ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

DESIGN AND IMPLEMETATION OF RISC-V BASED CORE FOR ARTIFICIAL INTELLIGENCE APPLICATIONS

SENIOR DESIGN PROJECT

Yusuf TEKİN
Ahmet Tolga ÖZKAN

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

JANUARY 2026

ISTANBUL TECHNICAL UNIVERSITY
ELECTRICAL-ELECTRONICS FACULTY

DESIGN AND IMPLEMETATION OF RISC-V BASED CORE FOR ARTIFICIAL INTELLIGENCE APPLICATIONS

SENIOR DESIGN PROJECT

Yusuf TEKİN
040200043
Ahmet Tolga ÖZKAN
040190067

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

Project Advisor: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

JANUARY 2026

İSTANBUL TEKNİK ÜNİVERSİTESİ
ELEKTRİK-ELEKTRONİK FAKÜLTESİ

YAPAY ZEKA UYGULAMALARI İÇİN RISC-V TABANLI ÇEKİRDEK TASARIMI VE GERÇEKLENMESİ

LİSANS BİTİRME TASARIM PROJESİ

Yusuf TEKİN
040200043
Ahmet Tolga ÖZKAN
040190067

Proje Danışmanı: Prof. Dr. Sıddıka Berna ÖRS YALÇIN

ELEKTRONİK VE HABERLEŞME MÜHENDİSLİĞİ BÖLÜMÜ

OCAK 2026

We are submitting the Senior Design Project Report entitled as “DESIGN AND IMPLEMETATION OF RISC-V BASED CORE FOR ARTIFICIAL INTELLIGENCE APPLICATIONS”. The Senior Design Project Report has been prepared as to fulfill the relevant regulations of the Electronics and Communication Engineering Department of Istanbul Technical University. We hereby confirm that we have realized all stages of the Senior Design Project work by ourselves and we have abided by the ethical rules with respect to academic and professional integrity .

Yusuf TEKİN .....
040200043

Ahmet Tolga ÖZKAN .....
040190067

FOREWORD

We would like to express our sincere gratitude to our advisor, Prof. Dr. Sıddıka Berna Örs Yalçın, for her invaluable guidance and support throughout this project. Her mentorship and the time she dedicated to us have had a significant impact not only on this thesis but also on our future careers. We are also grateful for the resources provided by Istanbul Technical University, which were essential for the realization of this work. Finally, we thank our families and loved ones for their patience and unwavering support.

January 2026

Yusuf TEKİN
Ahmet Tolga ÖZKAN

TABLE OF CONTENTS

Page

ABBREVIATIONS

AES	**:** Advanced Encryption Standard
AI	**:** Artificial Intelligence
ASIC	**:** Application-Specific Integrated Circuit
BRAM	**:** Block Random Access Memory
CCE	**:** Categorical Cross-Entropy
CNN	**:** Convolutional Neural Network
CPU	**:** Central Processing Unit
DNN	**:** Deep Neural Network
DoS	**:** Denial of Service
DSP	**:** Digital Signal Processing
FPGA	**:** Field-Programmable Gate Array
FPU	**:** Floating-Point Unit
FSM	**:** Finite State Machine
GCC	**:** GNU Compiler Collection
GNU	**:** GNU's Not Unix
HDL	**:** Hardware Description Language
IDS	**:** Intrusion Detection System
ISA	**:** Instruction Set Architecture
PE	**:** Processing Element
LUT	**:** Look-Up Table
MAC	**:** Multiply-Accumulate
MLP	**:** Multi-Layer Perceptron
MNIST	**:** Modified National Institute of Standards and Technology
NSL-KDD	**:** Network Security Laboratory - Knowledge Discovery in Databases
OS	**:** Operating System
R2L	**:** Remote to Local
RAM	**:** Random Access Memory
ReLU	**:** Rectified Linear Unit
RISC	**:** Reduced Instruction Set Computer
RISC-V	**:** Reduced Instruction Set Computer - V (Fifth Generation)
ROM	**:** Read-Only Memory
RTL	**:** Register-Transfer Level
TCL	**:** Tool Command Language
TFLu	**:** TensorFlow Lite Micro
U2R	**:** User to Root
UART	**:** Universal Asynchronous Receiver/Transmitter
USB	**:** Universal Serial Bus

SYMBOLS

α	**:** Alpha
β	**:** Beta
γ	**:** Gamma
ε	**:** Epsilon
θ	**:** Theta
μ	**:** Mu
σ	**:** Sigma
$\sum$	**:** Summation operator

LIST OF TABLES

Page

LIST OF FIGURES

Page

DESIGN AND IMPLEMENTATION OF RISC-V BASED CORE FOR ARTIFICIAL INTELLIGENCE APPLICATIONS

SUMMARY

In the era of the Internet of Things (IoT) and edge computing, the demand for deploying applications ns on resource-constrained devices has increased. Traditional cloud-based approaches for tasks such as Intrusion Detection Systems (IDS) often suffer from latency, bandwidth limitations, and privacy concerns. This thesis presents the design and implementation of a lightweight deep learning inference for network security applications on a RISC-V-based soft-core "Hornet". The primary objective is to validate the feasibility of running a complex, floating-point-heavy Multi-Layer Perceptron (MLP) on a soft-core processor within an FPGA environment.

The project utilizes the "Hornet" processor, a 32-bit RISC-V core originally designed at Istanbul Technical University. This core was enhanced with the Single-Precision Floating-Point ('F') extension (RV32IMF), which can support the precision required for neural network calculations. The hardware implementation was realized on a Xilinx Nexys Video FPGA board, featuring a custom-designed Wishbone bus system to integrate essential peripherals, including a 120,000-word memory subsystem, a Debug Interface, and a UART module for real-time communication.

On the software side, a deep learning model was developed using the NSL-KDD dataset to classify network traffic into five categories: Normal, DoS, Probe, R2L, and U2R. To address the dataset's severe class imbalance, a custom Focal Loss function was implemented in Python environment. Instead of using heavy frameworks like TensorFlow Lite Micro, a custom C inference engine was built from scratch for establishing the "minimal library" approach. Later in the project, some MLP layers are further optimized with a method called folding. The trained and folded weights and biases were extracted and compiled directly into the Hornet's memory.

Verification was conducted through a two-stage test suite. First, a behavioral simulation using Vivado XSim confirmed the functional correctness of the design. Second, a real-time hardware test suite was developed, where the test samples of the NSL-KDD dataset were transmitted from a host PC to the FPGA via UART. The hardware implementation achieved an accuracy of 78.87%, cathes the performance of the software model.

Lastly, an accelerator design is offered by lining up 16 MAC units and an additional memory to avoid memory bus latencies.

This study demonstrates that high-performance AI algorithms can be effectively deployed on open-source, memory-limited RISC-V hardware for secure and high-frequency edge devices.

RISC-V TABANLI YAPAY ZEKASI İŞLEMCİSİ TASARIMI VE GERÇEKLENMESİ

ÖZET

Nesnelerin İnterneti (IoT) ve sınır bilişim (edge computing) çağında, uygulamaların kaynak kısıtlı cihazlar üzerinde çalıştırılmasına yönelik talep artmıştır. Saldırı Tespit Sistemi (IDS) gibi görevler için kullanılan geleneksel bulut tabanlı yaklaşımlar, sıklıkla gecikme, bant genişliği sınırları ve gizlilik endişeleri gibi sorunlarla karşılaşmaktadır. Bu tez, ağ güvenliği uygulamaları için bir RISC-V yazılım tabanlı çekirdek (soft-core) olan "Hornet" üzerinde bir derin öğrenme tasarımını ve gerçeklemesini sunmaktadır. Temel amaç, karmaşık ve kayan nokta (floating-point) yoğunlukta bir Çok Katmanlı Perseptron (MLP) modelinin, bir FPGA ortamındaki yazılım tabanlı bir işlemci üzerinde çalıştırılabilirliğini doğrulamaktır.

Projede, aslen İstanbul Teknik Üniversitesi'nde tasarlanmış 32-bitlik bir RISC-V çekirdeği olan "Hornet" işlemcisi kullanılmıştır. Bu çekirdek, sinir ağı hesaplamaları için gereken hassasiyeti sağlamak amacıyla Tek Duyarlı Kayan Nokta ('F') eklentisi (RV32IMF) ile geliştirilmiştir. Donanım gerçeklemesi, 120.000 kelimelik bir bellek, Hata Ayıklama Arayüzü (Debug Interface) ve gerçek zamanlı iletişim için bir UART modülü dahil olmak üzere gerekli çevre birimlerini entegre eden özel tasarım bir Wishbone veri yolu sistemi kullanılarak Xilinx Nexys Video FPGA kartı üzerinde gerçekleştirilmiştir.

Yazılım tarafında, ağ trafiğini Normal, DoS, Probe, R2L ve U2R olmak üzere beş kategoriye sınıflandırmak için NSL-KDD veri seti kullanılarak bir derin öğrenme modeli geliştirilmiştir. Veri setindeki ciddi sınıf dengesizliğini gidermek için Python ortamında özel bir Odak Kayıp (Focal Loss) fonksiyonu uygulanmıştır. Gömülü sistem uygulaması için "minimal kütüphane" yaklaşımı benimsenmiştir. TensorFlow Lite Micro gibi ağır yapılar yerine, sıfırdan özel bir C çıkarım motoru inşa edilmiştir. Projenin ilerleyen aşamalarında, bazı MLP katmanları "katlama" (folding) yöntemiyle daha da optimize edilmiştir. Eğitilen ve katlanan ağırlıklar ile sapma değerleri çıkarılarak doğrudan Hornet'in belleğine derlenmiştir.

Doğrulama işlemi iki aşamalı bir test paketiyle gerçekleştirilmiştir. İlk olarak, Vivado XSim kullanılarak yapılan davranışsal simülasyon ile mantıksal doğruluk teyit edilmiştir. İkinci olarak, test örneklerinin bir bilgisayardan UART aracılığıyla FPGA'ya iletildiği gerçek zamanlı bir donanım testi geliştirilmiştir. Donanım gerçeklemesi %78,87 doğruluk oranına ulaşarak, yazılım modelinin performansını yakalamıştır.

