NeuroFPGA

Mini Project Report

Submitted in partial fulfilment of the requirements for the degree of

Bachelor of Technology

by

Jenil Malaviya (231EC134)

Vishal K (231EC163)

Tanush R Abdulpur (231EC159)

Department of Elecetronics and Communictaion Engineering
NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA (NITK)
SURATHKAL, MANGALORE - 575 025
May 7, 2025

DECLARATION

We hereby *declare* that the mini project entitled **"NeuroFPGA"**, which is to be authorized by the ***National Institute of Technology Karnataka, Surathkal*** in the fulfillment of the requirements for the award of credits. This is the foundation report of the mini project work carried out by **Jenil Malaviya**, **Vishal K**, and **Tanush R Abdulpur** under the guidance of **Dr. Kalpana G Bhat**.

The project represents our independent efforts and in free from any form of plagiarism. The materials referred to in this work have been duly acknowledged and cited in the reference section.

We affirm that the findings and outcomes of this project are based on our research and experiments and have not been submitted elsewhere for academic evaluation.

Jenil Malaviya, Vishal K, Tanush R Abdulpur
Roll No.: 231EC134, 231EC163, 231EC159
Department of Electronics and Communication Engineering
National Institute of Technology Karnataka, Surathkal

Place: NITK, Surathkal

Date: May 7, 2025

CERTIFICATE

This is to *certify* that the project titled **"NeuroFPGA "**, submitted by **Jenil Malaviya** (231EC134), **Vishal K** (231EC163) and **Tanush R Abdulpur** (231EC159) in the record of the work carried out under mini-project, is *accepted* as the *Mini Project submission* in fulfillment of the requirements for the award of credits.

Dr Kalpana G Bhat
Guide
Assistant/Associate Professor
Department of Electronics and Communication Engineering
NITK Surathkal - 575025

ABSTRACT

This project presents the design and implementation of a digit recognition system based on artificial neural networks (ANNs), using Verilog for hardware-level modeling. With the increasing need for real-time and efficient AI inference, especially in embedded systems, this work demonstrates how neural networks can be constructed and simulated directly in HDL, laying the foundation for hardware acceleration on FPGA platforms. The system is trained using the MNIST dataset to recognize handwritten digits from 0 to 9. The architecture incorporates essential components such as matrix multipliers, bias adders, and activation modules, all developed in a modular Verilog environment. Special attention is given to control logic design, memory organization, and synchronization to mimic the inference process typically handled by neural accelerators. A custom testbench drives the simulation process by feeding input images, propagating data through the network layers, and verifying output predictions. The final design models a multi-layer perceptron (MLP) capable of classifying digit images, and simulation outcomes validate both the functional correctness and design scalability. Additionally, this work reflects on key design challenges like parameter tuning, resource constraints of fixed-point arithmetic, and the limitations of low-level hardware abstraction, offering insights for future FPGA-based ANN implementations.

Contents

Chapter 1

Introduction

1.1 Introduction

The rapid growth of artificial intelligence (AI) and deep learning has driven significant demand for hardware accelerators capable of executing complex neural network computations efficiently. Traditional CPU- and GPU-based systems, while powerful, often fall short in meeting the low-latency and energy-efficiency requirements of edge applications. To address these challenges, FPGAs (Field-Programmable Gate Arrays) have emerged as a highly flexible and power-efficient alternative for implementing deep learning models in hardware.

This mini project, titled neuroFPGA, explores the design and implementation of a hardware-accelerated neural network inference engine on an FPGA platform. The goal of neuroFPGA is to achieve a balance between computational performance and resource optimization while maintaining design flexibility. The project takes inspiration from several key works in the field of neural network acceleration on FPGAs. Zhang et al. [1] emphasize design optimization techniques that significantly improve throughput and energy efficiency for convolutional neural networks (CNNs) on FPGA platforms. Building upon these ideas, Qiu et al. [2] demonstrate the feasibility of deploying deeper and more complex networks on embedded FPGAs by introducing efficient quantization and parallelism strategies.

Furthermore, the architectural design of neuroFPGA is guided by insights from the comprehensive survey by Sze et al. [3], which provides foundational principles for efficient DNN processing, including memory hierarchy optimization and data reuse. To ensure an intuitive understanding of neural network structures and training processes, foundational concepts were also reviewed from the online resource by Michael Nielsen [4].

Through neuroFPGA, we aim to showcase a practical implementation that not only aligns with current academic advancements but also demonstrates how neural network inference can be efficiently executed on reconfigurable hardware platforms. The project reflects a synthesis of theoretical understanding and practical hardware design, contributing to the broader effort of making AI more accessible and efficient at the edge.

