Department of Computer Engineering
Faculty of Engineering
University of Tripoli

A graduation project is submitted in partial fulfillment of requirements for the degree of Bachelor in Computer Engineering

Design and Implementation of Optimized CNN for Character Recognition on FPGA-based NIOS II Embedded Processor

By:
Nadia Anwar Ben Salem

Supervised by:
Dr. Mohamed Muftah Eljhani

Fall 2023

Design and Implementation of Optimized CNN for Character Recognition on FPGA-based NIOS II Embedded Processor

By:
Nadia Anwar Ben Salem

Approved by:

……………………………………
Dr. Mohamed Eljhani

……………………………………
Dr. Muharrem Drebi

……………………………………
Dr. Hussein Magboub

Department of Computer Engineering
Faculty of Engineering
University of Tripoli

Intellectual Property Rights Identification Form for Projects and Scientific Research

This form must be read and signed by students working on graduation projects, master's theses or any other research activities conducted at University of Tripoli / Faculty of Engineering / Department of Computer Engineering.

Intellectual property rights for projects and research activities and their results (such as graduation projects, master's theses, patents and any marketable research product) belong to the University of Tripoli/Department of Computer Engineering. These rights are subject to the laws, regulations and instructions of the University relating to intellectual property and patents.

I agree (Student's Name): ____

Student's ID: ____

As a condition of my participation in the research project entitled:
____
____
____

All intellectual property rights of the above-mentioned project or scientific research shall be attributable to the University of Tripoli/Department of Computer Engineering This requires me to inform the competent authority of the University of any invention or discovery that may result from such research and to be fully confidential therein and to work through the University to obtain the patent that may result from such research. I am also committed to placing the name of Tripoli University/Department of Computer Engineering and the names of all researchers involved in the research on any scientific bulletin for full research or its results, including publication of graduation projects, master's theses, doctorates, publication in journals, scientific conferences in general and posting on websites. I must adhere to the principles of copyright approved by the University of Tripoli/Department of Computer Engineering.

Student Signature: ____

Date: ____

Department of Computer Engineering
Faculty of Engineering
University of Tripoli

Plagiarism Declaration

I (Student's Name): ____

Student's ID: ____

hereby declare that I am the sole author of the graduation project entitled:
____
____
____

and that neither any part of the thesis nor the whole of the thesis has been submitted to any University or Institution for obtaining any degree / diploma / academic award.

This project was written by me and in my own words, except for quotations from published and unpublished sources which are clearly indicated and acknowledged as such. I am conscious that the incorporation of material from other works or a paraphrase of such material without acknowledgement will be treated as plagiarism, subject to the custom and usage of the subject, according to the University Regulations on Conduct of Examinations.

I shall be solely responsible for any dispute or plagiarism issue arising out of the graduation project.

Signature: ____

Date: ____

Abstract

Convolution neural network (CNN) character recognition has revolutionized the field of optical character recognition (OCR) technology, making it more accurate and efficient than ever before. While the impact of CNN character recognition has been overwhelmingly positive, there are also some limitations and challenges that need to be addressed in order to fully realize its potential. The software implementation, using a traditional CPU-based approach, faced challenges such as slower processing speeds and limited parallelism, which hindered real-time character recognition. In contrast, the hardware implementation effectively addressed these issues by leveraging parallelism, allowing multiple computations to be executed simultaneously.

This project explores the hardware implementation of CNN on NIOS II soft-core processor on FPGA. NIOS II and FPGA combination emerged as an optimal choice for this implementation due to their complementary strengths. This improvement is particularly impactful in applications requiring real-time image processing, such as autonomous driving systems, Digital Signal Processing (DSP) or live video surveillance, where reduced processing time can significantly enhance system responsiveness and efficiency. NIOS II processors offer flexibility and ease of integration, while FPGAs provide exceptional parallel processing capabilities, crucial for the efficient execution of CNNs.

The research involved the development of both software and hardware implementations of a CNN model that achieved 97.89% accuracy. The software implementation utilized a traditional CPU-based approach, while the hardware implementation leveraged the parallel processing capabilities of FPGAs. Comparative analysis demonstrated a significant performance improvement, with the hardware implementation achieving a speedup factor of approximately 115 times over the software counterpart. Additionally, the hardware design exhibited a power consumption of 137.42 mW, highlighting its efficiency. This project underscores the potential of FPGA-based accelerators in enhancing the efficiency of computationally intensive neural network tasks, offering insights into practical applications and future developments in the field of hardware-accelerated machine learning.

Acknowledgements

I would take this opportunity to appreciate those who contributed in preparation of this graduation project. Above all, I would like to express my gratitude to my project supervisor, Dr. Mohamed Eljhani for his support, constructive criticism, and constant motivation towards the success of this project. I would also like to acknowledge the support of faculty members of the Computer Engineering Department for offering me the conducive environment and resources for my study. My peers and friends deserve my gratitude as they have helped me a lot through their efforts and encouragement during this research.

Finally, I would like to express my deepest gratitude to my family for their unwavering support and encouragement throughout this academic journey. To my parents, whose constant belief in my abilities and dreams has been a source of immense motivation, I am profoundly grateful. Your love, patience, and sacrifices have provided the foundation upon which I have built this work.

A special thanks to my siblings for their understanding and support during the times when I was deeply immersed in this project. Your encouragement and presence have been a source of strength and comfort.

Table of Contents

List of Figures

List of Abbreviations

CNN	Convolutional Neural Network
FPGA	Field Programmable Gate Array
OCR	Optical Character Recognition
SOPC	System on a Programmable Chip
DSP	Digital Signal Processing
CPU	Central Processing Unit
GPU	Graphics Processing Unit
SoC	System on Chip
NLP	Natural Language Processing
BNN	Binary Neural Network
HLS	High Level Synthesis
RTL	Register Transfer Level
LPFP	Low Precision Floating-Point
EMNIST	Extended Modified National Institute of Standards
ASIC	Application Specific Integrated Circuits
LUT	Look Up Table
RAM	Random Access Memory
ROM	Read Only Memory
BRAM	Block Random Access Memory
LE	Logic Element
GPIO	General Purpose Input/Output
UART	Universal Asynchronous Receiver-Transmitter
ReLU	Rectified Linear Unit
SVD	Singular Value Decomposition
LCD	Liquid Crystal Display
MM	Memory Mapped
IP	Intellectual Property
HDL	Hardware Description Language

Introduction

This chapter provides a comprehensive overview of the research, including the problem statement, available solutions, project objectives, and an outline of the report structure. It sets the stage by presenting relevant information, theories, and prior work that the project builds upon. The chapter explores the origins, features, and approaches utilized to successfully complete the project. It delves into current research and related works on CNNs, particularly their application on the NIOS II processor and FPGA platforms. CNN is a deep learning algorithm, that had revolutionized many scientific and technological fields by providing powerful methods for processing and analyzing visual and spatial data. CNN shows excellent performance in solving complex computer vision problems including image classification, recognition, segmentation, objects and face detection. An increasing interest in CNNs is becoming more and more noticeable in medical imaging, as they have shown significant potential in the detection and diagnosis of breast cancer. In addition to breast cancer, CNNs have shown significant potential in the detection and diagnosis of several other medical conditions, including lung cancer, brain tumors, skin cancer, diabetic retinopathy, Alzheimer's disease, and cardiovascular diseases. The applications of CNN are much wider, they are also used in Natural Language Processing (NLP), astronomy, autonomous systems such as self- driving cars, security and surveillance, and much more.
However, the high performance of CNN algorithms comes with significant computational demands. These algorithms require a large number of parameters and involve extensive mathematical operations, which poses challenges for software implementation. CNNs require significant processing power and extensive memory to handle their large number of parameters and operations, which can be challenging on standard CPUs. Additionally, the scalability of CNNs on conventional CPUs is restricted due to their computational complexity, which requires massive parallel processing that CPUs are not optimized for. Additionally, limited memory bandwidth and parallelism capabilities in CPUs create bottlenecks during operations like convolution and pooling, leading to slower performance and increased resource usage. Optimizing CNNs for specific hardware restrictions can be complex, and the efficiency of the implementation is often dependent on the capabilities of underlying libraries and frameworks. In addition, while software implementations can utilize multi-threading, they may not fully take advantage of parallel processing as effectively as specialized hardware like Graphics Processing Units (GPUs) or Field-Programmable Gate Arrays (FPGAs).

One approach for optimization is model simplification. Techniques like pooling, pruning, and variations of conventional CNN algorithms such as lightweight architectures, and quantization have been developed to enhance resource efficiency. On the other hand, there is a growing trend towards using high-performance hardware platforms. GPUs and FPGAs are particularly noted for their capability to handle large-scale parallel operations. While GPUs are widely used for

CNN tasks due to their high performance and ease of use, choosing a NIOS II processor combined with FPGA for CNN implementation can offer significant advantages over GPUs, particularly in specialized applications such as embedded systems, real-time processing, edge computing, and low-power devices. The NIOS II soft-core processor can be configured and optimized for specific tasks, allows for tailored control and management of the CNN operations running on the FPGA. This setup provides exceptional customizability, enabling precise optimization of both the hardware and software aspects of the CNN implementation. FPGAs offer highly parallel processing capabilities and can be customized to accelerate specific CNN functions, leading to higher speed and improved power consumption compared to GPUs. Additionally, the integration of NIOS II with FPGA supports real-time processing and predictable performance, which are critical for applications with strict timing requirements. The combination of NIOS II and FPGA can also result in cost-effective solutions for large-scale implementations, where power consumption and thermal management are essential considerations. This approach allows for an efficient solution that maximize the strengths of both the processor and the FPGA, making it a powerful alternative to traditional GPU-based implementations for specific CNN applications.

FPGAs can be programmed to implement custom hardware accelerators tailored to the specific requirements of CNNs. Zhang proposed a high-performance FPGA-based accelerator for CNNs, demonstrating substantial improvements in speed and energy efficiency compared to CPU and GPU implementations [1]. Naveen Suda explored a hardware-software co-design approach for accelerating CNNs on FPGA-System on Chip (SoC) platforms, leveraging both FPGA fabric and NIOS II embedded processor to optimize performance [2]. Several studies have focused on implementing CNN-based handwritten character recognition systems on FPGA platforms with embedded processors. Jiang presented a handwritten digit recognition system using a CNN accelerator on an FPGA, controlled by a NIOS II processor. The system achieved real-time performance with high accuracy, demonstrating the feasibility of such implementations [3]. P. Wang implemented a binarized neural network (BNN) on FPGA for handwritten digit recognition. The BNN achieves 0.136 W power consumption and 18 μs image identification time. The accuracy rate is 85%, even without a batch normalization layer, the achieved accuracy is not as efficient as the one that was achived in this project [4].Jiang used HLS tools to deploy adaptable convolution and pooling IP cores on the ZYNQ7020 FPGA, achieving 74ms recognition time per digit and 98.89% accuracy at 100MHz. Siyu Zhu and Hu Huang implement a manual hardware-level CNN on an Intel Cyclone10 FPGA, achieving 0.0176 ms recognition time per digit and 97.57% accuracy at 150MHz. These works highlight the feasibility and efficiency of FPGA-based CNNs for real-time applications [5].Expanding on these advancements, several papers have specifically addressed CNN-based handwritten letter recognition on FPGA platforms. These studies typically focus on optimizing network architectures and implementation strategies to handle the broader and more complex problem of letter recognition compared to digit recognition. Ke Yu proposed a hardware optimization approach for OCR systems using Memory-Centric Computing and a “Memory-Tree” algorithm.

The algorithm was initially tested in C/C++ and OpenCV, then converted to RTL with Xilinx Vitis. The FPGA-based system recognized English capital letters and numbers in 34.24 µs and achieved a 77.87% reduction in power consumption compared to a traditional processor-based system [6]. De Oliveira developed a heterogeneous system for scene text character recognition, tested on the Terasic DE2i-150 platform, it achieved 65.5% accuracy and processed up to 11 frames per second while using only 11% of the FPGA's logic elements. This system balances performance with resource efficiency for embedded applications [7]. Furthermore, several studies have explored the use of fixed-point and floating-point data representations to optimize CNN performance on FPGA platforms. Wang proposed an 8-bit low-precision floating-point (LPFP) quantization method for FPGA-based CNN acceleration, achieving negligible accuracy loss (within 0.5%) without re-training. Their implementation on Xilinx FPGAs significantly improved throughput and DSP utilization, particularly for VGG16 and YOLO, compared to existing accelerators [8]. In their work, Jiang proposed a hardware-friendly quantization scheme for CNNs using improved logarithmic quantization, achieving high accuracy with negligible loss and efficient resource utilization on FPGA. The implementation on Zynq XC7Z020 reached 6.008W power consumption, demonstrating high resource efficiency [9]. Yanamala developed a 16-bit fixed-point FPGA accelerator on the PYNQ-Z2 board, achieving 82.45% speed more than NVIDIA Tesla K80 GPU. The design, optimized with techniques like Singular Value Decomposition (SVD), array partitioning, and loop unrolling, showed significant speed improvements over CPU and GPU implementations for MNIST and Tumor dataset classification [10].

The achieve of this research:

- Design CNN architecture for specific problem – handwritten characters (letters) recognition.
- Implement the proposed CNN design in python, train it with Extended Modified National Institute of Standards and Technology (EMNIST) dataset.
- Implement the CNN on FPGA-based NIOS II processor on DE2i-150 board to utilize the power of FPGA for high speed, parallel processing, high performance, energy efficiency, and reconfigurability for acceleration.
- Analyze the resource usage of the FPGA solution and make a comparison in performance and efficiency among hardware, and software implementation.

1.1 FPGA Platform

FPGA is the abbreviation of Field programmable Gate Array. It is a type of integrated circuit that can be programmed post-manufacturing by the user. FPGAs are different from traditional processors in that they do not have a fixed architecture; instead, they have programmable logic blocks and reconfigurable interconnects that can be customized for specific tasks. The flexibility, low latency, and high energy efficiency are main advantages of FPGAs, which made them

perfect for tasks that need custom hardware features, high speed, and parallel processing abilities.
Therefore, FPGA is widely used in producing highly customizable SoCs, ASIC verification, high performance computing etc.

1.2 FPGA Resources

Each resource plays a crucial role in enabling the FPGA to perform a wide range of functions, from basic logic operations to complex signal processing and communication tasks. The main resources are:

Look-up Table (LUT)

LUT works as a generator of functions. In the Cyclone IV GXarchitecture, 5-input LUTs are implemented. Wide multiplexers, along with other components like LUTs, flip flops, arithmetic, and carry chains, are combined to create a Logic Elelment (LE). LE serves as the primary tool for designing versatile combinatorial and sequential circuits on FPGA.

Digital Signal Processor (DSP)

FPGA-based DSP slices can execute a range of frequently utilized arithmetic functions. Using hardware parallelism with DSP can increase data throughput and efficiency for DSP applications. One DSP slice can be set up to carry out various arithmetic operations, such as 4-input addition, multiplication, multiply-accumulation, etc. The input and output data width can also be adjusted. The DSP slice is designed for minimal power usage, fast performance, compact dimensions, and flexibility.

Block Random Access Memory (BRAM)

BRAM is the primary memory component found on FPGA devices. They are dispersed and mixed with other customizable components such as DSP and LUT on FPGA. BRAM benefits from the flexibility provided by close interaction with DSP and LUT. In Cyclone IV GX devices, the block RAM can hold a maximum of 16 Kbits of data. It could be arranged in different memory block configurations. It could be either RAM or ROM. It can have one port or it can have two ports. Also there is ability to define the port width and number of lines.

FPGA architecture consist of thousand of LEs, known also as logic blocks, surrounded by a system of programmable interconnects, that routs signals between LEs. Input/output blocks interface between the FPGA and external devices. The FPGA architecture is shown in Figure 1.1.

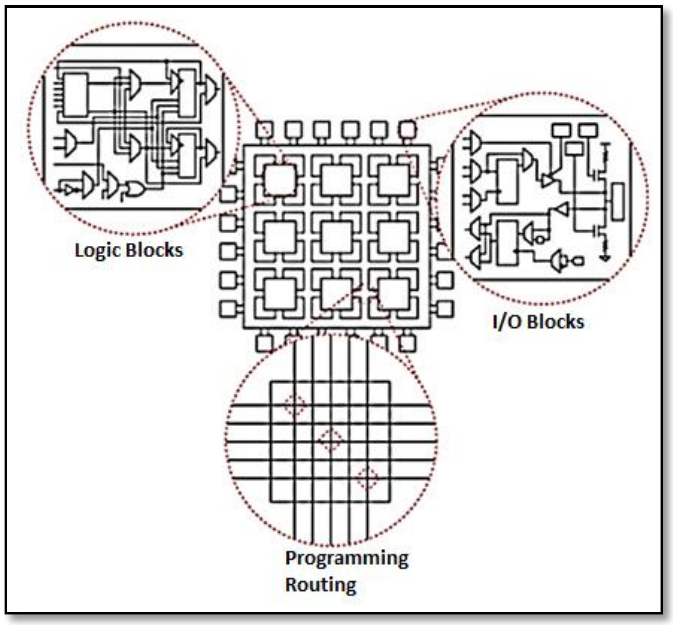


Figure 1.1 FPGA Architecture

1.3 NIOS II Processor

The NIOS processor, a soft processor core made by Intel (previously Altera), is intended for incorporation into FPGA designs. It can be personalized to fit specific application needs, making it a flexible option for embedded processing duties on an FPGA. NIOS II processor is based on the Harvard architecture, and incorporates many enhancements over the original NIOS architecture, making it more suitable for a wider range of embedded computing applications, from DSP to system-control.

1.3.1 Key Features of NIOS II

- **Customizable Architecture**

The NIOS II processor have multiple core variants such as fast, standard, and economy, each offering different balances of performance and resource usage. Also an advantage of NIOS is the ability of add custom instructions, to accelerate application-specific tasks.

- **Scalability**

The processor supports multiple address and data widths, allowing it to be scaled according to application requirements. Also NIOS can interface with various types of memory, including on-chip RAM, external memory, and cache, ensuring flexible memory management.

- **Integration**

  As a soft processor, the NIOS core is implemented within the FPGA fabric, enabling tight integration with other FPGA resources. It also supports a wide array of customizable peripherals, like timers, UARTs, and GPIOs.

1.4 Python in Machine Learning with Pytorch

Python is widely used in machine learning because of its ease of use, readability, and wide range of libraries and frameworks available. It offers a strong environment for manipulating data, creating visualizations, and developing machine learning models.

PyTorch was created by Facebook's AI Research lab as an open-source machine learning library. It offers a versatile and user-friendly system for deep learning, commonly utilized in research and production settings. PyTorch provides dynamic computation graphs, allowing for easier adjustments to the network behavior in real-time, as opposed to the static computation graphs found in TensorFlow.

PyTorch simplifies the process of creating, training, and deploying CNNs. Its flexible computational graph and wide range of libraries allow novices and professionals alike to create advanced deep learning models.

1.5 CNN

A Convolutional Neural Network is a deep learning model that excels in image classification tasks. CNNs extract image features through convolutional operations and utilize these features to categorize objects. The network is designed to automatically and adaptively learn spatial hierarchies of features through training. When classifying an image, the network aggregates the learned features to determine and vote for the most likely class the image belongs to.

Deep learning algorithms operate in two phases: training and inference. In the training phase, CNNs use a dataset of labeled images to learn, employing the backpropagation algorithm to update the network's parameters. Once the model is well-tuned and trained, it is used to classify new data samples, a process known as inference.

During inference, the structure and parameters of the neural network remain fixed, and the model processes each new data sample. Consequently, optimizing the inference phase is a key focus, as it directly impacts the efficiency of classifying new inputs.

This thesis is organized in the following structure:

- Chapter 1 is an introduction that describes the problem. It introduces the structure of CNN. Include an introduction to FPGA and Nios II soft-core processor. It also include literature review, which also including researches of resource optimized CNN models.

- Chapter 2 describes the main tools utilized in this project for software development and hardware design.
- Chapter 3 presents the proposed methodology, that covering the design and implementation of CNN system model, with the software and hardware implementation steps.
- Chapter 4 is dedicated to presenting the results, and comparative analysis of the project.
- Chapter 5 includes conclusion and future work.

Experimental Background

Developing and fine-tuning a CNN demands a strong experimental setup with specialized software tools for software development and hardware design. This chapter describes the main tools used in the project: Virtual Studio Code, ModelSim, Quartus Prime, and Eclipse, all of which were crucial at various points in the development process.

2.1 Virtual Studio Code

Visual Studio Code (VS Code) is a well-regarded open-source integrated development environment (IDE) created by Microsoft, known for its versatility, wide range of features, and compatibility with various programming languages. VS Code played a critical role in setting up and training the CNN in Python for this project. The streamlined coding process was enhanced by the IntelliSense and code auto completion features, and effective troubleshooting was made easier by the integrated debugging tools. The Git integration within the IDE facilitated seamless version control, aiding in collaboration and project management. Furthermore, the lightweight design and ability to work across different platforms of VS Code ensured a quick and efficient development process, allowing users to personalize the environment using numerous extensions. This makes it a crucial tool for handling the intricate tasks required in CNN development and training.

2.2 ModelSim

ModelSim is a popular simulation tool used for verifying digital logic designs, especially those created in hardware description languages such as VHDL, Verilog, and SystemVerilog. Created by Mentor Graphics, ModelSim is well-known for its robust capabilities that enable to test and troubleshoot hardware designs prior to being implemented on physical devices such as FPGAs or ASICs. ModelSim played a crucial role in the simulation of the hardware accelerator for this project, as it provided detailed waveform visualization that was essential for verifying the correct operation of the CNN design. By examining these waveforms, tracing signal behaviors and interactions within the accelerator was enable, allowing for early detection and resolution of potential issues before synthesis. This made ModelSim an indispensable tool in ensuring the functionality and reliability of the hardware implementation.

2.3 Quartus Prime

Quartus Prime is a comprehensive software suite developed by Intel (formerly Altera) for the design, synthesis, and implementation of digital circuits on FPGAs, and SoCs. It is widely recognized for its powerful toolset that supports the entire FPGA design flow, from initial concept to final hardware deployment.

2.3.1 Features of Quartus Prime

- **Design Entry and Synthesis**:
  Quartus Prime offers multiple options for design entry, including schematic capture, hardware description languages like VHDL and Verilog, and block-based design. Once the design is captured, the software performs synthesis, converting the high-level code into a netlist that represents the logic gates and connections needed to implement the design on an FPGA.

- **Optimization and Resource Management**
  The tool includes advanced optimization algorithms that focus on improving the performance and resource utilization of the design. Quartus Prime can optimize for various factors such as speed, power consumption, and area, helping to ensure that the design meets the specific constraints of the target FPGA.