Son olarak, bellek veri yolu gecikmelerini önlemek amacıyla 16 adet MAC (Çarp-Topla) birimi ve ek bir bellek sıralanarak bir hızlandırıcı tasarımı sunulmuştur.

Bu çalışma, yüksek performanslı yapay zeka algoritmalarının, güvenli ve yüksek frekanslı sınır cihazları için açık kaynaklı ve bellek kısıtlı RISC-V donanımları üzerinde etkin bir şekilde uygulanabileceğini göstermektedir.

1. INTRODUCTION

1.1 Purpose of the Project

The primary goal of this project is to show that any complex artificial intelligence algorithms can be run effectively on resource-constrained hardware by implementing a Multi-Layer Perceptron (MLP) for Intrusion Detection on a custom RISC-V processor ("Hornet"). This work aims to prove that critical security tasks can be performed locally on a chip without relying on cloud servers, thereby improving speed and privacy. The project shows that common software AI models (Python) can be replicated to low-level hardware execution (C and HDLs), and further can be accelerated.

The final version of the project is uploaded to GitHub [10].

1.2 Literature Review

Previous works have commonly handled AI workloads using either dedicated external accelerators [11] or by leveraging the RISC-V Vector (V) instruction set extension [12]. However, these studies are centered around Convolutional Neural Network (CNN) models for image processing tasks. While there exists some discussion regarding Multi-Layer Perceptron (MLP) implementation on soft-core processors [13], a complete FPGA-based implementation specifically tailored for real-time network security purposes has not yet been achieved.

A critical challenge in the domain of Intrusion Detection Systems (IDS) is the nature of the data itself. While the classification of network attacks can be effectively handled via Deep Neural Networks (DNN) [14], existing datasets often suffer from severe class imbalance issues [15]. This imbalance often leads to poor detection rates for minority attack classes in standard hardware implementations, necessitating specialized handling in both software training and hardware inference.

In this project, we utilize an MLP architecture, a fundamental neural network structure, to demonstrate that memory-intensive models can be effectively deployed for IDS on resource-constrained edge devices. Unlike commercial solutions that rely on hard-core processors (such as the ARM Cortex-A9 found in Xilinx Zynq SoCs), this work adopts an open-source RISC-V approach. This allows for precise customization, such as the inclusion of the Single-Precision Floating-Point (F) extension, ensuring that the processor is optimized specifically for the computational demands of the AI.

2. BACKGROUND INFORMATIONS

2.1 Deep Learning and Neural Networks

Since the early 1950s, classical machine learning algorithms have been employed to address a range of regression and classification problems. As famously observed by Gordon Moore in 1969, the number of transistors in a microchip, and thus the computational power, had nearly doubled every two years. This exponential growth in processing capabilities made mathematical computations, once considered impossible to realize, appear to be feasible through computers. This convergence led to the use of more complex algorithms, such as Gradient Descent and the Perceptrons, which effectively replaced the previous generation of simpler machine learning models. This ultimately triggered the development of the first true deep learning architecture: the Multi Layer Perceptron (MLP).

2.1.1 Multi Layer Perceptron Architecture

A perceptron (Figure 2.1) is a type of artificial neuron that was developed in the 1950s by Frank Rosenblatt. Although numerous efficient and accurate models are employed in contemporary research, the foundational principles of the Multi-Layer Perceptron (MLP) architecture remain valid. The activity of each neuron is determined by first calculating a weighted sum of its inputs and an additional (Equation 2.1)

$$z = \left(\sum_{j=1}^{n} x_j w_j \right) + b \tag{2.1}$$

Equation 2.1: Weighted Sum / Activation Potential (z)

where x_j is the j-th input, w_j is the corresponding weight, and b is the bias term.

This result is then passed through a non-linear activation function (such as ReLU or sigmoid) to introduce the non-linearity essential for modeling complex, non-linear patterns (Equation 2.2).

$$\text{output} = \sigma(z) = \frac{1}{1+e^{-z}} \tag{2.2}$$

Equation 2.2: Sigmoid Activation Function (σ)

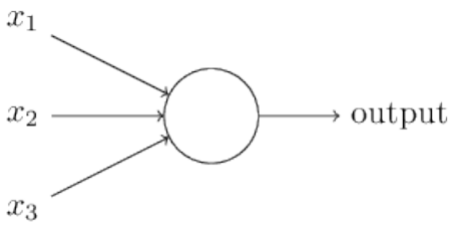

Figure 2.1 : A Single Perceptron Neuron [5].

An MLP processes input data across multiple layers of interconnected neurons (Figure 2.2). The network is typically composed of three main types of layers: Input, Hidden (Dense), and Output.

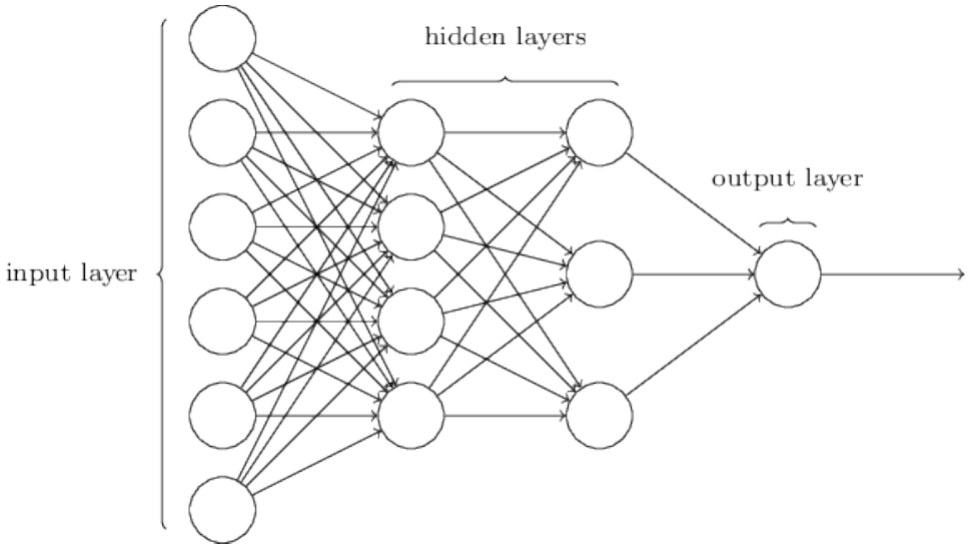

Figure 2.2 : Common MLP Structure [5].

2.1.1.1 Dense Layer

The Dense layer, also known as a fully connected layer, is the fundamental building block of an MLP. In this layer, every neuron is connected to every neuron in the preceding layer. Mathematically, the operation performed by a dense layer can be expressed as a matrix multiplication followed by vector addition (Equation 2.3).

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{2.3}$$

Where $\mathbf{x}$ is the input vector of size N, $\mathbf{W}$ is the weight matrix of size $M \times N$, $\mathbf{b}$ is the bias vector of size M, and $\mathbf{y}$ is the output vector of size M. This linear transformation allows the network to map input features into high-dimensional spaces where they can be separated.

2.1.1.2 Activation Functions

While the dense layer provides linear mapping, activation functions introduce the necessary non-linearity to the network.

Rectified Linear Unit (ReLU): The ReLU function is the most commonly used activation function in modern deep learning due to its computational efficiency and ability to mitigate the vanishing gradient problem [16]. It outputs the input directly if it is positive; otherwise, it outputs zero (Equation 2.4).

$$f(x) = \max(0, x) \tag{2.4}$$

In hardware implementations, ReLU is particularly advantageous as it requires only a simple comparator.

Softmax: The Softmax function is typically placed at the output layer of a classification model. It converts a vector of raw scores (logits) into a probability distribution, where the sum of all outputs equals 1. For a vector z of length K, the Softmax of the i-th component is shown in the Equation 2.5.

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.5}$$

This function ensures that the output represents the probability of the input belonging to a specific class.

2.1.1.3 Batch Normalization

Batch Normalization is a technique used to stabilize and accelerate the training of deep neural networks [16]. It addresses the problem of internal covariate shift, where the distribution of layer inputs changes during training. BN normalizes the output of a previous layer by subtracting the batch mean (μ_B) and dividing by the batch standard deviation (σ_B) as shown in the Equation 2.6.

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{2.6}$$

After normalization, the data is scaled and shifted using two parameters, γ (scale) and β (shift) (Equation 2.7).

$$y_i = \gamma \hat{x}_i + \beta \tag{2.7}$$

While crucial during training, Batch Normalization layers are often "folded" into the preceding Dense layer weights during inference to reduce computational overhead in embedded hardware.

2.1.2 Training and Optimization Methodology

The network is trained by iteratively adjusting the weights and biases. This process involves three critical components: Loss, Backpropagation, and Optimization.

2.1.2.1 Loss Function: Focal Loss

Training begins with a forward pass to generate a predicted output. The difference between this prediction and the true label is typically determined by a cost function. While standard Cross-Entropy loss is effective for balanced datasets, it often performs poorly on highly imbalanced data (like NSL-KDD). To address this, we utilize the Focal Loss function [17].

Focal Loss reshapes the standard cross-entropy loss by adding a modulating factor. Mathematically, for a predicted probability p_t focal loss function can be defined as shown in Equation 2.8.

$$FL(p_t) = -\alpha_t (1 - p_t)^{\gamma} \log(p_t) \tag{2.8}$$

Where γ, focusing parameter, adjusts the rate such that easy examples are down-weighted, and α_t balances class frequencies.

2.1.2.2 Backpropagation

Once the loss is calculated, the backpropagation algorithm efficiently calculates the gradient with respect to all weights and biases in the network. This gradient indicates

the direction and magnitude of the steepest ascent of the error surface (Figure 2.3). This allows the error to be propagated backward from the output layer to the input layer.