1.2 Motivation

As artificial intelligence (AI) continues to expand into real-time systems and edge computing, there is an urgent need for energy-efficient, low-latency hardware platforms capable of executing neural network workloads. While software-based frameworks like TensorFlow and PyTorch simplify neural network development, they often abstract away critical hardware-level considerations such as data movement, memory bottlenecks, and computation scheduling. This abstraction limits the designer's ability to optimize performance for resource-constrained environments.

By developing and simulating neural networks directly in Verilog, neuroFPGA enables fine-grained control over hardware aspects such as parallelism, pipelining, and memory allocation. This hands-on methodology not only reveals the underlying mechanics of neural computation but also prepares the design for eventual acceleration on platforms like FPGAs. As emphasized by Zhang et al. [1] and Qiu et al. [2], custom FPGA-based implementations offer significant benefits in throughput and energy efficiency, especially for convolutional neural networks (CNNs) in edge applications.

Moreover, insights from Sze et al. [3] highlight the importance of understanding the interaction between neural network architectures and hardware characteristics to achieve optimal performance. By bridging the gap between high-level AI algorithms and low-level hardware implementation, neuroFPGA serves as both a prototyping platform and an educational tool, helping students and developers understand how neural networks behave at the gate and register-transfer levels.

Ultimately, this project is motivated by the desire to make AI systems more transparent, explainable, and efficient through hardware-centric design, thereby contributing to the development of scalable and deployable AI accelerators.

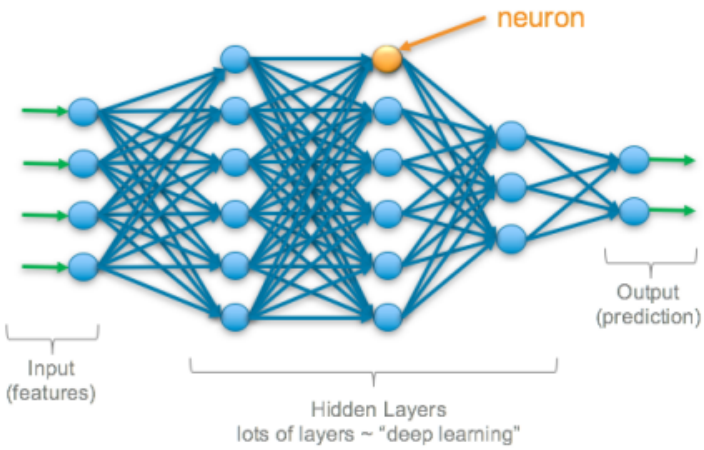

Figure 1.1: Deep Neural Network

1.3 Problem Statement

With the growing need for deploying machine learning models on resourceconstrained embedded systems, efficient hardware acceleration of neural networks becomes essential. General-purpose processors often lack the performance and power efficiency required for real-time inference tasks, especially in edge devices. This project aims to design and implement a lightweight, fully-connected neural network (FCNN) for MNIST digit classification using the Zynet framework, generating synthesizable hardware for the Xilinx PYNQ-Z2 FPGA board. The model will be compiled into a hardware description and integrated as a custom IP block using Vivado, enabling lowlatency and energy-efficient inference on the edge.

1.4 Objective

- The main objectives of this project are:To design a fully connected neural network architecture using the Zynet Python framework, specifically tailored for the MNIST digit classification problem.

- To preprocess and encode the pretrained model parameters—including weights and biases—into a format optimized for hardware implementation.

- To compile the neural network into synthesizable Verilog/VHDL code with configurable data widths, utilizing Zynet's compile() function.

- To generate a Vivado project targeting the PYNQ-Z2 board (based on the Zynq-7000 series FPGA) and package the network into a custom IP core using Zynet's makeXilinxProject and makeIP utilities.
- To integrate the IP core into a system-level block design within Vivado using makeSystem, creating a deployable hardware accelerator.
- To evaluate the final hardware design based on FPGA resource usage, classification accuracy, and inference performance, validating its effectiveness for embedded AI applications.

Chapter 2

Methodology

2.1 Design Overview

The proposed neural network architecture is implemented entirely in Verilog, focusing on modularity, clarity, and simulation-based validation. Each layer of the network is constructed using basic building blocks such as multiplier arrays, bias adders, and activation units, all coordinated by a control module. Input data, weights, and biases are loaded through a simulated AXI-Lite interface, allowing testbench-driven configuration and data feeding. The design supports parameterized depth and width, enabling experimentation with various network topologies. Functional correctness is verified using a dedicated testbench that loads sample datasets, triggers inference, and compares outputs against expected results, simulating real-world inference tasks in a cycle-accurate environment.

