recommendations for future work will be set forth.

# 2. Literature Review

## 2.1. Studies on Moore's Law and Its Limits

The concept of Moore's Law was introduced in 1965, and it stated that the number of transistors within an integrated circuit, the density of components, would typically double every two years. As we continue toward nanometer technology, it has become clear that this trend will not last indefinitely because Moore's Law will not hold true at the nano-scale for the same reasons that it held true at the micro-scale [1].

An examination published by Kuhn et al. (2011) which examined the scaling of devices below 10 nm, it was discovered that as devices get smaller, quantum mechanical effects such as electron tunneling through dielectric materials lead to substantially increased power dissipation due to increased leakage currents [2][3]. To illustrate this, with Intel's 10 nm process, almost 20% higher leakage was found compared to future generations (larger nodes) of chips.

In addition to the increase in static leakage currents due to quantum physics, it was also reported by the authors that the variation in threshold voltages for chips at this small scale exceeds 30% and that this variation is driven by atomic-level defects in the silicon crystal lattice [2].

The increasing cost to manufacture chips was discussed by Waldrop (2016) who estimates that it could cost over $700 million for an entire fabrication facility to be able to scale to 3 nm technology [4][5]. In 2020, Shalf proposed other means of meeting our ever-increasing processing needs, like using different designs (chiplets and 3D integration), but he concluded these methods are incremental improvements rather than exponential improvements and will lead to the need for an architecture change [6].

Tech Insights (2024) projects that TSMC's 2 nm process, expected in 2025, will incorporate III–V semiconductors like gallium arsenide to improve performance and energy efficiency, though these developments primarily address performance limits rather than the continuation of Moore's scaling trend [7].

The reduction of Moore's law is now fuelling research and development into Domain Specific Accelerators (DSA) such as ASICs, GPU's and programmable logic devices (FPGAs) which will

provide a method to attain higher computational throughput after the end of Moore's Law. Gordon Moore predicted that the number of transistors on the same silicon die would double approximately every two (2) years.

This phenomenon is frequently referred to as Moore's Law. Gordon Moore stated during an interview on April 2005 that, "You cannot continue this expectation indefinitely," due to the fact that the expectation cannot be supported over long periods of time. The most recent version of the International Technology Roadmap for Semiconductors (ITRS), which has been based upon and pioneered the semiconductor industry by the application of Moore's Law since 1998, was published in 2016. It has ceased prioritizing Moore's Law in its research and development strategy. Instead, it delineated what may be termed the More than Moore method, wherein application requirements dictate chip development rather than emphasizing semiconductor scalability. Application drivers encompass mobile devices, artificial intelligence, and data centers.

In 2016, IEEE initiated a road-mapping project, Rebooting Computing, designated as the International Roadmap for Devices and Systems (IRDS) [12][13]. Most prognosticators, including Gordon Moore, anticipate that Moore's Law will conclude by around 2025. Even if the semiconductor companies are trying their best to continue the buildout of a complex strategy for scaling down transistors while satisfying key metrics and design objectives, researchers should be focusing their target for the reasonable what-if moors low will soon end. There are a large number of research projects that have been ongoing for decades, trying to find a solution to the problem of Moore's Law's inevitable ending.

To address the issue of this disconcerting problem, researchers are trying to approach the solution in both the intermediate and long- term, with a multi-directional way of addressing the issue. The researchers are aiming to find a solution for the intermediate (10 years) evolutionary term and for the long (10–20 year).

The value of this research lies in the evolution of computing architecture using current semiconductor technology. In Figure 2 three axes represent different technology scaling paths that could be used to extract a prolonged (10–20 year) revolutionary strategic plan. The timing requirements for the intermediate term necessitate an evolutionary strategy focused on attaining advancements in manufacturing technology that sustain Moore's Law through existing complementary metal-oxide-

semiconductor (CMOS) technology. This approach will depend on novel computing architectures and sophisticated packaging technologies, including monolithic three-dimensional integration (constructing chips in three dimensions) and photonic co-packaging, to reduce data movement expenses [1][9] additional performance beyond the end of lithographic scaling. The immediate emphasis will be on the advancement of increasingly specialized designs and sophisticated packaging methods that organize existing components. In the intermediate future, focus will likely shift toward the development of CMOS-based devices that incorporate a third, vertical dimension, with advancements in materials and transistors aimed at enhancing performance through the creation of more efficient foundational logic circuits. The third axis signifies the potential to create novel computational models, including neuro-inspired and quantum computing, which address issues inadequately managed by digital computing [2]. The near-term focus of this research aim is on the rearrangement of current semiconductor computing technology taking natural brain-inspired computer techniques, such as machine learning.

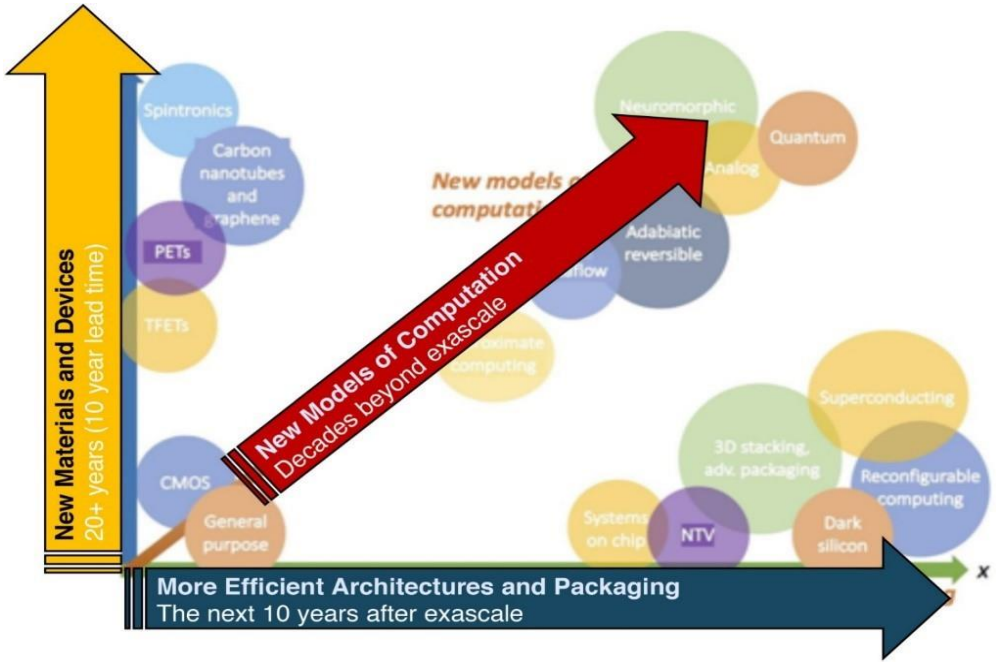


Figure 2 Research approach beyond moor's law[2]

The algorithms are less dependable for executing processes that need precise responses and repeatability, including "explainability" as well. Machine learning, a significant branch of AI, was described by Arthur Samuel in 1959 as the discipline that enables computers to learn autonomously without explicit programming. In the domain of machine learning, there exists a sector often known as brain-inspired computing. Given that the brain is now the most effective "machine" known for

learning and problem-solving, it is a logical source for a machine learning methodology. A brain-inspired computation refers to software or an algorithm that derives certain elements of its fundamental structure or functioning from the operational principles of the brain. This contrasts with efforts to construct a brain; instead, the software seeks to replicate some facets of our comprehension of brain function. Neural networks constructed by the machine-learning algorithm compute a weighted aggregation of the supplied data. The weighted sums represent the value scaling executed by the synapses and the aggregation of those values inside the neuron itself [14].

While scientists continue to investigate the intricacies of brain function, it is widely accepted that the primary computing unit of the brain is the neuron. The typical human brain has roughly 86 billion neurons. Neurons are interconnected by many incoming elements known as dendrites and an outgoing element referred to as an axon. The neuron receives impulses via the dendrites, analyzes them, and delivers a signal down the axon. The input and output signals are designated as activations. A neuron sends electrical signals (action potentials) to another neuron via a synapse at its end. Most neurons have multiple connections with other neurons. Signaling between the neuron and one of its nearby neighbors takes place at the synapse. Human brains usually have an estimated 100 trillion to 1000 trillion synapses [14].

The synapse allows for modulation of the electrical signal traveling through it. The degree to which this occurs is referred to as the synaptic weight (Wi). As a result of the modulation of electrical signals when a neuron sends an action potential to a neighboring neuron, the response of the neighboring neuron to the incoming signal varies based on the weight of the connection. As such, the weights of connecting units determine how a connecting neuron reacts to learning experiences when it is stimulated. Learning will take place to modify the connection(s) of the synapsis between two or more neurons in order to achieve a predetermined result without changing the overall structure of the neuron network (at least in respect to its basic fundamental configuration). Because of this capacity of the brain, it serves as a great model in creating a machine-learning system [14].

The rapid rate of growth in the AI/ML industry is causing companies and startup organizations that are developing increasingly sophisticated and powerful architectures in the digital computing space to direct their attention to this segment of the market.

The growing availability of commercial opportunity within the markets gives companies and venture capitalists significantly more attractive financial returns and potential earnings as compared with a large, mature market. The larger, stagnant market will continue to have a competitive environment, thus decreasing income for its participants over time. As a result, technology with high growth rates are receiving much greater attention, even when the overall market for that technology is relatively small.

In the future, the best way to achieve continuous improvement in performance will be by using different types of accelerators based on specialized architectures. The reason we state that this is the case, is that it typically takes about ten years for a revolutionary transistor design that has been demonstrated in a lab to be designed into a production process for commercial purposes. The U.S. Office of Science and Technology Policy (OSTP) document created by Robert Leland, examined the landscape of technologies that could replace CMOS and has identified numerous candidates that could be implemented [9]; however there are no substitutes currently being demonstrated in the lab. This means that we are currently ten years behind addressing this issue regarding a scalable post-CMOS technology solution.

Since there are no other options available, the only way for computer architects to continue building their hardware will be by using architectural specialization and improved packaging over the next ten years. Competing with an ever-developing general purpose computer ecosystem will make it more difficult for hardware architectures to prosper. Historically, the route of architectural specialization has produced little or no profit due to long lead times and costly development processes. Research done by Thompson and Spanuth [15] supports that diminishing returns from Moore's Law will promote the development of specialized architectures as a fruitful and economically viable replacement to universal computing systems. The selection made by computer architects will dramatically affect the manner in which algorithms are developed and the environments in which they will be run.

Thus, in the absence of further improvements in transistor technology or alternative devices to support the current trajectory of Technology Scaling, the only way for the computer architect to continue developing better performing computer systems is to utilize transistors in an optimized manner for the specific scientific problem being solved.

The use of Artificial Intelligence by semiconductor companies may present the single biggest opportunity to capture between forty and fifty percent (40-50%) of the cumulative value of their entire technology stack in the last three decades. To avoid the pitfalls of previously losing value captured due to operational limitations, semiconductor companies are required to adopt a new value creation model that provides specific, customised, and integrated solutions to focus on certain industries, referred to as "Micro-Verticals" [16].

There is a robust consensus that the diminishing of Moore's Law will result in a wider array of accelerators or specialized technologies than observed in the preceding thirty years[8].The advancement of artificial intelligence (AI) in executing cognitive processes such as perception, reasoning, and learning, together with its application across several domains, has necessitated reliance on hardware as a fundamental driver of innovation, particularly for logic and memory operations[16].

Examples of this trend are observable in smartphone technologies, which feature multiple specialized accelerators integrated on a single chip; in hardware used in large data centers, such as Google's Tensor Processing Unit (TPU), which optimizes the TensorFlow programming framework for machine learning applications; in field-programmable gate arrays (FPGAs) utilised in the Microsoft Cloud for Bing search and other applications; and in a diverse array of additional deep learning accelerators


Figure 3 Next wave of AI electronics driven application [16].

There are three universally accepted approaches to the solution of making a hardware accelerator. These are hardware-driven algorithm design, algorithm-driven hardware design, and co-developing hardware and algorithm [8].

Deep Neural Networks (DNNs), often known as deep learning, constitute a segment of the expansive domain of Artificial Intelligence (AI), defined as the science and engineering of developing intelligent computers capable of accomplishing objectives akin to human capabilities, as articulated by John McCarthy, the computer scientist who introduced the term in the 1950s[8].

There has been tremendous momentum in the realm of neuromorphic computing, and the largest contributor to the success of these types of solutions appears to be the continued advancements that have occurred in various areas of artificial intelligence (AI). The development of algorithms is significantly impacting the form factor of hardware (e.g., as it relates to semiconductor manufacturers, they will manufacture only the highest margin product and move forward with the top down approach of developing new AI Solutions (e.g. the creation of AI Design Tools to enable hardware accelerators to be created based on what workloads the hardware accelerator will handle) based on this potential, revenue opportunities would be impacted greatly and determine if a large share of future demand will include AI chips..

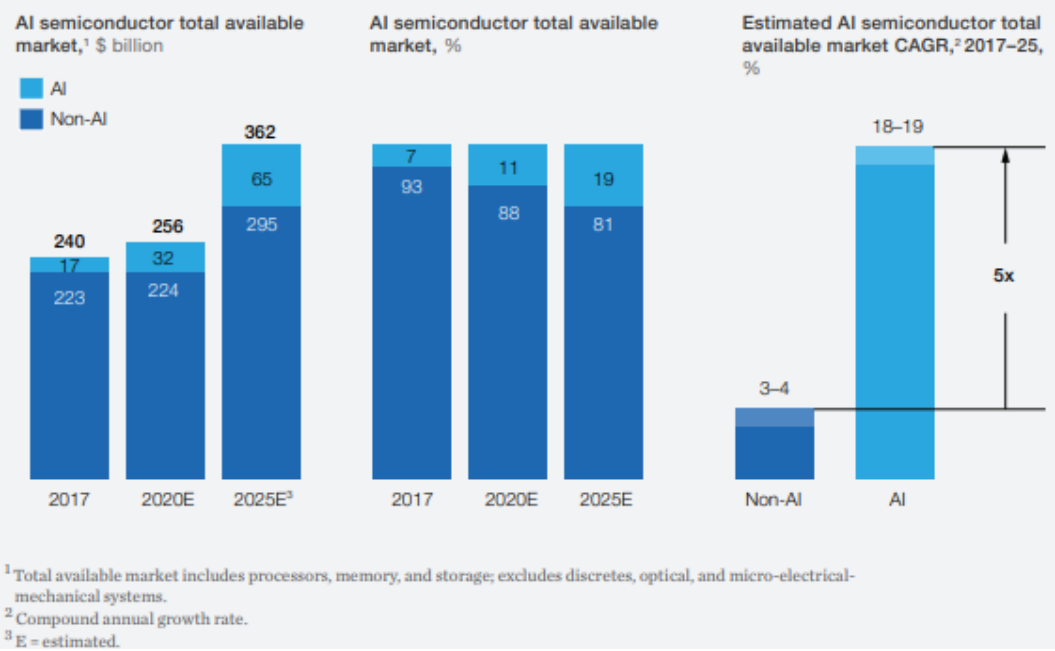


Figure 4 The growth of semiconductors associated with AI is anticipated to be fivefold compared to the growth of the rest of the industry [8].

Recent evolutions of AI hardware started with CPU processors in 70's, to FPGA in 80's, GPUs in 90's, and TPUs in 2010s'and other researches are going on light and quantum computing for advancements in artificial intelligence.

The training, inference, and Edge/IoT inference phases represent distinct markets for hardware accelerator developers. The artificial intelligence (AI) technology stack may be delineated into nine layers:

- Services – comprehensive solutions encompassing training data, models, hardware, and other components (e.g., voice recognition systems)
- Training – information provided to AI systems for examination
- methodologies, architecture, algorithms, and framework stack of the AI solution delineates the platform.
- The approaches are ways for optimizing the weights assigned to model inputs.
- The architecture refers to the systematic methodology for extracting characteristics from data, such as convolutional or recurrent neural networks.
- The algorithm stack contains a series of rules that incrementally adjusts the weights assigned to certain model inputs in the neural network during training to enhance

inferences.

- The framework is a software package designed to delineate designs and execute algorithms on the hardware via the interface.
- Interface - mechanisms inside a framework that establish and enable communication routes between software and the underlying hardware.
- Hardware – comprises head nodes and accelerators within its architecture. A head node is a component that orchestrates and organizes computations among accelerators. Accelerators are silicon processors engineered to perform extremely parallel processes essential for AI, facilitating simultaneous calculations.

From the technology stocks listed above, companies will find opportunities in the artificial intelligence (AI) market, with leaders already emerging. The opportunities in the existing market and the potential in new market creation comprise the areas of compute, memory, storage, and networking. In the computing area, accelerators for parallel processing, such as GPUs and FPGAs, represent the opportunities in the existing market, while workload-specific AI accelerators have potential for new opportunities. Companies can also benefit from designing high- bandwidth memory and on-chip memory (SRAM) in existing markets, while emerging non-volatile memory (NVM) can create new markets. The potential growth in demand for existing storage systems, as more data needs to be retained, also has market value in the current market. AI-optimized storage systems and emerging NVM as storage devices are the opportunities for new markets. In the networking area, infrastructure for data centers is beneficial in the current market, while searching for programmable switches and high-speed interconnects are new opportunities that should be sought by the companies.

The prospects will encompass both data centers and edge computing. The majority of compute growth in data centers will arise from increased demand for AI applications in cloud-computing environments. At this site, GPUs are currently utilized for nearly all training applications. We anticipate that they will soon start to cede market share to ASICs, resulting in a about equal distribution of the compute market between these alternatives by 2025.As ASICs are introduced into the market, GPUs will likely become increasingly tailored to satisfy the requirements of deep learning. Besides ASICs and GPUs, FPGAs will play a limited role in future AI training, mostly for specialized data center applications that need rapid market entry or customization, such as developing new deep learning applications. Currently, the CPU constitutes around 75 percent of the market for inference. They will cede

advantage to ASIC when deep learning applications proliferate. We anticipate a nearly equal distribution in the computer market, with CPUs representing 50% of demand in 2025 and ASICs comprising 40%. .Today, most edge training occurs on consumer laptops and computers; however, more and more devices will be able to collect data and train models on site under the right conditions. During the process of oil and gas exploration, geological data about what is located below the earth's surface can be collected while drilling wells; this collected data can then be used to assist with building computer model programs. At present there are nearly as many customers developing Machine Learning Accelerators for CPU technology and ASIC technology; however, our expectation is that once integrated ASICs are available in the form of SYSTEMS ON CHIPS (SOC) approximately 70% of future market demand will come from SOC/ASICs and approximately 20% will come from FPGAs to serve niche/cost sensitive applications. At this time, most edge computing devices (i.e., computers located at the edges/endpoints of a network) conduct inference using CPU or ASIC technology; however, as we expect that many new applications, including autonomous vehicles, will eventually be developed using GPUs, we forecast that by 2025, ASICs will account for approximately 70% of total edge inference market and GPUs will account for 20% [16].