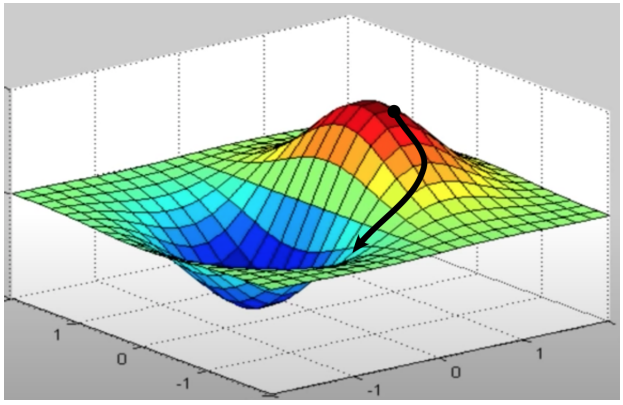

Figure 2.3 : Gradient Descent Optimization Visualization [6].

2.1.2.3 Advanced Optimization: Adam

To minimize the loss, the network utilizes the Adam (Adaptive Moment Estimation) optimizer instead of standard Stochastic Gradient Descent (SGD). Adam computes individual adaptive learning rates for different parameters from estimates of the first (m_t) and second (v_t) moments of the gradients [18]. The parameter update rule is given in Equation 2.9

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t \tag{2.9}$$

Where η is the learning rate and ε is a small scalar to prevent division by zero. This adaptive approach ensures faster convergence on the complex loss landscape of the IDS model.

2.1.2.4 Regularization: Dropout

To prevent overfitting, the Dropout regularization [19] is employed. During training, Dropout randomly sets a fraction p of the neurons to zero for each forward pass (Equation 2.10).

$$\tilde{\mathbf{h}} = \mathbf{r} \odot \mathbf{h}, \quad \text{where } \mathbf{r} \sim \text{Bernoulli}(p) \tag{2.10}$$

This forces the network to learn robust features that do not rely on specific individual neurons to prevent overfitting. During hardware inference, Dropout is deactivated.

2.1.3 Intrusion Detection System

An intrusion is defined as any unauthorized attempt towards the integrity, confidentiality, or availability of an information system. For instance, an attack designed to make a computer network, data server, or online service unresponsive to actual users is considered an intrusion. An Intrusion Detection System (IDS) is typically implemented as a software or hardware solution that designed to detect such malicious actions, thereby maintaining network security. [1]

IDS fundamentally represents a classification problem. During the detection process, network traffic and system parameters must be observed to identify whether an attack has occurred, and if so, to classify the attack type. Since every network presents a different number of monitored inputs, the features vary. Therefore, each system requires a unique approach (Table 2.1).

The Neural Network approach was selected for this project to validate the feasibility of implementing advanced stochastic algorithms on a memory-limited embedded device. The successful realization of such a stochastic method demonstrates that any other deterministic or low-cost stochastic detection algorithm can also be efficiently realized.

2.1.4 RISC-V

The Reduced Instruction Set Computer (RISC) is a CPU architecture designed to simplify the instruction set provided to the processing unit to execute computational tasks [20]. RISC-V is an open and free standard Instruction Set Architecture (ISA), originating from the University of California, Berkeley in 2010 as the fifth generation of RISC designs developed there since 1981. The establishment of the global RISC-V Foundation in 2015 significantly promoted the widespread dissemination of technical materials, specifications, and standardization across academic and research institutions.

The foundation of the entire RISC-V architecture, RV32I Base Integer ISA, is a mandatory subset that defines the minimum required instructions and features for any general-purpose RISC-V implementation, thereby ensuring binary compatibility across different hardware platforms. Key characteristics of RV32I include:

Table 2.1 : Comparison of different detection approaches (A: Anomaly-based, M: Misuse-based) [1]

Approach	**A/M**	**Accuracy**	**Completeness**	**Known attacks**	**Unknown attacks**	**Masquerade**	**Denial of service**	**Malicious use**	**Leakage**	**Attempted breakings**
Statistical	A	Low if threshold too low	Low if threshold is high	Yes	Yes	Yes	No	Yes	No	–
Predictive pattern generation	A	Low if event sequence is not listed	Low if event sequence is not listed	Yes	Yes	Yes	No	Yes	No	–
Neural networks	A	Low if window is small	Low if window is large	Yes	Yes	Yes	No	Yes	No	–
Sequence matching and learning	A	Low	High	Yes	Yes	Yes	No	Yes	No	–
Expert systems	M	High	Low	Yes	No	No	No	Yes	No	No
Model-based	M	High	Low	Yes	No	No	No	Yes	No	–
State transition analysis	M	High	Low	Yes	No	No	No	Yes	No	No
Pattern matching	M	High	Low	Yes	No	No	No	Yes	No	Yes

- **32-bit Architecture:** Specifies a 32-bit address space and memories, whereas a register file out of 32 registers.
- **Fundamental Operations:** Includes basic arithmetic (ADD, SUB), logical (AND, OR, XOR), load/store, and branching instructions required for all computational tasks.

The versatility required of a general-purpose CPU has driven the continued evolution of RISC-V through modular extensions to the base ISA [21]. These extensions are named according to the specialized capabilities they add:

- **Integer Multiplication and Division ('M' Extension):** This extension introduces specific instructions tailored for efficient multiplication and division operations.
- **Single-Precision Floating-Point ('F' Extension):** This extension provides the hardware support necessary to perform floating-point calculations with increased precision and speed. Prior to this, such computations were often approximated using fixed-point arithmetic, which compromises range and accuracy.

2.1.4.1 The Requirement Of Floating-Point Extension ('F') In This Project

The 'F' extension is crucial in applications that require high precision, particularly in the context of AI and signal processing. For example, in a DSP application or any Deep Learning model, weights, biases, and activation outputs are typically stored as floating-point numbers.

Table 2.2 : Comparison of Floating-Point Multiplier Delay Results on FPGAs [2]

Reference	FPGA family	Existing unit delay	Proposed unit delay
[22]	Spartan 6	49.797 ns	20.105 ns
[23]	Virtex 4	18.783 ns	16.721 ns
[24]	Spartan 3	29.72 ns	22.597 ns
[25]	Spartan 3E	34.333 ns	31.578 ns

Implementing the 'F' extension in a pipelined RISC-V design (Figure 2.5) enables accurate weight and bias calculations at high speed. (Table 2.2) presents results from various studies, clearly illustrating that the integration of a dedicated Floating-Point Unit (FPU) (Figure 2.4) significantly reduces the unit delay. This reduction in latency is crucial for models and functions that rely on floating-point arithmetic. By achieving

these shorter delays, the overall system maintains higher performance, thereby enabling it to meet the timing requirements of real-time applications.

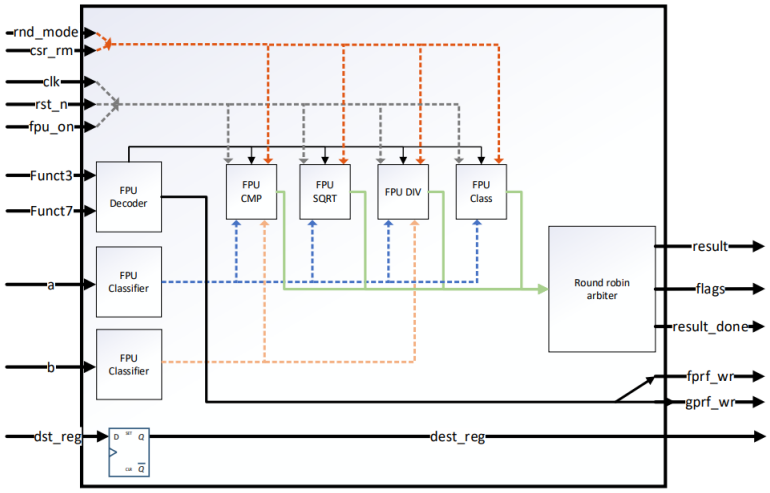

Figure 2.4 : A FPU design [7].

In this project, a RISC-V core with FPU is implemented to create an efficient simulation environment.

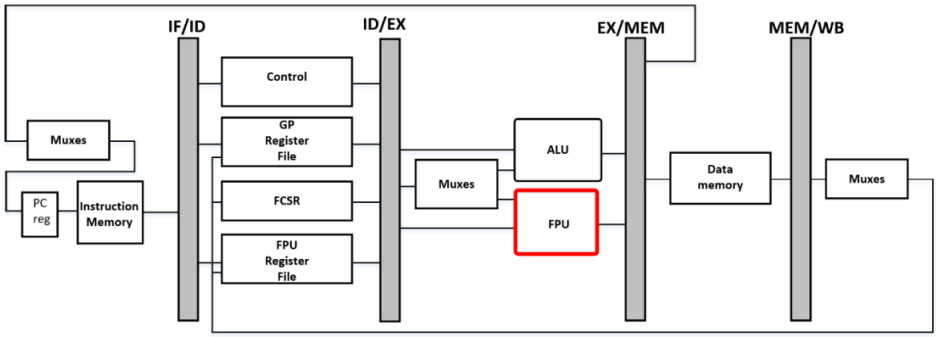

Figure 2.5 : FPU implementation on a RISC-V architecture [7].

2.1.5 HORNET

Hornet is a 32-bit processor that supports the RISC-V I and M instruction sets, meaning that it can process basic integer arithmetic and multiplication, and division. It was designed by students from Istanbul Technical University [26]. Later, it was updated to have the F extension, meaning that it has support to perform operations using floating point numbers [27]. Finally, the F extension, thereby the functionality and timing analysis of the FPU, was verified and tested completely according to the RV32IMF standards [28].

2.1.6 UART

Universal asynchronous receiver/transmitter (UART) is a widely used peripheral in digital design. It is one of the earliest and easiest-to-implement communication protocols, thereby a common element in the integrated circuit industry. Even today, it remains useful in many applications, such as serving as the default USB protocol for FPGAs.