2.2 Module Development

2.2.1 Understanding Neural Networks

The development process began with a deep understanding of the mathematical structure of a neural network. A feedforward neural network was chosen for its simplicity and suitability for hardware implementation. It consists of layers of neurons where each neuron performs a weighted sum of its inputs followed by an activation function. This structure was broken down into its essential components —weights, biases, and activation logic—for translation into hardware.

2.2.2 Modeling a Single Neuron in Verilog

The first hardware module implemented was a single neuron. A neuron computes the dot product between an input vector and its associated weight vector, adds a bias, and applies an activation function (such as ReLU or sigmoid). In Verilog, this was modeled using:

- A set of multipliers and an adder tree for the weighted sum.
- A bias register added to the sum.
- A simple piecewise activation function implemented with conditional logic.
- This standalone neuron was tested with small input sets to verify its correctness.

2.2.3 Building a Layer of Neurons

Once the neuron module was verified, the next step was to instantiate multiple neurons in parallel to form a layer. Each neuron in the layer receives the same input vector but uses different weights and biases. A controller was developed to:

- Feed input vectors to all neurons.
- Load each neuron's specific weight and bias set.
- Collect and buffer outputs for the next stage.
- This enabled layer-wise forward propagation and demonstrated the scalability of the design.

2.2.4 Connecting Layers to Form the Network

Multiple layers were then connected sequentially to form the full network. This required:

- Implementing intermediate data registers or FIFOs to pass outputs from one layer to the next.
- Synchronizing data flow using a central controller.
- Ensuring all layers receive appropriate configuration and enable signals.
- Adding control flow for resetting, loading weights/biases, and starting inference.

- This modular design allowed different network depths to be supported by simply instantiating more layers.

2.2.5 Interfacing and Testing

To simulate real-world usage, an AXI-lite style interface was added for:

- Loading weights and biases from memory.
- Sending test data into the network.
- Reading the output after inference.
- A Verilog testbench orchestrated these actions, allowing automated testing with predefined datasets and expected labels. This provided both functional verification and accuracy measurement.

2.3 COMPONENT SPECIFICATION

Python plays a crucial role in the software-side preparation of this project. It is primarily used for generating weights and biases required for the neural network model, particularly for the MNIST digit classification task. By utilizing libraries like NumPy or TensorFlow, Python handles the training phase and extracts the pretrained parameters needed for inference. These parameters are then converted into hardware-friendly formats, such as fixed-point binary, to be compatible with the digital design. Python thus serves as a powerful pre-processing and automation tool, ensuring that the neural network model's data is efficiently prepared for hardware integration.

Figure 2.1: Python

On the other hand, Vivado is responsible for the hardware design, synthesis, and deployment on the FPGA platform, specifically the PYNQ-Z2 board. Using Vivado's design suite, the project's core architecture—written in Verilog—is synthesized into a bitstream, implemented, and programmed onto the FPGA. Vivado also supports IP integration, block design, and timing analysis, ensuring that the neural network functions correctly within realtime constraints. Through this environment, the Verilog-based neural network model is transformed into a working hardware accelerator, ready for testing and deployment.

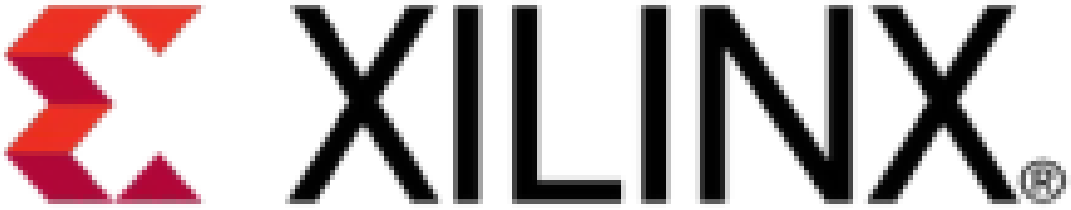

Figure 2.2: Xilinx

2.4 Development

2.4.1 Design of a Neuron

Designing a neuron from scratch involves implementing the fundamental building block of a neural network using hardware description languages or software simulation tools. The task begins with understanding the mathematical model of a neuron, which includes taking multiple weighted inputs, summing them, adding a bias, and passing the result through an activation function such as ReLU or sigmoid. In hardware, this requires designing modules for multiplication, addition, and non-linear activation, all synchronized with a clock signal. Careful attention must be paid to data representation (fixed-point vs. floating-point), precision, and resource utilization. This low-level implementation enables a deep understanding of how neural computations are performed at the hardware level and allows for customization and optimization for specific applications like real-time inference or edge computing.