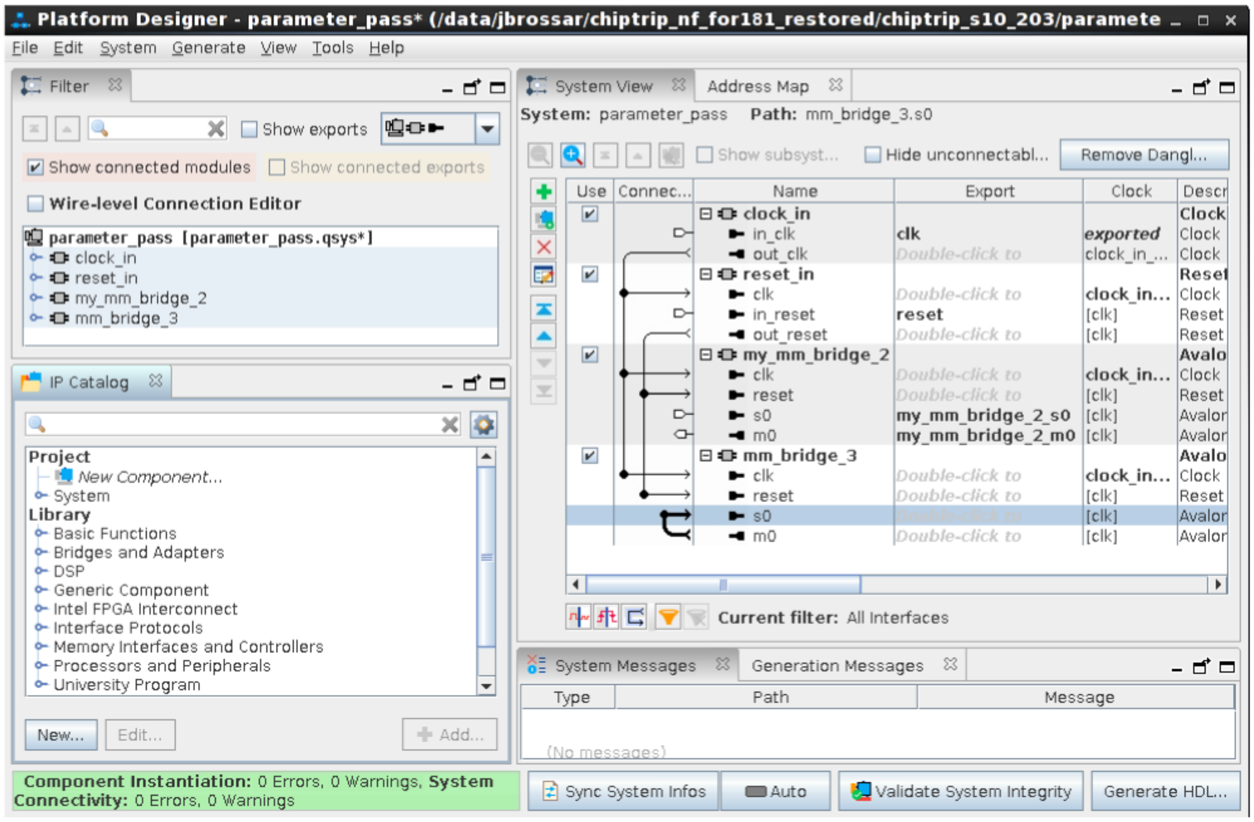
- **Place-and-Route**
  After synthesis, Quartus Prime performs place-and-route, where the synthesized netlist is mapped onto the physical resources of the FPGA. The tool determines the optimal placement of logic elements and routes the connections between them, taking into account timing constraints and the physical architecture of the FPGA.

- **Simulation and Verification**
  Quartus Prime integrates with simulation tools like ModelSim to allow for the verification of designs before and after synthesis. This ensures that the final implementation behaves as expected when deployed on hardware.

- **Device Support and IP Integration**
  Quartus Prime supports a wide range of Intel FPGA devices and includes an extensive library of pre-built Intellectual Property (IP) cores, which can be integrated into the design to add functionality without the need to develop it from scratch. These IP cores cover a variety of functions, including processors, memory controllers, and communication interfaces.

2.3.2 Platform Designer

Platform Designer, previously called Qsys, is a robust system integration tool in Intel's Quartus Prime software suite, aimed at making the creation of intricate FPGA-based systems easier. Designers can use a visually intuitive interface to link and set up different IP cores like processors, memory interfaces, and custom logic. Platform Designer streamlines numerous system integration tasks by automating the generation of interconnects, clock domain management, and maintaining data flow consistency across components. It further facilitates the development of personalized IP blocks and their smooth incorporation into current IP collections, allowing for adaptability and expandability in system planning. By simplifying the integration process, Platform Designer speeds up development cycles, facilitating the design of complex embedded systems with numerous components and peripherals. The graphical user interface of Platform Designer is shown in Figure 2.1.



[✓] Show connected modules

[ ] Wire-level Connection Editor

parameter_pass [parameter_pass.qsys*]
- clock_in
- reset_in
- my_mm_bridge_2
- mm_bridge_3

Project
- New Component...
- System

Library
- Basic Functions
- Bridges and Adapters
- DSP
- Generic Component
- Intel FPGA Interconnect
- Interface Protocols
- Memory Interfaces and Controllers
- Processors and Peripherals
- University Program

System: parameter_pass **Path:** mm_bridge_3.s0

Use	Connec...	Name	Export	Clock	Descr
✓		⊟ **clock_in**			**Clock**
		in_clk	**clk**	***exported***	Clock
		out_clk	*Double-click to*	clock_in_...	Clock
✓		⊟ **reset_in**			**Reset**
		clk	*Double-click to*	**clock_in...**	Clock
		in_reset	**reset**	[clk]	Reset
		out_reset	*Double-click to*	[clk]	Reset
✓		⊟ **my_mm_bridge_2**			**Avalo**
		clk	*Double-click to*	**clock_in...**	Clock
		reset	*Double-click to*	[clk]	Reset
		s0	**my_mm_bridge_2_s0**	[clk]	Avalon
		m0	**my_mm_bridge_2_m0**	[clk]	Avalon
✓		⊟ **mm_bridge_3**			**Avalo**
		clk	*Double-click to*	**clock_in...**	Clock
		reset	*Double-click to*	[clk]	Reset
		s0	*Double-click to*	[clk]	Avalon
		m0	*Double-click to*	[clk]	Avalon

Current filter: All Interfaces

Type	Path	Message
(No messages)		

Component Instantiation: 0 Errors, 0 Warnings, **System Connectivity:** 0 Errors, 0 Warnings

Figure 2.1 Platform Designer GUI

2.4 Eclipse Software

Eclipse, widely utilized in embedded systems and Java programming, is an open-source integrated development environment (IDE) for software development. It offers a strong foundation with a diverse range of tools and plugins for various stages of project development, making it a flexible option for developers handling intricate projects. Eclipse played a crucial role in the project, particularly in conjunction with the NIOS II Embedded Design Suite. It provided a comprehensive platform for developing and debugging the C++ code that ran on the NIOS II processor, which managed the FPGA-based CNN accelerator. Additionally, its integration with version control systems allowed for effective project management and collaboration.

Methodology

This chapter gives a thorough explanation of the methodology used in this project. This research work focuses on exploring how CNN can be successfully implemented on NIOS II processor with a hardware accelerator, and details the steps and tactics used to accomplish this objective. A CNN was designed and implemented in Python to train on the dataset. The weights and biases learned during training were then extracted and used to create a hardware accelerator, that was connected as custom IP to NIOS II processor. This method connects software-guided training with physical hardware, improving the CNN's effectiveness and productivity.

3.1 Preparing a dataset for model training

The dataset selected for this project is the EMNIST [12] dataset specifically designed for letters. This dataset is famous for its thorough representation of all 26 letters of the English alphabet, with each letter corresponding to a distinct class from A to Z. Each class contains 2,400 samples, providing a diverse and strong representation of handwritten letter variations. A scope of EMNIST dataset is shown in Figure 3.1. Dataset includes pictures with white letters on a black background. This color combination creates a sharp contrast between the characters and the background, making it easier to differentiate them for neural networks training. The dataset was initially in CSV file format.

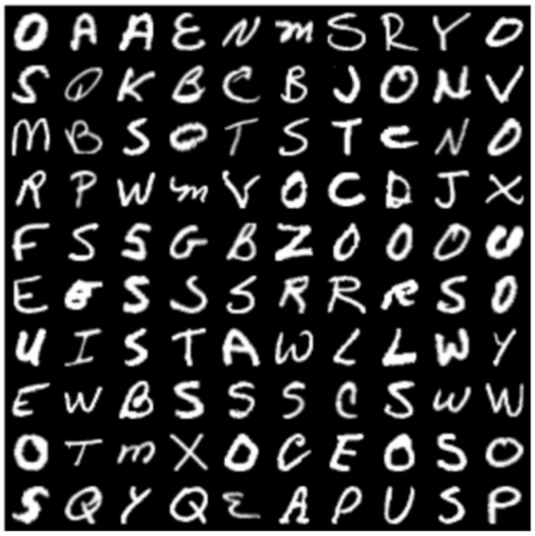


Figure 3.1 EMNIST Dataset

A MATLAB script was then employed to analyze the CSV file and extract every type of letters in a structured manner. Afterwards, the data that was retrieved was transformed into image form, with every image adjusted to a uniform size of 28×28 pixels.

3.2 Preprocessing of images for recognition

Tools from OpenCV library were used to preprocess the image. The *cv2.imread()* function was used for image loading in grayscale mode, converting it to a single-channel image instead of a 3-channel RGB image. To resize the image into 28×28 pixel resolution the *cv2.rezise()*function was used. Then the pixel values, which were originally range from 0 to 255 (grayscale), where 0 represents black, while 255 represents white, are normalized by dividing them by 255. This scales the pixel values to a range between 0.0 and 1.0, which is common in many image processing and machine learning tasks. After the preprocessing images were ready to feed the CNN for training and testing.

3.3 CNN Layers

A CNN is organized into layers. For a character recognition task, the network starts with an input layer that receives the image of character and ends with an output layer that provides values representing the probability of different classes. Between these layers, there are several hidden layers, including convolutional layers, activation functions, pooling layers, and fully connected layers, which process and transform the data. An example of this structure is shown in Figure 3.2.

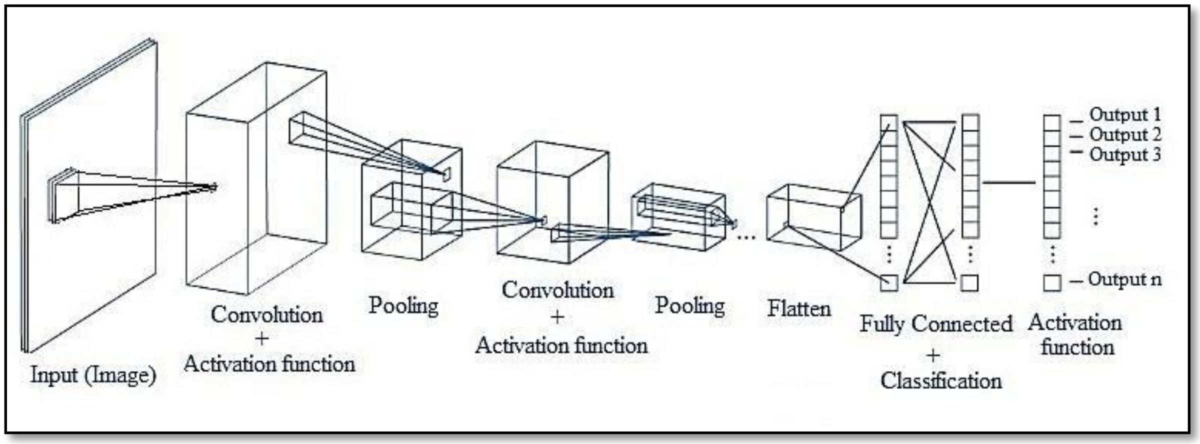


Figure 3.2 Structure of CNN

3.3.1 Convolution Layer

The convolutional layer is primarily used for feature extraction in neural networks. It accomplishes this by initially applying a convolution function, followed by an activation function on the resulting output. Multiple convolutional layers are typically utilized to progressively extract and refine features from the input data.

A convolutional layer has M input channels and N output channels. Each input channel contains a feature map sized $W_f \cdot H_f$. The $M \cdot W_f \cdot H_f$ input convolves with a convolution kernel sized $M \cdot W_k \cdot H_k$ and produces a $W_f \cdot H_f$ output feature map in one of the output channels. Figure 3.3 shows a convolution with a single kernel. Convolution kernels contains trained weights of the neural network. Convolution with N such kernels produces an output sized $N \cdot W_f \cdot H_f$.

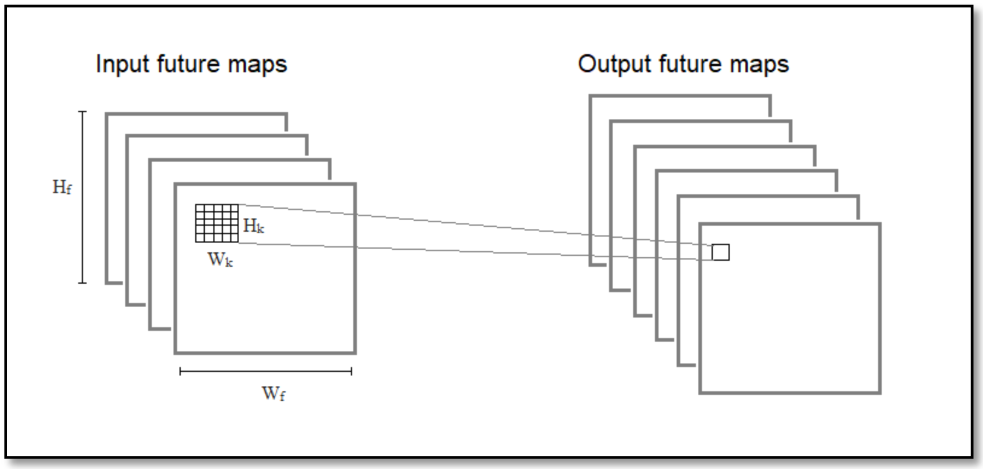


Figure 3.3 Structure of Convolutional Layer

W_f is the feature map width, and H_f is the feature map height. W_k is the kernel width, and H_k is the kernel height. For each pixel in input C and output G, the expression is shown in Equation 3.1, where K represents the convolution kernel.

$$G[n,x,y] = \sum_{i=-\frac{W_k}{2}}^{\frac{W_k}{2}} \sum_{j=-\frac{H_k}{2}}^{\frac{H_k}{2}} \sum_{m=0}^{M-1} = C[m,x+i,y+j] \cdot K[n,i,j] \qquad (3.1)$$

3.3.2 Padding

The size of output feature maps will shrink due to convolution. Padding is a technique used to control the spatial dimensions of the output feature maps. When applying a convolution operation, the filter or kernel slides over the input data, and padding helps manage the size of the resulting feature map.

In convolution there are mainly two types of padding: valid padding and same padding. Valid padding, also known as no padding, involves applying the convolution operation without adding any extra pixels to the input. This results in a smaller output feature map compared to the input, as the convolution filter is only applied where it fully overlaps with the input data. On the other hand, same padding, also known as zero padding, involves adding extra pixels (usually zeros) around the input's border. This approach ensures that the output feature map retains the same spatial dimensions as the input, allowing the convolution operation to cover the entire input, including its edges. Figure 3.4 shows how different types of padding affect the size of the output future maps.

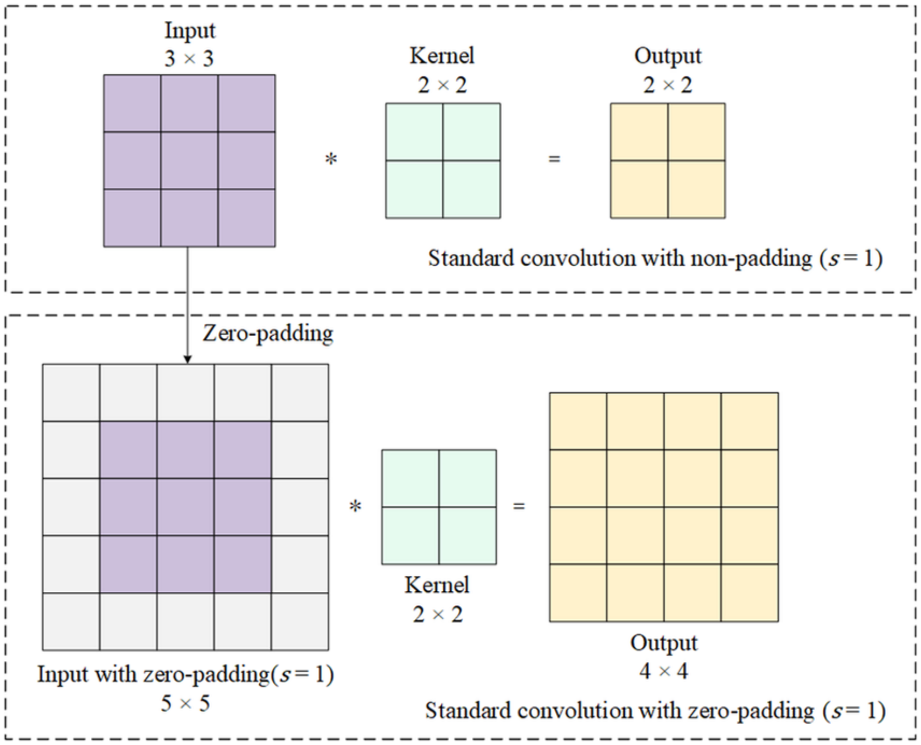

Input
3 × 3
Kernel
2 × 2
Output
2 × 2
Standard convolution with non-padding (s = 1)
Zero-padding
Input with zero-padding(s = 1)
5 × 5
Kernel
2 × 2
Output
4 × 4
Standard convolution with zero-padding (s = 1)

Figure 3.4 Convolution with and without zero padding

3.3.3 Stride

Stride determines the number of steps the convolution kernel moves during the convolution process and also defines the factor by which the output is downscaled. Figure 3.5 illustrates a 2D convolution using a 2 by 2 convolution kernel with a stride of 1, and 2.

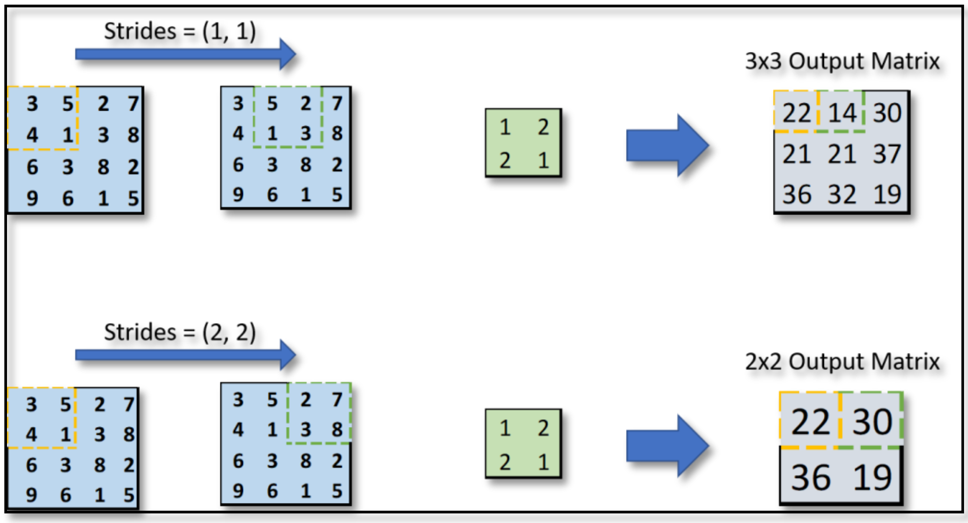


Figure 3.5 Convolution with different strides

To determine the spatial dimensions of the output feature maps from a given input size $W_{in} \times H_{in}$, kernel size $W_k \times H_k$, stride S, and padding P, the output size $W_{out} \times H_{out}$ can be calculated as shown in Equation 3.2.

$$W_{out} = \left\lceil \frac{W_{in} - W_k + 2P}{S} \right\rceil + 1 \quad , \quad H_{out} = \left\lceil \frac{H_{in} - H_k + 2P}{S} \right\rceil + 1 \qquad (3.2)$$

3.3.4 Activation Function

An activation function is applied to the feature map and the result is forwarded to the next layer as input. This function introduces nonlinearity to the network, allowing it to learn more complex patterns and high-order polynomials. By incorporating nonlinearity, the activation function enhances the network's ability to handle complicated tasks and improve its overall learning capability. Most common activation function is Rectified Liner Units (ReLU), shown in the Equation 3.3.

$$ReLU(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} \qquad (3.3)$$

3.3.5 Pooling Layer

Pooling layer segment the input data into smaller regions, known as pooling windows or receptive fields, and apply an aggregation operation within each region, such as taking the maximum or average value. This process reduces the spatial dimensions of the feature maps, resulting in a more compact and condensed representation of the input data. By decreasing the size of the feature maps, pooling layers help to simplify the data and lower computational demands, while maintaining the essential features for further analysis.
The difference between max-pooling and average-pooling layers output is shown in Figure 3.6.

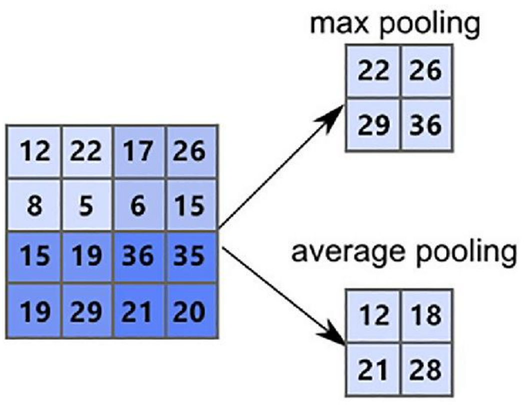


Figure 3.6 Max pooling vs. Average pooling

For pooling layer with stride of 2 the output future map is reduced into $\frac{1}{4}$ of the input size as result.

3.3.6 Fully Connected Layer

In a fully connected layer, the feature map from the previous layer is transformed into a linear structure. Each element of this feature map functions as a neuron, and every neuron is fully connected to all neurons in the subsequent layer. In a fully connected layer with M input neurons and N output neurons, and for each neuron in input X and output Y, the expression is shown in Equation 3.4, where W represents the weight of each connection, and B represents the bias of each output neuron.

$$Y[n] = \sum_{m=0}^{M-1} X[m] \,.\, W[m,n] + B[n] \tag{3.4}$$

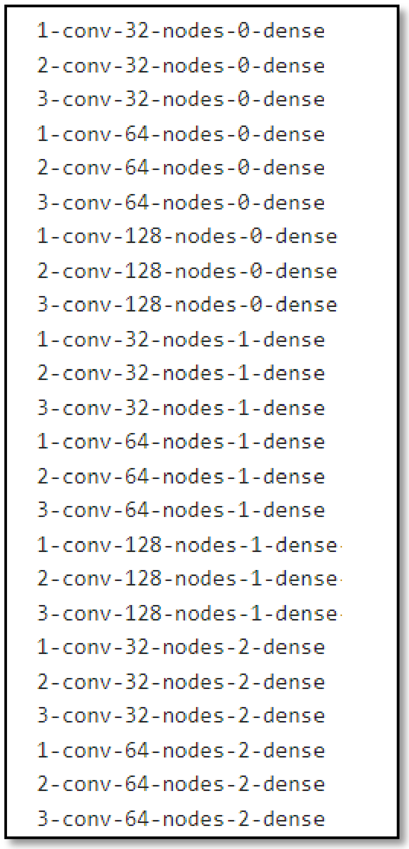
In a CNN, there can be multiple fully connected layers, each typically followed by a ReLU activation function.
However, the final fully connected layer usually employs a softmax activation function to output a probability distribution over the possible classes. The softmax function converts the raw output scores from the network into probabilities, which sum to one, making it ideal for multi-class classification problems.