The basic job of the UART is to organize parallel data for serial transmission and then reassemble it upon reception. It does this by wrapping each data byte in extra bits, a process called framing. The most common format, often called 1-8-1 (Figure 2.6), is shown below:

- **Start Bit (1 bit):** A single bit placed at the beginning of the data. This bit signals the receiving device that a new data frame is about to start. It acts as an alert.
- **Data Bits (8 bits):** These are the actual 8 bits of information being sent. In the 1-8-1 format, the data is transmitted starting with the Least Significant Bit (LSB) first.
- **Stop Bit (1 bit):** A final bit added at the end of the data. This signals that the entire frame is complete, allowing the receiver to prepare for the next start bit.

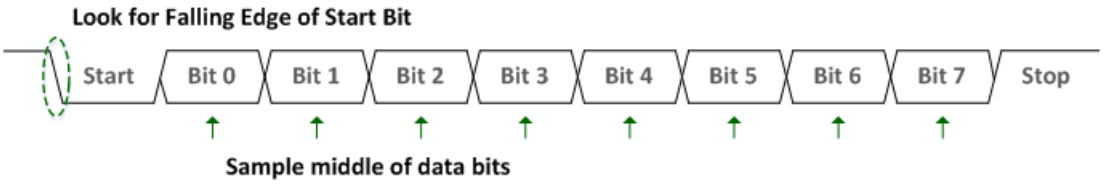

Figure 2.6 : 8-bit UART protocol [8].

2.2 Field-Programmable Gate Array (FPGA) and Toolchain

A Field-Programmable Gate Array (FPGA) is a type of reprogrammable integrated circuit that serves as a highly flexible prototyping platform for digital designs. Unlike rigid Application-Specific Integrated Circuit (ASIC) designs, an FPGA offers a cost-effective alternative by preferring an architecture based on the Look-Up Table (LUT). In the context of combinational logic, the LUT effectively functions as the electronic implementation of a truth table.

To put it simply, any behavior achieved by interconnecting logic gates (such as AND, NOR, etc.) can be efficiently realized by a LUT. Consequently, FPGAs provide the designer with the crucial option of having fully re-programmable logic, offering immense post-manufacturing flexibility. In this project, a Nexys Video FPGA Board (Figure 2.6) is used to implement the design.

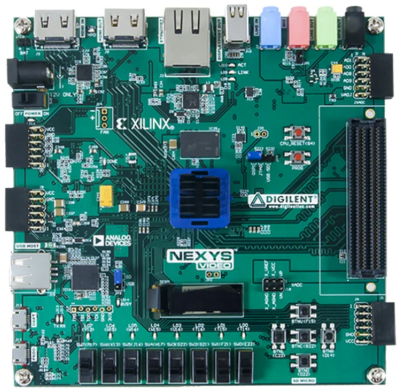

Figure 2.7 : Digilent Nexys Video FPGA Board [9].

Xilinx is a semiconductor company renowned for inventing the first commercially viable field-programmable gate array. The company primarily supplies programmable logic devices and an integrated design suite for synthesis, routing, and implementation. In this project, a Xilinx FPGA-based board, Nexys Video, is utilized for hardware implementation and testing of the RISC-V core. The entire design flow, from RTL to bitstream generation, is managed using the Vivado Design Suite.

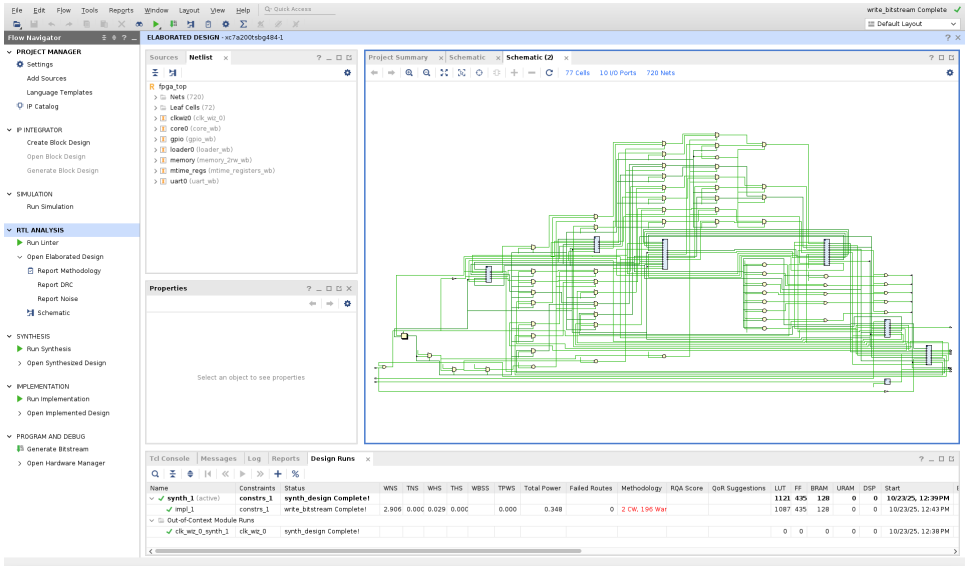

Figure 2.8 : Vivado Design Suite.

The Vivado Design Suite (Figure 2.8) is a toolchain that handles the entire digital design flow, including Register-Transfer Level (RTL) simulation, synthesis, implementation (placing and routing), and bitstream generation. Throughout this project, Vivado has been utilized for both the behavioral simulation and verification of the design, as well as the final physical realization of the design on the FPGA. Specifically, Version 2025.1 was chosen for all development and testing phases.

3. SOFTWARE LEVEL EVALUATION OF MLP

In this chapter, the design and implementation of the MLP will be discussed in three sections: designing the model using Python, implementing the model inference in C, and preparing the final RISC-V machine code. Subsequently, the creation of both software and hardware simulation environments for verifying this neural network model will be covered.

However, later on in the project (Chapter 5), during the hardware acceleration attempt, the initial model revealed a significant bottleneck that resulted in an inefficient and overly complex accelerator design. To address this, the software model was redesigned to eliminate the need for costly calculations on the hardware. As this optimization represented a major turning point in the project, both the initial (unfolded) and the final (folded) designs are presented to highlight the critical impact of algorithmic optimizations on hardware complexity.

3.1 Software Design: Python

Using Python libraries like TensorFlow (Keras), Scikit-learn (Sklearn), and Pandas is the most widely recognized and industry-standard approach for training neural networks. TensorFlow provides robust optimization and modeling tools, while Scikit-learn offers essential functionalities for data composition, scaling, encoding, and standard performance reporting.

A practical advantage of employing Python is its cross-platform support. The scripts and commands demonstrated below function consistently across cloud servers (such as Google Colab and Jupyter Notebooks) and various operating systems (like Windows, Ubuntu, and Fedora). The directory structure and terminal commands provided are based on our Ubuntu 22.04 LTS development system. For Windows users, references to `python3`/`pip3` should typically be substituted with `python`/`pip`.

Required installations:

```
pip3 install pandas
```

```
pip3 install numpy
pip3 install tensorflow
pip3 install matplotlib
pip3 install pydot
sudo apt install graphviz
```

The complete Python code can be found in the references as a GitHub link. [29].

3.1.1 Dataset and Data Preprocessing

This study's experiments are conducted using the NSL-KDD dataset, a refined version of the original KDD Cup 1999 dataset. The NSL-KDD dataset addresses critical issues, such as the problem of redundant records and excessive class imbalance, found in its predecessor, making it more suitable for evaluating intrusion detection models.

The dataset contains 41 input features extracted from network traffic, categorized into basic, content-based, time-based, and host-based features. Each record is labeled either as a normal connection or as one of various types of attacks.

3.1.1.1 Attack Categories

The malicious activities in the dataset are categorized into four major families [30], each representing a different attack vector and intent:

- **Denial of Service (DoS):** is an attack type where the attacker makes some computationaly expensive, resource-intensive requests, thereby the network becomes too busy to handle regular requests (e.g., *neptune*, *smurf*).
- **Probing (Probe):** An information-gathering attack to bypass the security of a network (e.g., *nmap*, *ipsweep*).
- **Remote to Local (R2L):** happens when an attacker does not have an account on the machine but sends packets over the network to exploit a vulnerability to gain local user access (e.g., *guess_passwd*, *ftp_write*).
- **User to Root (U2R):** The attacker starts with a normal user account and exploits system vulnerabilities to gain root/superuser privileges (e.g., *buffer_overflow*, *rootkit*). Even though some methods are offered to improve the detection of U2R attacks [3], it's often the hardest attack type to detect.

To process the dataset, each attack type is assigned to an attack family as it shown in the (Table 3.1). Between the test and train datasets, the attack types differ to simulate the real-life data realistically.

Table 3.1 : Taxonomy of Attacks in NSL-KDD: Training vs. Test Sets [3]

Attack Family	**Training Set (Known Attacks)**	**Test Set Only (Novel Attacks)**
DoS (Denial of Service)	Back, Land, Neptune, Pod, Smurf, Teardrop	Apache2, Mailbomb, Processtable, Udpstorm, Worm
Probe (Surveillance)	IPsweep, Nmap, Portsweep, Satan	Mscan, Saint
R2L (Remote to Local)	Ftp_write, Guess_passwd, Imap, Multihop, Phf, Spy, Warezclient, Warezmaster	Httptunnel, Named, Sendmail, Snmpgetattack, Snmpguess, Sqlattack, Xlock, Xsnoop
U2R (User to Root)	Buffer_overflow, Loadmodule, Perl, Rootkit	Ps, Xterm

3.1.1.2 Data Distribution

The `KDDTrain+.csv` file is used for model training, and the `KDDTest+.csv` file is used for testing and validation. The data exhibits significant class imbalance between the majority classes (Normal and DoS) compared to the minority classes (U2R and R2L). This severe imbalance causes low accuracy for standard classification algorithms, motivating the need for advanced techniques such as the Focal Loss function.