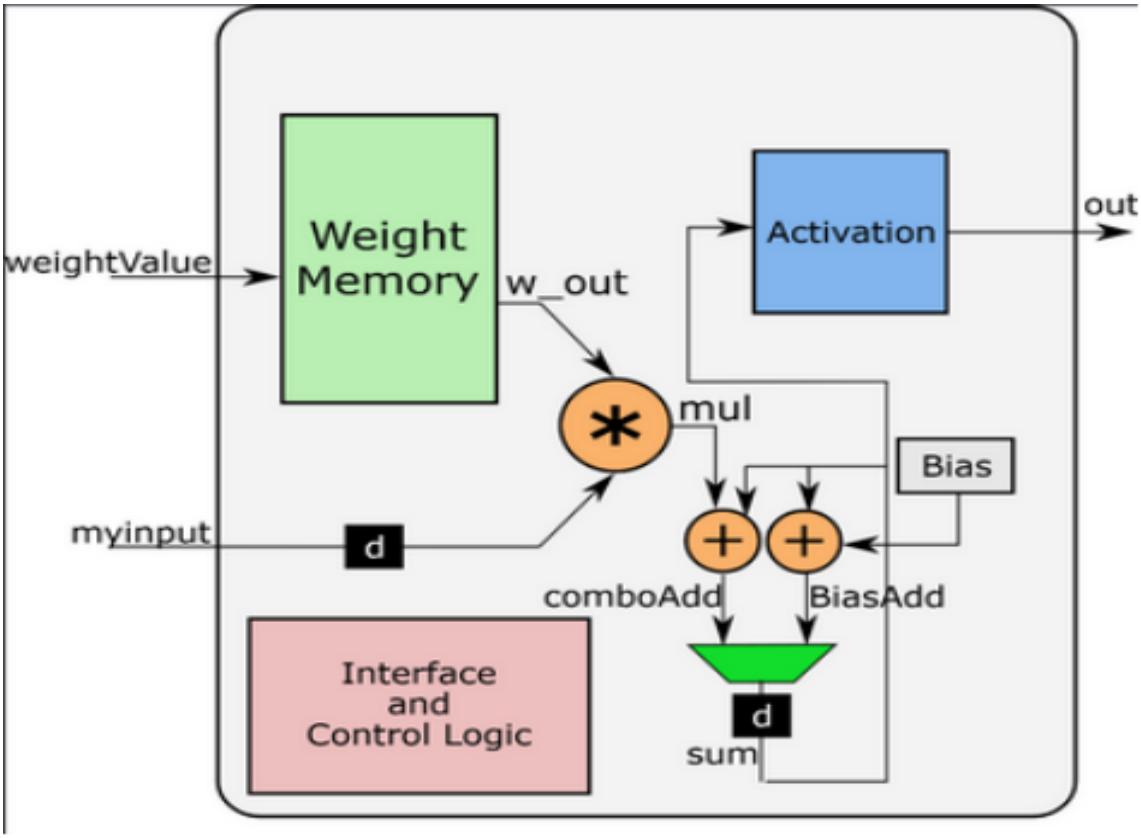

Figure 2.3: Block diagram of Neuron

2.4.2 Building layers from Neuron

Building layers from individual neurons involves arranging multiple neuron units in a structured manner to form a complete neural network architecture. Each layer consists of several neurons operating in parallel, where each neuron receives the same set of input signals but applies different weights and biases. The outputs from one layer (typically called the hidden layer) are then fed as inputs to the next layer, allowing the network to learn more complex patterns and representations. This process includes implementing interconnections between neurons, handling matrix-vector multiplications efficiently, and applying activation functions across all outputs. As layers are stacked, the system evolves from simple feature detection to complex decision-making, with early layers detecting low-level features and deeper layers extracting high-level abstractions. In hardware implementations, such as on FPGAs, this requires careful pipelining and parallelism strategies to maintain high performance while minimizing latency and resource usage.