For an input vector z, the softmax function is defined as in Equation 3.5. In the equation K is the number of classes, z_i is the input score for class I, and e is the base of the natural logarithm.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1,2, \dots \dots , K \tag{3.5}$$

3.4 CNN Design

In this research, different CNN structures were investigated to find the best model for letter recognition. Various configurations were tested, such as adjusting the number of layers, filter sizes, number of channels, and types of activation functions. For the convolution layer ReLU was selected as the activation function because of its simplicity and computational efficiency, which are essential for handling large datasets. In contrast to functions such as Sigmoid or Tanh, ReLU assists in addressing the issue of vanishing gradient, allowing for quicker and more efficient learning in deep neural networks. The network's capacity to generate sparse activations enhances generalization, and its nonlinearity enables capturing intricate patterns in character shapes. As the pooling layer the max pooling was selected instead of average pooling because it helps maintain essential features like edges and corners, which are important for character differentiation. Max pooling keeps strong features by choosing the highest value in each window, improving recognition accuracy, while average pooling may weaken these important details. Initially, structures with fewer layers were tried, followed by deeper networks to evaluate their impact on performance. Tests showed that while deeper networks could achieve higher accuracy, they had tendency to over fit. Less complex models, however, failed to achieve satisfactory accuracy. The architectures that was tested are shown in Figure 3.7. After various rounds of testing, the design that demonstrated the most ideal balance between generalization and performance included 8 layers. This setup consistently delivered better outcomes in important evaluation measures like accuracy, precision, and loss.



```
1-conv-32-nodes-0-dense
2-conv-32-nodes-0-dense
3-conv-32-nodes-0-dense
1-conv-64-nodes-0-dense
2-conv-64-nodes-0-dense
3-conv-64-nodes-0-dense
1-conv-128-nodes-0-dense
2-conv-128-nodes-0-dense
3-conv-128-nodes-0-dense
1-conv-32-nodes-1-dense
2-conv-32-nodes-1-dense
3-conv-32-nodes-1-dense
1-conv-64-nodes-1-dense
2-conv-64-nodes-1-dense
3-conv-64-nodes-1-dense
1-conv-128-nodes-1-dense
2-conv-128-nodes-1-dense
3-conv-128-nodes-1-dense
1-conv-32-nodes-2-dense
2-conv-32-nodes-2-dense
3-conv-32-nodes-2-dense
1-conv-64-nodes-2-dense
2-conv-64-nodes-2-dense
3-conv-64-nodes-2-dense
```

Figure 3.7 CNN tested architectures

A crucial tool for model optimization was TensorBoard, which provided visualizations of accuracy and loss graphs across all architectures, enabling effective performance tracking and comparison. TensorBoard was used to choose the most fitting architecture because it allowed for real-time monitoring of accuracy, validation accuracy, and loss on a single graph. This comprehensive view helped identify the best-performing models, detect overfitting early, and guide hyperparameter tuning, ensuring optimal model performance and effective architecture selection. Figure 3.8 presents graphs for multiple trained architectures, accuracy, loss, validation accuracy, and validation loss, allowing for a comparative analysis of their performance.

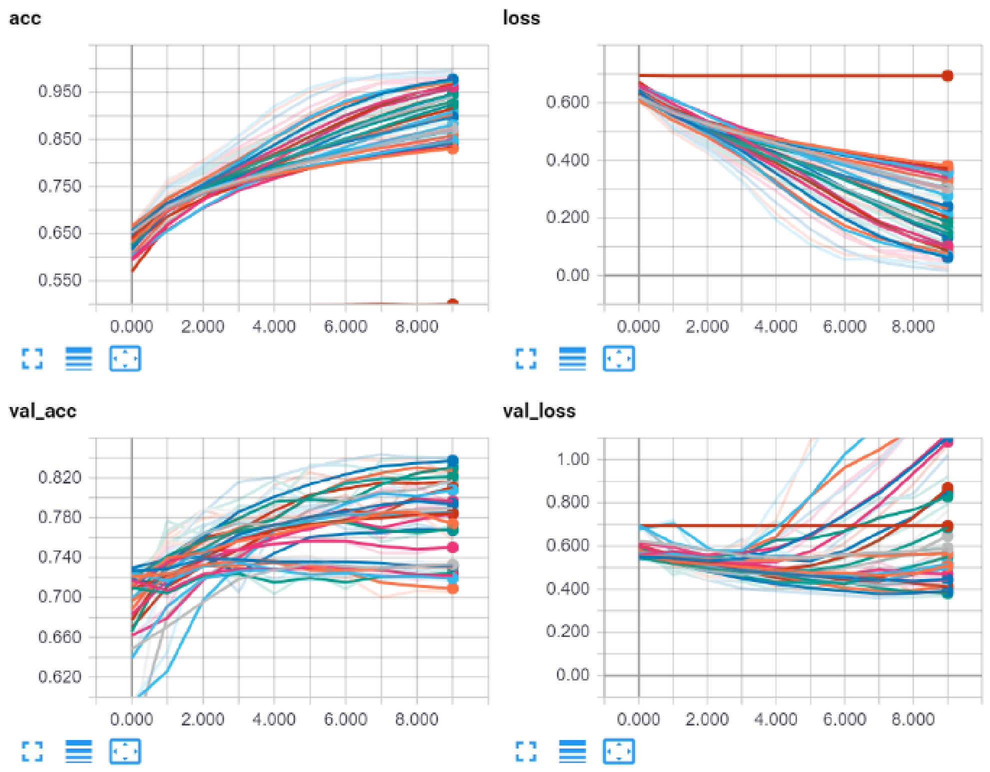


Figure 3.8 Different Architecture Graphs

As a result of this tests, valid padding (no padding) was selected for the convolutional layer, resulting in the filter being applied only where it completely overlapped with the input data. Therefore, the resultant feature map is reduced in size compared to the input as no extra pixels are included along the edges to preserve the original dimensions. Additionally, a stride of 1 was selected, meaning the filter moves one pixel at a time across the input, ensuring detailed feature extraction while still contributing to the reduced output size. In first convolution layer the kernel size was set to 5×5, in two other layers kernel size was reduced to 3×3. Max pooling was chosen with a stride of 2. This implies that during the pooling operation, the maximum value was chosen within every 2×2 section of the input, and the filter shifted by two pixels at a time.The architecture of CNN that was chosen is presented in Figure 3.9

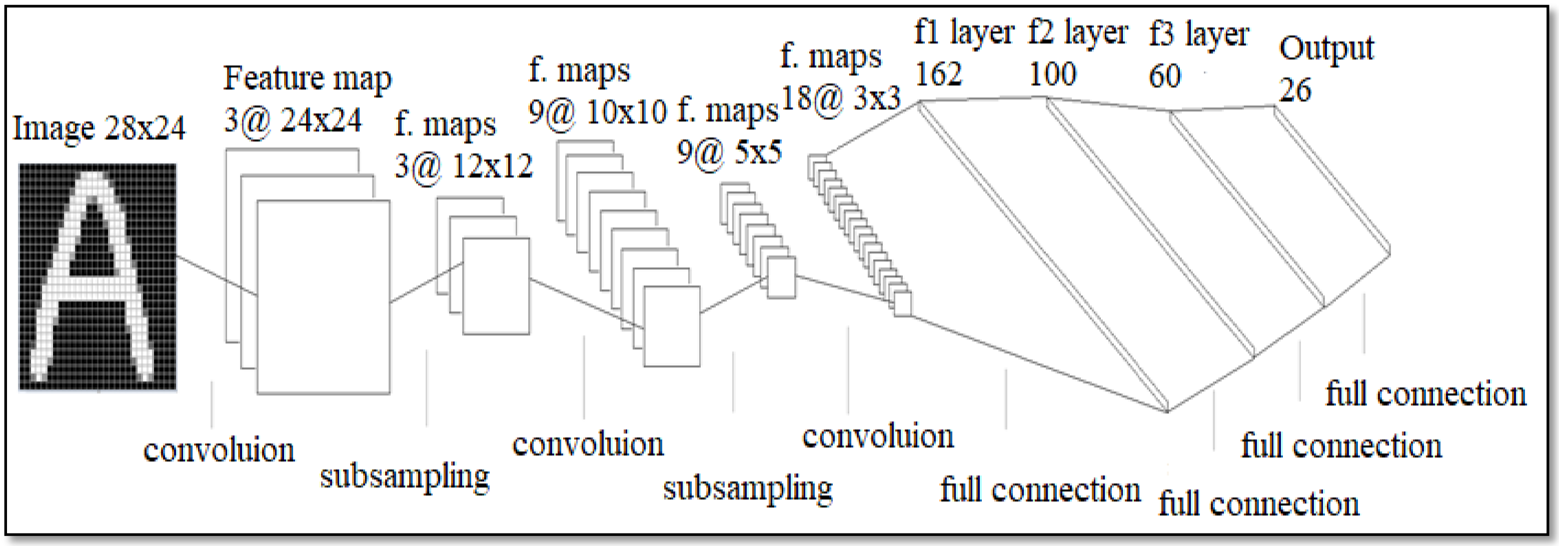


Figure 3.9 CNN Architecture

3.5 Training Process of the CNN

The image data is presented to the network and passed through the network layers. The first step in a CNN is to detect and investigate the unique features and structures of the objects to be differentiated. Filter matrices are used for this. Once a neural network has been modeled, these filter matrices are initially still undetermined and the network at this stage is still unable to detect patterns and objects. The networks are trained once during development and testing. After that, they are ready for use and the parameters no longer need to be adjusted.

Backward propagation is an algorithm used in CNN straining. It is responsible for updating the network's weights and biases in order to minimize the error in predictions.

First step in CNN training is forward propagation, which starts with image applied into the network, then the layers computation is execute in sequential way in the same manner that was explained in the section of the report, then the output is generated by the output layer, which is typically a vector of logits representing class scores for classification tasks.

After that the loss is calculated by the loss function, which measures the difference between the predicted output and the true labels. Loss functions for classification tasks include cross-entropy loss, the expression is shown in Equation 3.5, they$_i$ is the true label, and $\hat{y}_i$ is the predicted probability for class i.

$$Loss = -\sum_i y_i \log(\hat{y}_i) \qquad (3.5)$$

The next step is the backward propagation, that goal is to compute the gradient of the loss with respect to each parameter in the network. This is achieved using the chain rule of calculus to propagate the error backward through the network. As the name indicate it is propagating the error backward from the output layer to the input layer.

At output layerthe gradient of the loss is computed with respect to the input to the softmax layer which is implemented in Equation 3.6. For $\delta^{(L)}$ being the error at the output layer, $\hat{y}$ the predicted output, and y the true label.

$$\delta^{(L)} = \hat{y} - y \tag{3.6}$$

For the fully connected layer, gradients of the weights and biases are computed. The operations are represented mathematically in Equations 3.7-3.9.

$$\nabla W^{(l)} = \delta^{(l+1)}.(a^{(l)})^T \tag{3.7}$$

$$\nabla b^{(l)} = \delta^{(l+1)} \tag{3.8}$$

$$\delta^{(l)} = (W^{(l+1)})^T.\delta^{(l+1)} \cdot f'(z^l) \tag{3.9}$$

where $\nabla W_{(l)}$ and $\nabla b_{(l)}$ are the gradients of weights and biases, $\delta_{(l)}$is the error term for layer l, $a_{(l)}$ is the activation from the previous layer, $W_{(l+1)}$ are the weights of the next layer, and $f'(z^{(l)})$ is the derivative of the activation function applied to the pre-activation $z^{(l)}$.

Pooling layers transmit error signals by distributing the gradient to the max value locations (in max pooling) or equally (in average pooling) due to their lack of weights.
For convolutional layers, compute the gradient is done with respect to the filters, as shown in Equation 3.10, and 3.11.

$$\nabla W_{conv}^{(l)} = \delta_{conv}^{(l+1)} * a_{conv}^{(l)} \tag{3.10}$$

$$\delta_{conv}^{(l)} = \left(\delta_{conv}^{(l+1)} * W_{conv}^{(l+1)}\right) \cdot f'\left(z_{conv}^{(l)}\right) \tag{3.11}$$

where $*$ denotes the convolution operation, $\delta^{(l)}$ is the error term for the convolutional layer, $a^{(l)}$ is the activation map from the previous layer, and $f'(z^{(l)})$ is the derivative of the activation function applied to the convolutional layer's pre-activation map.

After calculating the gradients in the backpropagation process, the following step in training a CNN is adjusting the network's parameters (weights and biases) to reduce the loss function. This is accomplished using optimization algorithms. Two widely used optimization algorithms are Stochastic Gradient Descent (SGD) and Adaptive Moment Estimation (Adam). The computation of gradiens for weights and bias is shown in Equation 2.8, and 2.9.

$$W := W - \eta.\nabla W \tag{2.8}$$

$$b := b - \eta . \nabla b \tag{2.9}$$

where η is the learning rate.
The optimization algorithm that was used in the project for CNN training is Adam optimizer. It is an advanced optimization algorithm that combines the benefits of two other extensions of SGD: AdaGrad and RMSProp. It computes adaptive learning rates for each parameter. Adam is characterized by its adaptive learning rates and bias correction. It adjusts the learning rates for each parameter based on the estimates of the first and second moments of the gradients, making it particularly well-suited for problems with sparse gradients. Additionally, the bias correction terms ensure that the estimates $\hat{m}^t$ and $\hat{v}^t$ are unbiased, which is especially beneficial during the initial steps of training. This combination allows Adam to provide faster and more stable convergence compared to other optimization algorithms.

Mini-batch training and epochs are important ideas that enhance the efficiency and effectiveness of training neural networks. During mini-batch training, both the forward and backward propagation tasks are carried out for every mini-batch of data, resulting in quicker and more frequent adjustments to the model's parameters compared to analyzing the entire dataset simultaneously. This approach offers a trade-off between the computational speed of handling numerous samples simultaneously and the strength of incorporating some diversity in the training process, akin to stochastic training. Epochs are defined as the act of feeding the complete training dataset into the network repeatedly. This step-by-step method guarantees that the model gets multiple chances to adjust its weights and biases, gradually enhancing its performance. Epochs contribute to the network's understanding of the training data by aiding in a deeper grasp of patterns and features, resulting in a more precise and dependable model.

3.6 Software Implementation of CNN

The CNN was implemented in Python using the PyTorch library. A *CharacterDataset* class was defined to initialize the dataset by reading image paths from the root directory, storing them, and assigning labels to each class (corresponding to each folder). Each image from the dataset and its corresponding label was retrieved by index, after which the images were loaded and preprocessed. The dataset was then split into training and testing sets, with 80% of the data allocated for training and 20% for testing. A DataLoaders were created for both the training and testing sets, batching images in groups of 600 and shuffling the training data. Shuffling the data is a beneficial technique in CNN training because if the data is fed into the network class by class, the network is more likely to memorize the data rather than learn the underlying patterns.CNN class was defined, that consist of three convolutional layers, subsequent ReLU activations, and max-pooling layers. The first layer has 3 output channels, the second has 9, and the third has 18. Following the convolutional layers, the output is flattened and processed through three fully connected layers, decreasing the data to 100 units, then to 60 units, and ultimately generating an output of 26 units, representing the 26 character classes. The implementation of CNN in Python is shown in Figure 2.10.

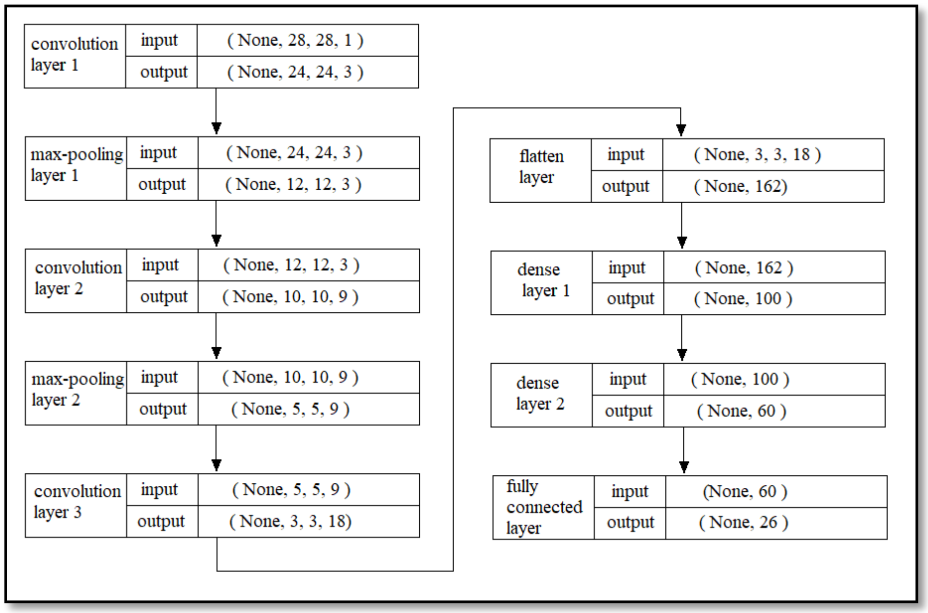
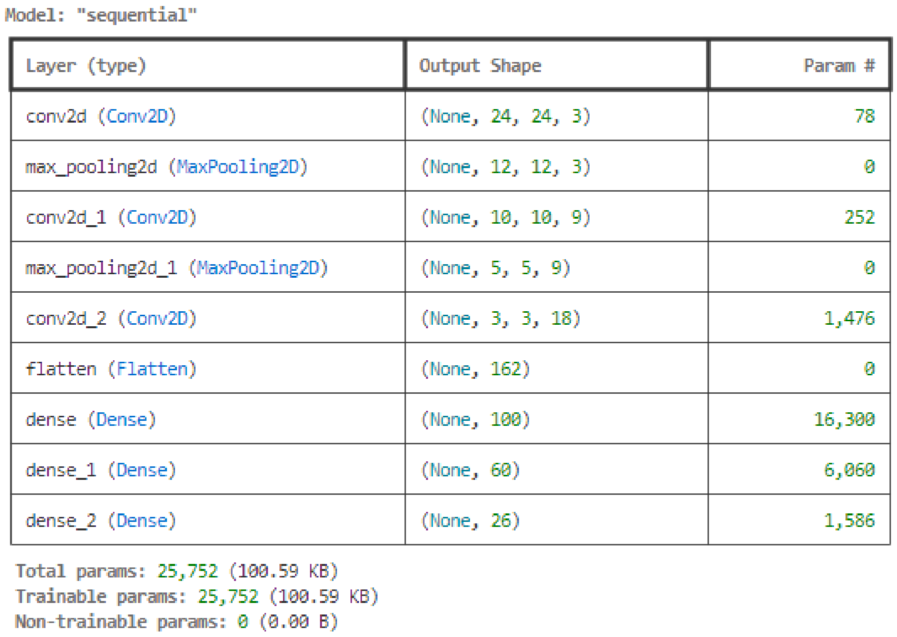


Figure 3.10 CNN Implementation in Python

The model is then initialized and displayed in the terminal using the *model.summary()* function, which provides a detailed overview of the model’s architecture, including the layers, output shapes, and the number of trainable parameters in each layer. The overview of the model as generated by the Python is shown in Figure 3.11.


```
Model: "sequential"
┌──────────────────────────────────┬─────────────────────────┬──────────────┐
│ Layer (type)                     │ Output Shape            │      Param # │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ conv2d (Conv2D)                  │ (None, 24, 24, 3)       │           78 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ max_pooling2d (MaxPooling2D)     │ (None, 12, 12, 3)       │            0 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ conv2d_1 (Conv2D)                │ (None, 10, 10, 9)       │          252 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ max_pooling2d_1 (MaxPooling2D)   │ (None, 5, 5, 9)         │            0 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ conv2d_2 (Conv2D)                │ (None, 3, 3, 18)        │        1,476 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ flatten (Flatten)                │ (None, 162)             │            0 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ dense (Dense)                    │ (None, 100)             │       16,300 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ dense_1 (Dense)                  │ (None, 60)              │        6,060 │
├──────────────────────────────────┼─────────────────────────┼──────────────┤
│ dense_2 (Dense)                  │ (None, 26)              │        1,586 │
└──────────────────────────────────┴─────────────────────────┴──────────────┘
 Total params: 25,752 (100.59 KB)
 Trainable params: 25,752 (100.59 KB)
 Non-trainable params: 0 (0.00 B)
```
Figure 3.11 Model Summary in Python

The loss function was defined as CrossEntropyLoss, and the Adam optimizer was used for training with a learning rate of 0.001. The training process runs for 100 epochs, during which the model iterates over the training dataset, performing forward and backward passes to optimize the model parameters. After each epoch, the loss and accuracy were calculated. The test function evaluates the trained model on the test dataset, and when the desired accuracy was achieved, the final trained model was saved to a file. The code also stored the model's parameters, which consist of the weights and biases for each layer, in text files. These weights and biases are essential for determining the behavior of the convolutional and fully connected layers, within the network, the pooling layers do not have any parameters. During training, the network adjusts weights to decrease loss, and biases help move the activation function to better fit the data. These stored parameters were next utilized in FPGA implementation of CNN.

3.7 Hardware Implementation of CNN

NIOS II processor combined with a hardware accelerator was selected to execute the CNN in hardware. This combination enables the CNN to run efficiently by transferring the computationally heavy duties to the accelerator, with the NIOS processor managing control and coordination. This arrangement combines the advantages of the processor and FPGA, allowing for strong performance of CNN in hardware. Figure 3.12 provides comprehensive visual representation of the architecture and components that forms the system.