DNNs consist of multiple processing layers and, thus may include multiple types of computations. Since the majority of the DNN layers weights will compute a weighted sum, many computations will be multiply-accumulate (MAC) operations. In general, MAC operations are determined by the layer geometry; however, since the DNN configurations may vary greatly based upon an application's requirements, DNN configurations will demonstrate tremendous variability.

A DNN's architecture/configuration can take many different forms. Variability in configuring DNNs allows for achieving maximum DNN performance (i.e., accuracy) and cost-effectively, respectively, and for adapting quickly to the rapid changes in application workloads and maintaining close functional compatibility with the software environment through the global network. The architecture of DNN processors is generally defined as a rectangular grid of processing elements (PEs), which typically feature MAC units for computing and may include local storage accessed by the PE. There are different architectures for DNN processors based upon the number of storage tiers, their available capacity, and the method of connecting processor to memory via a network.

To compare various competing architectures, there must be some evaluation of many metrics. In the

same manner, performance estimates of hardware accelerator designs require utilising all available performance measures as well as being aware of the accuracy of the dataset utilised and what specific tasks each design will perform. A simple DNN can be designed to function with the lowest possible power consumption, throughput, and cost, but may ultimately fail to provide optimal performance for large workloads. Conversely, a processor could potentially be rated only as having multipliers, reflecting on-chip bandwidth not being accounted; therefore, when marketed, it may demonstrate low cost, high throughput, rapid rates of accuracy, and low chip power, when compared to other similar products may be inaccurate. The test plan/architecture must be documented; and, the type of simulations of the architecture must also be documented, including the total number of images evaluated. Adequate efficiency assessment is essential; otherwise, it results solely in unrealistic and fanciful solutions. The book [14] summarizes the evaluation method to determine if a DNN system is a suitable option for a certain application as follows.

The determination of accuracy is dependent upon the performance of the hardware to execute the desired functionality. Latency and throughput establish the capability for the hardware to operate in a timely manner (real-time) and process information without delay. Energy/power consumption will primarily affect the physical size of the hardware due to the processing capability associated with that size. Cost will be primarily driven by the chip size and the required external memory bandwidth, and therefore, the overall investment in this solution. The term flexibility refers to the variety of functions that a given hardware design can perform. Scalability is the evaluation of whether a given design will enable the same development (design) to be used for various deployments (e.g., in the cloud, at the edge), and whether the system can be easily scaled in size relative to the size of the DNN model.

With an end to Moore's Law and the consequent reduction of development into new technologies, the economic feasibility for specialized architectural solutions will outweigh that of general-purpose systems. The impact of this change will significantly alter algorithms and programming environments. In the absence of any groundbreaking new transistor or device to facilitate ongoing technology scaling, the sole instrument available to a computer architect for achieving further performance enhancements is to utilize transistors more efficiently by tailoring the architecture to the specific scientific problem.

This project aims to integrate all design metrics for the creation of an Amharic handwriting recognition application utilizing a hardware accelerator implemented on an FPGA.

## 2.2. Research on FPGA-Based Acceleration for Machine Learning

The rapid development of machine learning (ML), particularly deep neural networks (DNNs), has driven demand for more efficient computing platforms. In his paper in 2015, LeCun and others demonstrated that deep neural networks (DNN) provided state-of-the-art results on image recognition and natural language processing (NLP), but at the significant cost of high computing requirements [9]. An analysis by Strubell and others in 2019 found that training a single large DNN had energy demands equivalent to that of several lifetimes of vehicles and raised questions regarding sustainability and practicality when using DNNs at a large scale.

According to Hennessy and Patterson, general-purpose CPUs were not well suited to ML tasks due to their reliance on a sequential execution model [10]. In comparison, GPUs, such as the NVIDIA RTX 4070 Ti, can achieve higher throughput due to their ability to exploit massive parallelism; however, GPUs are also very power-hungry devices, using around 285 watts per GPU[11]. Jouppi and others introduced the Google Tensor Processing Unit (TPU) to address these issues by providing efficient solutions for ML tasks; however, TPUs are not reconfigurable and, as such, cannot support a wide range of different workloads [12].

FPGA can be seen as a compromise between the flexibility offered by GPU technology and the faster execution speed of GPU technology. In addition to being able to support many different configurations of tasks and producing similar levels of throughput compared to CPUs, FPGAs also consume significantly less power than GPUs, while still producing similar levels of throughput performance compared to CNNs once they have been optimised using such methods as quantisation, pruning, amortisation and pipelining [14]. Jacob et al. (2018) [15] illustrated the advantages of quantising a model to reduce it to a size that is 75% smaller than the original model, while only losing 1% accuracy from the original version of the network. Reuther and colleagues (2018) [16] recently collected information regarding development efforts for accelerator platforms and concluded that the FPGA is a viable platform for accelerator development due to its ability to be configured to process various types of data. Reuther's research also indicates that the slow rate of adoption of FPGAs is due in part to the complexity associated with designing FPGA-based accelerators and the limited number of engineers skilled in both ML and hardware design.

Currently, while the use of FPGAs for ML workloads is growing, most of the work done has concentrated on common datasets like ImageNet and MNIST, while less common datasets, such as handwriting recognition applications for under-represented languages, have seen little research thus far [17].

### 2.3. Studies on Amharic Handwriting Recognition

Although the technology behind Deep Learning has improved handwriting recognition to an extent, there are not many studies that have analysed the Handwriting Recognition System (HRS) with regard to an under-researched language such as the Amharic Language. Amharic is the official language of Ethiopia; it is written using a syllabic script, and there are 231 different characters in the Amharic writing system. There are 33 consonants, and each orthographic consonant may be modified by 7 different vowels [17].

Assabie and Bigun (2009) were researchers who were pioneers in performing offline handwritten recognition of Amharic through the use of Hidden Markov Models (HMMs). They were able to achieve around 85% accuracy when utilizing small datasets; however, their study utilized simpler feature extraction methods and thus created restrictions on how accurately the researchers could perform this task [18]. Teferi et al. (2019) utilized deep learning methods to perform offline handwritten recognition of Amharic. In comparing the results to previous studies, the authors found that the CNN structure required increased computational limits and generated lower than 90% recognition accuracy due to the limitations of datasets available at that time [19]. Tesfaye (2022) increased the recognition rates of handwritten Amharic through the use of CNN networks and was able to achieve recognition rates between 85% - 90% through the use of roughly 3000 samples in the dataset [20]. While it was determined that this amount of data allows for a small portion of the size of the MNIST dataset, data scarcity and variability within classes were cited as significant hindrances [20].

Unlike Latin and Chinese script collections, which typically have over 5000 samples; this means that the generalisation ability of deep learning models is significantly limited when comparing general Amharic resources to a larger and more varied global sample dataset compared to that used by the Latin and Chinese character sets [17, 19, 20]. Alemu and Doe (2023) stated that, although there have

not been significant investigations of hardware acceleration and Design related specifically to Amharic and other Ethiopian societies; on the contrary, hardware acceleration related to designing and testing infrastructure focuses heavily on Globally Dominant Languages [21]. General-purpose FPGA tools may support CNN-based hardware acceleration, such as Xilinx's Vitis AI; however, this support is not geared towards the specific characteristics of syllablic scripts, particularly Amharic [22].

### 2.4. Gaps in the Literature

The author of this thesis was inspired to conduct his research based on the literature reviewed in this thesis, which points to three key deficiencies:

1. The variety of Languages Within the Ethiopic Family: Although there is a lot of research that has been performed on Ethiopic in Amharic, Most research on Amharic focuses on a very limited number of languages and thus, Amharic represents a very limited selection of the Ethiopic language family and consequently offers a smaller cultural and linguistic value to society[21].
2. Data Set Sample Size Limitations: There are a number of small Data Set of Handwritten Amharic available, with most of the available Web-Sourced Data Sets (currently available) containing between 3000-5000 samples., thus preventing enough training of Deep Convolutional Neural Network (DCNN) forHandwritten Amharic[19][20].
3. Co-Design of Hardware and Algorithms in a Collaborative Manner: FPGAs have been shown to be effective in improving the speed of Machine Learning and in that respect, however; there are little to no studies which have been done on the combined development of Machine Learning Algorithms and FPGA Hardware that work together to build a combined solution to the task of Amharic Character Recognition[16][22].

To fill the gaps that have been identified above, a script based Co-Design of Neural Networks and FPGAs is required to develop a solution that addresses the Data Scarcity Issues, increases the Energy Efficiency of the solution, and addresses the specific language of the task.

# 3. Methodology

## 3.1. Image processing

To initiate the image processing task, Amharic handwriting has also been collected. The range of collected data is from ʊ to ፖ, and a total of 250 samples representing various examples of handwriting were collected.

Figure 5 illustrates the complete workflow of the character extraction process and contains an overall view of the workflow associated with converting handwritten Amharic text images into segmented character samples.

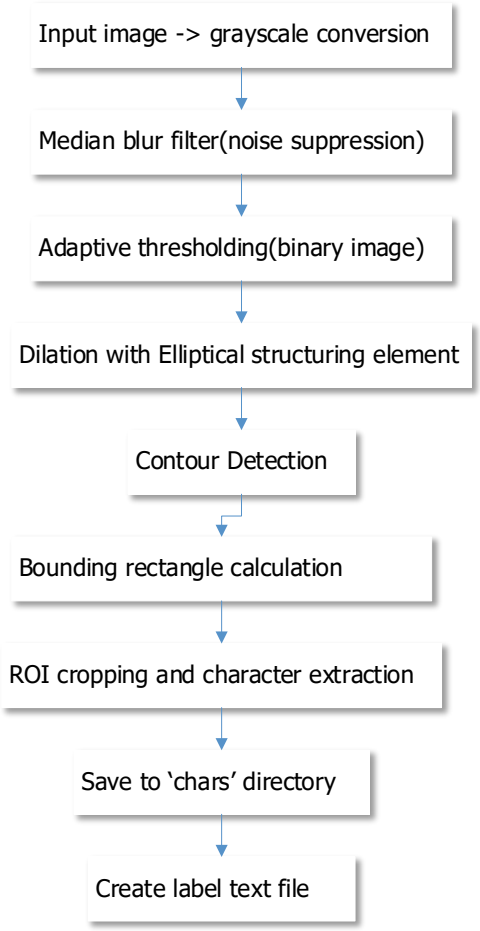


Figure 5 Amharic Handwritten Character Extraction Workflow

To read an image using OpenCV in Python, you can use the cv2.imread() function.

```
def read_image(img_path):
    image = cv2.imread(img_path)
    return image
```

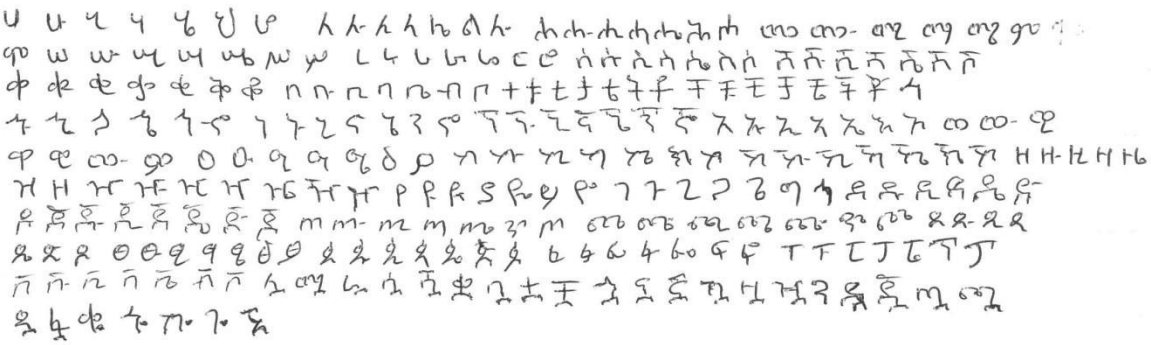

Figure 6 Amharic handwriting sample data

The cv2.imread() method loads an image into memory via its filename or file path specified by the variable img_path; once loaded successfully, you can extract each letter as indicated in the function below. Many methods of image processing are combined to do this and save each letter in separate files as shown below.

```
Parameters:
    img (numpy.ndarray): The input image from which characters will be
extracted.
    output_dir (str): The directory path where the extracted character images
will be saved.
def extract_text_chars(img,output_dir):

    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    blur = cv2.medianBlur(gray, 7)
    thresh = cv2.adaptiveThreshold(blur, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY_INV, 7, 11)
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
    dilate = cv2.dilate(thresh, kernel, iterations=1)
    cnts = cv2.findContours(dilate, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    if len(cnts) == 2:
        cnts = cnts[0]
    else:
        cnts = cnts[1]
    chars_path = os.path.join(output_dir,'chars')
    if not os.path.exists(chars_path):
        os.makedirs(chars_path)
    for char_idx, character in enumerate(cnts, start=-len(cnts)):

        x, y, w, h = cv2.boundingRect(character)
        roi = img[y:y + h, x:x + w]
        filename = 'char'+str(char_idx)+ '.jpg'
        resized_image = cv2.resize(roi, (28, 28))
```

```
save_img(chars_path,filename=filename,img=resized_image)
```

To extract individual characters from a sample image, a systematic image processing approach was implemented. Initially, to decrease the number of image dimensions and improve processing efficiency, the image is transformed to a single channel (grayscale). Next, the image undergoes a "Median Blur" operation (removing noise), while keeping the edges of the text intact. Once noise has been removed from the image, "adaptive thresholding" is performed (threshold value is based on the average brightness of neighbouring pixels) to create a binary image (black/white) with the text characters clearly being separated from the background.

Then, an elliptical "Structuring Element" (a defined sized element that can enlarge or reduce an object based on the size of the structuring element) is created and utilized to "dilate" the thresholded image (connects the text characters together). This enables the connected characters to be large enough to allow for easy individual detection and extraction, via a Contour extraction operation.

Upon the conclusion of the "dilation" process and the boundaries of the character images identified in the previous steps, a "chars" directory is produced within the Output directory to hold character images. An iteration through all identified contours occurs in order to create a bounding box for each detected character. The Character Image is then retrieved by cropping the image from which the character was located based on the bounding coordinates of the bounding rectangle. The character extracted will be saved as a(n) image named uniquely based on the index number where it is saved in the chars directory.

In this code example, Character Labelling and the Character Extraction can be separated as the Labels of the character images extracted are contained in a text file vs being connected with the character image.

To further explain, the code example produces the extraction of the character image from the image file at a certain stage, after the extraction has been completed, the labels (contained within the text file) are produced. Therefore, to store in a manner that will relate the labels and the respective character images, the text files and images of extracted characters are created in different folders but in corresponding sequences, so therefore one can look at the sequence of the

text files to locate specific labels for specific character images.

image

image_80\010000.bin.png

image_80\010001.bin.png

image_80\010002.bin.png

image_80\010003.bin.png

image_80\010004.bin.png

image_80\010005.bin.png

image_80\010006.bin.png

image_80\010007.bin.png

image_80\010008.bin.png

image_80\010009.bin.png

label

image_80\010000.gt.txt

image_80\010001.gt.txt

image_80\010002.gt.txt

image_80\010003.gt.txt

image_80\010004.gt.txt

image_80\010005.gt.txt

image_80\010006.gt.txt

image_80\010007.gt.txt

image_80\010008.gt.txt

image_80\010009.gt.txt

The image contains in ".png" format each Amharic character extracted from handwriting samples I have. And the txt contains the Amharic name of the character in the image.

 the corresponding label text for this image is "ኽ"

## 3.2. Methodology of the Proposed System

The proposed system is implemented on the Intel DE1-SoC development board, which integrates an ARM-based hard processor system (HPS) running Linux together with FPGA fabric. The methodology combines offline neural network training with hardware-accelerated inference to achieve real-time handwritten Amharic character recognition.

First, handwritten Amharic character images are collected and preprocessed to extract normalized 28×28 grayscale character samples. These samples are used to train a neural network classifier on a CPU/GPU workstation. Once training converges, the learned parameters (weights and biases) are converted into FPGA-compatible memory files and deployed to the DE1-SoC platform using the Quartus design workflow.

Verilog HDL is the hardware description language that was utilized for the development and hardware design of the artificial neural network architecture, incorporating Intel Quartus Prime's synthesized reference design within the FPGA architecture to provide forward propagation-based acceleration capability.

The software application created by this FPGA architecture using an operating system such as Linux will run solely on the ARM HPS platform, converting the C code into a C application, allowing for connection and interaction with the FPGA hardware from the User Space through standard access methods to memory. This will enable transmission of pre-character image data to the FPGA accelerator via the established memory mappings for the recognition process, and retrieval of the associated classification results via the same mappings.

TheARM HPS architecture will function as both the control and display interfaces between the two systems which include the C application on the HPS and the FPGA-based accelerator. The C application has the capability of computing and displaying the accuracies for each of the different input data types during the test, thereby establishing proof of functionality of the FPGA-based recognition system.

The complete process outlined lends itself to the effective performance and deployment of the artificial intelligence neural network generated by simulation (and/or otherwise) on the DE1-SoC

devices. The speed of inference provided by the FPGA accelerators will be coupled with the ability of the ARM processors to handle the data transfer, determine finger print/unit matches, and provide the user interface to the device. The combination of these two devices will produce superior performance and flexibility for handwriting recognition applications within embedded devices.