NSL-KDD	**Total**	**Normal**	**Others**
KDDTrain+	125,973	67,343	58,630
KDDTest+	22,543	9,710	12,833

Table 3.2 : Distribution of normal and other samples in NSL-KDD dataset

The total distribution of the NSL-KDD dataset for binary and multiclass classification is presented in Table 3.2 and Table 3.3, respectively. As the tables illustrate, while Normal and DoS dominate both sets, minority classes like R2L and U2R are significantly underrepresented, particularly in the training data. Therefore, specialized preprocessing and training techniques are required.

Class	Training Set	Percentage	Test Set	Percentage
Normal	67,343	53.458%	9,710	43.075%
DoS	45,927	36.458%	7,457	33.080%
Probe	11,656	9.253%	2,421	10.740%
R2L	995	0.790%	2,754	12.217%
U2R	52	0.041%	200	0.888%
Total	**125,973**	**100%**	**22,543**	**100%**

Table 3.3 : NSL-KDD train and test data distribution [4]

3.1.2 Loss Function and Optimization

In MLPs, the loss function plays a crucial role in the learning process by measuring the difference between the model's predictions and the true labels. During training, the MLP updates its weights to minimize this loss (using algorithms like Gradient Descent), which enables the network to generalize patterns in the data.

The Categorical Cross-Entropy (CCE) loss is commonly used to measure prediction errors for classification problems. It quantifies the difference between the predicted probability distribution and the true distribution. The CCE loss is expressed as it shown in the (Equation 3.1).

$$\mathscr{L}_{CE} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{3.1}$$

Where C is the number of classes, y_i is the true label (represented as a one-hot vector), and $\hat{y}_i$ is the model's predicted probability for class i.

However, the NSL-KDD dataset displays a severe class imbalance. In such scenarios, standard CCE loss may lead the model to focus disproportionately on the majority classes (Normal and DoS), resulting in poor generalization and poor performance on critical minority classes (R2L and U2R).

Even if it is not the point of this project, to overcome this, Focal Loss (Equation 3.2) is employed. Originally proposed for object detection [31], Focal Loss is specialized for highly imbalanced datasets and helps the model learn harder-to-classify minority samples.

$$\mathscr{L}_{FL} = -\alpha_t(1-\hat{y}_t)^{\gamma}\log(\hat{y}_t) \tag{3.2}$$

where $\hat{y}_t$ is the estimated probability of the model for the true class t, α_t is a weighting factor (tuned in the Python script) to address the class imbalance, and γ is the focus parameter that adjusts the rate at which easy examples are downweighted. Demonstrating how the focus parameter (γ) down-weights the loss contributed by easy samples ($\hat{y}_t \to 1$) is shown in the (Figure 3.1) [31].

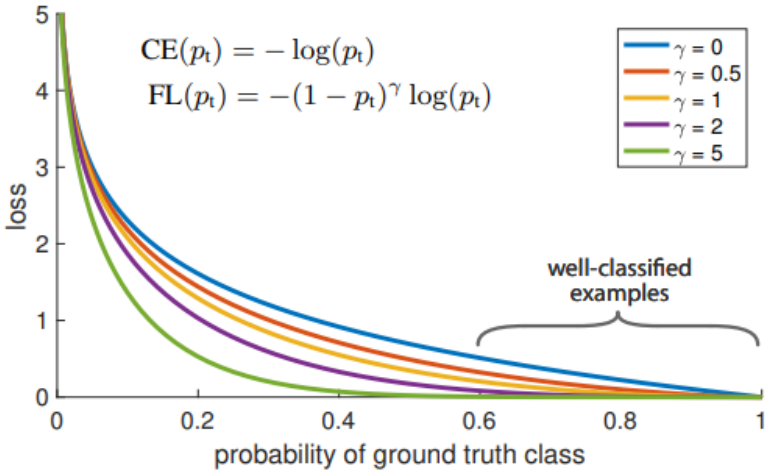

Figure 3.1 : How focal loss parameters affect the function.

The implementation of the custom Focal Loss function with respect to the Equation (3.2) in TensorFlow/Keras is shown below:

```
def focal_loss(gamma=2., alpha=None):
    alpha = tf.constant(alpha, dtype=tf.float32)

    def loss_fn(y_true, y_pred):
        epsilon = tf.keras.backend.epsilon()
        y_pred = tf.clip_by_value(y_pred, epsilon, 1. - epsilon)

        cross_entropy = -y_true * tf.math.log(y_pred)
        alpha_weight = tf.reduce_sum(alpha * y_true, axis=1,
        keepdims=True)
        focal_weight = tf.math.pow(1 - y_pred, gamma)
        loss = alpha_weight * focal_weight * cross_entropy

        return tf.reduce_mean(tf.reduce_sum(loss, axis=1))

    return loss_fn
```

3.1.3 Model Architecture and Training

The MLP model was constructed using the Sequential API from the Keras framework. The objective was to design a lightweight yet deep architecture suitable for deployment on the resource-constrained Hornet RISC-V platform [29].

The model consists of an input layer followed by four hidden layers, reducing in size from 256 to 32 neurons, and terminates with a 5-neuron output layer (corresponding to the five attack classes). Crucially, Batch Normalization and Dropout layers were integrated after each hidden layer to stabilize training, accelerate convergence, and prevent overfitting. The architecture is defined as follows (Figure 3.2):

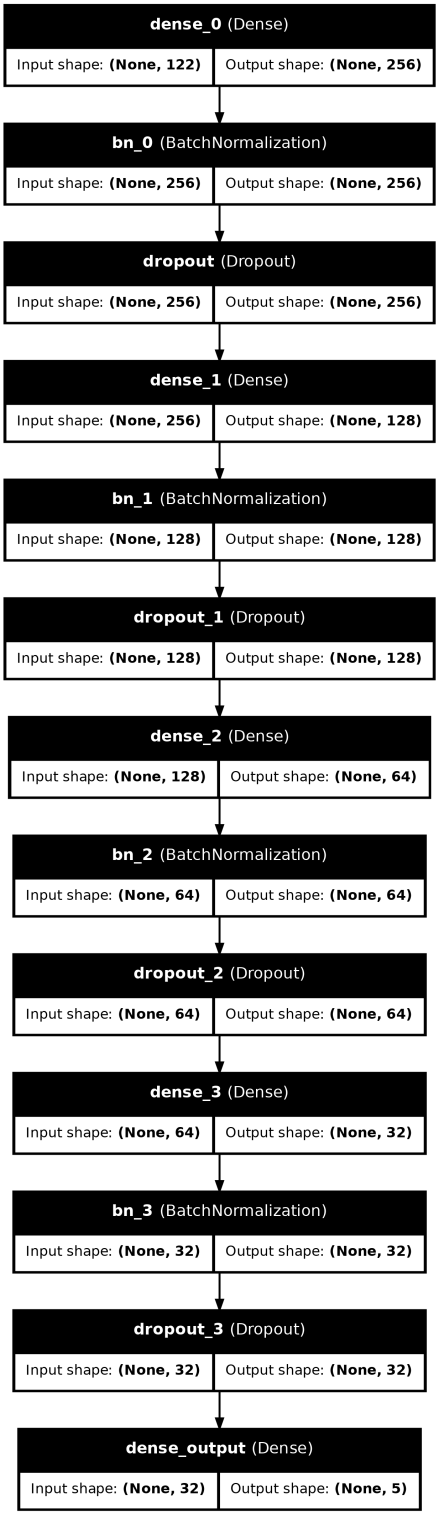

Figure 3.2 : Architecture of the MLP model for Intrusion Detection.

The model was compiled and trained using the Adam optimizer, a standard choice for deep learning models. As previously detailed in the loss function discussion, the custom Focal Loss function was employed. The training hyperparameters were set to optimize performance on the challenging minority classes:

```
model.compile(
    optimizer=Adam(learning_rate=1e-4),
    loss=focal_loss(gamma=2., alpha=[0.2, 0.25, 3.0, 1.0, 0.15]),
    metrics=['accuracy']
)

# Training execution
history = model.fit(
    X_train_processed, y_train_cat,
    validation_split=0.1,
    epochs=100,
    batch_size=512,
    verbose=2
)
```

The network was trained for 100 epochs with a batch size of 512, using 10% of the training data for validation. Following training, the model's performance was evaluated on the test set.

```
# Evaluation on Test Data
y_pred_probs_mlp = model.predict(X_test_processed, verbose=0)
y_pred_classes_mlp = np.argmax(y_pred_probs_mlp, axis=1)
y_true_classes_mlp = y_test_encoded
```

This evaluation stage generates the necessary data for computing the Precision, Recall, and F1-score metrics, which are common metrics for classification tasks.

3.1.4 Evaluation Metrics and Performance Analysis

This section serves to establish the golden model for the project. The outcomes obtained here will act as the benchmark for comparing the results of the C implementation and the post-hardware verification.

To evaluate the performance of the MLP model, the predictions are analyzed using a Confusion Matrix and several derivative metrics. The primary metrics utilized in this study are Precision, Recall, and the F1-Score.

Precision measures the accuracy of positive predictions, quantifying the proportion of true identifications relative to all positive predictions. It is calculated as shown in (Equation 3.3) where TP refers to True Positive, and FP refers to False Positive.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3.3}$$

Recall assesses the model's ability to identify all relevant instances within the dataset. It is defined as the ratio of true positives to the total number of actual positive samples, as expressed in (Equation 3.4) where TP refers to True Positive, and FN refers to False Negative.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3.4}$$

Finally, the **F1-Score** represents the harmonic mean of Precision and Recall. It provides a single metric that balances both concerns, offering a robust measure for performance on imbalanced datasets. The formula is provided in (Equation 3.5).

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.5}$$

The overall classification results are summarized in (Table 3.4), and the specific prediction counts are detailed in the Confusion Matrix in (Table 3.5).