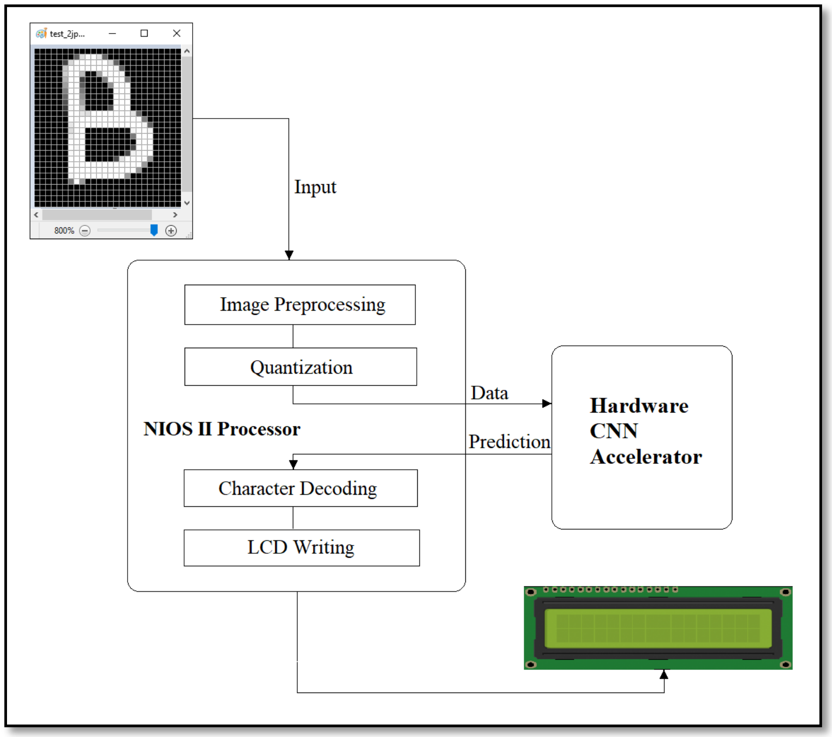
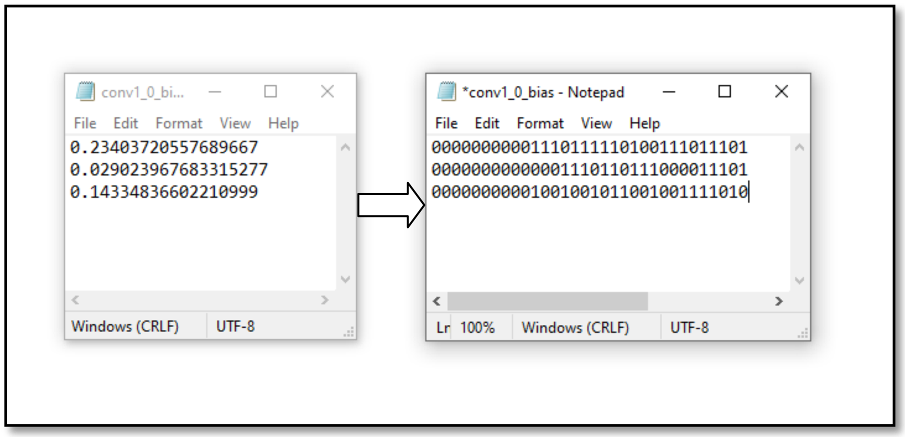


Figure 3.12 System Model Overview

3.7.1 Pixel Conversion to Fixed-Point Format

In the hardware implementation, pixel values were changed to a specified fixed-point format, Q8.24, to improve computational efficiency and accuracy in processing. Each pixel value was represented in the Q8.24 format as a 32-bit number, with 8 bits dedicated to the integer number part with the most significant bit for sign, and 24 bits dedicated to the fraction part. This transformation enables accurate mathematical calculations on the pixel values without increasing memory usage. By utilizing this structure, speedy and precise processing of pixel data was guaranteed, particularly in settings where floating-point calculations are expensive or not possible. The conversion of the pixels from floating to fixed point format is known as quantization.

Also the parameters extracted from trained model (weights and biases) were converted into fixed-point format, the conversion ensured that the parameters, initially in floating-point format, could be accurately represented in the memory of the FPGA-based accelerator. The conversion was done by Python script, the converted biases of first convolutional layer are represented in Figure 3.13, from the architecture of CNN is clear that first layer have only 3 biases because it has 3 output channels.


conv1_0_bi...
File Edit Format View Help
Windows (CRLF) UTF-8
*conv1_0_bias - Notepad
File Edit Format View Help
Ln 100% Windows (CRLF) UTF-8

```
0.23403720557689667
0.029023967683315277
0.14334836602210999
```

```
00000000001110111110100111011101
00000000000001110110111000011101
00000000001001001011001001111010
```

Figure 3.13 Quantization of CNN Biases

The algorithm used for Quantization was

1. Number scaling to fixed point representation, by multiply the floating point number by 2^F, where F is the fractional length of the variable, in the design is 24.
2. Round the scaled value to the nearest integer to handle any fractional part.
3. Clamp the rounded value to fixed-point range (32-bit).

3.7.2 Modules of Hardware Accelerator

Separate module was created in SystemVerilog Hardware Description Language (HDL) for each layer of the neural network for the CNN accelerator. This modular design method made it easier to handle the intricate calculations needed for convolution, pooling, and fully connected layers.

The modular representation improved the design and implementation process, making debugging and optimization of each layer easier. All individual models were connected by top-level module, which managed the overall execution sequence, ensuring that data transitioned smoothly from one layer to the next, and synchronized operations to maintain the integrity of the computation. The top module have several inputs from the NIOS II processor and outputs that are send to the processor, the block diagram of top module is represented in Figure 3.14.

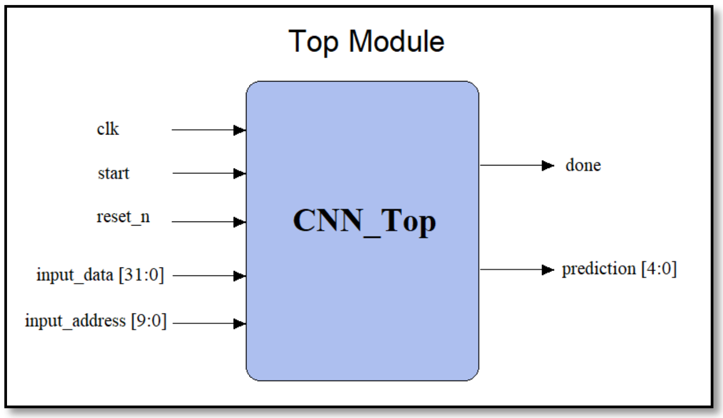


Figure 3.14 Block Diagram of Top Module

In the implemented design, each convolutional layer is connected to two Read-Only Memory (ROM) modules: one storing the weights and the other storing the biases. Each ROM is implemented as a lookup table where the memory array serves as the table, and the address input is used to select which data value to output. The only exception to this setup is the pooling layers, which do not require any weights or biases. Additionally, the design includes three Random Access Memory (RAM) modules. One RAM module is dedicated to storing the input image pixels received from the NIOS processor, which consists of 784 pixels. The other two RAM modules are reused throughout the network to store the output from each layer temporarily before passing it on to the next layer. After the image data is processed through the network, the final output prediction is generated. This prediction, represented as a number, is then sent back to the NIOS processor for further use.

State machine is crucial in the design, for managing the flow of data and ensuring that the entire neural network operates synchronously and efficiently. It guarantees that each step is executed in the correct sequence, preventing any data collisions or mismanagement of resources. The state machine that was implemented in the *CNN_Top* module is shown in Figure 3.15.

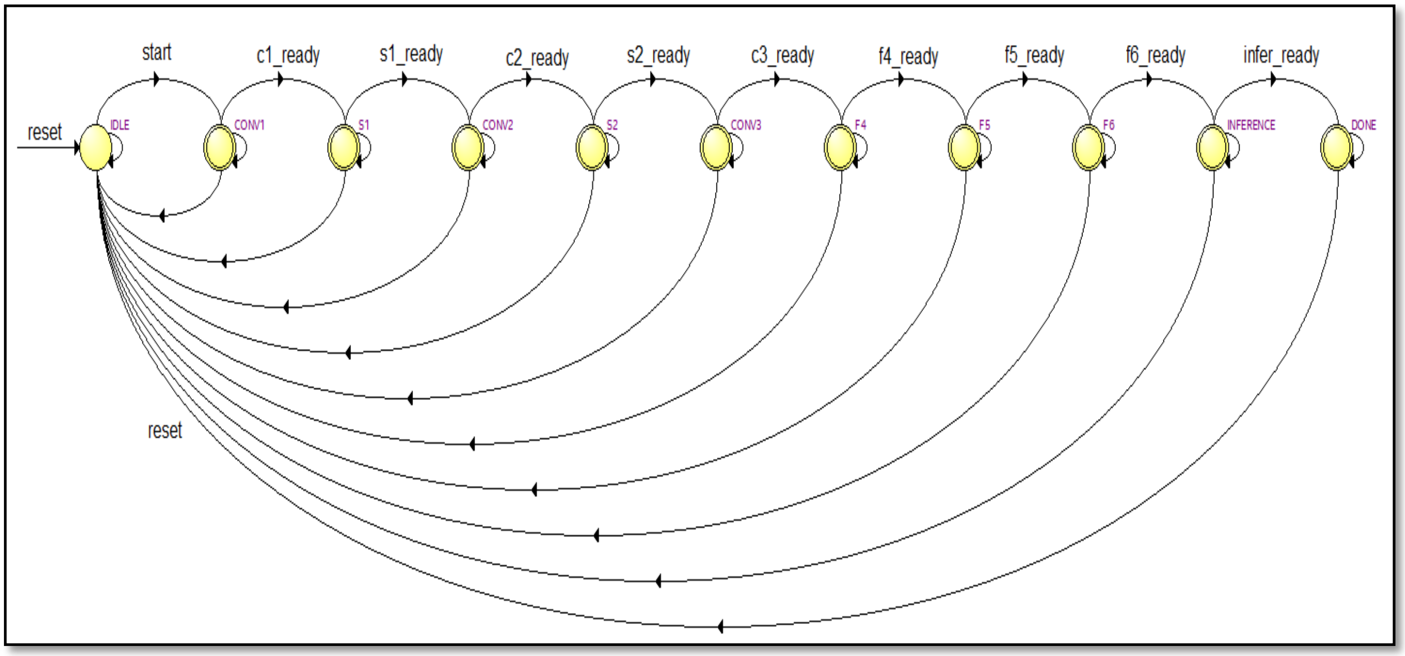


Figure 3.15 CNN_Top Module State Machine

3.7.2.1 Convolution Module

The *convolution_layer* modules for all three convolutional layers were implemented in a very similar manner, with the main differences being the size and number of filters, as well as the number of input and output channels. Each module features a state machine that manages the loading of input data, weights, and biases from the memory modules. Additionally, the modules perform the fixed-point multiplication and addition operations required for the convolution process,each module also incorporates a ReLU activation function to introduce non-linearity into the model.

The operation of each convolution_layer module is controlled by a state machine that follows a precise sequence:

- The module begins processing only when it receives a start signal.
- During processing, the convolution results are passed through the ReLU activation function, ensuring that only positive values are propagated forward.
- After completing the computations, the module sends the processed output data to the memory unit.
- Once all outputs have been sent, the module then issues a finish signal to indicate that all operations are complete.

This sequence ensures that the convolutional layers operate efficiently, with outputs being properly stored after applying the ReLU function and before the completion of each process. The state machine that control this flow is illustrated in Figure 3.16.

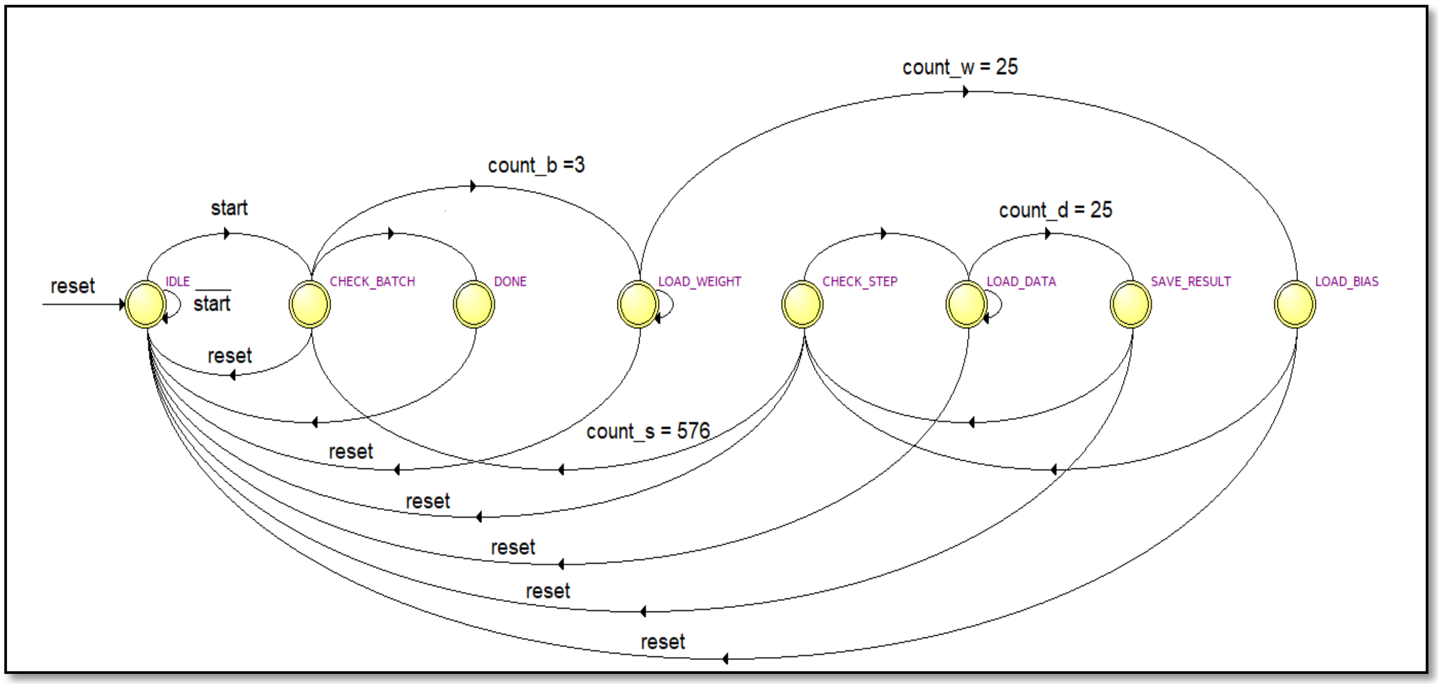


Figure 3.16 State Machine of Convolutional Layer Module

Where *count_b* tracks the number of channels being processed, *count_s* counts the number of output pixels generated, *count_w* counts the number of weights within the kernel, and *count_ d* tracks the range of input data being used for multiplication with the kernel.
Module *kernel_mult* is responsible for multiplication and addition of the kernel and input feature map, and also for the application of ReLU activation function.
In the implemented design, the input image is processed by dividing it into smaller overlapping regions, also known as frames, Each of these frames is then multiplied with the convolutional kernel to produce a feature map. The input image is divided into frames using a sliding window technique. This window moves across the image in small steps, equal to the stride length, ensuring that each frame overlaps with its neighboring frames. The size of the window matches the dimensions of the kernel.

3.7.2.2 Pooling Module

The pooling module is designed to reduce the spatial dimensions of the input while maintaining the most important features. The module takes input data and processes it in 2×2 blocks, where the maximum value of the four pixels is selected. The module's internal state machine manages the flow of data through several states, including loading the input data, performing the max pooling operation, and storing the resulting pooled value into memory. The state machine also handles the address calculations for both input and output data locations. The module operates in batches, iterating through each step and batch until all the data is processed, after that it indicating the end of the task by making the ready signal active high.

3.7.2.3 Fully Connected Module

The design of hardware accelerator contain 3 fully connected modules, the first two implement identical algorithm, the only difference between them is the number of input data and neurons (output data). They are designed to implement a fully connected layer in a neural network, where each neuron in the layer is connected to all neurons in the previous layer. This module processes input data by multiplying it with corresponding weights and then accumulating the results for each neuron. The computation includes adding a bias term and applying a ReLU activation function to ensure non-linearity. The state machine controls the sequence of operations, including loading data, weights, and biases, performing multiplication, accumulation, and applying ReLU, which is very similar to the convolution module. The module outputs the final computed value for each neuron and store it in memory.
Last fully connect module differs from the previous in several key ways, reflecting its role as the final output layer of the CNN. This module does not include a ReLU activation function, as it directly produces the final output values that will be used for prediction. The accumulator sums the products of input data and weights across all connections for each neuron, with the bias added before the final result is stored. Once the computations for all neurons are complete, the output values are stored in memory, and the module signals that the CNN processing is finished. This streamlined design is optimized for output generation, focusing on accurately producing the final prediction results.

3.7.2.4 Max_Value Module

The Max_Value module is designed to identify the index of the maximum value within a set of 26 values, which is are the output values from last fully connected layer. The module sequentially compares each input value to the current maximum value stored in the temp register. The state machine controls the flow of operations through several states: it starts in the IDLE state, then proceeds to CHECK_STEP, where it checks if all values have been compared. In the COMPARE state, it updates the temp register and the max index if the current value is greater than the stored maximum. The state machine is shown in Figure 3.17.

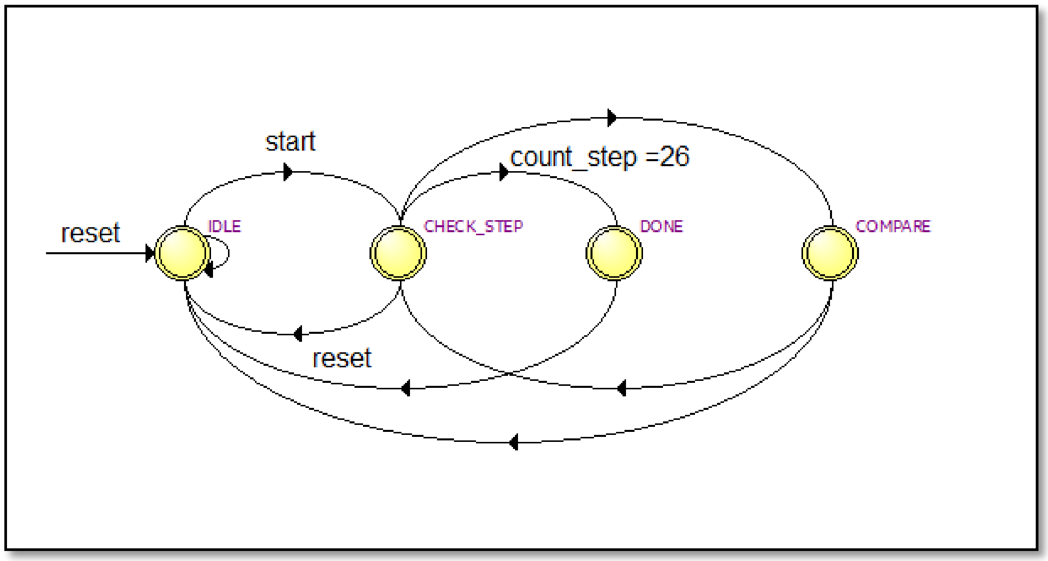


Figure 3.17 State Machine of Max_Value Module

Once all values have been processed, the module moves to the DONE state, at which point the done signal is asserted, and the index of the maximum valueis output. This module is crucial for determining the final prediction in a classification task, where the index of the highest value corresponds to the predicted letter class.

3.7.2.5 Interface Module

All the modules been packaged as custom IP core to create a hardware accelerator. Additionally, a top-level module has been implemented to interface between the hardware accelerator and the NIOS II processor, managing the communication and data exchange between them. This top-level module, which includes registers for data, address, write enable, and start signals, allows the processor to send input data to the CNN, control the computation process, and retrieve the final results. Specifically, the module includes logic for handling read and write operations from the NIOS II processor, using the processor's address, read, and write signals to control the flow of data. The CNN computation is initiated by the start signal, and upon completion, the output prediction (inference) and a done signal are sent back to the processor. Data transaction between processor(master) and accelerator(slave) adopts Advanced eXtensible Interface (AXI) protocol [13]. AXI protocol ensures high speed data transformation from point to point. Figure 3.18 shows the interface between NIOS and custom IP.

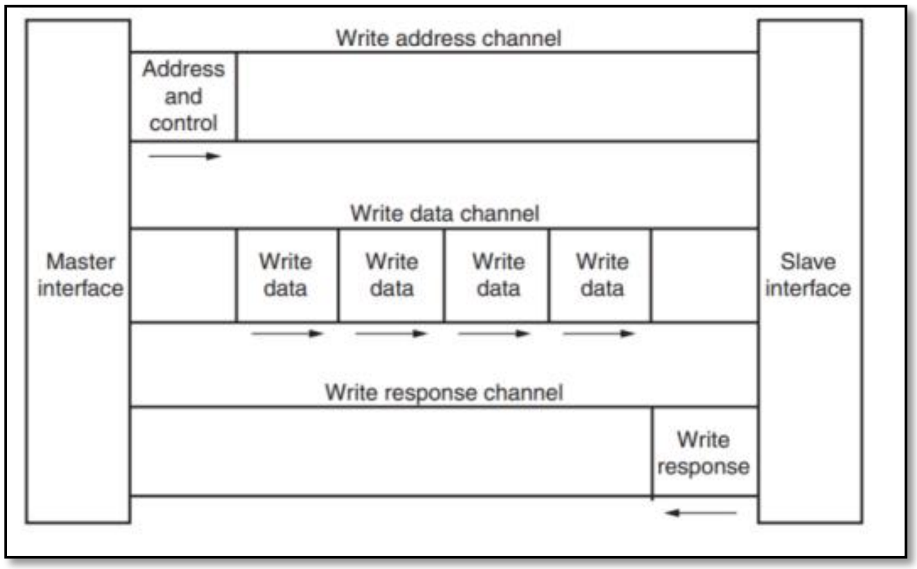


Figure 3.18 Interface between NIOS II and Accelerator

This design ensures efficient and accurate execution of CNN operations, leveraging the processing power of the custom IP cores while maintaining easy control via the NIOS II processor.

Custom IP core

IP core is a specialized digital design block created to carry out specific tasks in an FPGA or an ASIC. Custom IP cores are specially designed to fit the specific needs of a particular project or application, as opposed to standard IP cores that are readily available for general purposes. These cores are configurable to execute a variety of tasks. Developing a customized IP core consists of defining its functionality, coding it in hardware description languages such as Verilog or VHDL, and integrating it into the larger system with tools like Intel's Platform Designer. Tailored IP cores allow for optimized, hardware solutions designed for specific applications, providing the ability to boost performance, lower energy usage, and fully utilize the potential of the FPGA or ASIC they are integrated into.

3.7.3 Integration of CNN Accelerator with NIOS II Processor

After completing the design and packaging of the CNN hardware accelerator as a custom IP core, Platform Designer in the Quartus Prime software was utilized to incorporate this IP into a complete system[14]. The custom accelerator was included with important elements like the NIOS II processor, internal clock, UART for serial communication, a timer, on-chip RAM memory, LCD controller, and performance counter. This fusion developed a fully operational system that efficiently employs the hardware accelerator for performing high-performance CNN computations under the supervision of the NIOS II processor. The architecture of NIOS II processor is represented in Figure 3.19 [11].