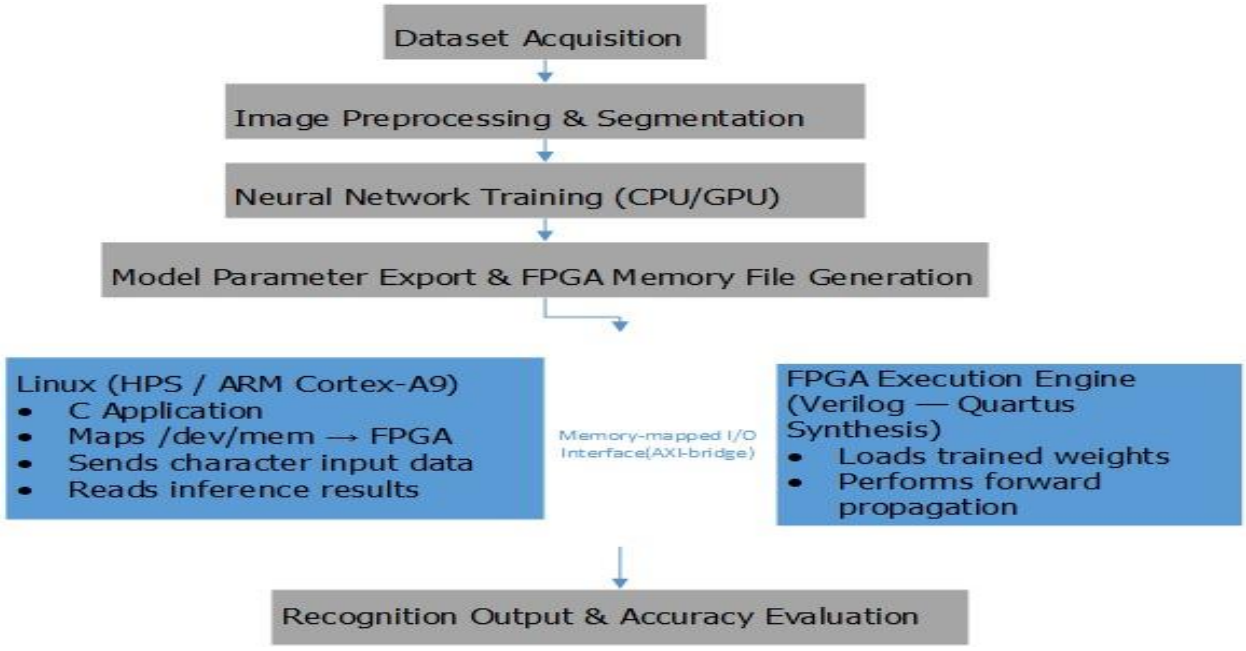


Figure 7 Overall System Architecture for FPGA-Accelerated Handwriting Recognition

## 3.3. Implementation

Neural network implementation using custom-built neural hardware is referred to as a "Hardware Implementation" of a Neural Network. Custom hardware (hardware optimized for performing neural computations) is usually purpose-built to perform the calculations required to run and train neural networks. The raw design of these specialized HWs is usually optimized for high-speed parallel processing of large amounts of data while allowing for the most efficient routing and storage of input data used to perform calculations.

Compared to hardware implementations, software implementations provide more flexibility and programmability than hardware implementations. Software-based methods, such as TensorFlow and PyTorch, allow for the easy construction, training, and modification of neural networks using traditional programming languages. In contrast, software implementations have a general-purpose architecture that allows for an expanded range of applications compared to hardware implementations.

The overall cost of hardware implementations can be significant due to the requirement for custom-built HW components (designing and manufacturing ASICs or buying high-performance GPUs). However, hardware implementations have better (faster and more energy efficient) performance than software implementations due to their specific HW design and the many advantages of a specialized HW architecture. Additionally, hardware implementations require greater resources (knowledge and money) compared to software implementations, which allows for a much higher level of flexibility and ease of development. Choosing between hardware and software implementations will ultimately depend upon an individual application, its performance requirements, and available resources.

An example of our neuron model for the Amharic handwriting recognition task is illustrated below. This model's input processing will prepare the data to be presented to the neuron. Input processing will also be responsible for features that are processed (e.g. normalization, amplification of the input signal, or feature extraction.)

```
// Pseudocode-level description for neuron.v

on reset:
  w_addr = all 1's (prepares for first write)
  wen = 0
  r_addr = 0
  sum = 0
  outvalid = 0

// Load weights when config selects this neuron
if (weightValid && (config_layer_num == layerNo) && (config_neuron_num ==
neuronNo)):
  w_in   = weightValue[dataWidth-1:0]
  w_addr = w_addr + 1
  wen    = 1
else:
  wen    = 0

// Bias handling
if pretrained:
```

```
  bias = concat(biasReg[addr][dataWidth-1:0], zeros(dataWidth)) // inline
scaling
else if (biasValid && selected_by_config):
  bias = concat(biasValue[dataWidth-1:0], zeros(dataWidth))

// Stream processing
if (rst || outvalid):
  r_addr = 0
else if (myinputValid):
  r_addr = r_addr + 1

// Pipeline alignment
myinputd    <= myinput              // align with memory read stage
weight_valid <= myinputValid
mult_valid   <= weight_valid
mux_valid    = mult_valid

// Multiply
mul <= signed(myinputd) * signed(w_out)  // 2*dataWidth

// Accumulate with saturation
if (rst || outvalid):
  sum <= 0
else if (r_addr == numWeight and muxValid_f): // end-of-stream edge
  // sum <- sum + bias (saturated)
  if (positive(bias) and positive(sum) and negative(sum + bias)): sum = MAX_POS
  else if (negative(bias) and negative(sum) and positive(sum + bias)): sum =
MAX_NEG
  else sum = sum + bias
else if (mux_valid): // normal MAC
  // sum <- sum + mul (saturated)
  if (positive(mul) and positive(sum) and negative(sum + mul)): sum = MAX_POS
  else if (negative(mul) and negative(sum) and positive(sum + mul)): sum =
MAX_NEG
  else sum = sum + mul

// Detect the last multiply has just been applied
muxValid_d <= mux_valid
muxValid_f <= (!mux_valid) && muxValid_d

// Output valid when bias add just happened
sigValid <= ((r_addr == numWeight) && muxValid_f) ? 1 : 0
outvalid <= sigValid

// Activation
if (actType == "sigmoid"):
  out <= Sig_ROM( top_bits(sum) )
else:
  out <= ReLU(sum)
```

The Neuron module takes N inputs and N corresponding weights as input and produces an output based on the weighted sum of the inputs.

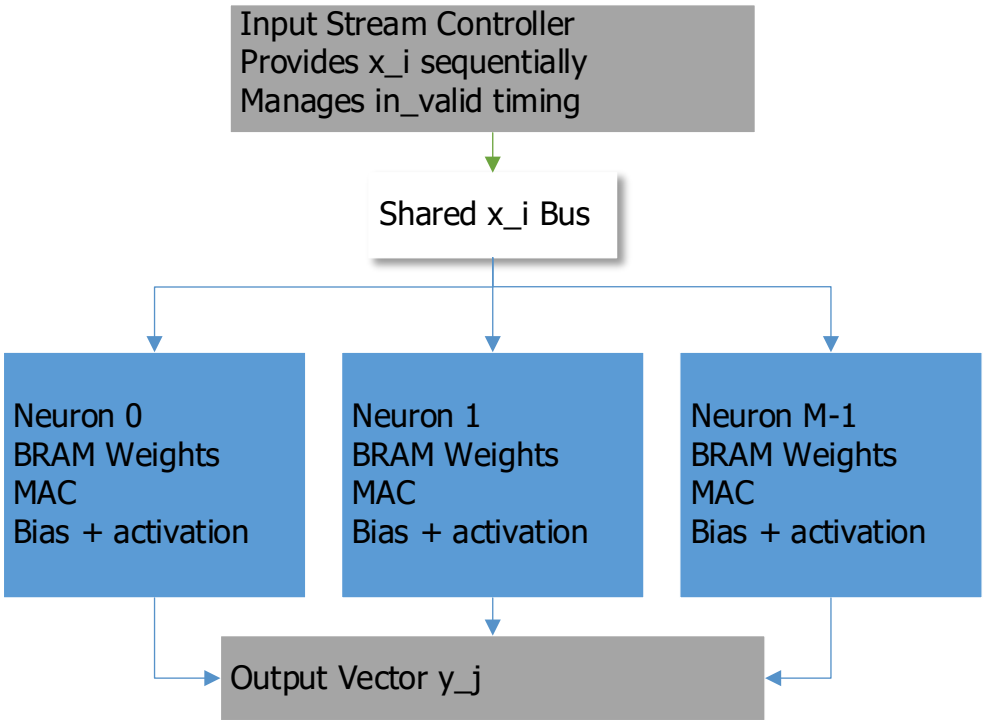


Figure 8 The neuron module

The neuron module implements a streaming multiply–accumulate (MAC) neuron with activation. It receives a sequence of numWeight inputs on myinput asserted with myinputValid, and for each input, it fetches the corresponding weight from an internal Weight_Memory block using r_addr as the read address while ren = myinputValid drives streaming. Weights are written into this memory during configuration when weightValid is asserted, and the config_layer_num and config_neuron_num select this neuron; in that case, w_in captures weightValue[dataWidth-1:0], w_addr increments, and wen pulses to store the weight. Bias is either preloaded from a file when pretrained is defined (via $readmemb(biasFile, biasReg)) or written at runtime when biasValid is asserted for the selected neuron; in both cases, it is left-shifted by dataWidth bits ({bias[dataWidth-1:0], zeros(dataWidth)}) so that it lives in the same numeric scale as the 2×dataWidth accumulator. The datapath pipelines input and weight to keep them aligned: myinput is delayed one cycle into myinputd while Weight_Memory produces w_out; on the next cycle, the module computes mul = $signed(myinputd) * $signed(w_out), a 2×dataWidth product. The running sum sum accumulates these mul values with explicit saturation protection: when adding mul or the final bias, if both operands share a sign and the provisional result flips sign, the sum is clamped to the maximum positive or minimum negative representable value at the 2×dataWidth width. This saturating addition is applied both during the MAC loop (sum + mul) and in the finalization step (sum + bias). Control and timing are coordinated with a small valid pipeline:

weight_valid follows myinputValid, mult_valid follows weight_valid, and mux_valid equals mult_valid, marking when mul is valid to add. The read address r_addr increments with each myinputValid and resets on rst or once outvalid fires. An edge detector (muxValid_d and muxValid_f) detects the "last-multiply-applied" moment at the end of the input stream (r_addr == numWeight), triggering the single-cycle bias add and asserting sigValid, which drives outvalid for one cycle. The output out then applies the selected activation: if actType == "sigmoid,", a small Sig_ROM is indexed by the top sigmoidSize bits of sum to produce the output; otherwise, a ReLU module takes the 2×dataWidth accumulator and emits a rectified result.

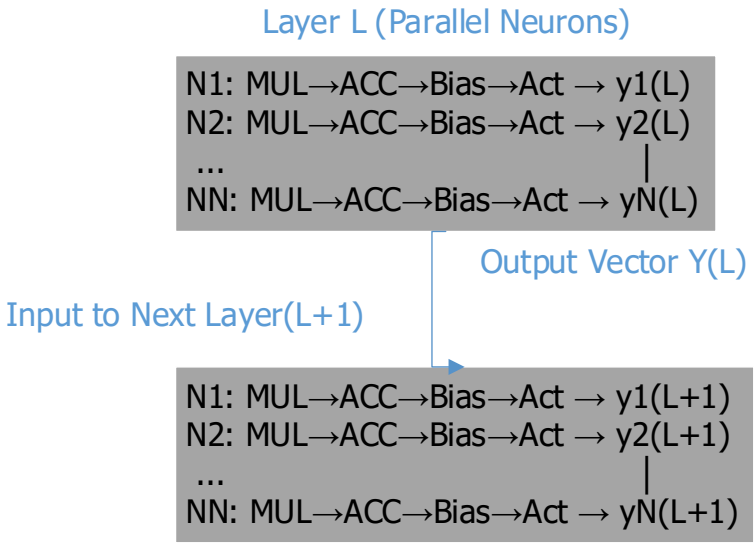


Figure 9 scalable and pipelined architecture

In hardware mapping terms, the mul operation infers a DSP block, the per-neuron weight memory infers a block RAM (M10K), the sigmoid ROM typically infers ROM in MLAB or M10K depending on size, and the adders, comparators, saturators, counters, and valid/control registers map to ALMs (logic and flip-flops). The design maintains a one-cycle alignment between input and weight (hence myinputd), pipelines the multiply, and uses the valid chain so accumulation occurs precisely for each product; outvalid rises only after the final bias has been safely added and the activation result is ready. Bias alignment via left-shifting by dataWidth ensures accumulator-domain consistency since products live at 2×dataWidth. In short, for each inference, the neuron streams numWeight inputs, performs a DSP-based signed multiply per input and a saturating logic-based add into sum, adds the bias with saturation once at the end, applies activation, pulses outvalid, and resets its input address counter to prepare for the next input vector.

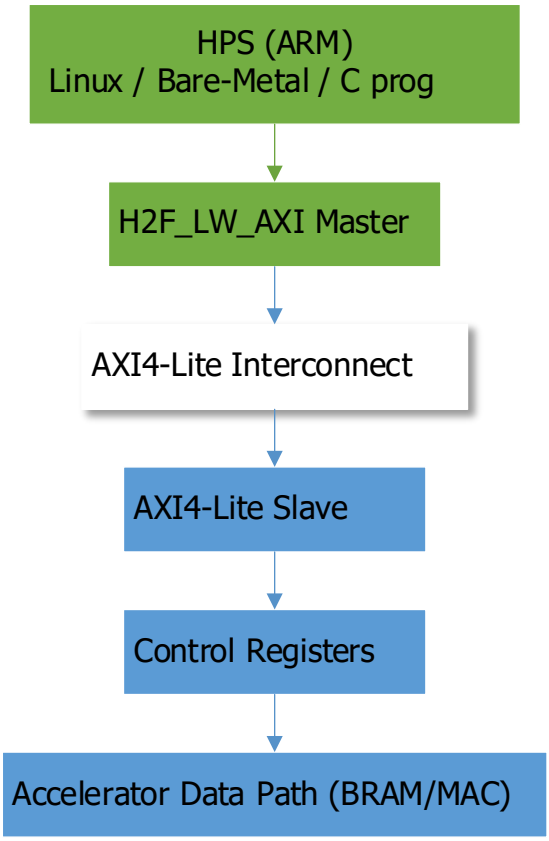


Figure 10 Memory Bus & BRAM-Based Design

In a fully connected neural network, also known as a dense network or a fully connected layer, each neuron in a particular layer is connected to every neuron in the previous layer.

Let's consider a simple example with two layers: an input layer and a hidden layer. When there are 'n' number of input neurons and 'm' number of hidden neurons, all 'n' number of input neurons will provide input to each of the 'm' number of hidden neurons. Therefore, every input neuron will have its output included in the calculation of the outputs of the 'm' number of hidden neurons.

For each neuron in the hidden layer, the output is the sum of the product of the weights and the outputs of each of the 'n' inputs, as well as a possible bias added to the product. The resulting output from the summation and potential bias will then propagate through an activation function before it is sent to the next layer.

Because of the way that this structure is built, it allows the network to learn and understand non-linear

features of the data that it is receiving.

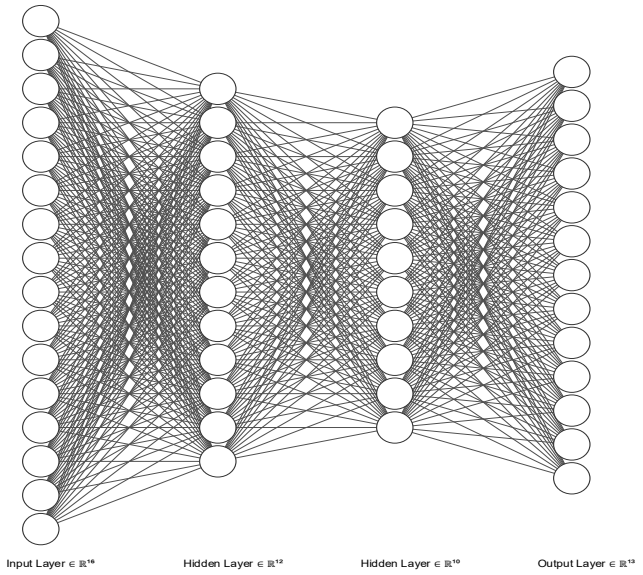


Figure 11 A neural network

The input information that a neural network will receive will typically be pixel information (which is interpreted in an image) and made from light values (intensity) and will all thus contain positive values because of the nature of how pixels have been established from the range of colours that can be represented as an image. As such, weights and biases may be any value of real numbers (including both positive and negative numbers and fractions).

A neural network can use two different formats to store and work with floating-point numbers, such as for weights or biases and these formats will be the IEEE floating-point format (also referred to as IEEE 754), which consists of both a single precision (32-bit) format and double precision (64-bit) format.. Various formats are available for the representation of fractional values, but with greatly varying levels of accuracy/fidelity. As a result, floating-point formats are a good option when the desired output is a large range of values or needs to support a very high degree of accuracy.

Fixed-point number representations allocate fixed-length bit sequences for both fractional and integral

segments of a value. The way the binary point is placed within the numeric sequence determines the value's scale factor. In comparison to floating-point number representations, Fixed-point formats are more efficient to implement in terms of resource and power requirements because of their higher computational efficiency and smaller memory footprint. However, true fixed-point formats offer less range and reduced degree of accuracy, potentially limiting the ability of the neural networks to adequately learn and interpret different types of data. Thus, fixed-point formats are often employed in environments where there are resource limitations or there are specific constraints on the hardware available.

Floating-point number representations enable the representation of very large numbers at greater precision than fixed-point format; however, combining floating-point representations with resource-limited environments or hardware restrictions can produce significant resource usage and difficulty in manipulation.

Fixed-point formats enable larger neural networks to run on resource-constrained devices while still providing a computationally efficient implementation of the model.

Fixed-point formats on FPGAs may introduce the risk of losing accuracy when compared to floating-point representations. However, when operating normally and avoiding overflows, a 32-bit fixed-point format implementation will perform better than a 32-bit floating-point solution because fixed-point formats are easier to implement in hardware and require fewer clock cycles and resources to compute results.

FPGA design utilizing fixed-point formats will enable designers to customize the number of bits and scale factors according to the application, allowing for the attainment of the desired trade-offs between range, accuracy, and resource usage.

Through the appropriate selection of parameters, designers can customize the fixed-point logic implemented in their designs to achieve optimal performance based on their respective applications and to customize it for the particular algorithms used, as well as to optimize computation with a view toward different signal processing tasks.

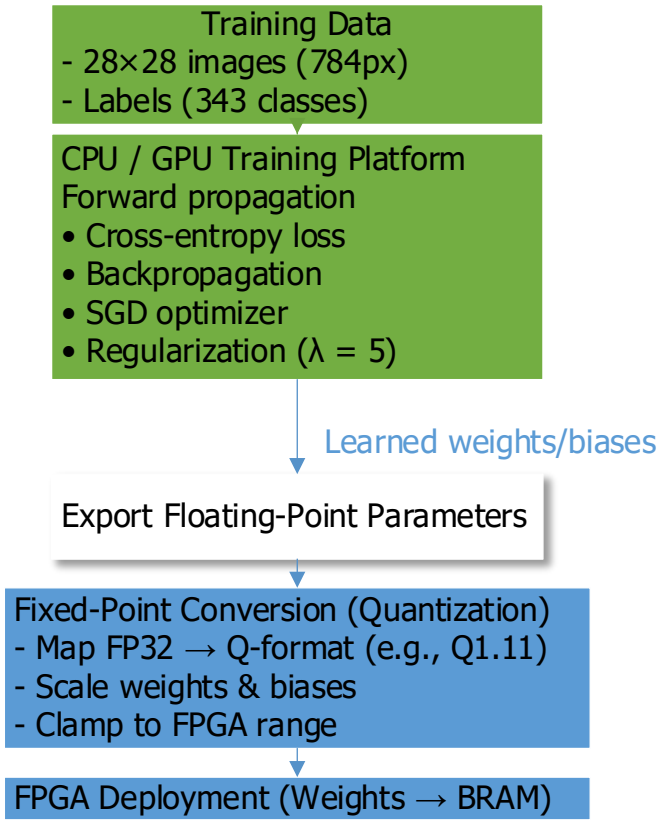


Figure 12 FPGA deployment setup

When using fixed-point representation, one must first specify the total number of bits for the representation, the number of bits allocated for the integer portion of the number, and the number of bits allocated for the fractional portion. The total number of bits assigned to the fixed-point representation determines both its range of possible values and its precision. For example, Amharic handwriting recognition uses a fixed-point representation having a width of 16 bits Integer. The range of integer values available depends on the number of bits allocated for the integer portion. The more bits allocated to the integer portion, the greater the range of integers. Conversely, allocating more bits to the integer portion decreases the available precision for the fractional portion. The number of bits allocated to the fractional portion of a fixed-point number determines the precision of the fractional value being represented by that fixed-point number. Allocating more bits to the fractional portion increases the precision of fractional values but decreases the total range of fractional values. The total number of bits will be the sum of the number of integer bits and the number of fractional bits. An example of a fixed-point representation using a total of 16 bits would have 8 bits allocated for the integer portion and 8 bits allocated for the fractional portion. This would be referred to as a Q8.8

format (or Qm.n notation, where m is the number of integer bits and n is the number of fractional bits). The bus width of the memory interface and there are 8bit, 16 bit, and 32bit most common options, along with 512 bit when designing FPGA circuits. For the purpose of this research, we have used a 16 bit memory bus interface.