Table 3.4 : MLP Classification Performance Report on the Test Set

Class (ID)	Precision	Recall	F1-Score	Support
0 (DoS)	0.9657	0.8369	0.8967	7,460
1 (Probe)	0.8531	0.6981	0.7678	2,421
2 (R2L)	0.9396	0.1241	0.2192	2,885
3 (U2R)	0.6667	0.2388	0.3516	67
4 (Normal)	0.6879	0.9700	0.8050	9,710
Weighted Avg.	**0.8297**	**0.7863**	**0.7550**	**22,543**

The overall model accuracy of 78.63% appears reasonable, but further investigation on the other metrics points out:

- **Minority Class Performance:** Classes 2 (R2L) and 3 (U2R) have the lowest Recall scores (0.1241 and 0.2388, respectively). This is expected due to the extreme underrepresentation of these attack types in the training data.
- **Majority Class Performance:** Classes 0 (DoS) and 4 (Normal) achieve high Recall (0.8369 and 0.9700), indicating the model is highly effective at identifying common and normal traffic.

A common visualisation method of neural networks, the confusion matrix (Table 3.5), illustrates the Actual/Predicted ratio for each attack type.

Table 3.5 : Confusion Matrix for MLP Test Predictions

Actual \Predicted	0 (DoS)	1 (Probe)	2 (R2L)	3 (U2R)	4 (Normal)
0 (DoS)	6243	49	0	0	1168
1 (Probe)	165	1690	0	0	566
2 (R2L)	0	20	358	3	2504
3 (U2R)	0	0	16	16	35
4 (Normal)	57	222	7	5	9419

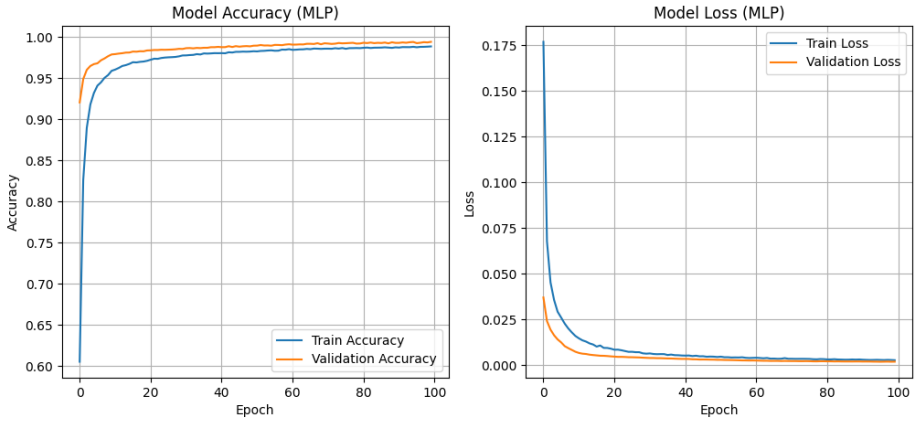

Figure 3.3 : Model Training History: Accuracy and Focal Loss during 100 epochs.

The (Figure 3.3) illustrates the convergence of both training and validation loss, confirming that the model did not suffer from significant overfitting, thereby rationalizing the use of `KDDTest+.csv`.

3.2 Software Design: C

To utilize the Python/TensorFlow model on the target hardware, it must be converted to C. A common tool for this purpose is TensorFlow Lite Micro (TFLu), a library from TensorFlow designed for embedded systems. This library is designed to be more effective for models such as CNNs in general.

However, TFLu was not used for this MLP design. The decision was based on avoiding the extra libraries and complexity that TFLu brings. A primary goal was to use as few C libraries as possible. This "minimal library" approach is crucial for generating the final machine code, as explained in the next section.

3.2.1 Custom C Functions

To build the MLP in C without including the standard C math library (`math.h`), simple math functions were written first. Replacements were needed for the exponential (`expf`) and square root (`sqrtf`) functions, as they are required for Softmax and Batch Normalization.

Approximations were defined for these functions. The `expf_approx` function uses a Taylor series expansion to calculate e^x, while `sqrtf_approx` uses the Babylonian method (a form of Newton's method) to find a square root.

In addition to the math functions, all core neural network layer functions were also written by hand. This includes:

- `relu`: The activation function.
- `dense_affine`: Calculates the weighted sum for a dense layer.
- `bn_infer`: Performs the Batch Normalization calculation.
- `softmax_stable`: The final activation function for classification.

For the final hardware test setup, simple functions for UART (to send and receive data) and Interrupts (to handle events) were also required. This test setup will be discussed in a later chapter.

3.2.2 Parameters and Test Data

An important point is that the neural network is not trained on the final RISC-V chip. The chip only runs the finished model (a process called inference).

To get the model data, two functions from the Python script are used:

1. **extract_and_print_c_parameters**: This function takes the trained weights, biases, and batch normalization parameters from TensorFlow and saves them as simple C arrays in flattened format. These are stored in a header file (e.g., `MLP_weightsBN_light.h`) to be compiled with the C code.

2. **save_full_testset**: To verify that the C code is working correctly, this function saves the test inputs (from `inputs.txt`) and the correct answers (from `labels.txt`) from the dataset.

3.2.3 MLP Inference Function in C

The main C function that runs the entire MLP is `model_infer`. This function replicates the exact steps of the Python model, but it uses only the simple, custom-written C functions.

The function takes one input vector `x` (representing one network packet with 122 features) and passes it through each layer sequentially.

The process inside `model_infer` is as follows:

1. **Define Buffers**: First, temporary arrays (buffers) are declared to hold the output values of each layer.
2. **Layer 0**: The dense layer calculation (`dense_affine`) is performed, the `relu` activation is applied, and the result is run through `bn_infer` (Batch Normalization).
3. **Layers 1-3**: This process is repeated for the next three hidden layers, feeding the output of the previous layer into the input of the next.
4. **Layer 4 (Output)**: A final `dense_affine` calculation is performed, and the `softmax_stable` function is applied to get the final five class probabilities.
5. **Argmax**: Finally, a loop iterates through the five output probabilities to find the index (0-4) with the highest value. This index is the predicted class, which is returned by the function as an integer.

3.2.4 C Code Verification

To verify the C implementation at the software level, a `main` function was created. This function's purpose is to feed every input vector from `inputs.txt` into the `model_infer` function. The resulting predictions are written to a separate file, `label_results.txt`. This text file is then compared against the ground-truth `labels.txt` file using a simple Python script, `compare.py`.

Before this verification can be run, the C source code must be compiled into an executable file. This is done using the GNU Compiler Collection, or `gcc`. The `gcc` command processes the human-readable `.c` files (which contain the `model_infer` function and others) and translates them into machine code that the processor can understand. A typical compilation command on a Linux system is:

```
gcc inference.c -o inference_test
```

This executable is then run from the terminal (`./inference_test`) to generate the `label_results.txt` file for accuracy checking.

As shown in the terminal output below, the accuracy of the final C code is 78.37%. This represents a minor error ratio of 0.33% from the original Python model's accuracy. This small error rate is an expected and acceptable trade-off, resulting from the use of custom, low-iteration math approximations (like `expf_approx`) instead of the high-precision `math.h` library. An error rate of 3 in a thousand is considered negligible in exchange for significantly lower memory (RAM of RISC-V core) usage and the prevention of potential stack overflows.

```
Toplam 22543 ornek islendi.
Sonuclar label_results.txt dosyasina yazildi.

UNIX> python3 compare.py
Correct: 17668/22543
Accuracy: 78.37%
```

3.3 Machine Code Generation with RISC-V GNU Toolchain

The C code developed in the previous section (`model_infer`) must be compiled into machine code that the Hornet processor understands. This requires a specific cross-compiler. The RISC-V GNU Toolchain [32] is an open-source suite of tools that includes `gcc` and `binutils` (the assembler and linker) configured for the RISC-V architecture.

The following steps simplifies the complex procedure of the installation.

3.3.1 Prerequisites

Before building, the system needs essential build tools. These can be installed on Ubuntu with the following command:

```
sudo apt-get install git autoconf automake autotools-dev curl \
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential \
bison flex texinfo gperf libtool patchutils bc zlib1g-dev \
libexpat-dev
```

3.3.2 Installation Steps

The installation is performed in three main stages: downloading, configuring, and building.

1. **Download the Source Code**: The toolchain is cloned from its official GitHub repository. After cloning, the submodules must also be downloaded for the sake of simplicity and time efficiency.

```
git clone https://github.com/riscv/riscv-gnu-toolchain
cd riscv-gnu-toolchain
git submodule update --init --recursive
```

2. **Configure the Build**: This is the most critical step. The `./configure` script is used to specify the processor's architecture. For this project, a "Newlib" toolchain is built with multilib support for the RV32IMF architecture and the ILP32F Application Binary Interface (ABI).

```
./configure --prefix=/opt/riscv --with-abi=ilp32f
--with-arch=rv32imf --enable-multilib
```

   - `-prefix=/opt/riscv`: This sets the installation directory. `/opt/riscv` is a common location for custom-built tools.
   - `-enable-multilib`: Allows the toolchain to build libraries for multiple architecture variations, improving flexibility.
   - `-with-arch=rv32imf`: Specifies the default target architecture (RV32 + Integer + Multiply + Single-Float).
   - `-with-abi=ilp32f`: Specifies the default ABI (Integer/Long/Pointers are 32-bit, Floating-point uses hardware registers).

3. **Compile the Toolchain**: The `make` command is used to compile the entire toolchain. This process is very long, often taking an hour or more. To speed it up, the `-j` flag is used to run multiple compilation jobs in parallel, based on the number of available threads.

```
# Example: Use 8 threads (adjust the number as needed)
make -j8

# Or, automatically use all available processor cores
make -j$(nproc)
```

4. **Update the System Path**: After the build finishes, the `PATH` variable must be updated so the terminal can find the new compiler.

```
# Add this line to the end of your ~/.bashrc file
export PATH="$PATH:/opt/riscv/bin"

# Apply the changes to the current terminal session
source ~/.bashrc
```

3.3.3 Compiling the Firmware with a Makefile

After installing the toolchain, the C code (`model_infer`) must be compiled. This entire process, which turns the C files into machine code for the RISC-V processor, is automated using a `Makefile`.