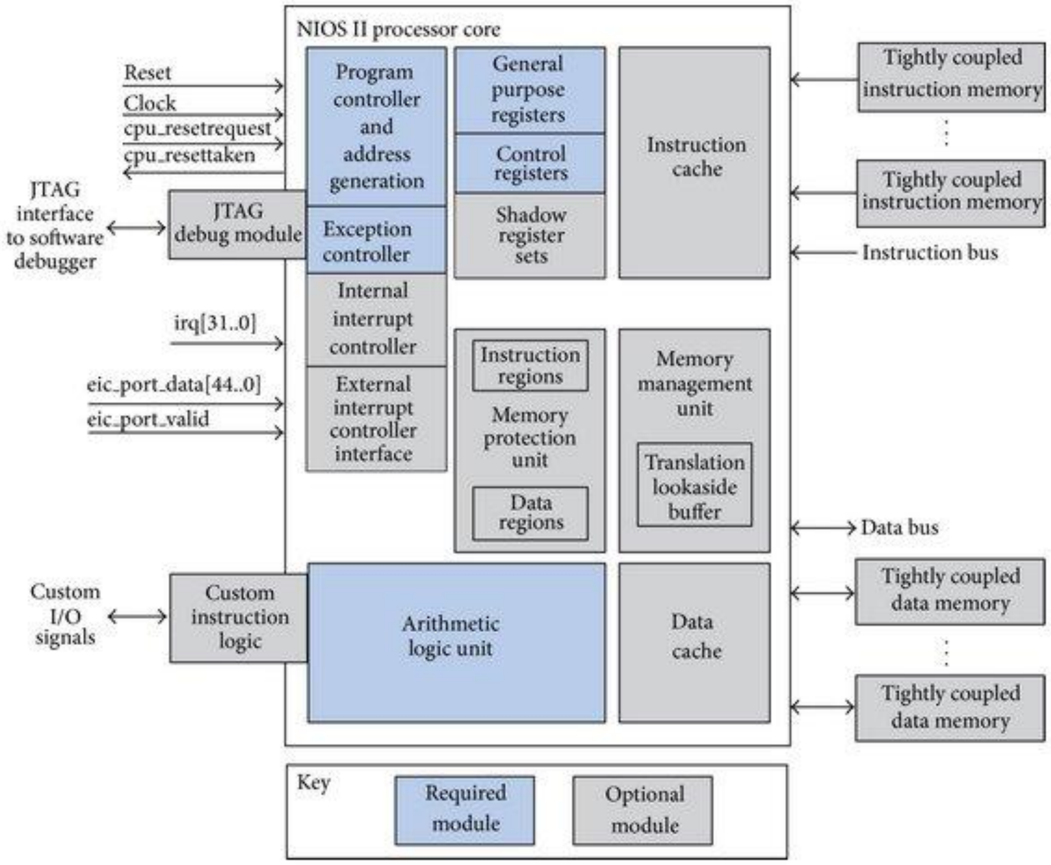
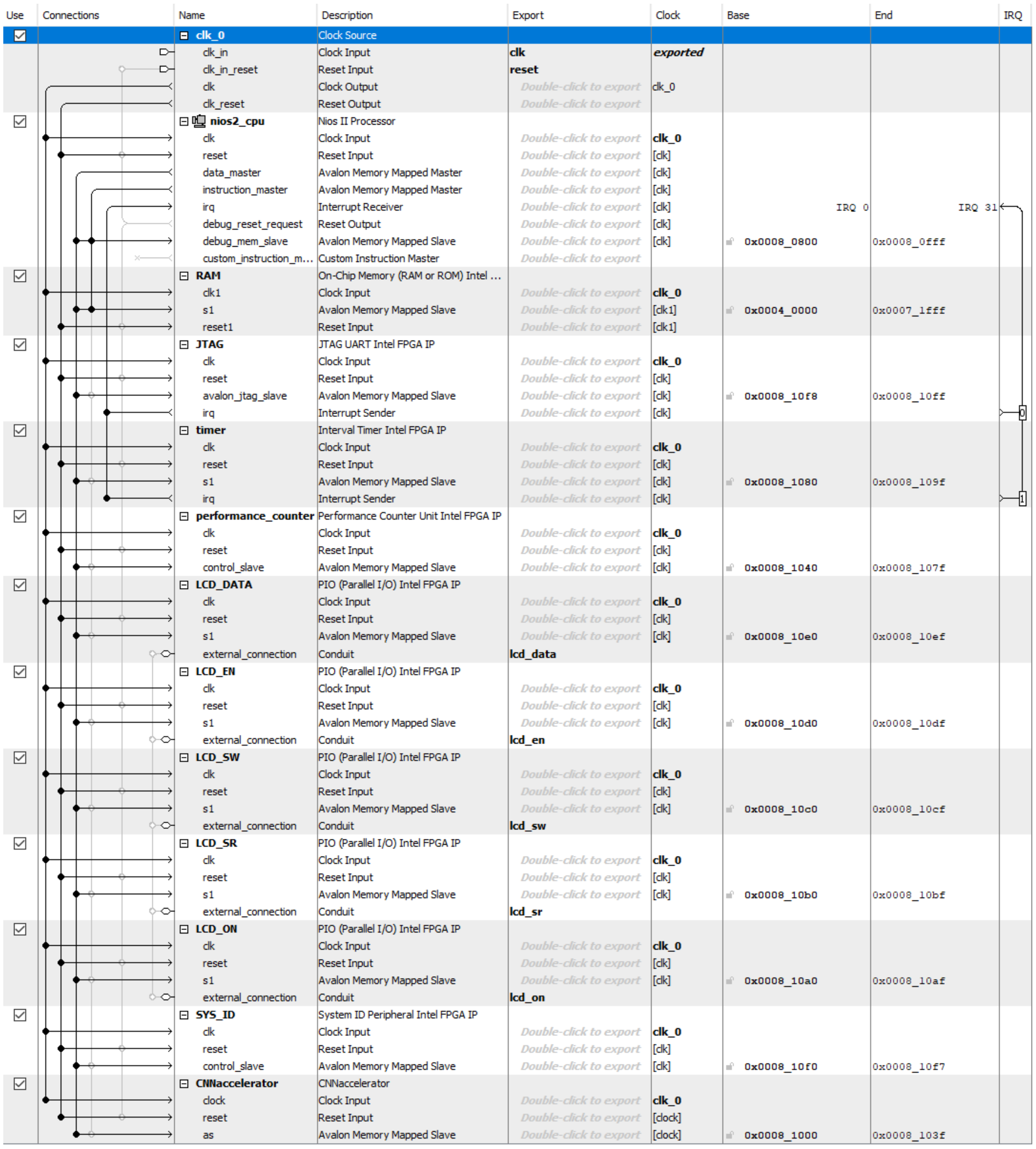


Figure 3.19 NIOS II Processor Architecture

Utilizing Platform Designer made it easy to connect and configure components, guaranteeing smooth data flow and system control. Figure 3.20 shows complete system with all connections, base address for each IP, and the interrupts.



Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
☑		clk_0	Clock Source					
		clk_in	Clock Input	clk	exported			
		clk_in_reset	Reset Input	reset				
		clk	Clock Output	Double-click to export	clk_0			
		clk_reset	Reset Output	Double-click to export				
☑		nios2_cpu	Nios II Processor					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		data_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		irq	Interrupt Receiver	Double-click to export	[clk]	IRQ 0	IRQ 31	
		debug_reset_request	Reset Output	Double-click to export	[clk]			
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0008_0800	0x0008_0fff	
		custom_instruction_m...	Custom Instruction Master	Double-click to export				
☑		RAM	On-Chip Memory (RAM or ROM) Intel ...					
		clk1	Clock Input	Double-click to export	clk_0			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk1]	0x0004_0000	0x0007_1fff	
		reset1	Reset Input	Double-click to export	[clk1]			
☑		JTAG	JTAG UART Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0008_10f8	0x0008_10ff	
		irq	Interrupt Sender	Double-click to export	[clk]			0
☑		timer	Interval Timer Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0008_1080	0x0008_109f	
		irq	Interrupt Sender	Double-click to export	[clk]			1
☑		performance_counter	Performance Counter Unit Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		control_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0008_1040	0x0008_107f	
☑		LCD_DATA	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0008_10e0	0x0008_10ef	
		external_connection	Conduit	lcd_data				
☑		LCD_EN	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0008_10d0	0x0008_10df	
		external_connection	Conduit	lcd_en				
☑		LCD_SW	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0008_10c0	0x0008_10cf	
		external_connection	Conduit	lcd_sw				
☑		LCD_SR	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0008_10b0	0x0008_10bf	
		external_connection	Conduit	lcd_sr				
☑		LCD_ON	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0008_10a0	0x0008_10af	
		external_connection	Conduit	lcd_on				
☑		SYS_ID	System ID Peripheral Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		control_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0008_10f0	0x0008_10f7	
☑		CNNaccelerator	CNNaccelerator					
		clock	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clock]			
		as	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0008_1000	0x0008_103f	

Figure 3.20 View of the System in Platform Designer

One of the key advantages of using Platform Designer is its ability to allow users to customize on-chip RAM to meet specific project requirements. In this project, the flexibility of Platform Designer enabled the configuration of on-chip RAM tailored to the needs of the CNN accelerator. The RAM size was customized to 40KB, ensuring sufficient memory for efficient data storage and processing while optimizing resource usage within the FPGA. This level of

customization is crucial for achieving a balanced design that meets both performance and resource constraints.
Once the system design is successfully completed, the next important task is to perform the pin assignment for the FPGA. This stage is essential to ensure proper and effective interaction among hardware components after downloading the system into FPGA. The pins are assigned due to the board manual [15].
After the design is downloaded into FPGA, the subsequent task involves coding the NIOS processor using C++ language. This programming is essential for managing all connected peripherals to the FPGA. The software guarantees that the FPGA and its accompanying components cooperate seamlessly to achieve the desired goals.
The main functions of the NIOS II processor are to initializes the LCD, processes an image, sends it to the CNN accelerator, waits for inference to complete, and displays the result on the LCD. The program starts by initializing the LCD through a series of commands to set its mode and ensure it's ready for operation. It then preprocesses an image by normalizing pixel values and quantize them, which is required for the CNN accelerator. After resetting the CNN accelerator to ensure it’s in a known state, the program sends the image data to the accelerator. It then starts the CNN processing and waits for the processing to complete by polling a status register. Once the CNN has finished, it reads the inference result, converts it to a character, and displays the result on the LCD. Additionally, performance counters are used to measure the execution time of the CNN processing.

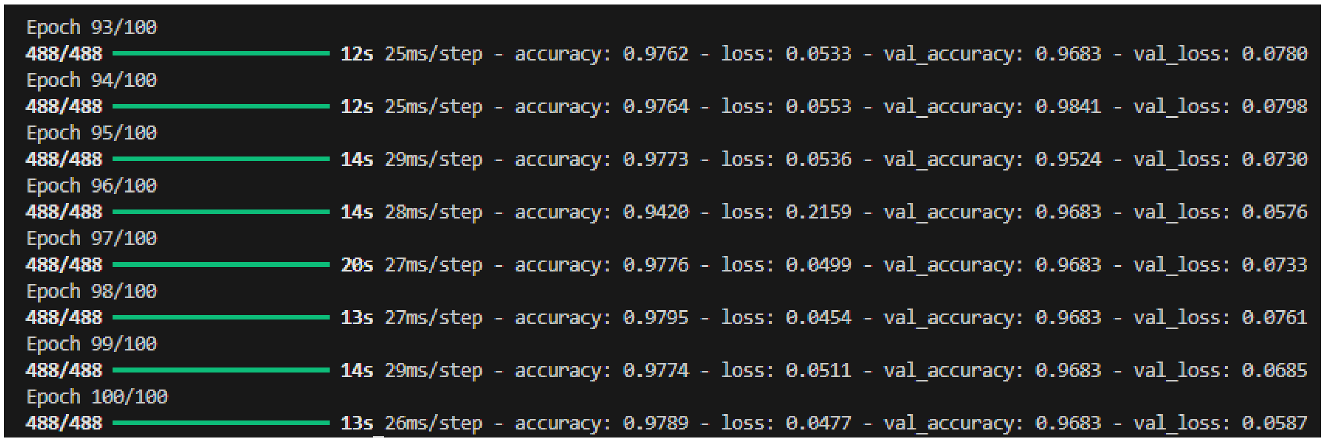
This chapter described a multi-stage approach to implementing and deploying a CNN using both software and hardware components. Initially, the CNN was designed and trained in Python using PyTorch framework. During training, the network learns from labeled data, adjusting its weights and biases to optimize performance. This trained model was then tested on a handwritten letters to evaluate its accuracy and generalization capabilities. Following this, the chapter transitions to hardware implementation. The trained CNN's weights and biases were integrated into a hardware design that includes a NIOS II processor and a custom FPGA-based CNN accelerator. The NIOS processor is programmed to manage data transfer between the accelerator and system components, while the accelerator performs CNN computations efficiently in hardware. By combining Python-based model training with FPGA hardware acceleration, this approach achieves a balance of flexibility, ease of development, and high-performance real-time inference.

Results and Discussion

This chapter demonstrate and examine the results of executing and utilizing the CNN on software and hardware platforms. The results and discussion are organized to offer a complete summary of the system's performance, encompassing the CNN model trained in Python, its hardware acceleration on the FPGA, and its integration with the NIOS processor. It starts by discussing the CNN's performance metrics and accuracy as assessed during testing, then evaluate the operational efficiency of the hardware accelerator. The conversation will focus on important discoveries, comparing software-based and hardware-accelerated implementations, and investigating the impact of these results on practical applications.

4.1Python Implementation and Results

The designed CNN was implemented in Python using the Visual Studio Code platform. After normalizing the dataset and splitting it into training and testing sets, the model was trained and evaluated. The highest level of accuracy was reached by making several changes to the CNN structure and adjusting different training parameters, such as the number of epochs, type of optimizer, and learning rate. Making iterative changes was crucial to improve the model's performance and reach maximum accuracy. Figure 4.1 show the accuracy that was achieved for training with Adam analyzer, with 100 epochs, and learning rate equal to 0.001.



```
Epoch 93/100
488/488 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9762 - loss: 0.0533 - val_accuracy: 0.9683 - val_loss: 0.0780
Epoch 94/100
488/488 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9764 - loss: 0.0553 - val_accuracy: 0.9841 - val_loss: 0.0798
Epoch 95/100
488/488 ━━━━━━━━━━━━━━━━━━━━ 14s 29ms/step - accuracy: 0.9773 - loss: 0.0536 - val_accuracy: 0.9524 - val_loss: 0.0730
Epoch 96/100
488/488 ━━━━━━━━━━━━━━━━━━━━ 14s 28ms/step - accuracy: 0.9420 - loss: 0.2159 - val_accuracy: 0.9683 - val_loss: 0.0576
Epoch 97/100
488/488 ━━━━━━━━━━━━━━━━━━━━ 20s 27ms/step - accuracy: 0.9776 - loss: 0.0499 - val_accuracy: 0.9683 - val_loss: 0.0733
Epoch 98/100
488/488 ━━━━━━━━━━━━━━━━━━━━ 13s 27ms/step - accuracy: 0.9795 - loss: 0.0454 - val_accuracy: 0.9683 - val_loss: 0.0761
Epoch 99/100
488/488 ━━━━━━━━━━━━━━━━━━━━ 14s 29ms/step - accuracy: 0.9774 - loss: 0.0511 - val_accuracy: 0.9683 - val_loss: 0.0685
Epoch 100/100
488/488 ━━━━━━━━━━━━━━━━━━━━ 13s 26ms/step - accuracy: 0.9789 - loss: 0.0477 - val_accuracy: 0.9683 - val_loss: 0.0587
```

Figure 4.1 CNN Accuracy during training

Table 4.1 show the finale results of CNN training after 100 epochs. The closeness of accuracy on the training data and the validation data suggests that the model is not overfitting to the training set. It indicate that it is likely to perform well on truly unseen data. Also the small training loss means that the model is effectively learning from the training data and making accurate predictions on it. Small validation loss means that the model is also performing well on data it hasn't seen during training.

Table 4.1 Results of CNN training

Accuracy	97.89%
Validation Accuracy	96.83%
Loss	0.0477
Validation Loss	0.0587

4.1.1Tensorflow and Keras Libraries

TensorFlow is an open-source machine learning framework developed by Google. It provides a comprehensive ecosystem for building and deploying machine learning models, particularly deep learning models. TensorFlow supports a variety of neural network architectures and algorithms, making it versatile for tasks such as image recognition, natural language processing, and more. It offers flexibility and scalability, with capabilities for both research and production environments.
Keras is a high-level neural networks Application Programming Interface (API) that runs on top of TensorFlow. It provides an intuitive and user-friendly interface for defining neural network architectures, training models, and evaluating performance.
TensorFlow and Keras were used to enhance the model evaluation and visualization processes, by create the confusion matrices, and graphs including loss curves and accuracy plots. The crated confusion matrices is shown in Figure 4.2.

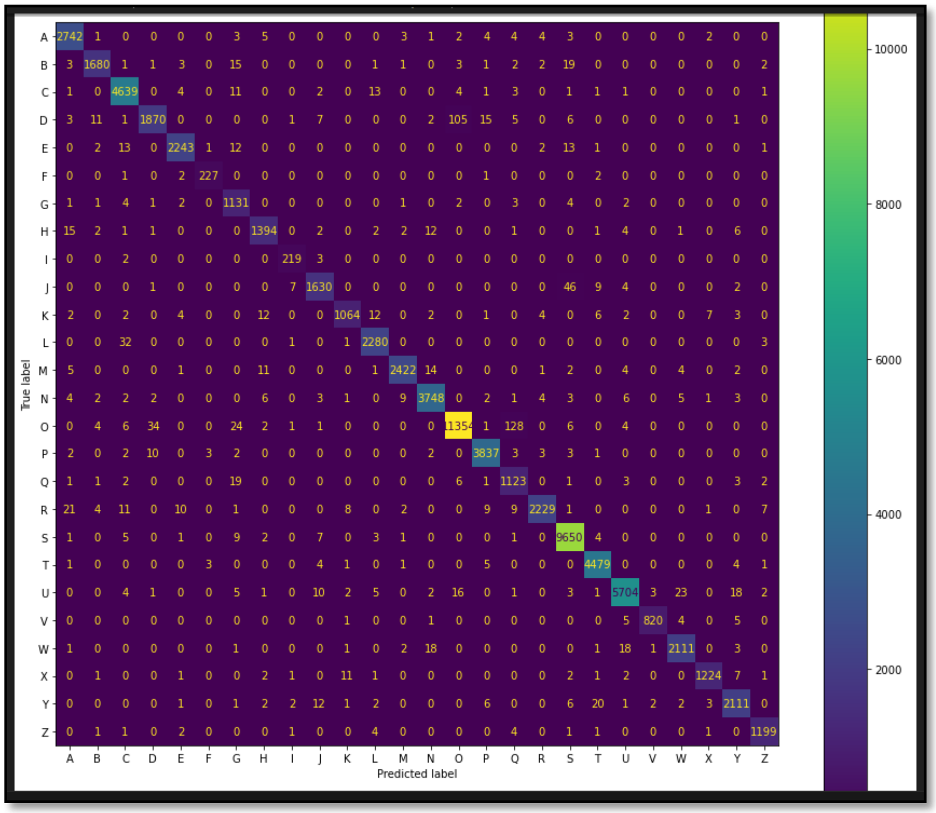


Figure 4.2 Confusion Matrices

Confusion matrices were instrumental in identifying areas where the model was making errors and guiding improvements. The confusion matrix for the final design, shown in Figure 3.2, revealed that the letter "D" was most frequently confused with the letter "O." Despite this, "D" was correctly predicted 94.68% of the time. On the other hand, the letter "O" was often confused with the letter "Q," but incorrect predictions were relatively infrequent.
In Figure 4.3 the loss and accuracy graphs are shown, accuracy graph show that both training and validation accuracy reach around 97% and stabilize after 20 epochs, indicating good generalization without overfitting. The loss curves decrease sharply early on, with training loss continuing to slightly decline while validation loss stabilizes, which indicate that the model is well-trained.

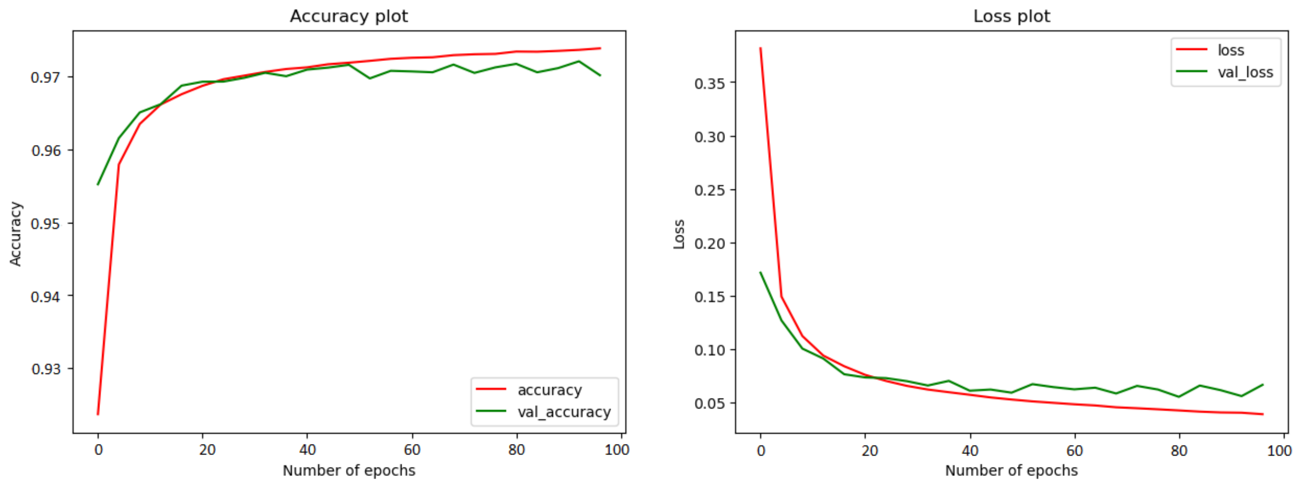
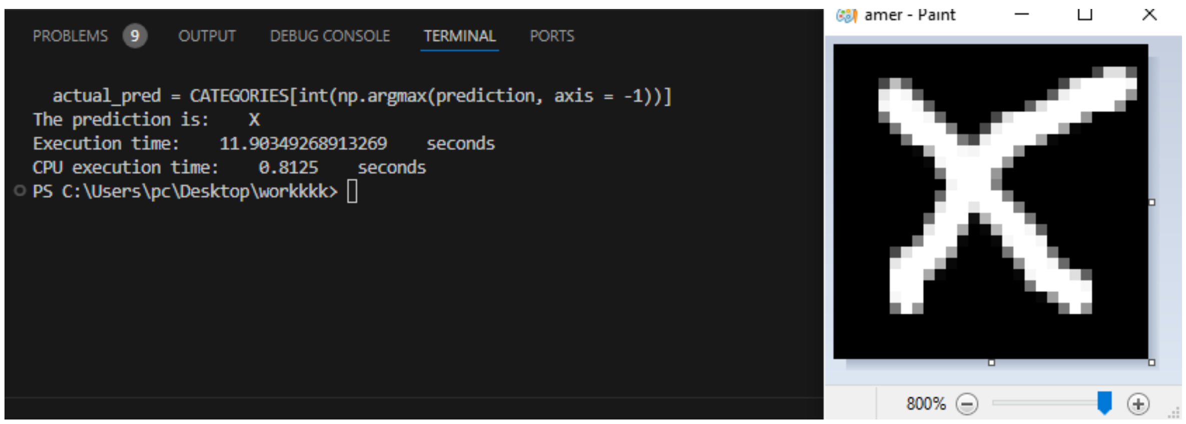


Figure 4.3 Accuracy and Loss Graphs

Another important factor in CNN training is the quality of the dataset. For effective training, a well-balanced dataset is crucial, meaning each class should have an equal number of samples. This balance ensures that the model doesn't become biased toward any particular class. The idea of implementing a CNN to recognize all characters (numbers, uppercase, and lowercase letters) was abandoned because the classes in EMNIST dataset were not balanced, leading to significantly lower accuracy. Additionally, the dataset should be sufficiently large to provide the model with enough examples to learn meaningful patterns and generalize well to new data.