Avoiding the use of IP cores is preferable. The use of memory functions through BRAM results in full design control, customization of the memory function to application requirements, and optimum organization of memory, data width, and addressing to maximize performance and resource utilization to support application requirements. The majority of IP cores will include a lot of features/functions that are not required by the particular application. Therefore, building a memory function through BRAM reduces unnecessary resource consumption, allowing your application to be designed smaller and more efficiently, using FPGA resources to a higher degree of efficiency and thereby accommodating more complicated features and/or modules. Additionally, using BRAM directly in your design allows you to control memory access and management. Control of the addressing, read/write, and retrieval of data gives you the ability to tune your design to achieve the best performance and the lowest latency.

Paralleling and pipelining techniques are more easily implemented by using coding styles that leverage BRAM. Usually, IP cores are tied to the vendor/toolchain of the respective FPGA they were created. When using coding styles that are specifically designed to leverage BRAM, your design will mostly be portable and will not be dependent upon vendor-specific IP. Consequently, you can easily migrate to other FPGA families or toolchains as your design evolves, giving your design greater flexibility and adaptability. Designing a memory function through a specific coding style based on BRAM provides designers with an excellent learning experience and gives them the chance to better understand FPGA architecture, memory management, and other low-level design factors associated with FPGA design. Students pursuing FPGA studies will utilize their experience with the technology in conjunction with Designers improving their skills in FPGA design. In the comparison between using IP Cores versus coding styles based on BRAM, a designer must remember that even though coding styles based on BRAM allow for the greatest amount of customization and control over the design, they also require the highest amount of effort and expertise to implement. The complexity of the design, available resources, available time, and whether or not the design must be customized for a specific use will determine if IP Cores or BRAM-based coding styles will be used to create your design.

The level of detail in the implementation and the usage of resources will be determined by your choice of neuron model, desired performance characteristics, and the level of resources available on the FPGA. To create an efficient and expandable Neuron Architecture on an FPGA requires consideration of design timing constraints, design memory requirements, and how to optimize system performance.

In order to create a scalable architecture for fpga's Neuron Designs, developers may use Pipeline Techniques to achieve Parallel Processing for multiple Neurons. Each pipeline stage corresponds to an individual neuron, and the flow of data through the various pipeline stages allows for multiple inputs and outputs for multiple automated and human neurons to be processed simultaneously. Developers must provide a mechanism for synchronizing the various Pipeline stages so that the data moves seamlessly through the pipeline and can be synchronized and handshaked properly between stages. Developers must also design and create the Control/Configuration Logic to control the neuron design, including weights, threshold levels, selection of activation functions, and so on.

At this point in the process of selecting neuron design activation functions and determining their corresponding levels of complexity and resource utilization, we can create various Outputs from each respective Neuron based upon the resulting values from the Activation Functions. The resulting Outputs can be either binary (0,1) or continuous (within a specified range).

## 4. Result and discussion

This section discusses our results from the deployment of our hardware solution on a Field Programmable Gate Array (FPGA) that was developed specifically for the task of handwriting recognition on Amharic text via the use of a deep neural network (DNN). We have created and trained several models of DNN specifically designed for the characteristics of Amharic handwriting and have thoroughly examined both simulated and hardware performance to confirm the effectiveness of these models.

To train and evaluate our handwriting recognition system, we developed and utilized a dataset that consisted of 72,000 images for training and 8,000 images for testing and validation purposes. The network weights/biases were originally generated through our software implementation prior to transitioning to the hardware.

When we performed training for 30 epochs, we achieved an accuracy rate of 83% for the testing dataset using the software implementation. Each model was built using a 5-layer architecture that included one input layer (with 784 neurons), two hidden layers (with 30 neurons each), and one output layer (with 343 neurons). This architecture was designed based on the specific characteristics of the Amharic character set, and by using this specific architecture, we have achieved excellent accuracy and have been able to learn the various styles of writing that exist within Amharic text.

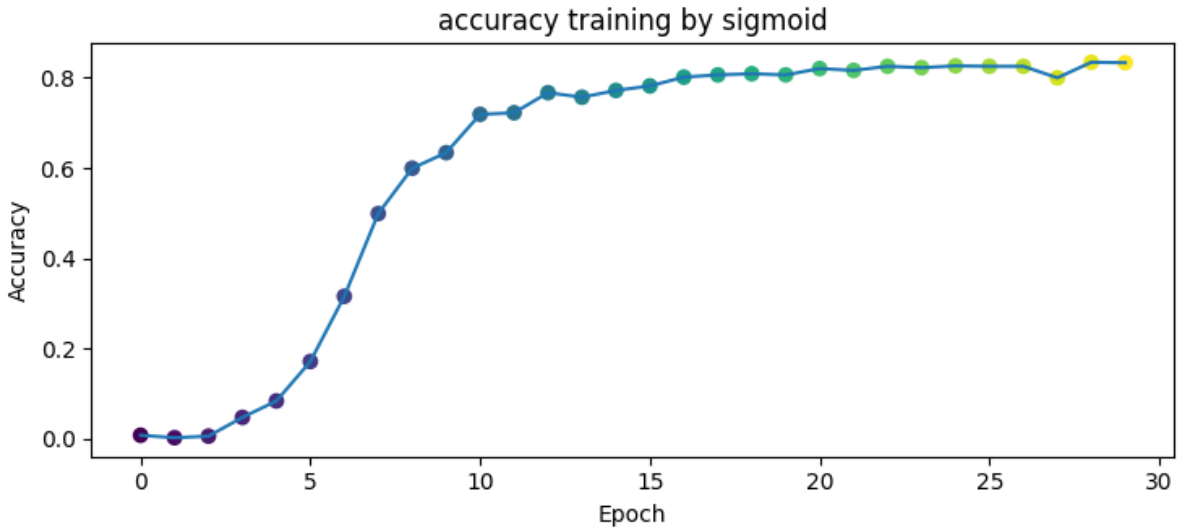


Figure 13 accuracy training by sigmoid

To train a neural network utilizing a specific approach for the identification and interpretation of Amharic handwriting, our team has created a neural network with an architecture that includes five layers: 784, 30, 30, 343, and 343.

In conducting our study, we employed a methodology that consists of several steps including: Data collection/preprocessing, determination of an appropriate loss function and optimization method, establishment of a network architecture, and hyperparameter tuning. The specific methodologies and techniques chosen during the training of the neural network were aimed at maximizing the accuracy and improving the performance of the model for recognizing Amharic handwriting.

We monitored the information regarding the Neural Network's accuracy after every epoch during our training process. An epoch is typically one complete round of going through the Training Dataset with the Neural Network. The objective of plotting out the Neural Networks performance (accuracy) at the 30th epoch was to give you insight into how the network learned over the entire course of the training and provide you with information on when the Neural Network may have converged and where there may be an opportunity for improvement.

As indicated in this paper, we used a Lambda value of five. In essence, this means we placed an importance factor (weight) on how much regularization (L1 or L2) affected the Total Loss Function (Loss) of the network. By using the Lambda value to balance two key features of our Neural Network:

1. Complexity/Number of parameters; and 2. Ability to generalize to new/unseen Amharic handwriting data (preventing overfitting, while still increasing the Classification Performance of the NN on new/unseen handwriting samples).

Our research has produced a very accurate and high-performing neural network model for Amharic handwriting recognition. The methods used and results obtained from this research will help provide increased information and guidance for building neural networks to recognize characters in general, and Amharic handwriting in particular.

A comparison of the weight distribution of the neural network input layer to other input layers shows that there are many weights close to zero. While the majority of weights were distributed between -1 and 1 in this layer, our random selection of weight values very effectively positioned the majority of weights in the input layer near zero. It is essential to note that more than 175 instances of weighted inputs were observed within this range. Therefore, we can conclude that almost all input layer weights were moving towards zero. As noted previously, this concentration supports the assumption that the weight distribution in the input layer reflects the statistical characteristics of our random selection of weight values, which are typically very small random values assigned to weights.

The significant number of weights located close to zero can be attributed to the characteristics of the input data and/or the characteristics of the preprocessing performed on the input. It is important to note that the weights are also influenced by the specific features of the input data as well as the weight of those features for the task.

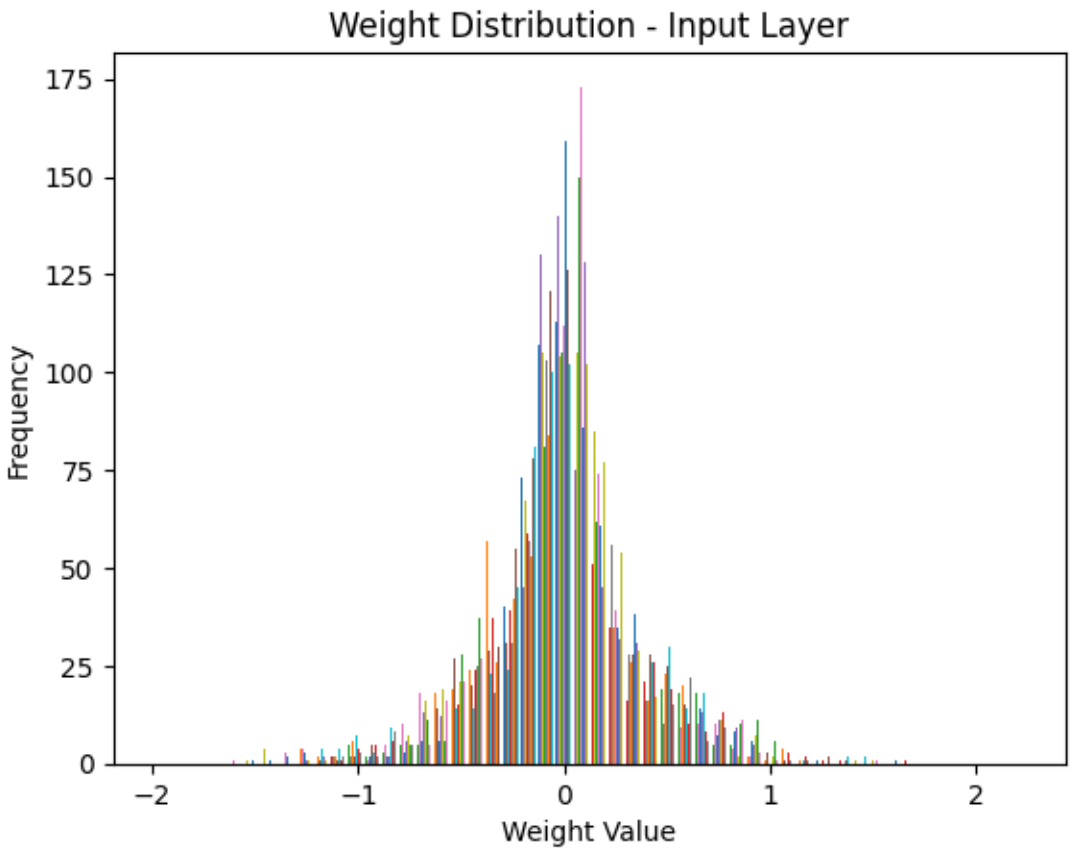


Figure 14 weight distribution – input layer

The weights in Layer 2 are dispersed such that they have an even distribution between -4.5 and +4.5 (the range) indicating that the weights of this layer have achieved a more complex range of weight values as a result of the learning and adaptation process occurred during training. Therefore, Layer 2 has developed a more extensive weight range compared to the weights in Layer 1.

The histogram reveals that the maximum weight frequency occurs around the weight of -1. There are multiple weights that exist in a cluster near the weight value of -1; this indicates that there are specific neurons within Layer 2 that respond strongly to particular types of input. These neurons may be able to identify or extract particular features/characteristics of the handwritten Amharic characters. An even distribution of weights for the learning process for Layer 2 suggests that Layer 2's different neurons have learned to respond to different combinations of feature groups from the Amharic handwritten input data. Since there are many different weights present in Layer 2, the learning capacity of this network will have a broader ability to understand many different aspects of the handwritten Amharic characters, thereby increasing the likelihood of being able to correctly identify the handwritten

Amharic characters.

Layer 2's weight distribution pattern shows the network developing and beginning to generate more complex representations than were initially established in Layer 1. In addition, the presence of separate weight clusters and the increased dynamic range indicate that the network started creating and/or recognizing more advanced aspects of the Amharic Handwriting Dataset.

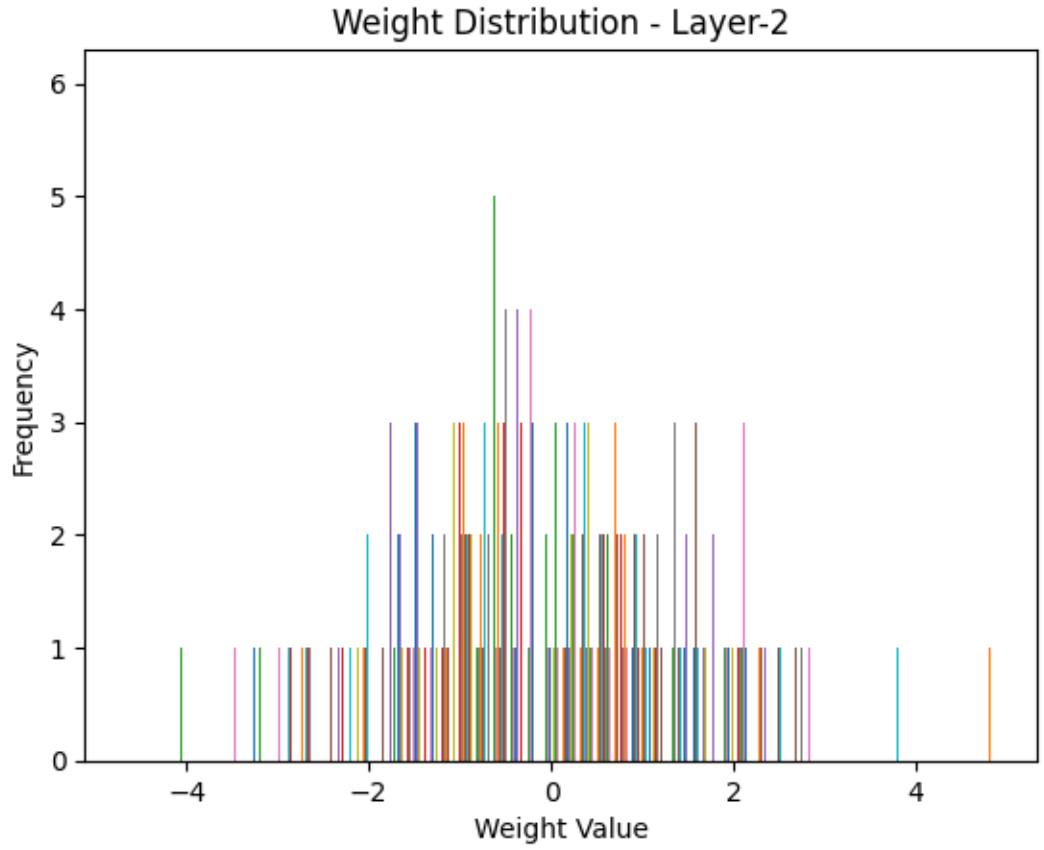


Figure 15 Weight distribution – layer -2

The weights of the 3rd layer are said to be affected by a Gaussian distribution from -3.0 to +3.0, which means that weight values for each neuron will be almost equally likely between -3.0 and +3.0 or not likely at all to have weights further away than about 3.0 from 0.0 (see distribution of weights in 3rd layer). The majority of the weights (i.e. maximum frequency) are centered around 0.0; therefore, in the 3rd layer, there are some neurons (or groups of neurons) that will respond more to the specific characteristics associated with the weight of 0.5 than to other weights. Some weights are also clustered between (-1.5) and (+1.5); this means that there are also clusters of weight values that are strongly associated with these weights.

As indicated by the Gaussian distribution of weights in the 3rd layer, there have been further transformations or refinements made by the neural network on the representations obtained from the previous layers. These transformations or refinements indicate that specific neurons in the 3rd layer are responding preferentially to specific features or attributes of the input. By exhibiting a Gaussian weight distribution, it is reasonable to conclude that the network has created a more specialized and more discriminative representation of the given Amharic handwriting data. The result of these additional transformations/refinements is a heavier emphasis on the extraction of higher-level features that will aid in forming reliable recognition of the characters contained within the original writing.