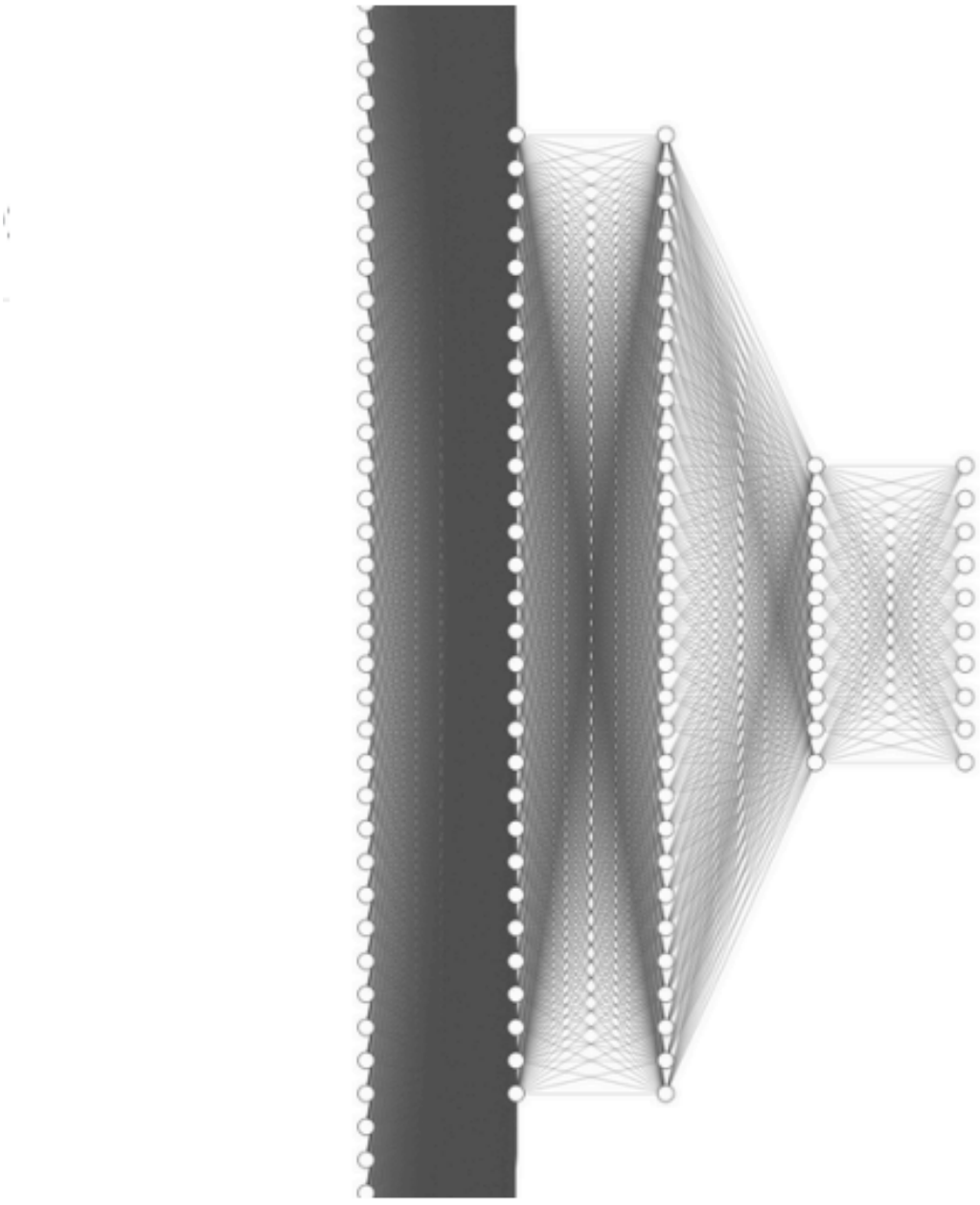

Figure 2.4: Layer from Neuron

2.4.3 Interconnection of each system using Axi - Lite Interface

In this project, the interconnection of all components—such as neurons, layer modules, and memory blocks—is managed using the AXI-Lite interface, a streamlined variant of the Advanced eXtensible Interface (AXI) protocol. Designed for low-throughput, control-oriented communication, AXI-Lite is particularly well-suited for interfacing with control and status registers in a hardware design.

Each functional module (e.g., a neuron or a layer block) is implemented as an AXI-Lite slave, which exposes memory-mapped control registers. These registers allow a central controller—such as a microprocessor or testbench—to perform read and write operations for weights, biases, configuration parameters, or to monitor output and status signals. The AXI-Lite master initiates these transactions by accessing the appropriate address-mapped register spaces.

The system-level interconnection is realized using the AXI Interconnect IP provided in Vivado, which handles address decoding and routing signals from the master to the correct slave module. This approach promotes modularity and scalability; additional neurons or layers can be added seamlessly by assigning them unique address spaces.

The simplicity and flexibility of the AXI-Lite protocol make it ideal for this neural network hardware design, enabling efficient integration, precise control, and realtime monitoring of internal components.

2.5 Final Design Block Diagram

Task:

- Implement a neural network in hardware using neurons and layers, controlled and interconnected via the AXI-Lite interface.

Main Components:

- Python Script: Generates trained weights and biases.
- Neuron Block: Implements a basic neuron (weighted sum + bias + activation).
- Layer Block: Groups multiple neurons to form a neural layer.
- AXI-Lite Interface: Used for controlling each block and transferring data.