4.1.2CNN Testing in Python

After successfully training the model to achieve satisfactory accuracy, it was tested on some handwritten letters. These letters were created using the Paint application and then imported into Visual Studio Code for testing by the CNN. Figure 4.4 illustrates the model's prediction for the letter "X". The code also provides the execution time and the CPU time for the prediction, which latter on is compared with the hardware accelerator timing.



```
  actual_pred = CATEGORIES[int(np.argmax(prediction, axis = -1))]
The prediction is:    X
Execution time:    11.90349268913269    seconds
CPU execution time:    0.8125    seconds
PS C:\Users\pc\Desktop\workkkk>
```

Figure 4.4 CNN Testing in Python

Several letters with several types of handwriting were tested and the CNN shows a very good performance.

4.2Hardware Accelerator Model Simulation

The hardware accelerator was written in SystemVerilog and simulated using ModelSim platform, which is a simulation tool used for verifying HDL designs, such as VHDL, Verilog, and SystemVerilog. It provides a powerful environment for debugging and validating digital circuits by allowing users to simulate and analyze waveforms, ensuring the correct functionality of designs before implementation.

The CNN accelerator model was extensively tested through simulation to ensure its functionality. During the testing, each layer of the CNN was individually verified to confirm it operates correctly. Figure 4.5 shows how the computation process begins when the start signal is asserted (active high). When the start signal is asserted, at the first positive edge of the signal, the system transitions from the IDLE state to the CONV1 state. This transition indicates that the computation for the first convolutional layer is starting.

Figure 4.6 shows the final stages of the computation. Once all layers have completed their operations, the system transitions from the INFERENCE state to the DONE state. At this point, the done signal is set high, indicating that the CNN has finished processing and the prediction result is now available.

Also the simulation shows that the prediction was correct, the image that was entered to the CNN accelerator was containing letter ‘E’, and the prediction was ‘00100’ which represents ’E’.

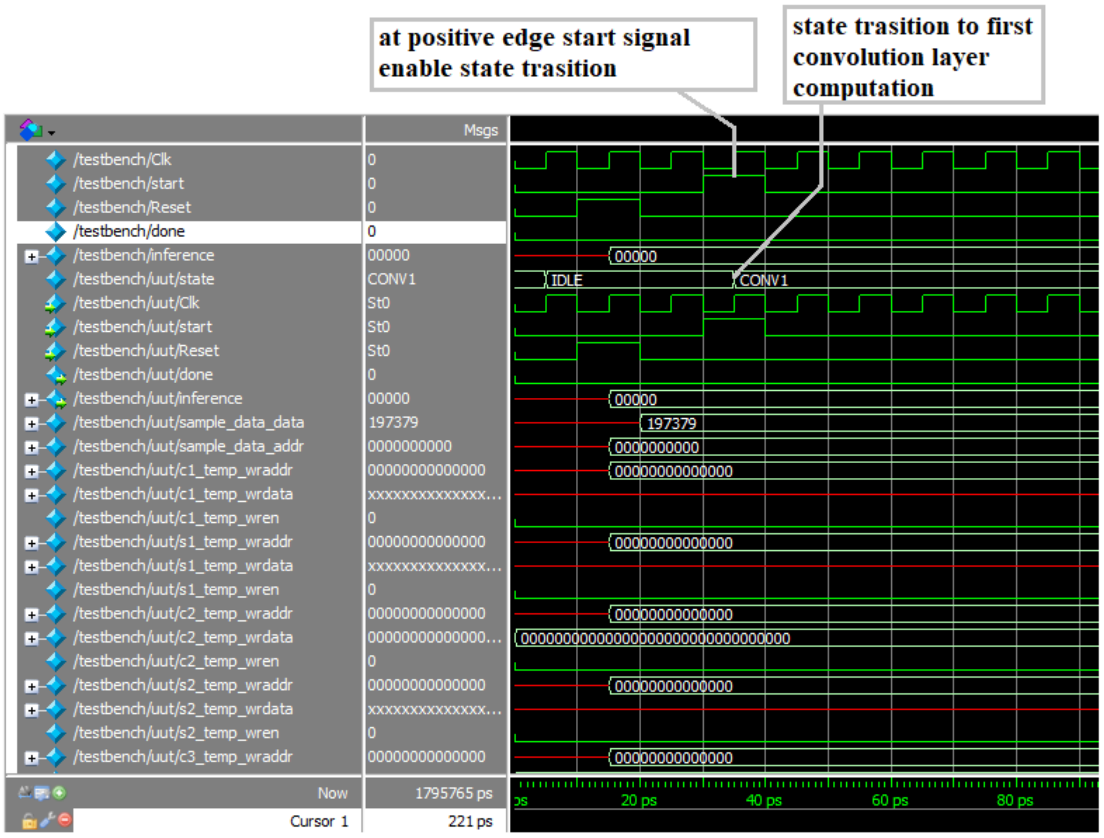

at positive edge start signal enable state trasition
state trasition to first convolution layer computation
/testbench/Clk
/testbench/start
/testbench/Reset
/testbench/done
/testbench/inference
/testbench/uut/state
/testbench/uut/Clk
/testbench/uut/start
/testbench/uut/Reset
/testbench/uut/done
/testbench/uut/inference
/testbench/uut/sample_data_data
/testbench/uut/sample_data_addr
/testbench/uut/c1_temp_wraddr
/testbench/uut/c1_temp_wrdata
/testbench/uut/c1_temp_wren
/testbench/uut/s1_temp_wraddr
/testbench/uut/s1_temp_wrdata
/testbench/uut/s1_temp_wren
/testbench/uut/c2_temp_wraddr
/testbench/uut/c2_temp_wrdata
/testbench/uut/c2_temp_wren
/testbench/uut/s2_temp_wraddr
/testbench/uut/s2_temp_wrdata
/testbench/uut/s2_temp_wren
/testbench/uut/c3_temp_wraddr
Msgs
IDLE
CONV1
Now 1795765 ps
Cursor 1 221 ps

Figure 4.5 Simulation Waveforms of Convolution Start

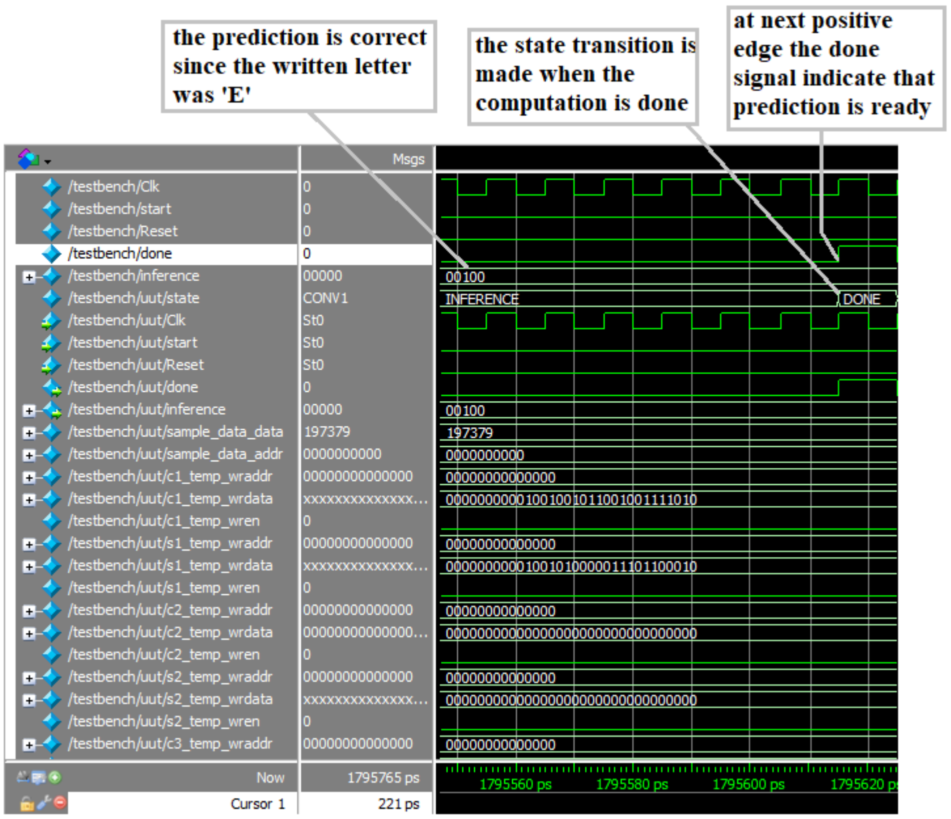

the prediction is correct since the written letter was 'E'
the state transition is made when the computation is done
at next positive edge the done signal indicate that prediction is ready
/testbench/Clk
/testbench/start
/testbench/Reset
/testbench/done
/testbench/inference
/testbench/uut/state
/testbench/uut/sample_data_data
/testbench/uut/c1_temp_wrdata
/testbench/uut/s1_temp_wrdata
/testbench/uut/c3_temp_wraddr
00100
INFERENCE
DONE
Now 1795765 ps
Cursor 1 221 ps

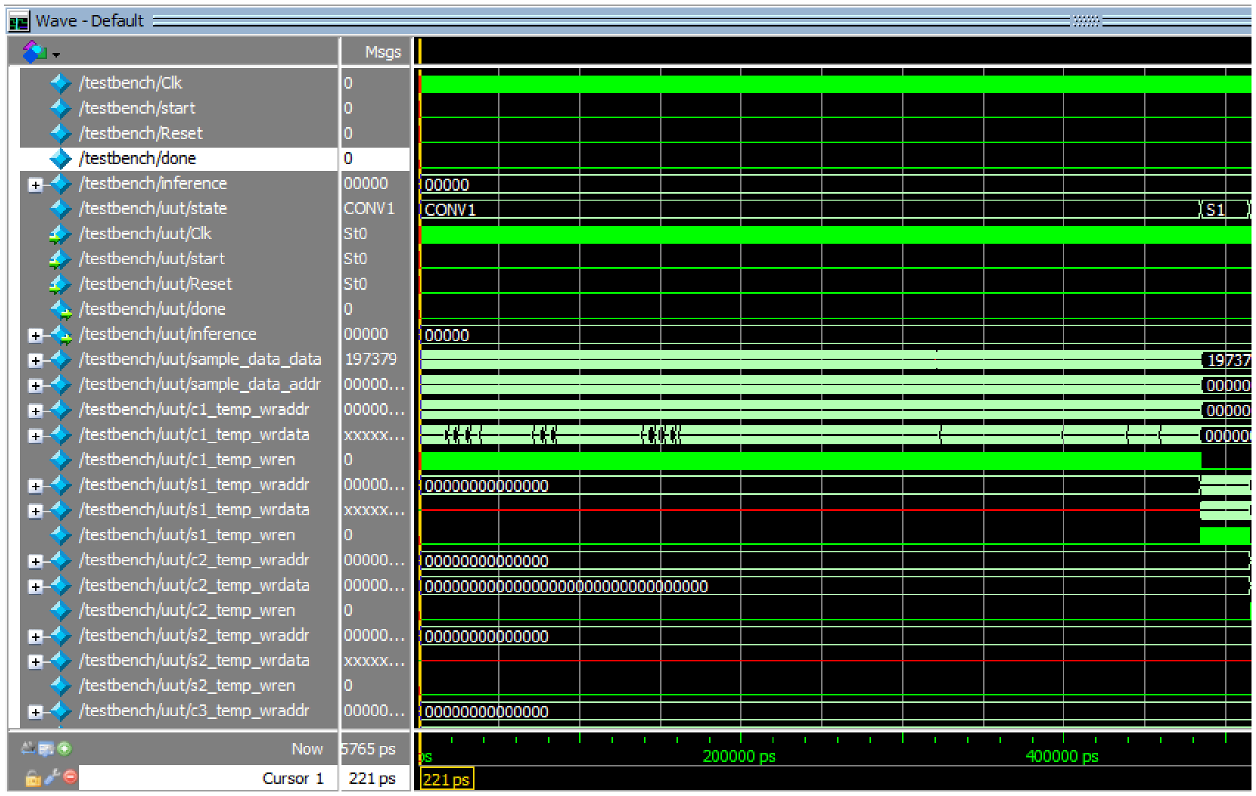
Figure 4.6 Simulation Waveforms of Prediction Output

Table 4.1 presents the output binary numbers and their corresponding letters, illustrating the mapping between the binary outputs of the CNN and the assigned letter labels.

Table 4.1 Mapping of Output Binary Numbers to Assigned Letters

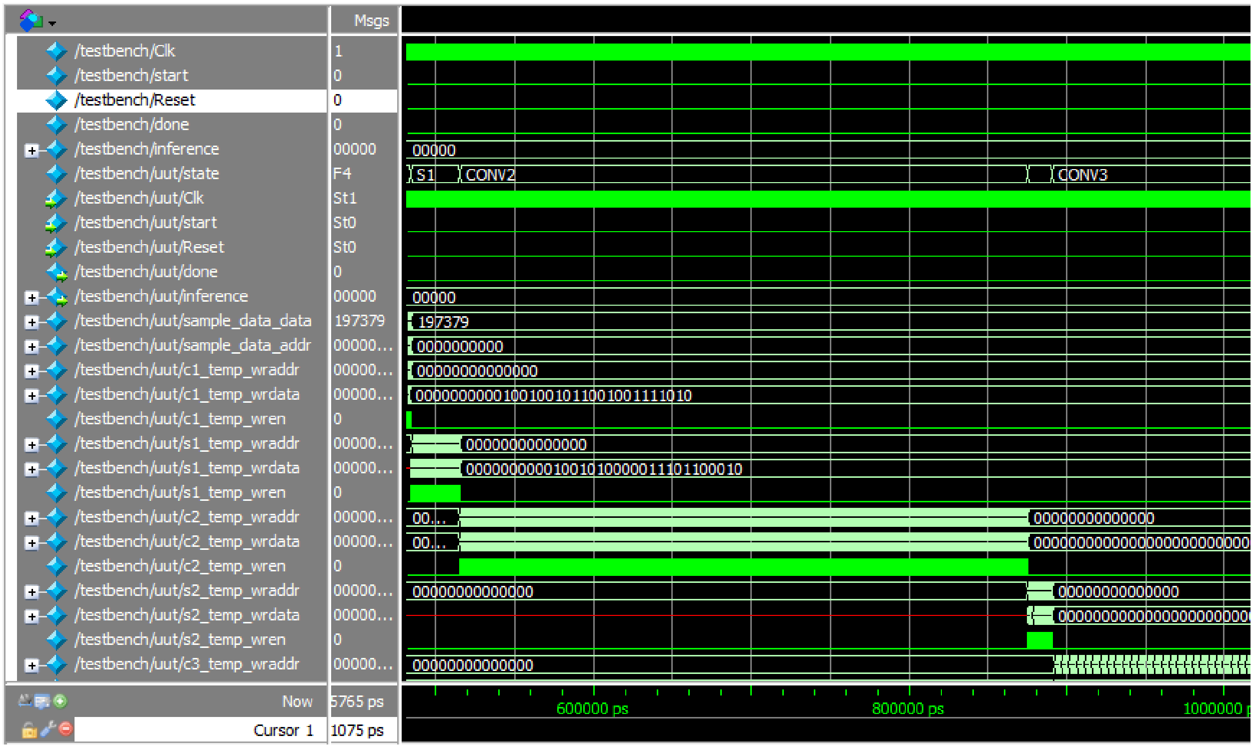
Binary output	Assigned letter
00000	A
00001	B
00010	C
00011	D
00100	E
00101	F
00110	G
00111	H
01000	I
01001	J
01010	K
01011	L
01100	M
01101	N
01110	O
01111	P
10000	Q
10001	R
10010	S
10011	T
10100	U
10101	V
10110	W
10111	X
11000	Y
11001	Z

The simulation waveforms clearly depict the amount of time each layer takes for computation, as illustrated in Figures 4.7, 4.8, 4.9. The convolution layers and the first fully connected layer were the most time-consuming, the other two fully connected layers were decreasing in time because the number of neurons was less. The pooling layers were comparatively less demanding in terms of computation time.



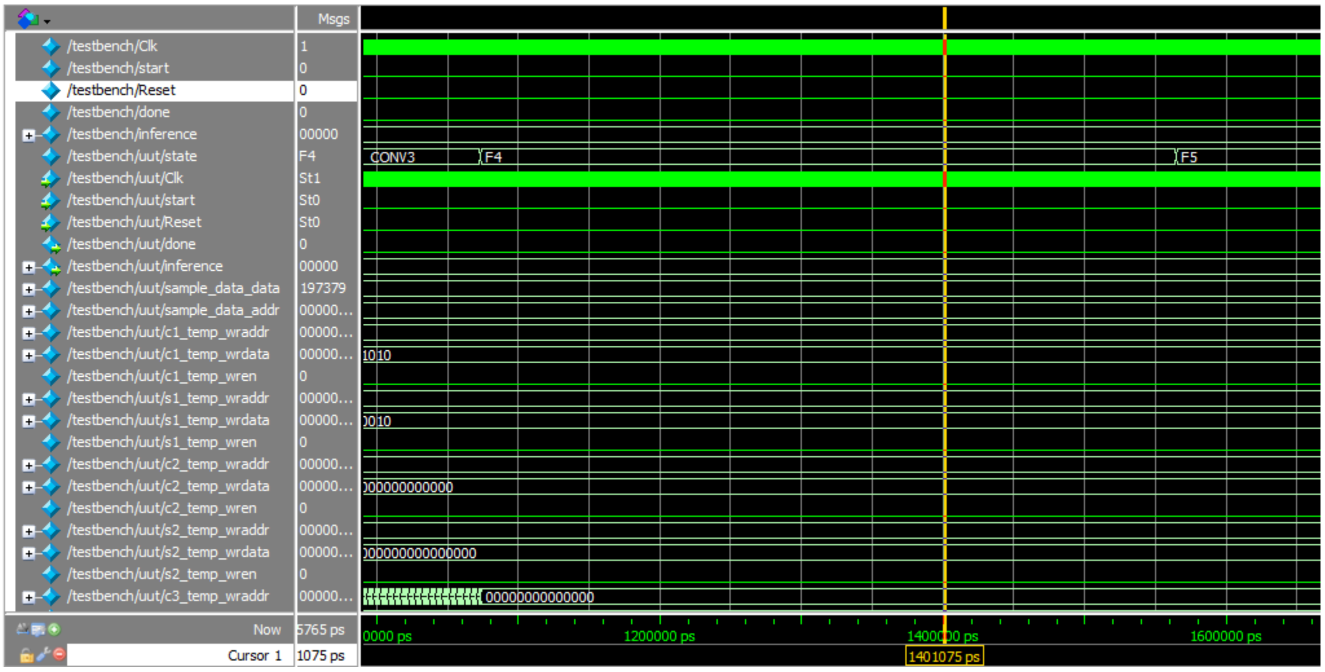
```
Wave - Default
                                    Msgs
/testbench/Clk                      0
/testbench/start                    0
/testbench/Reset                    0
/testbench/done                     0
/testbench/inference                00000
/testbench/uut/state                CONV1
/testbench/uut/Clk                  St0
/testbench/uut/start                St0
/testbench/uut/Reset                St0
/testbench/uut/done                 0
/testbench/uut/inference            00000
/testbench/uut/sample_data_data     197379
/testbench/uut/sample_data_addr     00000...
/testbench/uut/c1_temp_wraddr       00000...
/testbench/uut/c1_temp_wrdata       xxxxx...
/testbench/uut/c1_temp_wren         0
/testbench/uut/s1_temp_wraddr       00000...
/testbench/uut/s1_temp_wrdata       xxxxx...
/testbench/uut/s1_temp_wren         0
/testbench/uut/c2_temp_wraddr       00000...
/testbench/uut/c2_temp_wrdata       00000...
/testbench/uut/c2_temp_wren         0
/testbench/uut/s2_temp_wraddr       00000...
/testbench/uut/s2_temp_wrdata       xxxxx...
/testbench/uut/s2_temp_wren         0
/testbench/uut/c3_temp_wraddr       00000...
Now                                 5765 ps
Cursor 1                            221 ps
```

Figure 4.7 Simulation Waveforms of CONV1 to S1 transition



```
                                    Msgs
/testbench/Clk                      1
/testbench/start                    0
/testbench/Reset                    0
/testbench/done                     0
/testbench/inference                00000
/testbench/uut/state                F4
/testbench/uut/Clk                  St1
/testbench/uut/start                St0
/testbench/uut/Reset                St0
/testbench/uut/done                 0
/testbench/uut/inference            00000
/testbench/uut/sample_data_data     197379
/testbench/uut/sample_data_addr     00000...
/testbench/uut/c1_temp_wraddr       00000...
/testbench/uut/c1_temp_wrdata       00000...
/testbench/uut/c1_temp_wren         0
/testbench/uut/s1_temp_wraddr       00000...
/testbench/uut/s1_temp_wrdata       00000...
/testbench/uut/s1_temp_wren         0
/testbench/uut/c2_temp_wraddr       00000...
/testbench/uut/c2_temp_wrdata       00000...
/testbench/uut/c2_temp_wren         0
/testbench/uut/s2_temp_wraddr       00000...
/testbench/uut/s2_temp_wrdata       00000...
/testbench/uut/s2_temp_wren         0
/testbench/uut/c3_temp_wraddr       00000...
Now                                 5765 ps
Cursor 1                            1075 ps
```

Figure 4.8 Simulation Waveforms of state transitions



```
                                      Msgs
/testbench/Clk                        1
/testbench/start                      0
/testbench/Reset                      0
/testbench/done                       0
/testbench/inference                  00000
/testbench/uut/state                  F4
/testbench/uut/Clk                    St1
/testbench/uut/start                  St0
/testbench/uut/Reset                  St0
/testbench/uut/done                   0
/testbench/uut/inference              00000
/testbench/uut/sample_data_data       197379
/testbench/uut/sample_data_addr       00000...
/testbench/uut/c1_temp_wraddr         00000...
/testbench/uut/c1_temp_wrdata         00000...
/testbench/uut/c1_temp_wren           0
/testbench/uut/s1_temp_wraddr         00000...
/testbench/uut/s1_temp_wrdata         00000...
/testbench/uut/s1_temp_wren           0
/testbench/uut/c2_temp_wraddr         00000...
/testbench/uut/c2_temp_wrdata         00000...
/testbench/uut/c2_temp_wren           0
/testbench/uut/s2_temp_wraddr         00000...
/testbench/uut/s2_temp_wrdata         00000...
/testbench/uut/s2_temp_wren           0
/testbench/uut/c3_temp_wraddr         00000...
                                Now   5765 ps
                           Cursor 1   1075 ps
```

Figure 4.9 Simulation Waveforms of layers state transitions

4.3 Register Transfer Level

After simulating the CNN accelerator in ModelSim, the design was compiled in Quartus Prime platform, and the RTL was generated. RTL was used to describe the circuit's structure and behavior at a level that can be synthesized into hardware. Figure 4.10 illustrates the complete system, including the NIOS processor, memory, CNN accelerator, and other associated components.