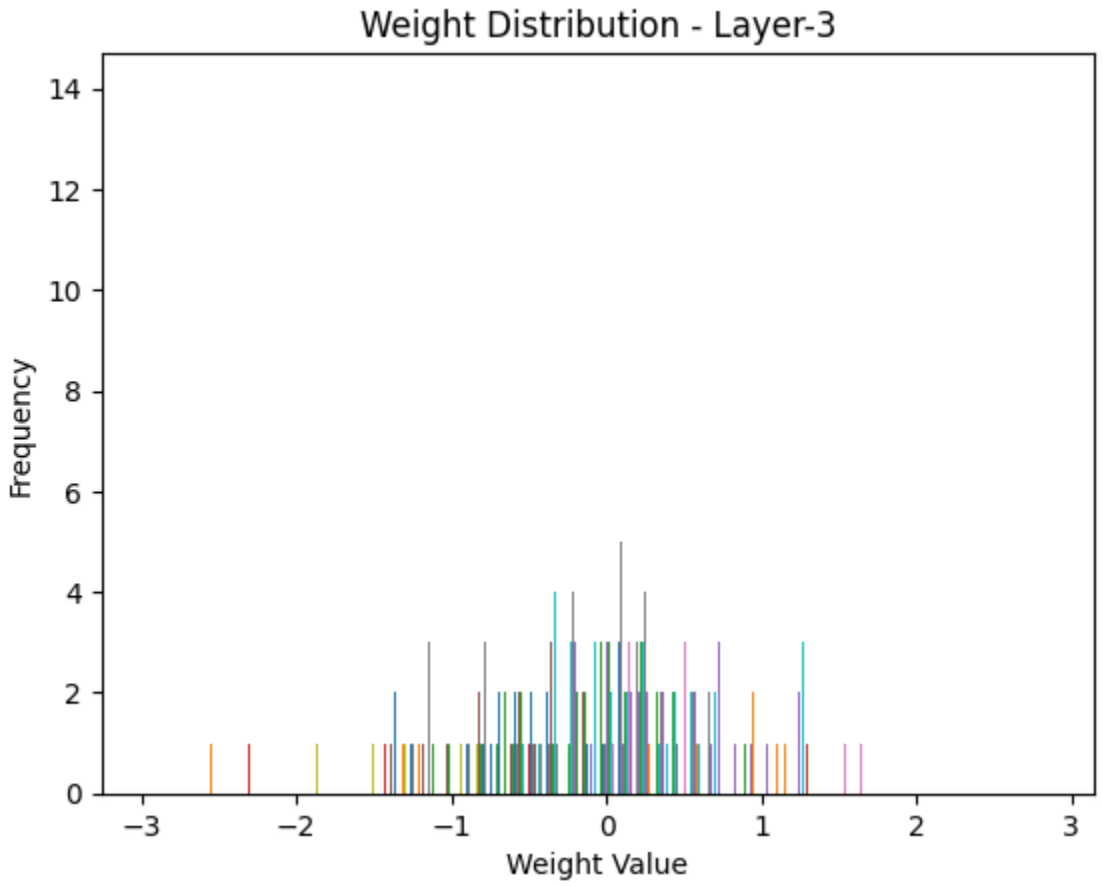


Figure 16 Weight distribution -layer-3

In the fourth and final layer, the weight distribution graph indicates that the weights are distributed Gaussian between -1 and 1. The distribution has a mean of 0 and standard deviation. Most weights are distributed between -0.5 and +0.5, indicating that the network has a tendency to learn smaller weight values in the last layer. The greatest number of weights occur at a weight value close to -0.2 (approximately 200 weights), suggesting that a particular group of neurons or connections in

layer four may have a higher response rate to patterns or features that correspond with that value.

The weight distribution in the final layer of a neural network shows that the information learned from the other layers has been effectively combined or consolidated into a coherent model by examining the Gaussian distribution of the final layer weights. The high concentration of weights close to -0.5 and +0.5 also indicates that the model has learned to consider/input certain features (or patterns) of the input image with moderate relative weights when computing the output of the final layer or when producing a prediction/classification.

Examining the weight distribution of the last layer provides an understanding of how the neural network will classify or predict an incoming Amharic handwriting sample based on the overall degree of activation of each neuron during the activation of the last layer. Each of these weights will affect the final activation of all the neurons in a neural network; thus, each weight will influence the predicted/classified sample's classification.

The final layer weights contain training (knowledge) for the neural network based upon which discriminating features were detected by the neural network while attempting to classify the letters and words of the Amharic alphabet.

By using the weight distribution from the last layer to evaluate the neural network's performance on Amharic handwriting samples, the evaluator will gain an understanding of the neural network's ability to generalize to unknown Amharic handwriting samples, which will help identify any possible biases present or areas of improvement for the way in which the neural network derives its final predictions/classifications.

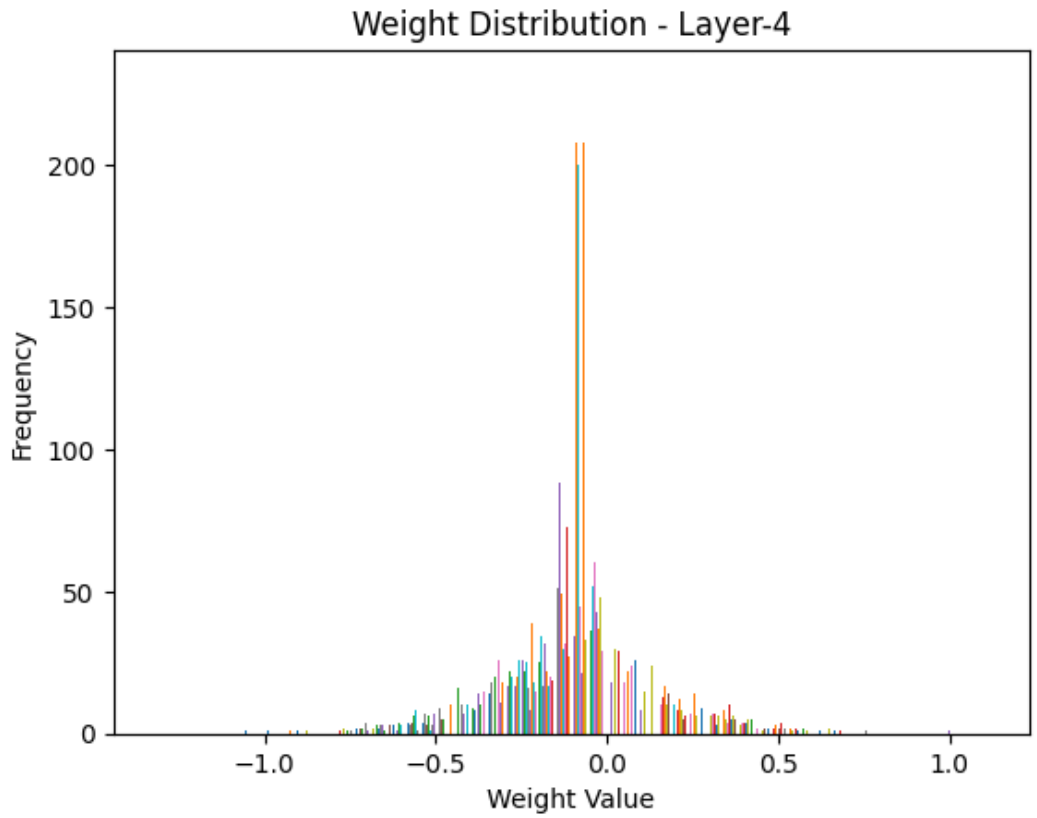


Figure 17 Weight distribution -layer-4

The Final Layer Weights Distribution Captures Information About the Learning of the Network and How Well it is Capable of Classifying Amharic Handwriting.

### 4.1. Hardmax Module and the Final Character Mapping

The structure of the Network contains an output layer that is linked to a Hardmax module, which is essentially responsible for determining which neuron produced the greatest output value regarding the character. The maximum output represents the final classification of an Amharic Character.

When the Network is passed through all layers up until this point, the final output from the Network will typically take the form of a vector of probabilities where each index represents a specific class/character category. Additionally, using the probabilities from all the output neurons, the Hardmax module selects the maximum probability (maximum neuron) and produces the final classification of that character in one step, as opposed to previous classifications based on many points of decision making.

With Hardmax, the Network can provide the classification of a character with the most confidence at that point in time, and this simplifies the overall recognition process; therefore Hardmax is effectively a form of "post-processing" to take the Network's output and convert it into a single class assignment instead of a continuous probability distribution.

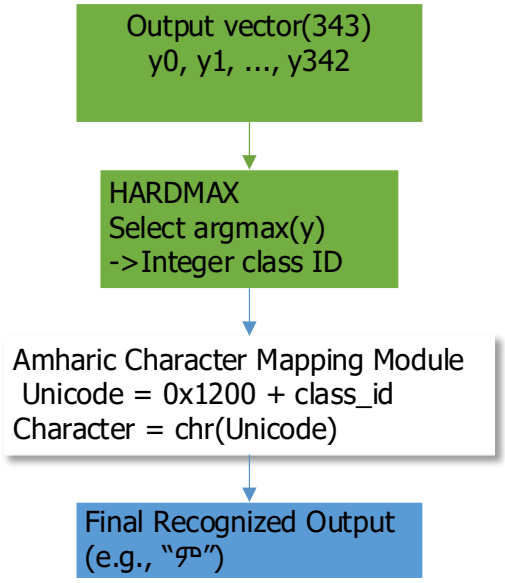


**Figure 18 shows how characters composed of AMHARIC handwriting have been mapped using the HARDMAX module.**

It is worth noting that the entire system architecture and HARDMAX module were both thoroughly tested and implemented using Xilinx vivado 2018.2. Vivado is a well-known software package for designing and implementing FPGA designs and includes all of the tools required for creating, simulating and deploying hardware designs.

To confirm that our Verilog design met customer expectations, we performed a simulation using Vivado. The simulation allowed us to compare our design to customer requirements prior to deploying it on an FPGA.

After simulation verification was completed, we synthesized and implemented our Verilog design with Vivado to generate a working hardware platform to recognize Amharic handwriting with FPGA hardware. The FPGA implementation of our system utilized the HARDMAX module (the

HARDMAX Neural Network) to provide a reliable way to capture and classify Amharic handwritten text. We simulated the architecture of the neural network to verify that it functioned correctly, met timing constraints, and provided the desired performance prior to implementation onto the FPGA.

Once the simulation of the architecture was completed and verified using Vivado, we moved to a physical implementation of the design on actual hardware. We used the Intel DE1-SoC Development Board for our hardware implementation. The Intel DE1-SoC Development Board is not supported by Vivado, so the design was implemented using Intel Quartus Prime 18. The design constraint files required some adjustments to be compatible with the environment of the Intel FPGA. Though we employed different software tools, the functional characteristics of the design remain unchanged due to the similar methodology used to develop and implement the design.

Using the Quartus tool chain to synthesise and implement our design, we fully realised the Amharic handwriting recognition system on DE1-SoC hardware, allowing us to confirm the practical applicability of our design and demonstrate that an implementation of a hardmax-based neural network could operate successfully on real FPGAs.

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 1 1 1 1
1 1 1 1 1 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 1 1 1 0 0 1 1 1
1 1 1 1 0 0 1 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 1 1 0 0 1 1
1 1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 1 1 0 0 1 1
1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 0 0 0 1 1 1 1 1 0 0 0 1
1 1 0 0 0 1 1 1 1 0 0 0 0 1 1 1 0 0 0 1 1 1 1 1 0 0 1 1
1 1 0 0 0 1 1 1 1 0 0 0 1 1 1 1 1 0 0 0 1 1 0 1 0 1 1 1
1 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Figure 18 28*28 array of 1 and 0 for sample Amharic letter

To be effective, an Amharic character recognizer trained on a neural network must have an output layer that produces vectors of scores (probabilities) representing many possible Amharic characters. By taking the maximum of those probabilities (using an operation called argmax), the output letter can then be predicted with a high degree of accuracy. However, there is a difference between the indices returned by a neural network and what those indices correspond to in the Unicode representation for Amharic characters. The discrepancy is corrected by adding 0x1200 to the index value, where, for example, 29 means that the binary representation of the maximum probability contained a 1 in the 0-27 range (a total of 343 outputs available from the neural network). Thus, the binary representation associated with an index allows for proper identification of the output character and its associated Unicode code point.

To illustrate, let's run this example using the code written in Python: print(chr(29+0x1200)) would output the character "ም". The character labels that will be displayed or printed after the code execution is known as the Unicode character label. The offset of 0x1200 is added to the index value of the label in order to be able to offset to the Unicode code point that was specified and provide the means of identifying and validating the predicted character. Through correctly offsetting the index, the model can connect the neural network output to the expected Amharic character, so that the recognition results reflect the expected Unicode representation.

A list Amharic letters is defined, containing the Unicode characters that represent the Amharic letters within the specified range. It consists of 343 characters, fall within the range specified '\u1200' to '\u1357' or in decimal 4608 to 4951.

```
#for reference
for i in range(4608, 4951):
    print(chr(i) + " " , end= '')
```

The code will print all the characters that have been detected by the neural network.

ሀ ሁ ሂ ሃ ሄ ህ ሆ ሇ ለ ሉ ሊ ላ ሌ ል ሎ ሏ ሐ ሑ ሒ ሓ ሔ ሕ ሖ ሗ መ ሙ ሚ ማ ሜ ም ሞ ሟ ሠ ሡ ሢ ሣ ሤ ሥ ሦ ሧ ረ ሩ ሪ ራ ሬ ር ሮ ሯ ሰ ሱ ሲ ሳ ሴ ስ ሶ ሷ ሸ ሹ ሺ ሻ ሼ ሽ ሾ ሿ ቀ ቁ ቂ ቃ ቄ ቅ ቆ ቇ ቈ □ ቋ ቌ ቍ ቎ □ □ ቐ ቑ ቒ ቓ ቔ ቕ ቖ □ ቘ □ ቚ ቛ ቜ ቝ □ □ በ ቡ ቢ ባ ቤ ብ ቦ ቧ ቨ ቩ ቪ ቫ ቬ ቭ ቮ ቯ ተ ቱ ቲ ታ ቴ ት ቶ ቷ ቸ ቹ ቺ ቻ ቼ ች ቾ ቿ ኀ ኁ ኂ ኃ ኄ ኅ ኆ ኇ ኈ □ ኊ ኋ ኌ ኍ □ □ ነ ኑ ኒ ና ኔ ን ኖ ኗ ኘ ኙ ኚ ኛ ኜ ኝ ኞ ኟ አ ኡ ኢ ኣ ኤ እ ኦ ኧ ከ ኩ ኪ ካ ኬ ክ ኮ ኯ ኰ □ ኲ ኳ ኴ ኵ □ □ ኸ ኹ ኺ ኻ ኼ ኽ ኾ □ ዀ □ ዂ ዃ ዄ ዅ □ □ ወ ዉ ዊ ዋ ዌ ው ዎ ዏ ዐ ዑ ዒ ዓ ዔ ዕ ዖ □ ዘ ዙ ዚ ዛ ዜ ዝ ዞ ዟ ዠ ዡ ዢ ዣ ዤ ዥ የ ዩ ዪ ያ ዬ ይ ዮ ዯ ደ ዱ ዲ ዳ ዴ ድ ዶ ዷ ዸ ዹ ዺ ዻ ዼ ዽ ዾ ዿ ጀ ጁ ጂ ጃ ጄ ጅ ጆ ጇ ገ ጉ ጊ ጋ ጌ ግ ጎ ጏ ጐ □ ጒ ጓ ጔ ጕ □ □ ግ ጉ ጊ ጋ ጌ ግ ጎ ጏ ጠ ጡ ጢ ጣ ጤ ጥ ጦ ጧ ጨ ጩ ጪ ጫ ጬ ጭ ጮ ጯ ጰ ጱ ጲ ጳ ጴ ጵ ጶ ጷ ጸ ጹ ጺ ጻ ጼ ጽ ጾ ጿ ፀ ፁ ፂ ፃ ፄ ፅ ፆ ፇ ፈ ፉ ፊ ፋ ፌ ፍ ፎ ፏ ፐ ፑ ፒ ፓ ፔ ፕ ፖ

We will classify inputs by their defined classes or categories. Neural Networks will classify these inputs based on features factored from the inputs. Each class will correspond to an integer.

In this instance, we will classify the Amharic written digits (0, 1, 2, 3, ... 342) and assign a corresponding numerical (integer) value based on what the Neural Networks' predictions are for those Amharic written digits.

0 ሀ 1 ሁ 2 ሂ 3 ሃ 4 ሄ 5 ህ 6 ሆ 7 ሇ 8 ለ 9 ሉ 10 ሊ 11 ላ 12 ሌ 13 ል 14 ሎ 15 ሏ 16 ሐ 17 ሑ 18 ሒ 19 ሓ 20 ሔ 21 ሕ 22 ሖ 23 ሗ 24 መ 25 ሙ 26 ሚ 27 ማ 28 ሜ 29 ም 30 ሞ 31 ሟ 32 ሠ 33 ሡ 34 ሢ 35 ሣ 36 ሤ 37 ሥ 38 ሦ 39 ሧ 40 ረ 41 ሩ 42 ሪ 43 ራ 44 ሬ 45 ር 46 ሮ 47 ሯ 48 ሰ 49 ሱ 50 ሲ 51 ሳ 52 ሴ 53 ስ 54 ሶ 55 ሷ 56 ሸ 57 ሹ 58 ሺ 59 ሻ 60 ሼ 61 ሽ 62 ሾ 63 ሿ 64 ቀ 65 ቁ 66 ቂ 67 ቃ 68 ቄ 69 ቅ 70 ቆ 71 ቇ 72 ቈ 73 □ 74 ቊ 75 ቋ 76 ቌ 77 ቍ 78 □ 79 □ 80 ቐ 81 ቑ 82 ቒ 83 ቓ 84 ቔ 85 ቕ 86 ቖ 87 □ 88 ቘ 89 □ 90 ቚ 91 ቛ 92 ቜ 93 ቝ 94 □ 95 □ 96 በ 97 ቡ 98 ቢ 99 ባ 100 ቤ 101 ብ 102 ቦ 103 ቧ 104 ቨ 105 ቩ 106 ቪ 107 ቫ 108 ቬ 109 ቭ 110 ቮ 111 ቯ 112 ተ 113 ቱ 114 ቲ 115 ታ 116 ቴ 117 ት 118 ቶ 119 ቷ 120 ቸ 121 ቹ 122 ቺ 123 ቻ 124 ቼ 125 ች 126 ቾ 127 ቿ 128 ኀ 129 ኁ 130 ኂ 131 ኃ 132 ኄ 133 ኅ 134 ኆ 135 ኇ 136 ኈ 137 □ 138 ኊ 139 ኋ 140 ኌ 141 ኍ 142 □ 143 □ 144 ነ 145 ኑ 146 ኒ 147 ና 148 ኔ 149 ን 150 ኖ 151 ኗ 152 ኘ 153 ኙ 154 ኚ 155 ኛ 156 ኜ 157 ኝ 158 ኞ 159 ኟ 160 አ 161 ኡ 162 ኢ 163 ኣ 164 ኤ 165 እ 166 ኦ 167 ኧ 168 ከ 169 ኩ 170 ኪ 171 ካ 172 ኬ 173 ክ 174 ኮ 175 ኯ 176 ኰ 177 □ 178 ኲ 179 ኳ 180 ኴ 181 ኵ 182 □ 183 □ 184 ኸ 185 ኹ 186 ኺ 187 ኻ 188 ኼ 189 ኽ 190 ኾ 191 □ 192 ዀ 193 □ 194 ዂ 195 ዃ 196 ዄ 197 ዅ 198 □ 199 □ 200 ወ 201 ዉ 202 ዊ 203 ዋ 204 ዌ 205 ው 206 ዎ 207 ዏ 208 ዐ 209 ዑ 210 ዒ 211 ዓ 212 ዔ 213 ዕ 214 ዖ 215 □ 216 ዘ 217 ዙ 218 ዚ 219 ዛ 220 ዜ 221 ዝ 222 ዞ 223 ዟ 224 ዠ 225 ዡ 226 ዢ 227 ዣ 228 ዤ 229 ዥ 230 ዦ 231 ዧ 232 የ 233 ዩ 234 ዪ 235 ያ 236 ዬ 237 ይ 238 ዮ 239 ዯ 240 ደ 241 ዱ 242 ዲ 243 ዳ 244 ዴ 245 ድ 246 ዶ 247 ዷ 248 ዸ 249 ዹ 250 ዺ 251 ዻ 252 ዼ 253 ዽ 254 ዾ 255 ዿ 256 ጀ 257 ጁ 258 ጂ 259 ጃ 260 ጄ 261 ጅ 262 ጆ 263 ጇ 264 ገ 265 ጉ 266 ጊ 267 ጋ 268 ጌ 269 ግ 270 ጎ 271 ጏ 272 ጐ 273 □ 274 ጒ 275 ጓ 276 ጔ 277 ጕ 278 □ 279 □ 280 ጘ 281 ጙ 282 ጚ 283 ጛ 284 ጜ 285 ጝ 286 ጞ 287 ጟ 288 ጠ 289 ጡ 290 ጢ 291 ጣ 292 ጤ 293 ጥ 294 ጦ 295 ጧ 296 ጨ 297 ጩ 298 ጪ 299 ጫ 300 ጬ 301 ጭ 302 ጮ 303 ጯ 304 ጰ 305 ጱ 306 ጲ 307 ጳ 308 ጴ 309 ጵ 310 ጶ 311 ጷ 312 ጸ 313 ጹ 314 ጺ 315 ጻ 316 ጼ 317 ጽ 318 ጾ 319 ጿ 320 ፀ 321 ፁ 322 ፂ 323 ፃ 324 ፄ 325 ፅ 326 ፆ 327 ፇ 328 ፈ 329 ፉ 330 ፊ 331 ፋ 332 ፌ 333 ፍ 334 ፎ 335 ፏ 336 ፐ 337 ፑ 338 ፒ 339 ፓ 340 ፔ 341 ፕ 342 ፖ

## 4.2. Quantization and Accuracy Evaluation

When you're implementing a Neural Network, there's many things to consider. One of them is the type of Activation Function you want to use for each neuron in your Neural Network. A common setup for Activation Functions is the Sigmoid function, which will turn the weighted sums of your network's input into something between 0 and 1.