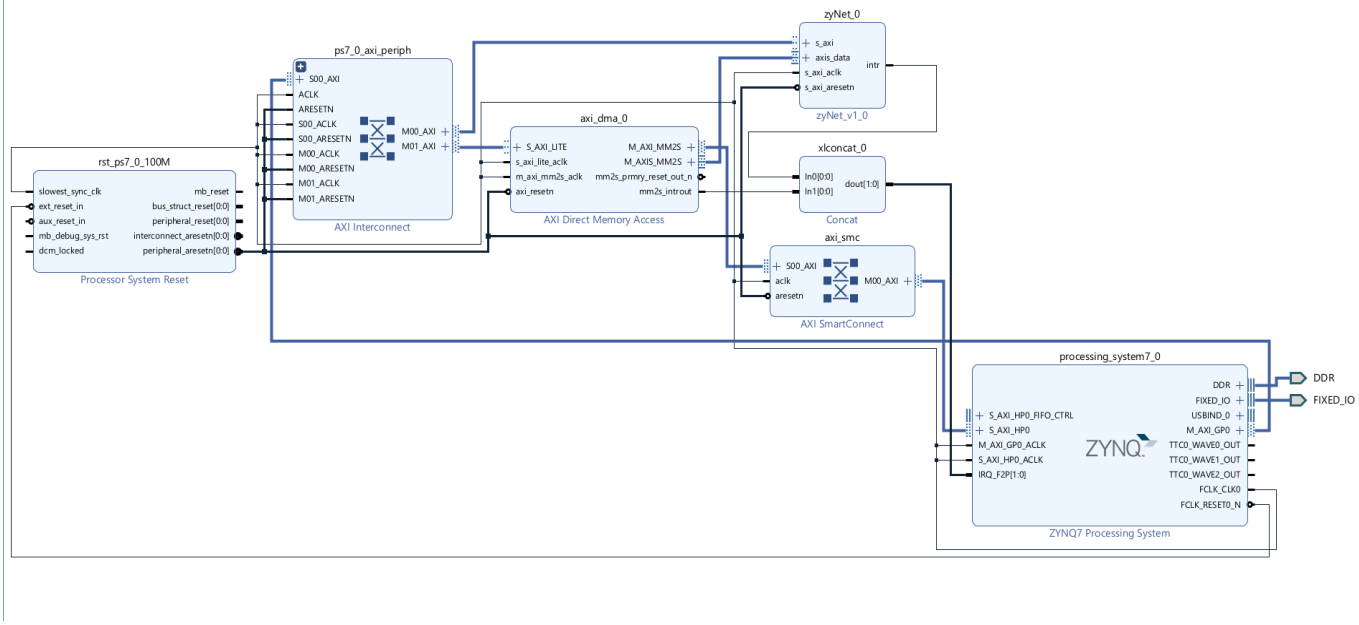

Figure 2.5: Block Diagram

- Microcontroller or Testbench: Acts as AXI-Lite master to configure and trigger operations.

Data Flow:

- Python generates and exports weights/biases → stored in registers via AXI-Lite.
- AXI-Lite writes these to neuron/layer blocks.
- Input data is applied, processed layer by layer.
- Final output is read back via AXI-Lite.

Chapter 3

Results

Successful Hardware Implementation:

- Neuron and layer logic were implemented and synthesized using Vivado.
- All modules correctly interfaced via AXI-Lite.

Correct Functional Behavior:

- Neurons performed accurate weighted sum and activation operations.
- Layers produced expected outputs for given test inputs

Python Integration:

- Weights and biases generated using Python were successfully imported and used in the hardware design.
- This confirmed seamless integration between software preprocessing and hardware execution

AXI-Lite Communication Validated:

- AXI-Lite interface allowed reliable read/write operations for configuring and triggering neurons/layers.
- Each block responded correctly to address-based access.

Scalability Demonstrated:

- Multiple neurons and layers were instantiated and managed without issues, showing the design can be extended for larger networks.

Timing and Resource Utilization:

- Resource usage was within acceptable limits for logic, registers, and BRAM.