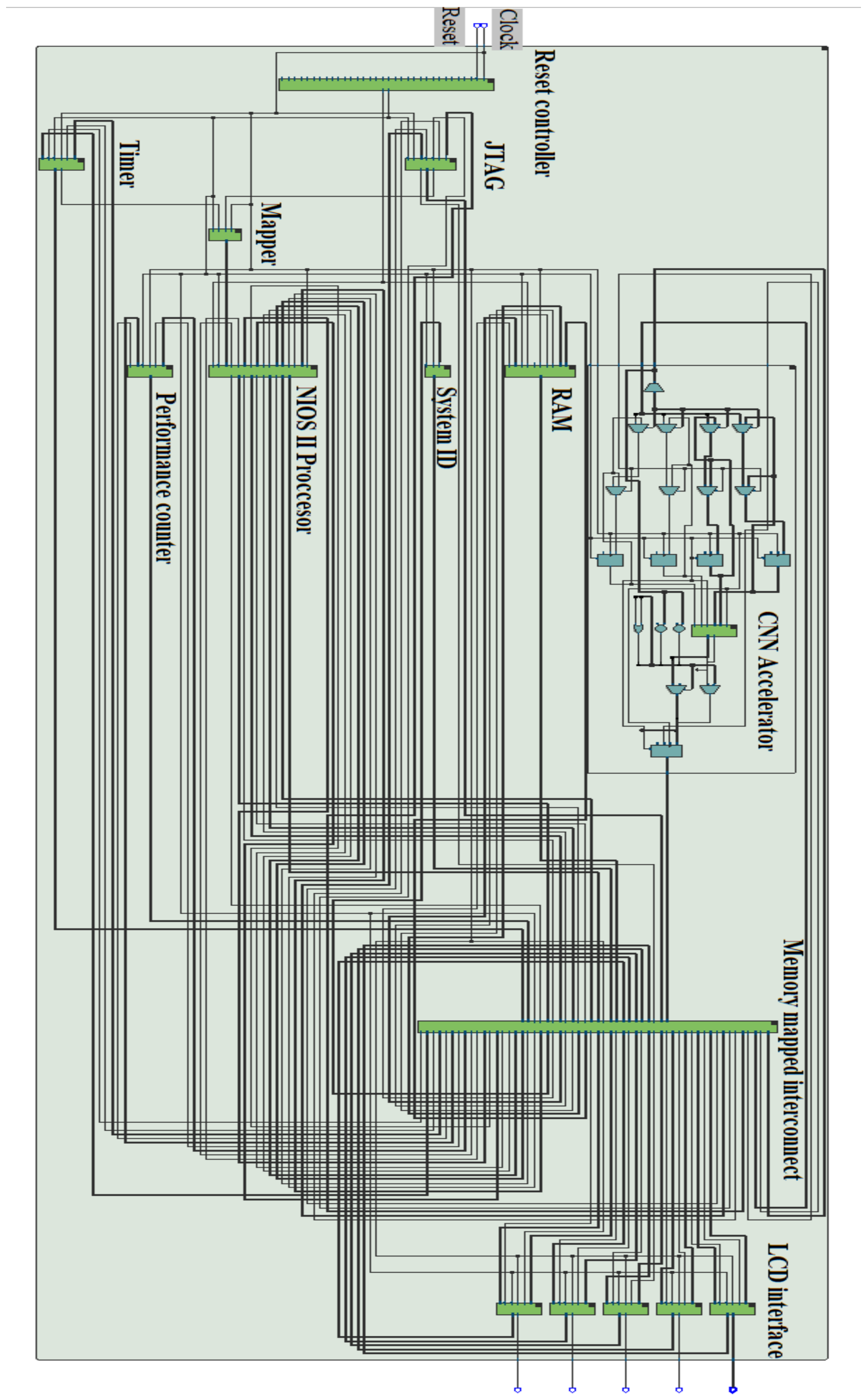


Figure 4.10 RTL of Complete System

4.4 Hardware Implementation of CNN and Results

After the system composed of NIOS processor and hardware accelerator was successfully instantiated, simulated and verified. It was implemented and tested using Cyclone IV GX4CX1550 FPGA evaluation platform, on the Intel (Altera previously) DE2i-150 board [13].Figure 4.11 illustrates the board, that contains many futures, one of them is LCD that also was used in the project.

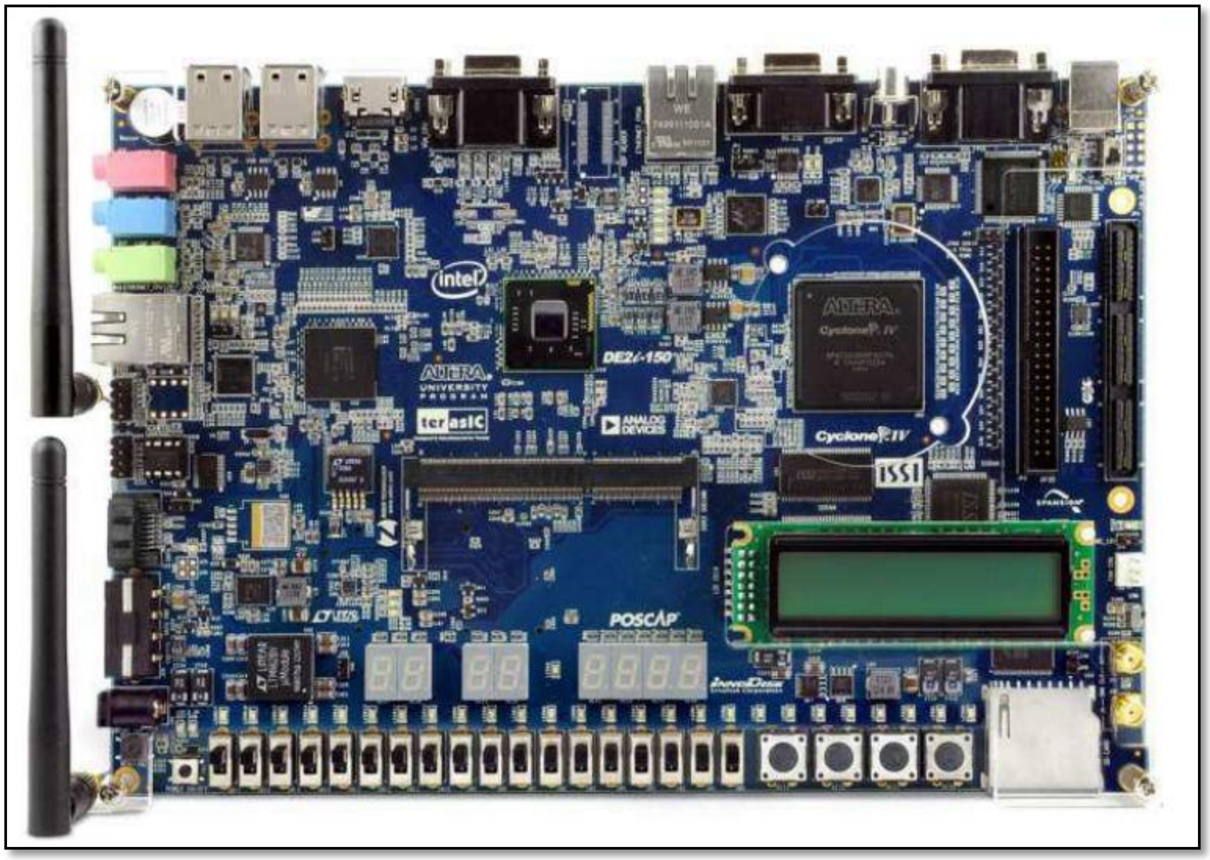
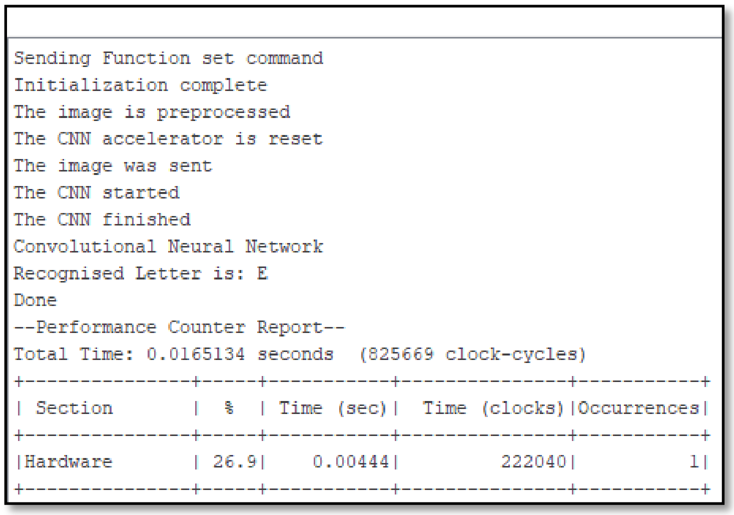


Figure 4.11 DE2i-150 Board

4.4.1 CNN Prediction Results

The CNN was tested on a set of 100 images of handwritten uppercase letters. The tests showed that 97 out of 100 images were correctly predicted, resulting in an accuracy of 97%. This accuracy is very similar to that of the software implementation, with the small difference likely due to the fixed-point representation of data. Figure 4.12 illustrates the CNN's prediction for the letter ‘E,’ which matches the prediction made by the software CNN.



```
Sending Function set command
Initialization complete
The image is preprocessed
The CNN accelerator is reset
The image was sent
The CNN started
The CNN finished
Convolutional Neural Network
Recognised Letter is: E
Done
--Performance Counter Report--
Total Time: 0.0165134 seconds  (825669 clock-cycles)
+---------------+-----+-----------+---------------+-----------+
| Section       |  %  | Time (sec)|  Time (clocks)|Occurrences|
+---------------+-----+-----------+---------------+-----------+
|Hardware       | 26.9|    0.00444|         222040|          1|
+---------------+-----+-----------+---------------+-----------+
```

Figure 4.12 Prediction of the CNN

Also report on performance counters is given, displaying the time taken and number of clock cycles used in the process. The hardware accelerator accounted for 26.9% of the total execution time, taking about 0.00444 seconds or 2,220,401 clock cycles, as shown in the report. The LCD display of prediction is shown in Figure 4.13.

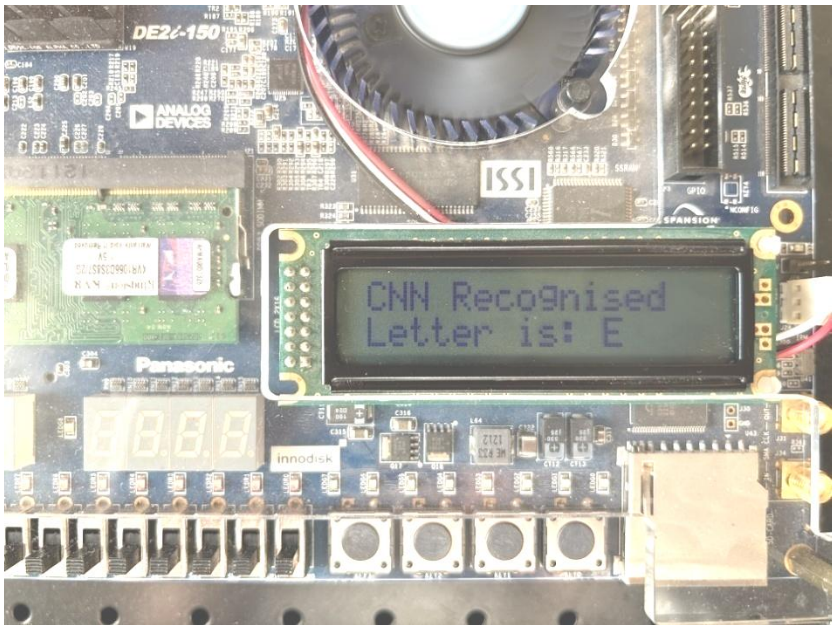


Figure 4.13 CNN Recognition Display on LCD

4.5 Timing Performance

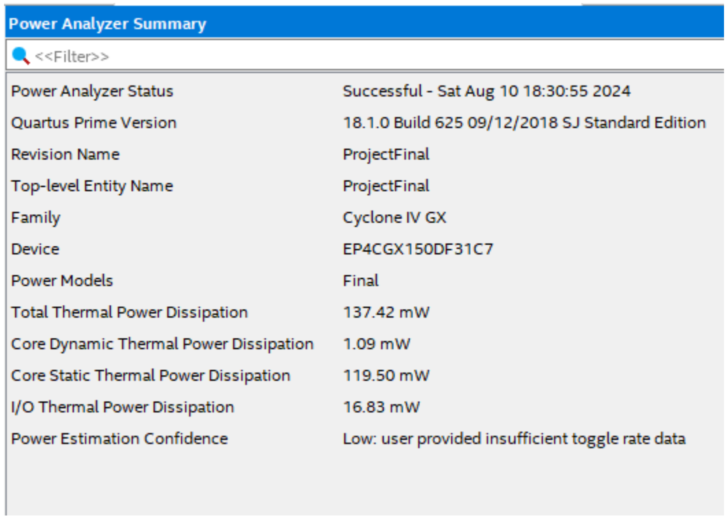
The implemented design is working under 50 MHz frequency. It takes 825669 clock cycles, and 0.0165134 seconds to deal with one image, while the hardware accelerator only without dealing with LCD display takes only 222040 clock cycles, and 0.00444 seconds. The speed-up was calculated, and Equation 4.1 shows the formula that was used.

$$Speedup = \frac{Execution\ time\ (Software)}{Execution\ time\ (Hardware)} \tag{4.1}$$

Compared with the software implemented CNN, which takes 0.8125 seconds to deal with same image, FPGA implementation shows an advantage in higher performance. It has a 115x speed-up than CPU.

4.6Power Consumption

Power play power analyzer used by Intel-FPGA Quartus Prime software tool. The power analyzer used to measure the thermal power dissipation for the model, t he power consumption in the design is 137.42 mW. The power consumption of different part is shown in Figure 4.14.


Power Analyzer Summary

<<Filter>>

Power Analyzer Status	Successful - Sat Aug 10 18:30:55 2024
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Standard Edition
Revision Name	ProjectFinal
Top-level Entity Name	ProjectFinal
Family	Cyclone IV GX
Device	EP4CGX150DF31C7
Power Models	Final
Total Thermal Power Dissipation	137.42 mW
Core Dynamic Thermal Power Dissipation	1.09 mW
Core Static Thermal Power Dissipation	119.50 mW
I/O Thermal Power Dissipation	16.83 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Figure 4.14 Power Consumption of System

The CNN was developed for use on both a software platform and an FPGA, with the FPGA running on the Cyclone IV GX FPGA board. The FPGA design on the board was able to meet all resource and timing constraints. The CNN on the FPGA showed much better performance than the software implementation, achieving faster processing speeds and shorter inference times. Moreover, the FPGA-powered CNN demonstrated reduced power usage, rendering it better suited for energy-saving tasks. Utilizing FPGA led to increased power efficiency by utilizing parallel processing for higher computational performance per watt. This not only boosted system performance but also increased scalability and adaptability, highlighting the benefits of hardware acceleration in deep learning applications.

Conclusion and Future Work

This project involved designing and implementing a specialized CNN for recognizing handwritten characters. Python-based software CNN was implemented. The model underwent successful training, obtaining accuracy of 97.89% and a loss of 0.0477. This performance is notably strong, especially when compared to other implementations in the field, highlighting the effectiveness and robustness of the model. The parameters of trained CNN were extracted and saved for hardware implementation. The CNN was modified for faster processing on hardware using an FPGA and NIOS processor combo, making use of SystemVerilog hardware description language for the execution. This hardware version used fixed-point arithmetic for data representation, for its efficiency and speed, as it requires fewer resources and performs faster than floating-point arithmetic. This approach also allows for better control over precision, making it ideal for resource-constrained environments like FPGAs. In particular, the system based on FPGA achieved a processing rate of 0.0165 seconds for each inference, representing a significant speedup of around 115 times compared to the software version.
Additionally, the FPGA system demonstrated efficiency in power consumption, with a measurement of 137.42 mW. This project shows the benefits of using FPGA-based approaches for deep learning tasks. The FPGA's noteworthy increase in speed and decrease in power usage, in comparison to CPU-based software methods, highlight the potential of hardware acceleration to improve performance and efficiency in real-world applications. Furthermore, this project showcases the effective combination of a soft-core processor and dedicated hardware, a synergy made possible by the FPGA. This combination exemplifies the flexibility and power of FPGA technology in enhancing the performance of CNN applications.

Future work will focus on advancing to more complex CNN architectures and exploring their application in real-time systems. This includes optimizing these advanced models for even faster processing and real-time performance in practical scenarios. Potential areas for development include tumor detection, where enhanced CNN models can improve early diagnosis through medical imaging; autonomous vehicles, where sophisticated CNNs can advance object detection and collision avoidance for safer navigation; real-time video surveillance, which can benefit from rapid analysis for security threat detection; speech recognition, where optimized CNNs can enhance the accuracy and responsiveness of virtual assistants; and medical image analysis, where advanced CNNs can aid in more precise and timely diagnostics. These applications aim to broaden the scope of FPGA-based deep learning systems, enhancing their utility in dynamic and high-demand environments.

References

[1]Zhang, C., et al. (2015) Optimizing FPGA-Based Accelerator Design for Deep Convolutional Neural Networks. Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, New York, NY, USA, February 2015, 161-170.

[2] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16). Association for Computing Machinery, New York, NY, USA, 16–25.

[3] Xiao, R., Shi, J., & Zhang, C. (2020). FPGA Implementation of CNN for Handwritten Digit Recognition. 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), 1, 1128-1133.

[4] P. Wang, J. Song, Y. Peng and G. Liu, "Binarized Neural Network Based On FPGA To Realize Handwritten Digit Recognition," 2020 IEEE International Conference on Information Technology,Big Data and Artificial Intelligence (ICIBA), Chongqing, China, 2020, pp. 1204-1207.

[5] Siyu Zhu, Hu Huang, Zhihong Hu, Qian Tian, "Design of handwritten digit recognition system based on FPGA," Proc. SPIE 11848, International Conference on Signal Image Processing and Communication (ICSIPC 2021), 1184812 (1 June 2021).

[6] Yu, Ke, Minguk Kim, and Jun Rim Choi. 2023. "Memory-Tree Based Design of Optical Character Recognition in FPGA" *Electronics* 12, no. 3: 754.

[7] L. A. de Oliveira Junior and E. Barros, "An FPGA-based Hardware Accelerator for Scene Text Character Recognition," *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, Verona, Italy, 2018, pp. 125-130, doi: 10.1109/VLSI-SoC.2018.8644776.
[8] Chen Wu, Mingyu Wang, Xinyuan Chu, Kun Wang, and Lei He. 2021. Low-precision Floating-point Arithmetic for High-performance FPGA-based CNN Acceleration. ACM Trans. Reconfigurable Technol. Syst. 15, 1, Article 6 (March 2022).

[9] Jiang, T., Xing, L., Yu, J. *et al.* A hardware-friendly logarithmic quantization method for CNNs and FPGA implementation. J Real-Time Image Proc 21, 108 (2024).

[10] Yanamala RMR, Pullakandam M. Empowering edge devices: FPGA-based 16-bit fixed-point accelerator with SVD for CNN on 32-bit memory-limited systems. *Int J Circ Theor Appl*. 2024; 1-28.

[11]Processor Architecture Intel. Available at: https://www.intel.com/content/www/us/en/docs/programmable/683836/current/processor-architecture.html.

[12] The EMNIST dataset (2024) NIST. Available at: https://www.nist.gov/itl/products-and-services/emnist-dataset.

[13] Zhe Li, Xiaoyu Wang, Xutao Lv, and Tianbao Yang. Sep-nets: Small and effective pattern networks. CoRR, abs/1706.03912, 2017.

[14] Chu, P.P. (2012) Embedded SoPC design with NIOS II processor and Verilog examples. Hoboken, N.J: Wiley.

[15] Altera. DE2i-150 FPGA System User Manual.

[16] Solai, P. (2020, June 22). How does Backpropagation work in a CNN? | Medium. *Medium*. https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c

[17] Nguyen, S. (2021, December 13). A gentle explanation of Backpropagation in Convolutional Neural Network (CNN). *Medium*. https://medium.com/@ngocson2vn/a-gentle explanation-of-backpropagation-in-convolutional-neural-network-cnn-1a70abff508b

[18] Linmoody. (2014, September 15). *How to Use Mathematical Equations in Your Thesis Current Grad Students*. Current Grad Students. https://gradstudents.carleton.ca/2014/use mathematicalequations-thesis/

Appendix A

The Source Code of CNN_ Top Module

```
module CNN_Top(
        input logic                     Clk, nios_reset_n, start, wen_g,
        input logic signed [31:0]       input_data_data,
        input logic [9:0]               input_data_addr,
        output logic                    done,
        output logic [4:0]              inference
);

        //Reset
        logic Reset;
        assign Reset = ~nios_reset_n;
        image data

        logic signed [31:0]     sample_data_data;
        logic [9:0]             sample_data_addr;
        // wren and address control
        logic [13:0]    c1_temp_wraddr;
        logic [31:0]    c1_temp_wrdata;
        logic                   c1_temp_wren;

        logic [13:0]    s1_temp_wraddr;
        logic [31:0]    s1_temp_wrdata;
        logic                   s1_temp_wren;

        logic [13:0]    c2_temp_wraddr;
        logic [31:0]    c2_temp_wrdata;
        logic                   c2_temp_wren;

        logic [13:0]    s2_temp_wraddr;
        logic [31:0]    s2_temp_wrdata;
        logic                   s2_temp_wren;

        logic [13:0]    c3_temp_wraddr;
        logic [31:0]    c3_temp_wrdata;
        logic                   c3_temp_wren;

        logic [13:0]    f4_temp_wraddr;
        logic [31:0]    f4_temp_wrdata;
        logic                   f4_temp_wren;
        logic [13:0]    f5_temp_wraddr;
        logic [31:0]    f5_temp_wrdata;
        logic                   f5_temp_wren;

        logic [13:0]    f6_temp_wraddr;
        logic [31:0]    f6_temp_wrdata;
        logic                   f6_temp_wren;

        logic                           inter_ram0_wren;
        logic                           inter_ram1_wren;
        logic                           c1_ready, s1_ready, c2_ready, s2_ready;
        logic                           c3_ready, f4_ready, f5_ready, f6_ready, infer_ready;

        logic [9:0]                     c1_data_addr;
        logic [13:0]                    s1_data_addr;
        logic [13:0]                    c2_data_addr;
        logic [13:0]                    s2_data_addr;
        logic [13:0]                    c3_data_addr;
        logic [13:0]                    f4_data_addr;
        logic [13:0]                    f5_data_addr;
        logic [13:0]                    f6_data_addr;
        logic [13:0]                    find_max_data_addr;
logic [9:0]             c1_weight_addr;
logic [13:0]            c2_weight_addr;
logic [13:0]            c3_weight_addr;
logic [13:0]            f4_weight_addr;
logic [13:0]            f5_weight_addr;
logic [13:0]            f6_weight_addr;

logic [9:0]             c1_bias_addr;
logic [9:0]             c2_bias_addr;
logic [9:0]             c3_bias_addr;
logic [9:0]             f4_bias_addr;
logic [9:0]             f5_bias_addr;
logic [9:0]             f6_bias_addr;

logic [13:0]            inter_ram0_wraddr;
logic [13:0]            inter_ram1_wraddr;
logic [13:0]            inter_ram0_rdaddr;
logic [13:0]            inter_ram1_rdaddr;


logic signed [31:0]     c1_weight_data;
logic signed [31:0]     c2_weight_data;
logic signed [31:0]     c3_weight_data;
logic signed [31:0]     f4_weight_data;
logic signed [31:0]     f5_weight_data;
logic signed [31:0]     f6_weight_data;

logic signed [31:0]     c1_bias_data;
logic signed [31:0]     c2_bias_data;
logic signed [31:0]     c3_bias_data;
logic signed [31:0]     f4_bias_data;
logic signed [31:0]     f5_bias_data;
logic signed [31:0]     f6_bias_data;

logic signed [31:0]     inter_ram0_rddata;
logic signed [31:0]     inter_ram1_rddata;
logic signed [31:0]     inter_ram0_wrdata;
logic signed [31:0]     inter_ram1_wrdata;
Ram temp0(
                                                  .Clk,
                                                  .rdaddr(inter_ram0_rdaddr),
                                                  .rddata(inter_ram0_rddata),
                                                  .wren(inter_ram0_wren),
                                                  .wraddr(inter_ram0_wraddr),
                                                  .wrdata(inter_ram0_wrdata));

Ram temp1(
                                                  .Clk,
                                                  .rdaddr(inter_ram1_rdaddr),
                                                  .rddata(inter_ram1_rddata),
                                                  .wren(inter_ram1_wren),
                                                  .wraddr(inter_ram1_wraddr),
                                                  .wrdata(inter_ram1_wrdata));

image_ram image(
                                                  .Clk,
                                                  .addr(sample_data_addr),
                                                  .read_data(sample_data_data));

conv1_weights_ram c1_weight(
                                                  .Clk,
                                                  .addr(c1_weight_addr),
                                                  .read_data(c1_weight_data));
```