A consideration with the way you store the Sigmoid function in Memory and how that affects the model's accuracy (your model is applying Sigmoid element-wise across all outputted neurons).

Depth registering the Sigmoid function means to precompute a range of values for different inputs, then place them in your Memory (the precomputed values) so that your Neural network can reference the computed values, as opposed to computing them in every single forward pass of the network.

The depth of the Sigmoid is also a factor in how to analyze the accuracy of your model as a function of the number of values precomputed. To increase the accuracy of your Neural Network, you can store a greater number of precomputed values for the Sigmoid function; however, this increase in the precomputed values comes with an increase in the Memory needed to store them. Therefore, finding the optimum depth that balances the accuracy of the model with the Memory used to store the model is the goal (optimal depth).

Finding that depth can be accomplished by varying the depth used in your model and measuring the resulted accuracy from validating it against your validation dataset. Look at how the accuracy of the model changes as the depth of the Sigmoid function changes to gain experience with how to determine an optimal level of precomputation for your specific Neural Network Architecture and Dataset.

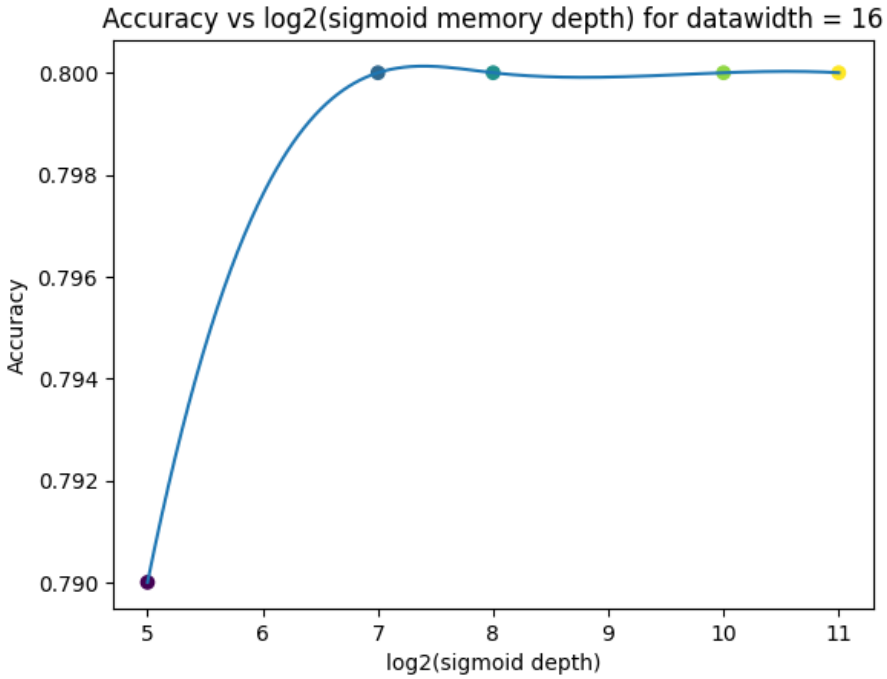


Figure 20 the relationship between the accuracy of a neural network design and the number of bits of memory depth

When a Neural Network is implemented in hardware, using a larger Data Path Size will generally improve Accuracy of the resulting model. It is directly proportional; as the size of Data Path increases, data processed during Forward Propagation receives more accuracy because of additional bits used to perform calculations. An example of this would be an output from a Sigmoid Function; by storing a greater number of decimal places (digits), it provides a better representation of the output from the Sigmoid Function.

Not only does a larger Data Path Size provide improved Precision for Forward Propagation computations, but it also reduces Quantization Errors. With fewer bits available to represent neuron signals, there exists the possibility of Rounding or Truncating the input values, which introduces Quantization Errors in the computations. Thus, when more bits are available for the Representation of Neuron Signals, there will be fewer Quantization Errors and hence the resultant computations will be more Accurate.

Also, with increased Data Path Size, the Input Range of a Neural Network is also Maximized without the risk of Overflow/Underflow. Therefore, Maximizing Accuracy from Levels for the

Fixtures used to Process these Inputs is Critical to effectively create a Hardware representation of a Neural Network.

Therefore, it is important to emphasise that while increasing the Data Path Size will, in General, increase the Accuracy of a Hardware implemented Neural Network when compared to a Small Data Path Size; other factors such as Neural Network Architecture, Training Methods & Characteristics of the Training Set will be factors that will affect overall Accuracy of the Neural Network.

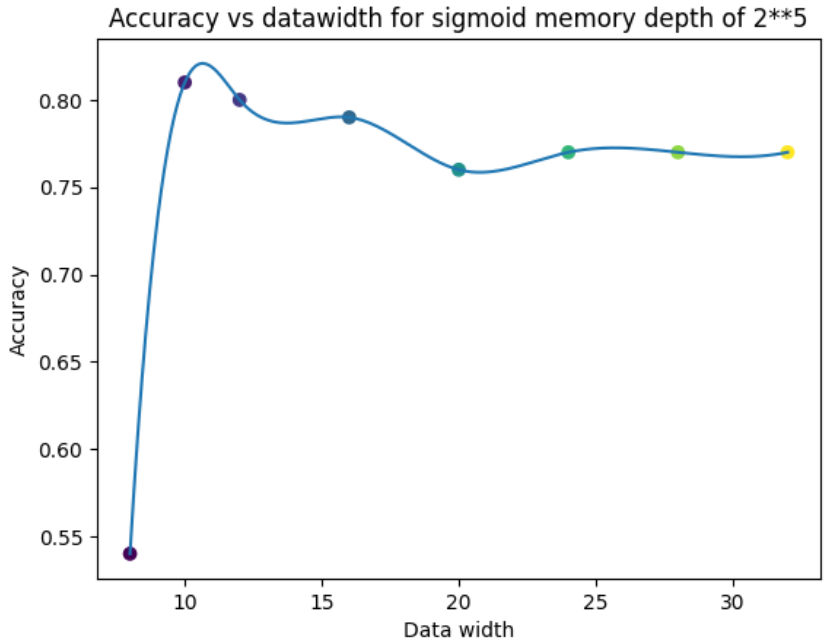


*Figure 19* Accuracy vs data width for sigmoid memory depth 2**5

## 4.3. FPGA Resource Mapping and Performance Evaluation

This section presents the implementation results and design decisions for a hardware neural network accelerator deployed on the DE1-SoC platform. The system accepts 784-dimensional inputs and produces a 250-class argmax prediction. The accelerator executes inside the FPGA fabric while a Linux application on the HPS performs configuration and test orchestration through the lightweight AXI bridge. The primary challenges were meeting device resource limits on a Cyclone V while sustaining correct arithmetic behavior and providing a robust validation path from user space.

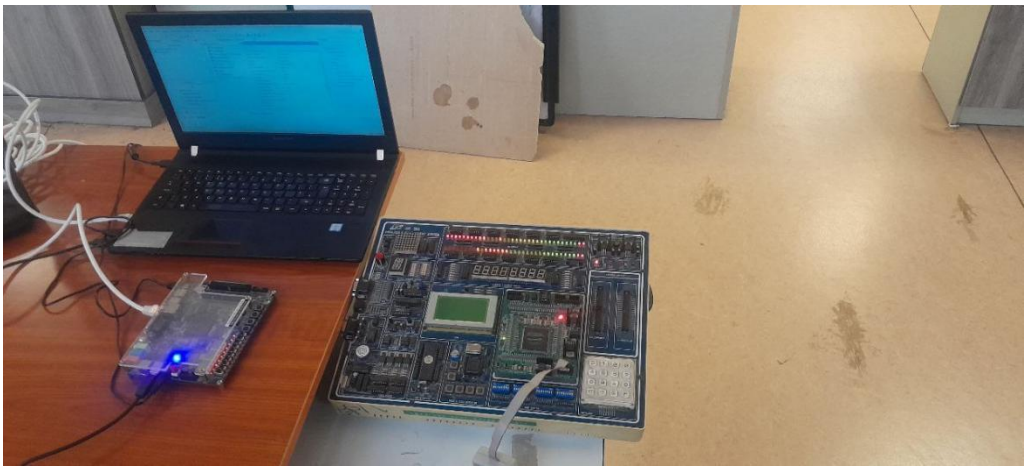

Figure 20 Lab different FPGA setup for test

The RTL organization follows a conventional separation of concerns. A compute core implements the neuron datapath and accumulations. A thin register bank exposes a small memory-mapped interface for streaming inputs, reading predictions, and issuing a software reset.

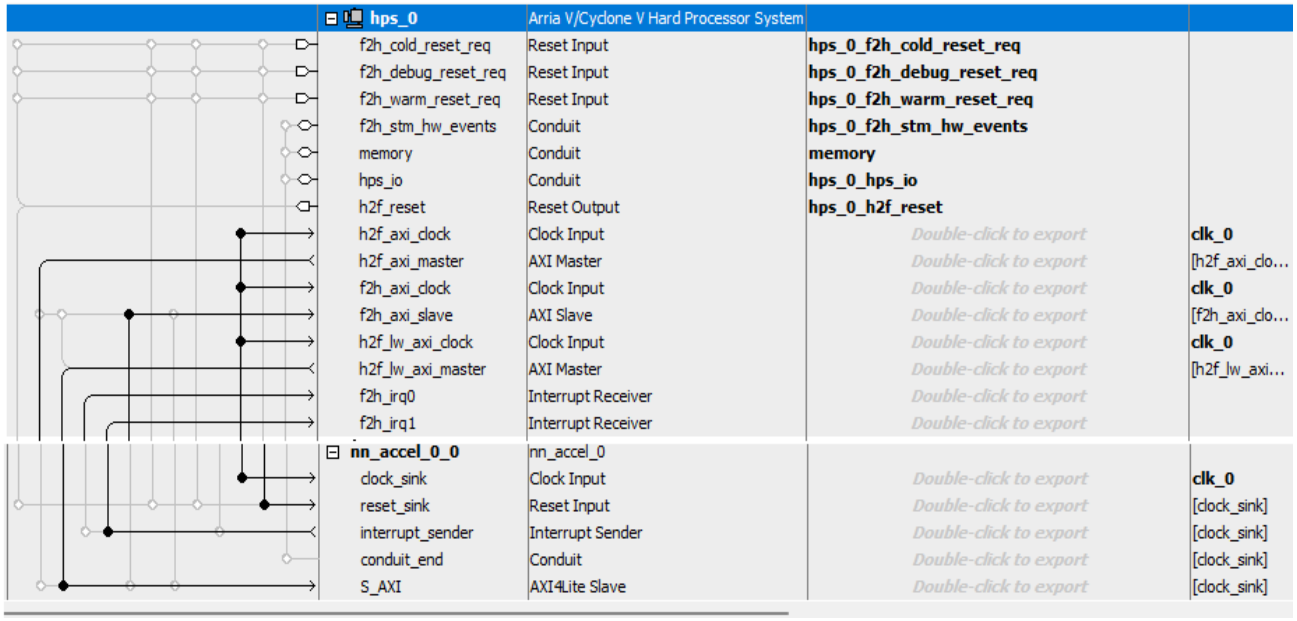


Figure 21 HPS and FPGA wire map on platform design of Quartus software

An interrupt and a read-to-clear status flag signal completion. Parameterization controls layer sizes, data path width, and activation choices. During synthesis, the compiler infers DSP blocks for multipliers and block memories for weights and biases; ensuring that this inference is critical for fitting the device.

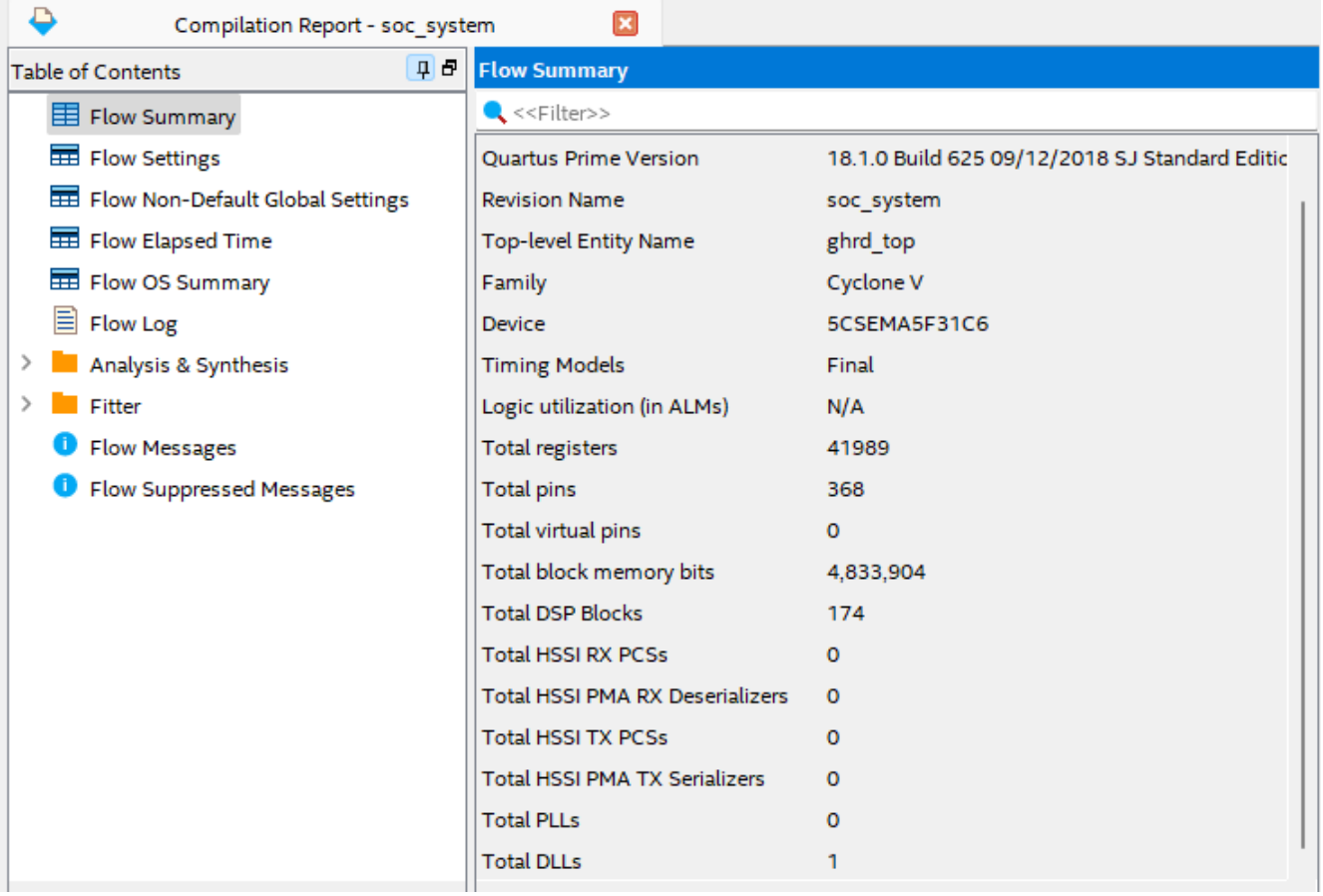


Figure 22 Failed compilation report for network 174-343 with data width 16

Compilation settings were tuned with the explicit goal of reducing logic and routing pressure while preserving functionality. At the analysis and synthesis stage, automatic RAM/ROM replacement, flexible memory size recognition, and automatic DSP replacement were enabled so that storage and arithmetic would map to the most efficient hard resources. State machine extraction and resource sharing were left on, allowing the compiler to consolidate logic where safe. On the fitter, the build targeted area optimization with aggressive effort and increased packing density of registers into logic blocks. Physical synthesis for combinational logic was enabled to restructure logic cones and reduce area. The fitter also promoted true clocks and high-fanout control signals to dedicated networks where beneficial, improving routing without increasing the logical footprint. When results were marginal,

router seeds were varied because alternate initial placements can lead to meaningfully different resource packing on arithmetic-heavy designs.

Several limiting errors surfaced during the exploration phase and guided subsequent adjustments. An early build failed with a message that the design required more LABs than were available. This outcome typically occurs when large arithmetic or memory structures are realized in logic rather than DSPs or block RAM, or when the network footprint simply exceeds the fabric capacity. Another build failed due to block RAM demand exceeding the available M10K blocks. In both cases, the response was twofold: first, confirm that inference was occurring as intended; second, reduce the footprint where it yielded the largest payoff for minimal algorithmic disturbance. Compiler reports were examined for any hints that multipliers or memories had been implemented. Attributes were introduced selectively to encourage weights into block RAM and, when M10Ks became the tighter constraint, to permit the smallest memories (such as biases) to inhabit distributed RAM. This light-touch steering preserved the structure of the design while trimming pressure on the most constrained resource.