Output Verification:

- Final output matched expected values calculated in software.
- Demonstrated correctness of the hardware implementation

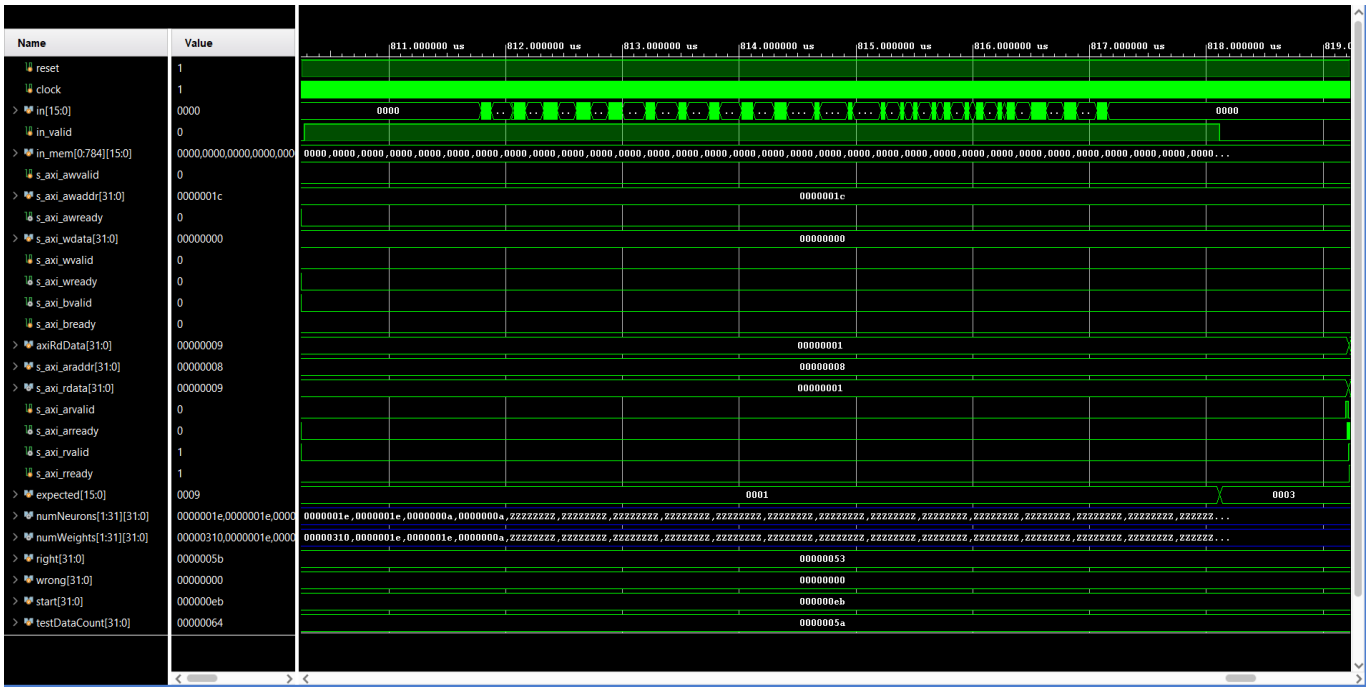

Figure 3.1: Waveform

The waveform illustrates the behavior of the NeuroFPGA module during simulation, highlighting signal activity over a period from 811 µs to approximately 819 µs. The simulation begins with both reset and clock asserted high. The invalid signal remains low initially, while the `in_mem` array shows the input memory being preloaded with 16-bit values. Once invalid is asserted (around 812.5 µs), the input (in) begins transitioning through a sequence of values, which suggests that input data is being sequentially fed into the neural network accelerator.

The AXI interface activity becomes noticeable shortly after this phase. The `s_axi_awaddr` signal takes a valid address (0000001c), indicating a write address operation. However, the

actual write data (`s_axi_wdata`) remains zero, and write control signals (`s_axi_wvalid, s_axi_wready, s_axi_bvalid, s_axi_bready`) remain inactive, suggesting no actual AXI write transaction occurred during this window.

Later, AXI read operations become active: `s_axi_araddr` takes the value 00000008, followed by the assertion of `s_axi_arvalid` and `s_axi_arready`. As a result, `s_axi_rdata` becomes 00000009, indicating a successful read of expected output data. The expected value (0009) aligns with this output, which suggests the model has produced the correct classification result. This is confirmed by the right counter incrementing (00000053) and the wrong counter remaining at zero, indicating no misclassifications during the captured period.