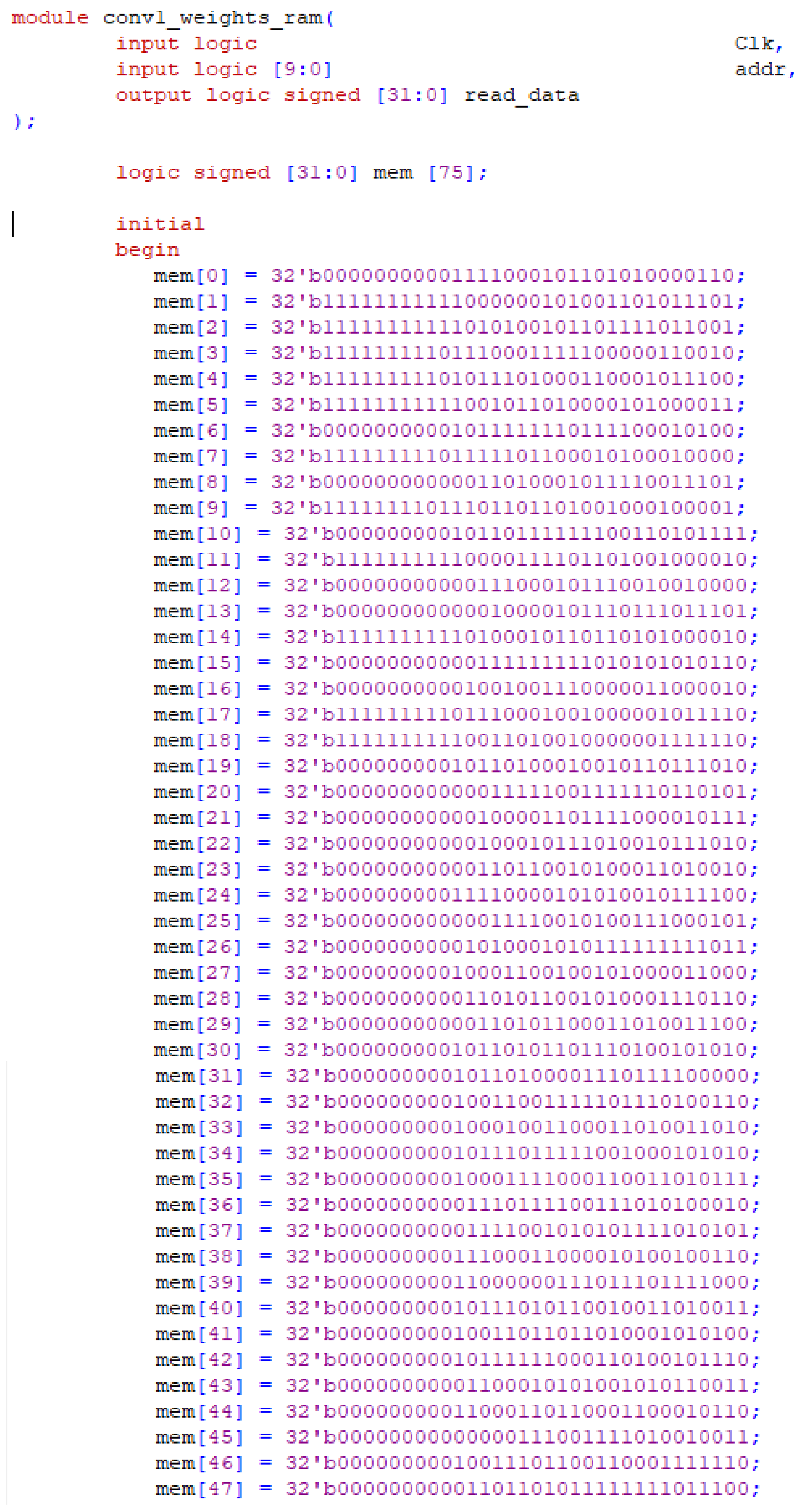
```
conv1_bias_ram c1_bias(
                                                .Clk,
                                                .addr(c1_bias_addr),
                                                .read_data(c1_bias_data));

Convolution_1 c1(
                                                .Clk, .start(start), .Reset,
                                                .curdata(sample_data_data),
                                                .curweight(c1_weight_data),
                                                .curbias(c1_bias_data),
                                                .data_addr(sample_data_addr),
                                                .weight_addr(c1_weight_addr),
                                                .bias_addr(c1_bias_addr),
                                                .temp_addr(c1_temp_wraddr), // save to ram0
                                                .temp_data(c1_temp_wrdata),
                                                .temp_wren(c1_temp_wren),
                                                .ready(c1_ready));
Pooling_1 s1(
                                                .Clk, .Reset, .start(c1_ready),
                                                .curdata(inter_ram0_rddata), // read from ram0
                                                .data_addr(s1_data_addr),
                                                .temp_addr(s1_temp_wraddr), // save to ram1
                                                .temp_data(s1_temp_wrdata),
                                                .temp_wren(s1_temp_wren),
                                                .ready(s1_ready));

conv2_weights_ram c2_weight(
                                                .Clk,
                                                .addr(c2_weight_addr),
                                                .read_data(c2_weight_data));

conv2_bias_ram c2_bias(
                                                .Clk,
                                                .addr(c2_bias_addr),
                                                .read_data(c2_bias_data));
Convolution_2 c2(
                                                .Clk, .start(s1_ready), .Reset,
                                                .curdata(inter_ram1_rddata), // read from ram1
                                                .curweight(c2_weight_data),
                                                .curbias(c2_bias_data),
                                                .data_addr(c2_data_addr),
                                                .weight_addr(c2_weight_addr),
                                                .bias_addr(c2_bias_addr),
                                                .temp_addr(c2_temp_wraddr), // save to ram0
                                                .temp_data(c2_temp_wrdata),
                                                .temp_wren(c2_temp_wren),
                                                .ready(c2_ready));
Pooling_2 s2(
                                                .Clk, .Reset, .start(c2_ready),
                                                .curdata(inter_ram0_rddata), // read from ram0
                                                .data_addr(s2_data_addr),
                                                .temp_addr(s2_temp_wraddr), // save to ram1
                                                .temp_data(s2_temp_wrdata),
                                                .temp_wren(s2_temp_wren),
                                                .ready(s2_ready));

conv3_weights_ram c3_weight(
                                                .Clk,
                                                .addr(c3_weight_addr),
                                                .read_data(c3_weight_data));

conv3_bias_ram c3_bias(
                                                .Clk,
                                                .addr(c3_bias_addr),
                                                .read_data(c3_bias_data));
```

```
Convolution_3 c3(
                                                .Clk, .start(s2_ready), .Reset,
                                                .curdata(inter_ram1_rddata), // read from ram1
                                                .curweight(c3_weight_data),
                                                .curbias(c3_bias_data),
                                                .data_addr(c3_data_addr),
                                                .weight_addr(c3_weight_addr),
                                                .bias_addr(c3_bias_addr),
                                                .temp_addr(c3_temp_wraddr), // save to ram0
                                                .temp_data(c3_temp_wrdata),
                                                .temp_wren(c3_temp_wren),
                                                .ready(c3_ready));

f4_weight_ram f4_weight(
                                                .Clk,
                                                .addr(f4_weight_addr),
                                                .read_data(f4_weight_data));

f4_bias_ram f4_bias(
                                                .Clk,
                                                .addr(f4_bias_addr),
                                                .read_data(f4_bias_data));

fullyconnect_4 f4(
                                                .Clk, .start(c3_ready), .Reset,
                                                .curdata(inter_ram0_rddata), // read from ram0
                                                .curweight(f4_weight_data),
                                                .curbias(f4_bias_data),
                                                .data_addr(f4_data_addr),
                                                .weight_addr(f4_weight_addr),
                                                .bias_addr(f4_bias_addr),
                                                .temp_addr(f4_temp_wraddr), // save to ram1
                                                .temp_data(f4_temp_wrdata),
                                                .temp_wren(f4_temp_wren),
                                                .ready(f4_ready));
f5_weight_ram f5_weight(
                                                .Clk,
                                                .addr(f5_weight_addr),
                                                .read_data(f5_weight_data));

f5_bias_ram f5_bias(
                                                .Clk,
                                                .addr(f5_bias_addr),
                                                .read_data(f5_bias_data));

fullyconnect_5 f5(
                                                .Clk, .start(f4_ready), .Reset,
                                                .curdata(inter_ram1_rddata), // read from ram1
                                                .curweight(f5_weight_data),
                                                .curbias(f5_bias_data),
                                                .data_addr(f5_data_addr),
                                                .weight_addr(f5_weight_addr),
                                                .bias_addr(f5_bias_addr),
                                                .temp_addr(f5_temp_wraddr), // save to ram0
                                                .temp_data(f5_temp_wrdata),
                                                .temp_wren(f5_temp_wren),
                                                .ready(f5_ready));

f6_weight_ram f6_weight(
                                                .Clk,
                                                .addr(f6_weight_addr),
                                                .read_data(f6_weight_data));

f6_bias_ram f6_bias(
                                                .Clk,
                                                .addr(f6_bias_addr),
                                                .read_data(f6_bias_data));
```

```
fullyconnect_6 f6(
                                                .Clk, .start(f5_ready), .Reset,
                                                .curdata(inter_ram0_rddata), // read from ram0
                                                .curweight(f6_weight_data),
                                                .curbias(f6_bias_data),
                                                .data_addr(f6_data_addr),
                                                .weight_addr(f6_weight_addr),
                                                .bias_addr(f6_bias_addr),
                                                .temp_addr(f6_temp_wraddr), // save to ram1
                                                .temp_data(f6_temp_wrdata),
                                                .temp_wren(f6_temp_wren),
                                                .ready(f6_ready));

find_max infer(
                                                .Clk, .Reset, .start(f6_ready),
                                                .curdata(inter_ram1_rddata), // read from ram1
                                                .data_addr(find_max_data_addr),
                                                .done(infer_ready),
                                                .idx(inference));


assign test_rddata = inter_ram0_rddata;

enum logic [3:0]{IDLE, CONV1, S1, CONV2, S2, CONV3, F4, F5, F6, INFERENCE, DONE} state, state_in;

always_ff@(posedge Clk)
begin
        if(Reset)
                state <= IDLE;
        else
                state <= state_in;
end

always_comb
begin
        state_in = state;
        unique case(state)
                IDLE:
                        if(start)
                                state_in = CONV1;

                CONV1:
                        if(c1_ready)
                                state_in = S1;

                S1:
                        if(s1_ready)
                                state_in = CONV2;

                CONV2:
                        if(c2_ready)
                                state_in = S2;

                S2:
                        if(s2_ready) // do not forget this line
                                state_in = CONV3;

                CONV3:
                        if(c3_ready)
                                state_in = F4;

                F4:
                        if(f4_ready)
                                state_in = F5;

                F5:
                        if(f5_ready)
                                state_in = F6;
                F6:
                        if(f6_ready)
                                state_in = INFERENCE;

                INFERENCE:
                        if(infer_ready)
                                state_in = DONE;

                DONE:
                        if(~start)
                                state_in = IDLE;
                default:
                        state_in = IDLE;


        endcase
end

// logic control
always_comb
begin
        done = 1'b0;

        unique case(state)

                DONE:
                        done = 1'b1;


                default:;

        endcase

end

// ram write control
always_comb
begin
        inter_ram0_rdaddr = 0;
        inter_ram0_wraddr = 0;
        inter_ram0_wrdata = 0;
        inter_ram0_wren = 0;
        inter_ram1_rdaddr = 0;
        inter_ram1_wraddr = 0;
        inter_ram1_wrdata = 0;
        inter_ram1_wren = 0;
        case(state)
                CONV1:begin
                        inter_ram0_wraddr = c1_temp_wraddr;
                        inter_ram0_wrdata = c1_temp_wrdata;
                        inter_ram0_wren         = c1_temp_wren;
                end

                S1:begin
                        inter_ram0_rdaddr = s1_data_addr;
                        inter_ram1_wraddr = s1_temp_wraddr;
                        inter_ram1_wrdata = s1_temp_wrdata;
                        inter_ram1_wren   = s1_temp_wren;
                end

                CONV2:begin
                        inter_ram1_rdaddr = c2_data_addr;
                        inter_ram0_wraddr = c2_temp_wraddr;
                        inter_ram0_wrdata = c2_temp_wrdata;
                        inter_ram0_wren         = c2_temp_wren;
                end
                S2:begin
                        inter_ram0_rdaddr = s2_data_addr;
                        inter_ram1_wraddr = s2_temp_wraddr;
                        inter_ram1_wrdata = s2_temp_wrdata;
                        inter_ram1_wren   = s2_temp_wren;
                end

                CONV3:begin
                        inter_ram1_rdaddr = c3_data_addr;
                        inter_ram0_wraddr = c3_temp_wraddr;
                        inter_ram0_wrdata = c3_temp_wrdata;
                        inter_ram0_wren         = c3_temp_wren;
                end

                F4:begin
                        inter_ram0_rdaddr = f4_data_addr;
                        inter_ram1_wraddr = f4_temp_wraddr;
                        inter_ram1_wrdata = f4_temp_wrdata;
                        inter_ram1_wren         = f4_temp_wren;
                end

                F5:begin
                        inter_ram1_rdaddr = f5_data_addr;
                        inter_ram0_wraddr = f5_temp_wraddr;
                        inter_ram0_wrdata = f5_temp_wrdata;
                        inter_ram0_wren         = f5_temp_wren;
                end

                F6:begin
                        inter_ram0_rdaddr = f6_data_addr;
                        inter_ram1_wraddr = f6_temp_wraddr;
                        inter_ram1_wrdata = f6_temp_wrdata;
                        inter_ram1_wren         = f6_temp_wren;
                end

                INFERENCE:begin
                        inter_ram1_rdaddr = find_max_data_addr;
                end

                DONE:;

        endcase
    end


endmodule
```

Appendix B

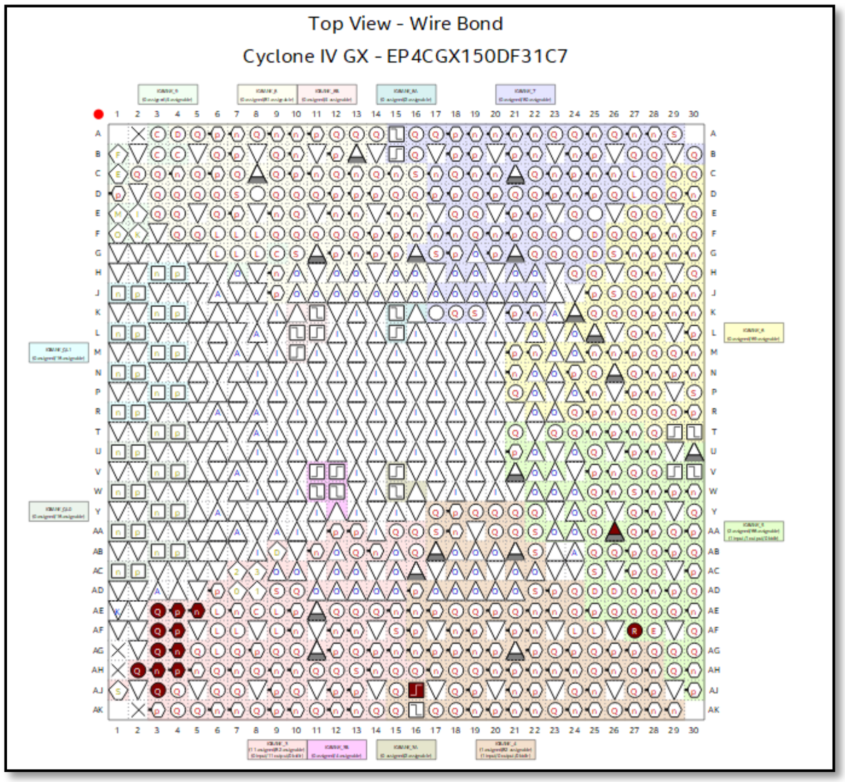
The Source Code of Weight ROM

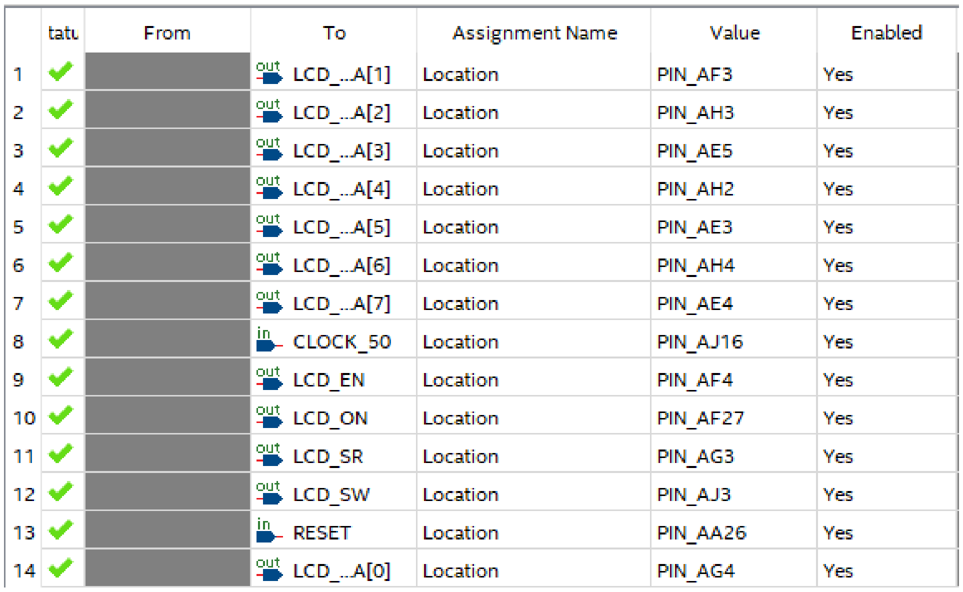


```
module conv1_weights_ram(
        input logic                                     Clk,
        input logic [9:0]                               addr,
        output logic signed [31:0] read_data
);

        logic signed [31:0] mem [75];

        initial
        begin
           mem[0] = 32'b00000000000111100010110101000110;
           mem[1] = 32'b11111111111000000101001101011101;
           mem[2] = 32'b11111111111010100101101111011001;
           mem[3] = 32'b11111111101110001111100000110010;
           mem[4] = 32'b11111111101011101000110001011100;
           mem[5] = 32'b11111111111100101101000010100011;
           mem[6] = 32'b00000000001011111110111100010100;
           mem[7] = 32'b11111111101111101100010100010000;
           mem[8] = 32'b00000000000001101000101111001101;
           mem[9] = 32'b11111111011101101101001000100001;
           mem[10] = 32'b00000000010110111111100110101111;
           mem[11] = 32'b11111111111000011110110100100010;
           mem[12] = 32'b00000000000011100010111001001000;
           mem[13] = 32'b00000000000001000010111011101101;
           mem[14] = 32'b11111111110100010110110101000010;
           mem[15] = 32'b00000000000011111111101010101010;
           mem[16] = 32'b00000000001001001110000011000010;
           mem[17] = 32'b11111111101110001001000001011110;
           mem[18] = 32'b11111111111001101001000000111110;
           mem[19] = 32'b00000000010110100010010110111010;
           mem[20] = 32'b00000000000001111100111111010101;
           mem[21] = 32'b00000000000010000110111100001011;
           mem[22] = 32'b00000000000010001011101001011101;
           mem[23] = 32'b00000000000011011001010001101001;
           mem[24] = 32'b00000000011110000101010010111100;
           mem[25] = 32'b00000000000001111001010011100010;
           mem[26] = 32'b00000000001010001010111111111011;
           mem[27] = 32'b00000000010001100100101000011000;
           mem[28] = 32'b00000000000110101100101000111011;
           mem[29] = 32'b00000000000011010110001101001110;
           mem[30] = 32'b00000000010110101101110100101010;
            mem[31] = 32'b00000000010110100001110111100000;
            mem[32] = 32'b00000000010011001111101110100110;
            mem[33] = 32'b00000000010001001100011010011010;
            mem[34] = 32'b00000000010111011111001000101010;
            mem[35] = 32'b00000000010001111000110011010111;
            mem[36] = 32'b00000000000111011110011101010001;
            mem[37] = 32'b00000000000111100101010111101010;
            mem[38] = 32'b00000000011100011000010100100110;
            mem[39] = 32'b00000000011000000111011101111000;
            mem[40] = 32'b00000000010111010110010011010011;
            mem[41] = 32'b00000000010011011011010001010100;
            mem[42] = 32'b00000000010111111000110100101110;
            mem[43] = 32'b00000000000110001010100101011001;
            mem[44] = 32'b00000000011000110110001100010110;
            mem[45] = 32'b00000000000000001110011110100100;
            mem[46] = 32'b00000000010011101100110001111110;
            mem[47] = 32'b00000000000110110101111111101110;
            mem[48] = 32'b00000000001011011000101101111111;
            mem[49] = 32'b00000000001101000000011011000101 0;
```

Appendix C

Pin Assignment of the System





	tatu	From	To	Assignment Name	Value	Enabled
1	✓		out LCD_...A[1]	Location	PIN_AF3	Yes
2	✓		out LCD_...A[2]	Location	PIN_AH3	Yes
3	✓		out LCD_...A[3]	Location	PIN_AE5	Yes
4	✓		out LCD_...A[4]	Location	PIN_AH2	Yes
5	✓		out LCD_...A[5]	Location	PIN_AE3	Yes
6	✓		out LCD_...A[6]	Location	PIN_AH4	Yes
7	✓		out LCD_...A[7]	Location	PIN_AE4	Yes
8	✓		in CLOCK_50	Location	PIN_AJ16	Yes
9	✓		out LCD_EN	Location	PIN_AF4	Yes
10	✓		out LCD_ON	Location	PIN_AF27	Yes
11	✓		out LCD_SR	Location	PIN_AG3	Yes
12	✓		out LCD_SW	Location	PIN_AJ3	Yes
13	✓		in RESET	Location	PIN_AA26	Yes
14	✓		out LCD_...A[0]	Location	PIN_AG4	Yes