A `Makefile` is a script that runs a list of terminal commands. The one used for this project defines the compiler, the C files to include (like `inference_light.c`, `uart.c`, `irq.c`, and the `crt0.s` startup file), and the rules (flags) for compilation.

`-ffp-contract=off` is the most important flag here and must always be included in the `Makefile` due to the lack of support for several instructions on Hornet, such as fused multiply-add operations. Hornet handles such instructions seperately

The `Makefile` used for inference:

```
CC32=riscv32-unknown-elf
CCFLAGS=-march=rv32imf -mabi=ilp32f -ffp-contract=off
-mstrict-align -malign-data=xlen -O3 -fno-math-errno
-T linksc.ld -lm -nostartfiles -ffunction-sections
-fdata-sections -Wl,--gc-sections -o inference_light.elf

build:
    $(CC32)-gcc inference_light.c uart.c irq.c crt0.s $(CCFLAGS)
    $(CC32)-objcopy -O binary -j .init -j .text -j .rodata -j
    .sdata inference_light.elf inference_light.bin
    ../../rom_generator inference_light.bin
    cp inference_light.data
    ../../memory_contents/memory_init.mem
```

This `Makefile` automates four main steps to create the final `memory_init.mem` file:

1. **Compile to an ELF file**: The first command runs `riscv32-unknown-elf-gcc`. This is the RISC-V compiler. It takes all the source files (`.c` and `.s`) and compiles them into a single file called `inference_light.elf`. This ELF file contains the compiled code and information about where each part should go in memory. This memory layout is defined by the `linksc.ld` file, which is called a linker script.

2. **Extract the Binary File**: The second command uses `objcopy`. This tool strips away all the extra information from the ELF file. It copies only the raw machine code instructions into a new file, `inference_light.bin`. This is the pure, runnable program.

3. **Generate the ROM File**: The third command runs a custom utility called `rom_generator`. This script reads the `inference_light.bin` file and converts the binary data into a text file of hexadecimal (hex) numbers. This hex format is required by the FPGA's memory (BRAM) and is named `inference_light.data`.

4. **Copy the Final File**: The last step simply copies the new hex file (`inference_light.data`) into the hardware simulation folder and renames it `memory_init.mem`.

4. DESIGNING A TEST SUITE

In this project, the neural network's hardware implementation was verified using two different methods.

The first method, behavioral simulation, provided a straightforward approach for single-input functional verification. However, this approach was found to be impractical for large-scale testing, as a single input/output simulation required as long as 30 minutes to complete on XSim Vivado.

Therefore, to validate the design against the full 22,543-sample test dataset, a second approach, a UART-based real-time test suite, was developed. Before detailing the testbenches, the hardware peripherals required for both the real-time suite and the behavioral simulation must be discussed.

Out of the box, Hornet offers 5 peripherals: Debug Interface, Loader, Memory, MTime Registers, and UART. For further debugging, it has a Tracer block that offers a continuous scan of core wires. To connect these peripherals, a simple Wishbone data bus was implemented.

The Wishbone Bus is one of the basic data bus designs commonly used in computer systems. It follows a standard Master-Slave protocol, which expects peripherals as slaves and the main controller (the RISC-V core, in this case) as master. The slave choices differ for the two separate test designs.

Before diving into the test suite designs, some alterations and definitions are required for several peripherals.

4.1 Peripherals and Tweaks Made

In this section, the changes that had to be made among the peripherals and drivers are discussed.

4.1.1 Debug Interface

The Debug Interface (Figure 4.1) is not a module that can be synthesized and implemented on an FPGA; it is a simulation-only module because it uses Verilog system tasks. Hence this peripheral is used only for behavioral simulation test.

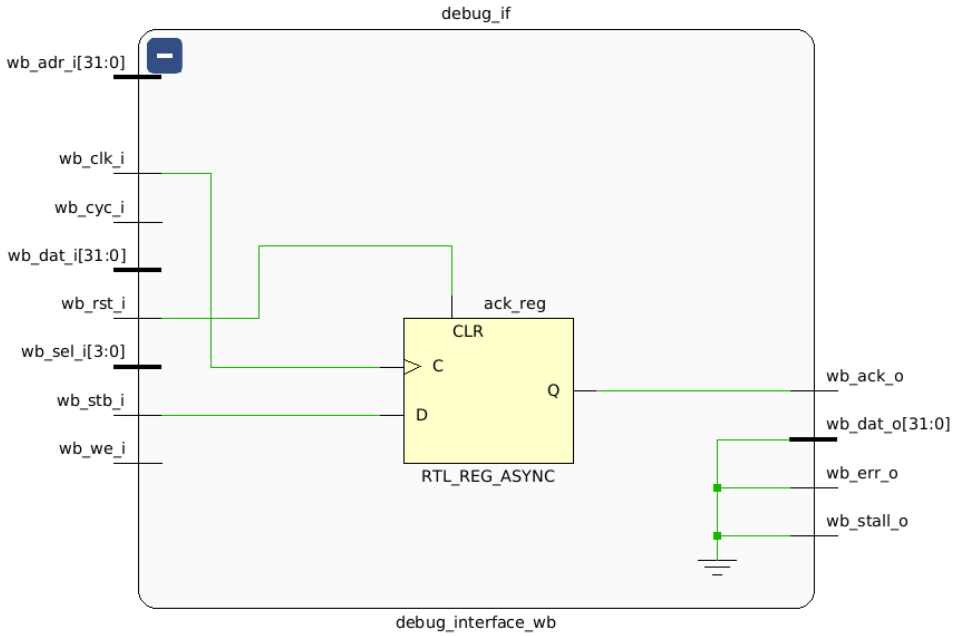

Figure 4.1 : Debug Interface Module of Hornet.

Finishing the behavioral simulation and printing whether the result is correct or not via the TCL Console is the main purpose of this module. To be able to do so, some parameters and design methods must be implemented in the design and the C code:

The debug interface has an address vector defined in the top module (`barebones_wb_top.v`) to be triggered by the core whenever a write instruction is sent to its specific address:

```
assign slave_adr_begin[3] = 32'h1000_8010;
assign slave_adr_end[3]   = 32'h1000_8010;
```

On the C code side, the checking process is done in the `main` function:

```
#define DEBUG_IF_ADDR 0x10008010

// ...

char *addr_ptr = (char*)DEBUG_IF_ADDR;
if(result == 0) *addr_ptr = 1;
else *addr_ptr = 0;
```

This process checks the inference result and, if it is correct (equal to 0 in this test), it writes a '1' to the memory-mapped address for the debug interface. When the core writes to this address, the debug module is triggered, which then halts the simulation and reports the success or failure.

4.1.2 Memory

The memory peripheral (Figure 4.2) must be compatible with the two test methodologies. However, the out-of-the-box memory configuration of the Hornet processor was insufficient for this project's needs. The generated machine code for the MLP inference (77,524 words) and the required run-time RAM (estimated at $\approx 40,000$ words) far exceeded the original BRAM size. Therefore, significant modifications to both the Verilog hardware modules and the software build system were required.

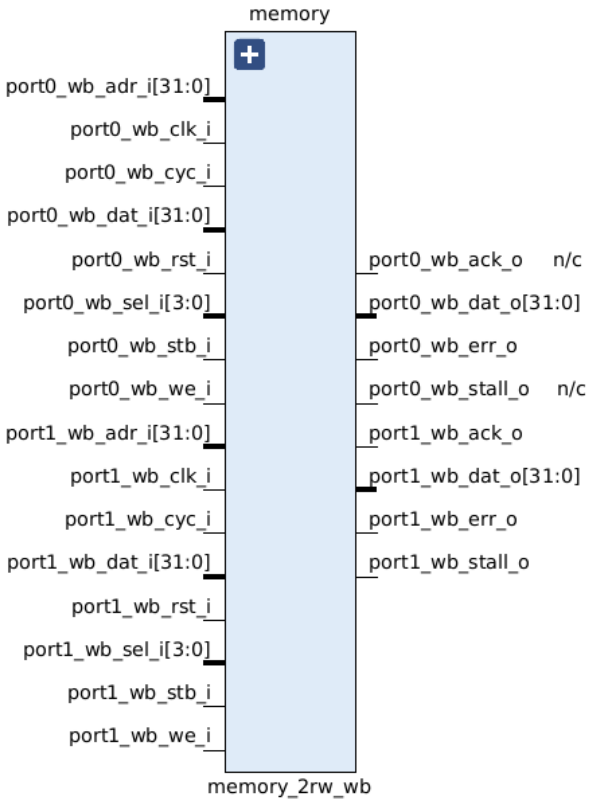

Figure 4.2 : HORNET Wishbone Memory Subsystem.

First, the memory and initialization parameters in the Verilog code were altered. The original Hornet design used a top-level parameter for `ADDR_WIDTH` and derived the `RAM_DEPTH` from it (e.g., as $2^{\text{ADDR\_WIDTH}}$). This method of using an exponential function was found to cause synthesis errors in newer versions of Vivado.

To fix this, the logic was inverted:

1. `RAM_DEPTH` was made the primary parameter, set directly to 120,000 words.
2. `ADDR_WIDTH` is now derived from `RAM_DEPTH` using the Verilog `$clog2` system function. This is more robust and synthesizes correctly.

This 120,000-word depth safely accommodates the 77,524-word machine code (ROM) and the $\approx 40,000$ words of data RAM, with a small buffer to ensure the process.

Next, the software build tools and drivers were updated to match this new hardware size. The `rom_generator` utility, which creates the memory initialization file, was modified to handle a 120,000-entry memory.