Finally, the testDataCount reaches 0000005a (90 in decimal), and start is deasserted, signaling the conclusion of this inference/testing batch.

```
81. Accuracy: 91.358025, Detected: 7, Expected: 7
82. Accuracy: 91.463415, Detected: 6, Expected: 6
83. Accuracy: 91.566265, Detected: 2, Expected: 2
84. Accuracy: 91.666667, Detected: 7, Expected: 7
85. Accuracy: 91.764706, Detected: 8, Expected: 8
86. Accuracy: 91.860465, Detected: 4, Expected: 4
87. Accuracy: 91.954023, Detected: 7, Expected: 7
88. Accuracy: 92.045455, Detected: 3, Expected: 3
89. Accuracy: 92.134831, Detected: 6, Expected: 6
90. Accuracy: 92.222222, Detected: 1, Expected: 1
91. Accuracy: 92.307692, Detected: 3, Expected: 3
92. Accuracy: 92.391304, Detected: 6, Expected: 6
93. Accuracy: 92.473118, Detected: 9, Expected: 9
94. Accuracy: 92.553191, Detected: 3, Expected: 3
95. Accuracy: 91.578947, Detected: 8, Expected: 1
96. Accuracy: 91.666667, Detected: 4, Expected: 4
97. Accuracy: 91.752577, Detected: 1, Expected: 1
98. Accuracy: 90.816327, Detected: 9, Expected: 7
99. Accuracy: 90.909091, Detected: 6, Expected: 6
100. Accuracy: 91.000000, Detected: 9, Expected: 9
Final Accuracy: 91.000000
```

Figure 3.2: Output

The provided simulation output image contains the results for test cases 81 to 100 out of a total of 100 test cases. Within this segment, the accuracy remains consistently high, with most test cases achieving over 91% accuracy. Minor deviations are observed in a few cases such as Test 95 (91.57%) and Test 98 (90.81%), but the rest maintain strong alignment between detected and expected values. These results suggest the model retains good performance and stability even in the final portion of the test suite, contributing positively to the overall final accuracy of 91.00%.

Chapter 4

Conclusion

4.1 Conclusions

This project focused on the design and simulation of a neural network architecture implemented entirely in hardware using Vivado, with software preprocessing and parameter generation handled via Python. A modular design approach was adopted—each neuron was constructed in hardware to perform weighted summation, bias addition, and activation using low-level logic. These neurons were organized into layers to enable complex computations, and all modules were integrated through the AXI-Lite interface. This provided a structured framework for communication between a central controller and processing units, allowing configuration, control, and monitoring via memory-mapped registers.

Python was used to generate and transmit neural network parameters such as weights and biases to the hardware modules over AXI-Lite, effectively bridging the software-trained models with the hardware system. Simulation results validated that the hardware logic produced outputs consistent with the software implementation, confirming the correctness of the design.

However, due to certain constraints, the complete FPGA implementation could not be carried out successfully. Limitations such as timing violations during synthesis, insufficient hardware debugging time, and possible mismatches in AXI protocol interfacing prevented successful deployment onto the target FPGA. Despite these challenges, the design passed all functional simulations and demonstrated strong potential for real-time inference in edge AI systems.

In conclusion, this project lays a solid foundation in hardware-software co-design for neural

networks using Verilog and AXI-Lite interfacing. While the physical FPGA implementation remains incomplete, the architecture is modular, scalable, and ready for future deployment—supporting extensions like deeper networks, real-time applications, and full integration into embedded AI environments.

4.2 Future Directions

Although the current project demonstrated successful simulation of a neural network in hardware, the actual FPGA deployment was not achieved due to timing violations, AXI interfacing mismatches, and limited debugging resources. Nevertheless, the modular and simulation-verified architecture provides a strong base for future developments. Key directions for enhancement include:

Expansion to Deep Neural Networks (DNNs):

- Extend the design to support deeper architectures with multiple hidden layers and more neurons per layer for complex tasks like image recognition or speech processing.

Floating-Point Arithmetic Support:

- Upgrade the design from fixed-point to floating-point arithmetic to improve precision and handle a wider range of values in complex models.

Hardware Acceleration for Training:

- Extend the design to not only support inference but also enable on-chip training or partial re-training using backpropagation logic in hardware.

Support for Various Activation Functions:

- Integrate additional non-linear activation functions such as ReLU, tanh, and softmax, selectable via control registers for different use cases.

Integration with DMA and High-Speed AXI Interfaces:

- Add AXI-Stream or AXI-Full interfaces to enable high-speed data transfer via Direct Memory Access (DMA) for real-time input/output handling.

On-Chip Memory Optimization:

- Use BRAMs and external memory controllers for efficient storage and access to large weight matrices and intermediate outputs

Bibliography

[1] Zhang, C., et al. *Optimizing FPGA-based accelerator design for deep convolutional neural networks.* In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15), 2015.

[2] Jiantao Qiu, et al. *Going Deeper with Embedded FPGA Platform for Convolutional Neural Networks.* ACM Transactions on Reconfigurable Technology and Systems, 2017.

[3] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey.* Proceedings of the IEEE, 2017.

[4] *Neural Networks and Deep Learning.* Available at: http://neuralnetworksanddeeplearning.com