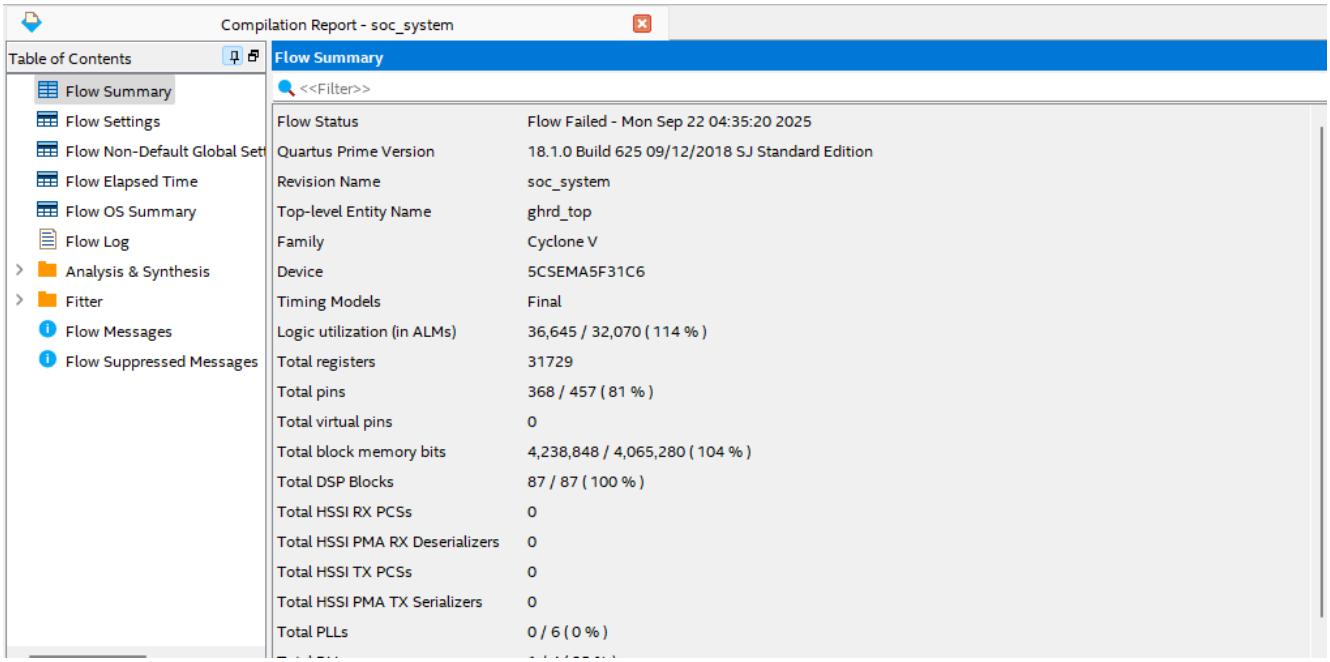


Figure 23 Failed compilation report for network 174-343 with data width 12

The single most impactful decision for fitting was the selection of a 12-bit signed fixed-point datapath in Q1.11 format. Moving to Q1.11 reduced memory widths, narrowed adders and multipliers, and

correspondingly lowered pressure on both logic and block RAM. This precision was chosen because it preserves the sign, provides a symmetric dynamic range around zero, and offers 11 fractional bits sufficient for typical normalized inputs and trained weights. After adopting Q1.11, the design fit the device consistently under the area-focused fitter recipe. This result highlights a general rule for embedded inference on mid-range FPGAs: a well-chosen fixed-point representation often provides a larger improvement in fit and timing closure than deeper architectural changes.

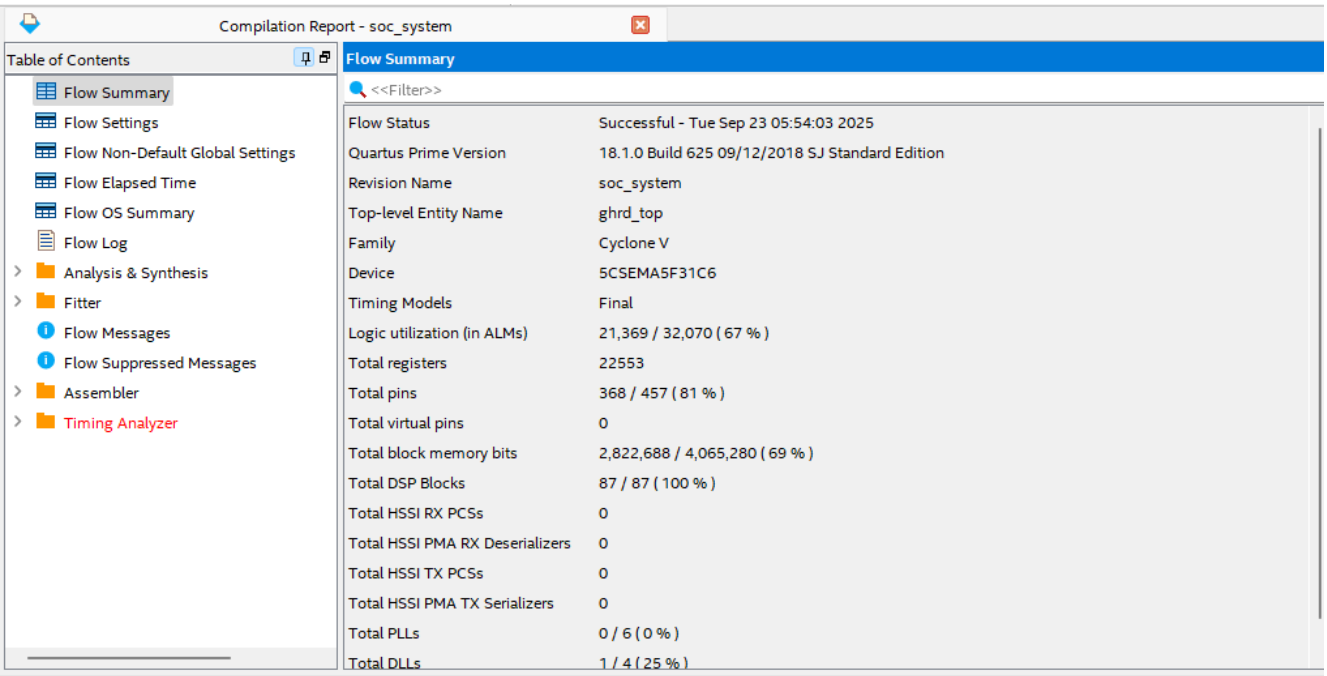


Figure 24 Successful compilation report for network 174-343 with data width 12

On the software side, a userspace Linux application was developed to act as a practical testbench for the accelerator. The program maps the lightweight AXI window via the standard /dev/mem interface and performs a minimal protocol: write a zero to the control register to clear the soft reset; perform exactly 784 writes to the input register, one per sample, using the lower 16 bits for the signed 12-bit value; poll a read-to-clear status register until completion is indicated; then read the prediction from a dedicated output register. This handshake mirrors the Verilog simulation harness but operates entirely from userspace, avoiding the need for a kernel driver during bring-up. Particular attention was paid to C language standard settings for the cross-compiler, enabling C99 so that loop variable declarations within for-statements are accepted. Where floating-point rounding helpers were used, linkage against the math library was added, although the final fixed-point quantization employed a branchless round-

to-nearest expression that does not require libm.

Data handling was treated with the same rigor as the hardware. Two input encodings were supported for validation. For simulation-parity tests, the input records were authored as binary strings using a format identical to that expected by readmemb: the first 784 tokens represent signed 12-bit two's-complement samples, and the final token encodes the expected class as a binary integer. The test program parses the tokens, sign-extends the 12-bit samples, and compares the hardware's argmax with the decoded label, reporting both per-sample results and running accuracy. For grayscale test sets, an alternative path maps values in the range [0, 255] to the signed interval [-1, 1] through a linear transform, then quantizes to Q1.11 with saturation before streaming into the accelerator. The program prints basic statistics after quantization, such as the minimum and maximum sample values, to confirm that the dynamic range is sensible; this simple step quickly reveals mismatches, such as providing only non-negative inputs to a network expecting zero-centered activations.

The problems that arise from using FPGA devices include issues such as resource usage and creating large projects on FPGA's. There are configuration pitfalls associated with using a large FPGA. For example, a project may not have had its configurations saved properly because the data file was in read-only mode causing any new compiled output to not reflect the changes made to the previous configuration file in the project's GUI. Once this was corrected by changing the project configuration file to a writable format the configurations did save properly and were used correctly when the project configuration was generated again. In this example, the Fitter converted additional logic to MLAB, resulting in an increase in overall physical device area, compared to the prior configuration that had Alms in the lower half of the device with large weight matrices placed in Block Ram and distributing only small memories among all M10k free Alms with minimal impact to the physical space of the FPGAs. Additional safeguards were provided by examining the router's seed during the configuration of the two borderline fits: while both seeds resulted in borderline fits, the use of a third seed frequently resulted in a successful fit without any need for alteration to the architectural layout of the hardware.

The configuration that delivered reliable fits used a single hidden stage with 250 outputs connected to a simple argmax selection. Arithmetic was mapped to DSP blocks, weights were realized in block RAM, and biases were permitted to occupy distributed RAM as needed. The fitter operated in an area-aggressive mode with extra effort and physical synthesis for combinational logic, and registers were

packed aggressively into logic blocks to improve density. On the HPS, the testbench executed batches of records placed alongside the binary; it streamed inputs, synchronized via a read-to-clear status, and retrieved argmax indices for comparison with labels. Collectively, these choices resulted in a deterministic, easily scriptable validation workflow with predictable resource characteristics.

The data and results demonstrate that resource-constrained neural inference can be achieved on a mid-range FPGA when the architecture is aligned with the silicon's strengths and the toolchain is guided toward efficient inference. DSP utilization remained high, confirming that multipliers stayed in hard blocks; block RAM utilization was likewise high, indicating correct memory inference for weights; and logic usage fell substantially after adopting a 12-bit datapath. The system behaved deterministically once fitted, and the userspace testbench consistently verified predictions against labeled data, reporting running accuracy akin to the simulation harness.


Figure 25 The user-space testbench

There are clear avenues for future improvements. If throughput beyond CPU-driven streaming is required, the data path can be augmented with DMA or streaming FIFOs to reduce software overhead and increase sustained bandwidth. If model capacity grows beyond what the device's on-chip RAM can accommodate, even at 12 bits weights can be staged from external memory into small on-chip caches under the control of a simple prefetcher, trading latency and complexity for capacity. Finally, the polling-based completion detection in userspace can be replaced with an interrupt-driven path using a userspace I/O device, reducing CPU utilization and latency jitter.

Finally, the project has evolved from a very large but fully functional first draft of a new implementation, to a new, smaller, more resource aware implementation that could now fit nicely inside of our target device. There were several major changes during this transition. First, as stated above, we validated the performance of the DSP and block RAM inference methods. Second, we aggressively based our implementation's size on a small but efficient size optimally fitting the area of our target chip. Third, we fixed our configuration persistence issues. Fourth, we converted to a 12-bit fixed point representation which provided a good tradeoff between accuracy and the overall size of our implementation. The final system is capable of producing complete inference outputs based on representative data, and provides a stable foundation upon which future improvements in throughput, capacity and software ergonomics can be built.

# 5. Conclusion

In this thesis, we have described, implemented, and tested a fully connected (dense) Neural Network for the recognition of Amharic handwritten text. An FPGA accelerated inference engine was also created in Verilog for implementation on an Intel Cyclone V System on Chip (SoC). The software model (784 → 30 → 30 → 343 → 343) obtained approximately 83% test accuracy after 30 training epochs. The hardware implementation was optimised for the constraints of the FPGA by using a 12-bit Q1.11 fixed-point data path, directing arithmetic to DSPs and large matrices to Block RAM, and a deterministic userspace testbench using a lightweight AXI window for streaming inputs and polling the Read-to-Clear Status Register for completion. All co-design decisions led to a compact, resource-aware implementation (matching simulation results for representative data) with predictable results for full inference on the target device.

While the DE1-SoC platform provided a functional and accessible development environment, its resource capacity constrained the size of the deployable neural network. As the only FPGA platform available during this research, the Cyclone V SoC limited experimentation with deeper or wider architectures that could further improve accuracy. Future work should therefore evaluate the same design on larger FPGA devices—such as those with increased block RAM, DSP count, or HBM/DDR bandwidth—to support the scaling of the neural model and to assess performance, inference latency, and resource utilization across different hardware tiers.

Collectively, the results validate that a resource-aware hardware/software co-design— including fixed-point quantization, careful memory mapping, and area-focused toolchain settings— can enable neural inference for an under-resourced script such as Amharic on mid-range FPGAs while providing energy-efficient operation suitable for offline digitization and OCR tasks.

## 5.1. Limitation of the research

The reported results currently present only aggregate accuracy (83%). The study lacks per-class precision/recall, confusion matrices, and statistical robustness checks (cross-validation, confidence intervals), which are necessary to diagnose failure modes across glyphs. While qualitative claims about DSP/BRAM/LUT trade-offs and a successful Q1.11 fit are documented, precise synthesizer/fitter

utilization percentages, measured power figures, and latency breakdowns (HPS vs. FPGA) are not included and should be added. Evaluation was performed under laboratory conditions with a userspace polling testbench. Robustness to temperature, supply variation, long-term drift, and interactive use (strict real-time latency) is yet to be evaluated.

### 5.2. Recommendations & Future Work

Replace userspace polling with AXI-DMA + interrupt-driven transfers or an AXI streaming FIFO / double-buffered flow to reduce host overhead and jitter. The document already notes DMA/FIFO as a future improvement— implement this to measure gains. Evaluate the system on real devices under environmental variation (temperature, supply). Deploy a prototype on a mobile or low-power edge board and collect field performance data (user study or pilot). Investigate on-device or hybrid low-precision training workflows (FPGA-assisted or mixed precision) and test the co-design methodology on related Ethiopic scripts (Tigrinya, Ge'ez) to show generalizability.

## 6. Reference

[1] J. M. Shalf and R. Leland, "Computing beyond moore's law," *Computer (Long. Beach. Calif).*, vol. 48, no. 12, pp. 14–23, 2015, ISSN: 00189162. DOI: 10.1109/MC.2015.374.

[2] R. T. Sataloff, M. M. Johns, and K. M. Kost, *Artificial Intelligence A Modern Approach*,ISBN: 9781626239777.

[3] D. Monroe, "Neuromorphic computing gets ready for the (really) big time," *Commun. ACM*, vol. 57, no. 6, pp. 13–15, 2014, ISSN: 15577317. DOI: 10.1145/2601069.

[4] S. Singh, A. Sarma, N. Jao, A. Pattnaik, S. Lu, K. Yang, A. Sengupta, V. Narayanan, and C. R. Das, "NEBULA: A Neuromorphic Spin-Based Ultra-Low Power Architecture for SNNs and ANNs," *Proc. - Int. Symp. Comput. Archit.*, vol. 2020-May, pp. 363–376, 2020, ISSN: 10636897. DOI: 10.1109/ISCA45697.2020.00039.

[5] S. Lekhwar, S. Yadav, and A. Singh, *Big Data Analytics in Retail*. Springer Singapore, 2019, vol. 107, pp. 469–477, ISBN: 9789811317460. DOI: 10.1007/978-981-13-1747-7_45. [Online]:http://dx.doi.org/10.1007/978-981-13-1747-7{_}45.

[6] V. Sze, Y. H. Chen, J. Einer, A. Suleiman, and Z. Zhang, "Hardware for Machine Learning: Challenges and Opportunities," *Proc. Cust. Integr. Circuits Conf.*, vol. 2017-April, 2017, ISSN: 08865930. DOI: 10.1109/CICC.2017.7993626. arXiv: 1612.07625.

[7] Cisco Public, "Cisco Global Cloud Index: Forecast and Methodology, 2015–2020," pp. 2015–2020, 2016. [Online]. Available: http :// www . cisco. com/c/ dam/ en/ us/ solutions / collateral / service - provider / global - cloud - index - gci / white - paper-c11-738085.pdf.

[8] J. Lee, J. Lee, D. Han, J. Lee, G. Park, and H.-j. Yoo, "7.7 LNPU: A 25.3TFLOPS/W Sparse Deep-Neural-Network Learning Processor with Fine-Grained Mixed Precision of FP8- FP16 - IEEE Conference Publication," *2019 IEEE Int. Solid- State Circuits Conf.,* -, pp. 142– 144, 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8662302.

[9] J. Shalf, "The future of computing beyond Moore' s Law Subject Areas:" *Philos. Trans. R. Soc.*, vol. 378, no. 20190061, pp. 1–14, 2020

[10] B. Fleischer, S. Shukla, M. Ziegler, J. Silberman, J. Oh, V. Srinivasan, J. Choi, S. Mueller,A. Agrawal, T. Babinsky, N. Cao, C. Y. Chen, P. Chuang, T. Fox, G. Gristede, M. Guillorn,H. Haynie, M. Klaiber, D. Lee, S. H. Lo, G. Maier, M. Scheuermann, S. Venkataramani,C. Vezyrtzis, N. Wang, F. Yee, C. Zhou, P. F. Lu, B. Curran, L. Chang, and K. Gopalakrishnan, "A Scalable Multi-TeraOPS Deep Learning Processor Core for AI Training and Inference," *IEEE Symp. VLSI Circuits, Dig. Tech. Pap.*, vol. 2018-June, no. Table 2, pp. 35–36, 2018. DOI: 10.1109/VLSIC.2018.8502276.

[11] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 1965.

[12] K. J. Kuhn et al., "Process technology variation," *IEEE Trans. Electron Devices*, vol. 58, no. 8, pp. 2197–2208, Aug. 2011.

[13] International Technology Roadmap for Semiconductors (ITRS), "2015 Report," 2015. [Online]. Available: http://www.itrs2.net

[14] M. M. Waldrop, "The chips are down for Moore's Law," *Nature*, vol. 530, no. 7589, pp. 144–147, Feb. 2016.

[15] International Business Strategies, "Semiconductor Cost Forecast," 2023.

[16] J. Shalf, "The future of computing beyond Moore's Law," *Philos. Trans. Royal Soc. A*, vol. 378, no. 2166, Feb. 2020.

[17] TechInsights, "TSMC 2nm Process Technology Outlook," 2024.

[18] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436– 444, May 2015.

[19] E. Strubell, A. Ganesh, and A. McCallum, "Energy and Policy Considerations for Deep Learning in NLP," in *Proc. 57th Annu. Meeting Assoc. Comput. Linguistics*, Florence, Italy, Jul. 2019, pp. 3645–3650.

[20] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Cambridge, MA, USA: Morgan Kaufmann, 2019.

[21] NVIDIA, "RTX 4070 Ti Specifications," 2025. [Online]. Available: https://www.nvidia.com

[22] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Toronto, ON, Canada, Jun. 2017, pp. 1–12.

[23] Xilinx, "Versal AI Core Series Benchmarks," Q1 2025.

[24] C. Zhang et al., "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Monterey, CA, USA, Feb. 2017, pp. 161–170.

[25] B. Jacob et al., "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 2, pp. 567–578, Feb. 2023.

[26] A. Reuther et al., "Survey of machine learning accelerators," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, Waltham, MA, USA, Sep. 2019, pp. 1–12.

[27] ResearchGate, "Amharic Handwriting Dataset Overview," 2025. [Online]. Available: https://www.researchgate.net

[28] Y. Assabie and J. Bigun, "Offline handwritten Amharic character recognition using HMM," in *Proc. Int. Conf. Pattern Recognit.*, Tampa, FL, USA, Dec. 2009, pp. 1–4.

[29] D. Teferi, T. Meshesha, and A. Abebe, "A deep learning approach to Amharic handwriting recognition," *J. Comput. Sci. Ethiopia*, vol. 5, no. 1, pp. 23–30, Jun. 2019.

[30] A. B. Tesfaye, "Deep learning for Amharic handwriting recognition," *Pattern Recognit. Lett.*, vol. 150, pp. 45–52, Oct. 2022.

[31] T. M. Alemu and J. P. Doe, "Hardware acceleration for under-resourced scripts: A survey," *Int. J. Comput. Vis.*, vol. 131, no. 3, pp. 789–810, Mar. 2023.

[32] Xilinx, "Vitis AI User Guide," 2023. [Online]. Available: https://www.xilinx.com

# 7. Appendix

The module neuron implements a streaming multiply–accumulate (MAC) neuron with activation for fixed-point inference. During configuration, per-neuron weights are written into an internal weight memory, and the bias is either preloaded from a file (pretrained) or written via the configuration interface. At run time, the neuron receives a sequence of numWeight input samples through myinput with myinputValid. For each input, it reads the corresponding weight from the on-chip memory, multiplies the signed input by the signed weight (one DSP-inferred multiply), and accumulates the result into a wide accumulator (sum) with explicit saturation logic to prevent overflow. After all inputs have been processed (detected via an internal read address and a small valid pipeline), the bias—scaled to match the accumulator width—is added once, also with saturation. The final accumulator value is then passed through a selectable activation: a small ROM-backed sigmoid (indexed by the top accumulator bits) or a ReLU unit for rectification. The output out is asserted together with a one-cycle outvalid pulse to mark the completion of the neuron's computation for that input vector.

Control is kept simple and fully synchronous: a read address r_addr advances on each valid input, a short valid pipeline aligns the memory read, multiply, and accumulate stages, and an edge detector marks the transition from the last MAC to the bias-add/activation cycle. Implementation maps naturally onto FPGA resources: the multiplier infers a DSP block, the weight storage infers block RAM, the sigmoid table maps to ROM (MLAB/M10K depending on size), and the adder/saturation, counters, and control are realized in ALMs and flip-flops. This organization delivers a compact, timing-friendly neuron that streams inputs, performs MAC with saturation, adds bias, applies activation, and emits a single validated output per input vector.

```
module neuron #(parameter
layerNo=0,neuronNo=0,numWeight=784,dataWidth=16,sigmoidSize=5,weightIntWidth=1,a
ctType="relu",biasFile="",weightFile="")(
    input           clk,
    input           rst,
    input [dataWidth-1:0]    myinput,
    input           myinputValid,
    input           weightValid,
    input           biasValid,
    input [31:0]    weightValue,
    input [31:0]    biasValue,
    input [31:0]    config_layer_num,
    input [31:0]    config_neuron_num,
    output[dataWidth-1:0]    out,
    output reg      outvalid
    );

    parameter addressWidth = $clog2(numWeight);

    reg         wen;
    wire        ren;
    reg [addressWidth-1:0] w_addr;
    reg [addressWidth:0]   r_addr;//read address has to reach until numWeight
hence width is 1 bit more
    reg [dataWidth-1:0]  w_in;
    wire [dataWidth-1:0] w_out;
    reg [2*dataWidth-1:0]  mul;
    reg [2*dataWidth-1:0]  sum;
    reg [2*dataWidth-1:0]  bias;
    reg [31:0]    biasReg[0:0];
    reg         weight_valid;
    reg         mult_valid;
    wire        mux_valid;
    reg         sigValid;
    wire [2*dataWidth:0] comboAdd;
    wire [2*dataWidth:0] BiasAdd;
    reg  [dataWidth-1:0] myinputd;
    reg muxValid_d;
    reg muxValid_f;
    reg addr=0;
   //Loading weight values into the momory
    always @(posedge clk)
    begin
        if(rst)
        begin
            w_addr <= {addressWidth{1'b1}};
            wen <=0;
        end
        else if(weightValid & (config_layer_num==layerNo) &
(config_neuron_num==neuronNo))
        begin
            w_in <= weightValue;
            w_addr <= w_addr + 1;
            wen <= 1;
        end
        else
            wen <= 0;
    end
```

```
    assign mux_valid = mult_valid;
    assign comboAdd = mul + sum;
    assign BiasAdd = bias + sum;
    assign ren = myinputValid;

    `ifdef pretrained
        initial
        begin
            $readmemb(biasFile,biasReg);
        end
        always @(posedge clk)
        begin
            bias <= {biasReg[addr][dataWidth-1:0],{dataWidth{1'b0}}};
        end
    `else
        always @(posedge clk)
        begin
            if(biasValid & (config_layer_num==layerNo) &
(config_neuron_num==neuronNo))
            begin
                bias <= {biasValue[dataWidth-1:0],{dataWidth{1'b0}}};
            end
        end
    `endif


    always @(posedge clk)
    begin
        if(rst|outvalid)
            r_addr <= 0;
        else if(myinputValid)
            r_addr <= r_addr + 1;
    end

    always @(posedge clk)
    begin
        mul  <= $signed(myinputd) * $signed(w_out);
    end


    always @(posedge clk)
    begin
        if(rst|outvalid)
            sum <= 0;
        else if((r_addr == numWeight) & muxValid_f)
        begin
            if(!bias[2*dataWidth-1] &!sum[2*dataWidth-1] & BiasAdd[2*dataWidth-
1]) //If bias and sum are positive and after adding bias to sum, if sign bit
becomes 1, saturate
            begin
                sum[2*dataWidth-1] <= 1'b0;
                sum[2*dataWidth-2:0] <= {2*dataWidth-1{1'b1}};
            end
            else if(bias[2*dataWidth-1] & sum[2*dataWidth-1] &
!BiasAdd[2*dataWidth-1]) //If bias and sum are negative and after addition if
sign bit is 0, saturate
            begin
```

```
                sum[2*dataWidth-1] <= 1'b1;
                sum[2*dataWidth-2:0] <= {2*dataWidth-1{1'b0}};
            end
            else
                sum <= BiasAdd;
        end
        else if(mux_valid)
        begin
            if(!mul[2*dataWidth-1] & !sum[2*dataWidth-1] & comboAdd[2*dataWidth-
1])
            begin
                sum[2*dataWidth-1] <= 1'b0;
                sum[2*dataWidth-2:0] <= {2*dataWidth-1{1'b1}};
            end
            else if(mul[2*dataWidth-1] & sum[2*dataWidth-1] &
!comboAdd[2*dataWidth-1])
            begin
                sum[2*dataWidth-1] <= 1'b1;
                sum[2*dataWidth-2:0] <= {2*dataWidth-1{1'b0}};
            end
            else
                sum <= comboAdd;
        end
    end

    always @(posedge clk)
    begin
        myinputd <= myinput;
        weight_valid <= myinputValid;
        mult_valid <= weight_valid;
        sigValid <= ((r_addr == numWeight) & muxValid_f) ? 1'b1 : 1'b0;
        outvalid <= sigValid;
        muxValid_d <= mux_valid;
        muxValid_f <= !mux_valid & muxValid_d;
    end


    //Instantiation of Memory for Weights
    Weight_Memory
#(.numWeight(numWeight),.neuronNo(neuronNo),.layerNo(layerNo),.addressWidth(addr
essWidth),.dataWidth(dataWidth),.weightFile(weightFile)) WM(
        .clk(clk),
        .wen(wen),
        .ren(ren),
        .wadd(w_addr),
        .radd(r_addr),
        .win(w_in),
        .wout(w_out)
    );

    generate
        if(actType == "sigmoid")
        begin:siginst
        //Instantiation of ROM for sigmoid
            Sig_ROM #(.inWidth(sigmoidSize),.dataWidth(dataWidth)) s1(
            .clk(clk),
            .x(sum[2*dataWidth-1-:sigmoidSize]),
            .out(out)
```

```
        );
        end
        else
        begin:ReLUinst
            ReLU #(.dataWidth(dataWidth),.weightIntWidth(weightIntWidth)) s1 (
            .clk(clk),
            .x(sum),
            .out(out)
        );
        end
    endgenerate

    `ifdef DEBUG
    always @(posedge clk)
    begin
        if(outvalid)
            $display(neuronNo,,,,"%b",out);
    end
    `endif
endmodule
```

The Weight_Memory module provides the per-neuron weight storage as a simple synchronous on-chip memory with parameterized depth and width. It exposes separate write and read, ports sharing a single clock: when wen is asserted, the value on win[dataWidth-1:0] is written into location wadd[addressWidth-1:0] at the rising edge of clk; when ren is asserted, the location addressed by radd[addressWidth-1:0] is synchronously read and the output wout[dataWidth-1:0] is updated on the next clock edge. The memory array is declared as reg [dataWidth-1:0] mem [numWeight-1:0], which synthesizes to on-chip RAM (M10K/MLAB) rather than logic when project settings permit. In “pretrained” builds, the memory contents are initialized from a text file at configuration time via $readmemb(weightFile, mem);, so no runtime writes are needed; otherwise, weights are loaded at runtime by asserting wen per address while software selects the neuron and layer externally.

Operationally, this module acts as a compact RAM where the neuron streams sequential radd addresses (one per input sample) to fetch weights during inference, while configuration logic or HPS software uses wadd/win/wen to populate weights ahead of time. Because the read is synchronous, the output value corresponds to read from the previous cycle, which the neuron aligns by delaying the corresponding input by one cycle. Typical FPGA mapping places the memory into block RAM resources, leaving adders, control, and multiply units to ALMs and DSPs elsewhere

in the design. The parameters numWeight, addressWidth, and dataWidth control capacity and precision, and weightFile selects the initialization source under the pretrained flow, making the

block reusable across layers and neurons with consistent timing and resource inference.

```
module Weight_Memory #(parameter numWeight = 3,
neuronNo=5,layerNo=1,addressWidth=10,dataWidth=16,weightFile="w_1_15.mif")
    (
    input clk,
    input wen,
    input ren,
    input [addressWidth-1:0] wadd,
    input [addressWidth-1:0] radd,
    input [dataWidth-1:0] win,
    output reg [dataWidth-1:0] wout);

    reg [dataWidth-1:0] mem [numWeight-1:0];

    `ifdef pretrained
        initial
        begin
            $readmemb(weightFile, mem);
        end
    `else
        always @(posedge clk)
        begin
            if (wen)
            begin
                mem[wadd] <= win;
            end
        end
    `endif

    always @(posedge clk)
    begin
        if (ren)
        begin
            wout <= mem[radd];
        end
    end
endmodule
```

The ReLU module implements a clocked rectified linear activation with fixed-point saturation. It accepts a wide accumulator input x that is 2*dataWidth bits (matching the neuron's MAC accumulator) and produces a dataWidth-wide output out. On each rising edge of clk, the module checks the sign of x using a signed comparison. If x is negative, the output is clamped to zero, implementing the standard ReLU behavior. If x is non-negative, the module extracts the fixed-point output slice from x by discarding weightIntWidth integer bits above the fractional boundary

plus the sign guard, i.e., it selects dataWidth bits starting at 2*dataWidth-1-weightIntWidth. Before The upper bounds for generating positive overflows from integers in the slice above guarantee that

if any of the upper weightIntWidth+1 bits (i.e., the highest-order bit of the integer) are "asserted" (i.e., set to 1), the resulting maximum value that can be represented in dataWidth bits will be clamped (i.e., limited) to the maximum possible value (in dataWidth bits) (i.e., with all higher-order bits set to 0 followed by all lower-order bits set to 1). The design ensures that sufficiently large positive accumulation within a system will not wrap around (i.e., exceed the maximum value represented in dataWidth bits), preserving stability for fixed-point numeric inference.. The design is fully synchronous (registered output), uses only simple logic (comparators, bit-slices, and muxing), and therefore maps efficiently to ALMs and flip-flops without requiring DSPs or block RAM.

```
module ReLU  #(parameter dataWidth=16,weightIntWidth=4) (
    input           clk,
    input   [2*dataWidth-1:0]   x,
    output  reg [dataWidth-1:0]  out
);


always @(posedge clk)
begin
    if($signed(x) >= 0)
    begin
        if(|x[2*dataWidth-1-:weightIntWidth+1]) //over flow to sign bit of
integer part
            out <= {1'b0,{(dataWidth-1){1'b1}}}; //positive saturate
        else
            out <= x[2*dataWidth-1-weightIntWidth-:dataWidth];
    end
    else
        out <= 0;
end

endmodule
```

The Sig_ROM module implements a parameterized lookup-table ROM for the sigmoid activation, providing a fixed-point output with synchronous read latency. It accepts an inWidth-bit input index x and returns a dataWidth-bit output out by reading from a pre-initialized memory mem. At elaboration time, the ROM contents are loaded from the file sigContent.mif using $readmemb, which encodes the sampled sigmoid curve in fixed-point format. Because x is treated as a signed value while the ROM address space is naturally unsigned, the module recenters the signed domain to an unsigned address by offsetting x by $2^{(inWidth-1):}$ if x >= 0 it computes y = x + $2^{(inWidth-1)}$, otherwise, y = x - $2^{(inWidth-1)}$. This effectively maps the signed range [$-2^{(inWidth-1)}$, $2^{(inWidth-1)}$ -1] to

the unsigned address range [0, $2^{inWidth}$ - 1]. The address y is registered on the rising edge of clk, and the ROM is read synchronously so that out = mem[y] is returned with one-cycle latency. The design is compact and synthesizes to on-chip ROM structures (often MLAB or M10K depending on size and tool settings), using only simple registers and a small adder/subtractor for the address recentering, which makes it well-suited to feed the neuron's activation stage with predictable timing and resource usage.

```
module Sig_ROM #(parameter inWidth=10, dataWidth=16) (
    input           clk,
    input   [inWidth-1:0]   x,
    output  [dataWidth-1:0]  out
    );

    reg [dataWidth-1:0] mem [2**inWidth-1:0];
    reg [inWidth-1:0] y;

    initial
    begin
        $readmemb("sigContent.mif",mem);
    end

    always @(posedge clk)
    begin
        if($signed(x) >= 0)
            y <= x+(2**(inWidth-1));
        else
            y <= x-(2**(inWidth-1));
    end

    assign out = mem[y];

endmodule
```

The file sigContent.mif provides the precomputed lookup values for the sigmoid activation used by Sig_ROM in the neural datapath. It is a plain text memory initialization file containing one binary value per line, ordered from the most negative to the most positive input index after Sig_ROM recenters the signed address space to an unsigned range. Each entry encodes the sigmoid output in fixed-point form (matching the dataWidth expected by Sig_ROM), starting near zero for large negative inputs, rising monotonically through the transition region, and saturating near the maximum representable value for large positive inputs. During synthesis/simulation, Sig_ROM loads this table with $readmemb("sigContent.mif", mem), so a read of mem[y] returns the sigmoid output for the quantized input corresponding to address y. This approach avoids runtime computation of the non-linear function, yielding a single-cycle,

deterministic activation stage that maps efficiently to on-chip ROM (MLAB/M10K), with the numeric behavior governed by this file's fixed-point samples.

```
0
0
0
0
0
0
0
0
0
0
0
1
100
1100
11110
1000100
10000000
10111011
11100001
11110011
11111011
11111110
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
11111111
```

The maxFinder module selects the index of the maximum value from a vector of fixed-point inputs presented as a single wide bus.The number of inputs (numInput) and the width of each input (inputWidth) define how many input entries there will be and how many bits each entry will be. When the signal i_valid goes high, the module takes a snapshot of the entire combined input sequence, i_data[(numInput*inputWidth)-1:0] and saves it into an internal buffer, also setting the starting maximum value (maxValue) to be the entry from the first lane of data. The module will then look through the data lanes each clock cycle by counting on a simple counter, keeping track of the maximum value of the lanes it has already examined, and comparing those values against maxValue; if it finds a lane with a value greater than maxValue, it will update the value of maxValue to the new value found in that lane, and also set the output (o_data) for the

corresponding lane number as the argument for the maximum value found. . After exactly numInput comparisons, the module raises o_data_valid for one cycle to indicate that o_data now holds the argmax index for that vector; the counter is reset, and the unit becomes ready for the next i_valid strobe. This design performs the reduction over time rather than fully in parallel, minimizing logic usage by reusing a single comparator path and a small amount of control. The internal input

buffering allows stable, cycle-by-cycle access to each lane without re-driving i_data. Resource mapping is lightweight: the comparator and multiplexing logic map to ALMs, the input buffer is implemented as registers (or could be inferred as distributed resources depending on synthesis), and there are no RAMs or DSPs required. Latency from i_valid to o_data_valid is numInput cycles plus one cycle for initialization, and throughput is one vector per numInput cycles, which is adequate for post-layer argmax while keeping area small.

```
module maxFinder #(parameter numInput=10,parameter inputWidth=16)(
input           i_clk,
input [(numInput*inputWidth)-1:0]   i_data,
input           i_valid,
output reg [31:0]o_data,
output  reg     o_data_valid
);

reg [inputWidth-1:0] maxValue;
reg [(numInput*inputWidth)-1:0] inDataBuffer;
integer counter;

always @(posedge i_clk)
begin
    o_data_valid <= 1'b0;
    if(i_valid)
    begin
        maxValue <= i_data[inputWidth-1:0];
        counter <= 1;
        inDataBuffer <= i_data;
        o_data <= 0;
    end
    else if(counter == numInput)
    begin
        counter <= 0;
        o_data_valid <= 1'b1;
    end
    else if(counter != 0)
    begin
        counter <= counter + 1;
        if(inDataBuffer[counter*inputWidth+:inputWidth] > maxValue)
        begin
            maxValue <= inDataBuffer[counter*inputWidth+:inputWidth];
            o_data <= counter;
        end
```

```
    end
end

endmodule
```