Finally, the linker script (`linksc.ld`) was adjusted to match this new hardware size. The memory management in the linker script is critical, as it must align with the hardware's word-based memory map. The linker, however, uses byte-addressing. Since one 32-bit word equals four bytes, the memory lengths defined in the linker script must be exactly four times the register count defined in Verilog. The linker's `MEMORY` definition, which maps the software sections into the hardware's address space, was configured as follows:

```
OUTPUT_ARCH(riscv)
MEMORY
{
    ROM(RX)   : ORIGIN = 0x00000000, LENGTH = 0x00050000
    RAM(WAIL) : ORIGIN = 0x00050000, LENGTH = 0x0001FFFC
}
```

This configuration sets the `ROM` section at the start of the address space with a length of 320KB, and the `RAM` section immediately after it, with a length of 128KB. These partitions can be thought of as instruction and data memory in order.

After this memory expansion, the memory-mapped addresses for other peripherals (like the Debug Interface and UART) had to be re-aligned, as their original addresses were now located inside the new, larger BRAM block.

4.1.3 UART

The UART peripheral (Figure 4.3) was discussed in detail in (Section 2). For the second test method, on-hardware verification, UART was chosen due to its wide support in FPGAs, and Hornet comes with a UART peripheral. However, several tweaks were required before using the UART. First, the transmitter and receiver pins had to be constrained according to the `fpga_top.v` file to align with the Nexys Video's reference manual (Figure 4.4).

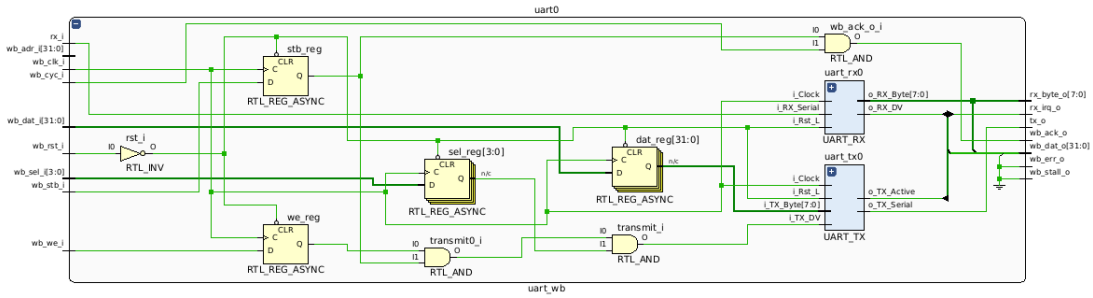

Figure 4.3 : UART of Hornet as a peripheral.

For the C implementation, the drivers (`uart.c`, `uart.h`, `irq.c`, `irq.h`) define the rx/tx functions, supported by Assembly lines for synchronization using stalls (NOP: No Operation). Moreover, these driver files are later referenced in the `Makefile` for the compiling process.

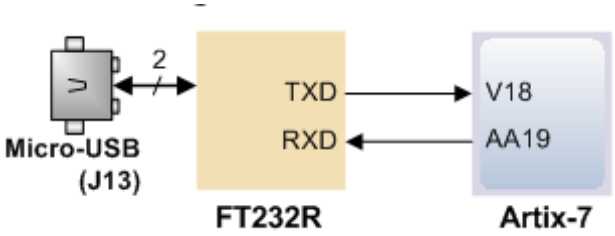

Figure 4.4 : UART of the Nexys Video [9].

4.2 Behavioral Simulation Using XSim

After the brief summary of the peripherals, the first test method is discussed in this section. The behavioral simulation is designed as illustrated in (Figure 4.5). Unlike `TRACER` (which was used only for early debugging), `MEMORY_2RW` and `DEBUG`

INTERFACE are the key components in this section. The debug address is set to 0x10008010 in both the C code and the Verilog top module.

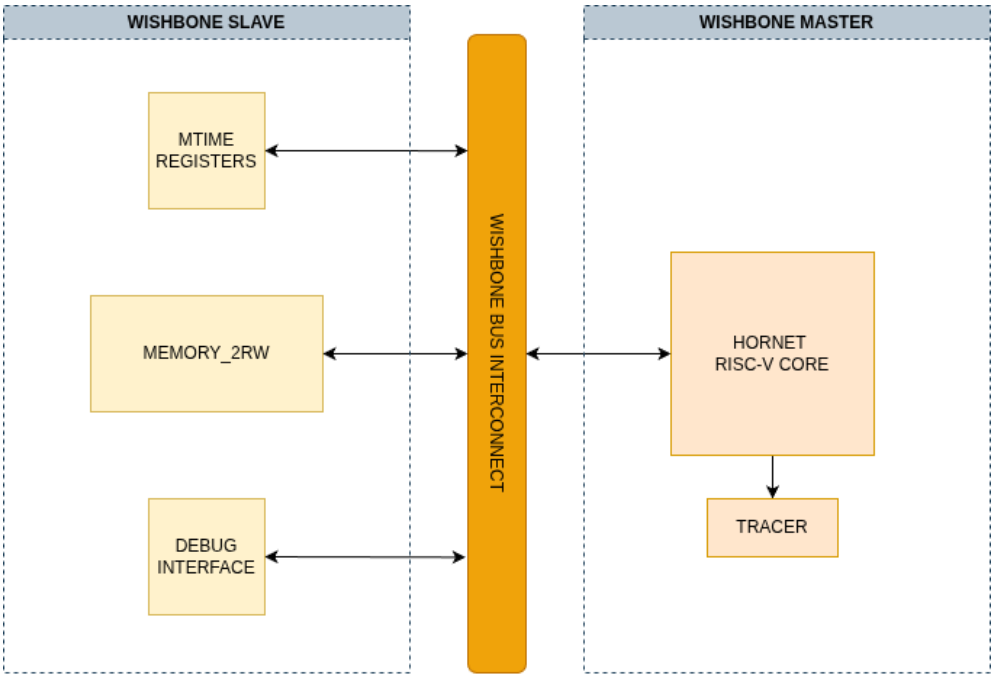

Figure 4.5 : High-Level Block Diagram of the Simulation Design.

After connecting the debug interface and designing the main C function accordingly, the XSim environment (Figure 4.6) was ready. This simulation was run five times, once for a sample of each attack type (Normal, DoS, Probe, R2L, and U2R), to confirm the core logic was working as expected.

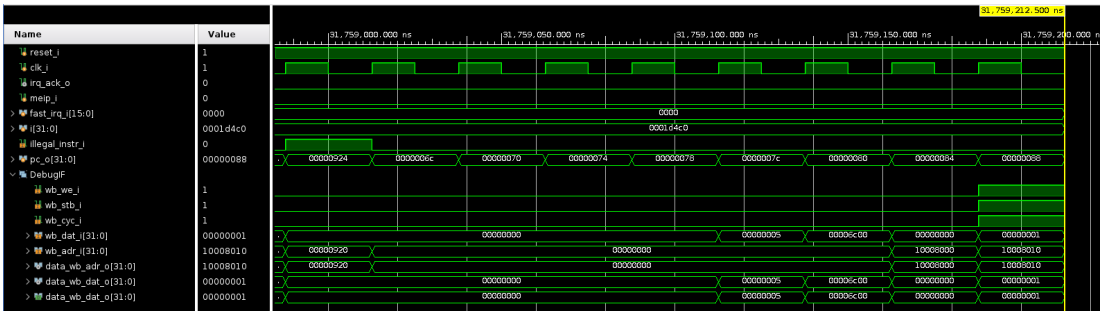

Figure 4.6 : XSim Waveform for a Single Input Test.

The simulation was successful, with the Debug Interface module printing a "Success!" message. This printing behaviour is triggered when the three wishbone signals wb_we_i, wb_stb_i, and wb_cyc_i go high all at once. However, the simulation's performance metrics highlighted a significant bottleneck:

- **Real-World Time:** It took 57 seconds of real-world time for the host computer (with 12 threads) to complete the simulation for a single input.
- **Simulation Time:** The actual Hornet simulation time was 31.76 ms (31,759,212,500 ps).
- **Host Resources:** The simulation process was resource-intensive, consuming over 11 GB of peak memory on the host machine.

While this method confirmed the design's functional correctness, the time per input (nearly one minute) made it impractical for testing the full 22,543-sample dataset. It would take around 374 hours (or $\approx 15,5$ real-life days) to complete all of the data. Additionally, storing ($22{,}543 \times 122 \times 4$) bytes more data for all the inputs would cause another memory problem. These limitations were the primary motivation for developing a clever engineering approach: the real-time hardware test suite.

4.3 On-Hardware Verification via UART

Implementing designs on hardware presents both distinct advantages and several challenges. To address these issues step-by-step, a BRAM-rich FPGA was required; therefore, the Nexys Video board was selected. It is worth noting that after successful debugging, the MLP was also verified on the Nexys 4 DDR, an entry-level educational FPGA. However, for the sake of simplicity and continuity in this report, all results presented are from the Nexys Video.

Before discussing the technical implementation, specific challenges must be addressed. First, the Hornet core with the FPU extension had never been implemented on an FPGA. Previously, it was tested in a verification environment and verified solely in simulation (FPGA implementation was only done before the addition of the FPU). Consequently, the previous clock frequency of 40 MHz might require adjustment. Furthermore, the UART peripheral had not been tested in conjunction with the FPU. Additionally, parameters such as the baud rate and OS-level USB drivers (Windows/Linux) lacked prior testing or documentation for this specific configuration.

To manage these ambiguities, the problems were tackled one at a time. This led to the use of intermediate test systems: `aes_main` and `MNIST`. AES [33] is a symmetric

encryption algorithm that had already been tested with Hornet's M extension using UART. This provided a baseline test that was memory-dependent and used UART, but did not require the FPU. MNIST [34], on the other hand, is a standard dataset used for introduction-level Convolutional Neural Network (CNN) training. While MNIST was previously used to verify Hornet in simulation, here it served to debug FPU-UART compatibility within a smaller memory footprint, verifying the system's flexibility.

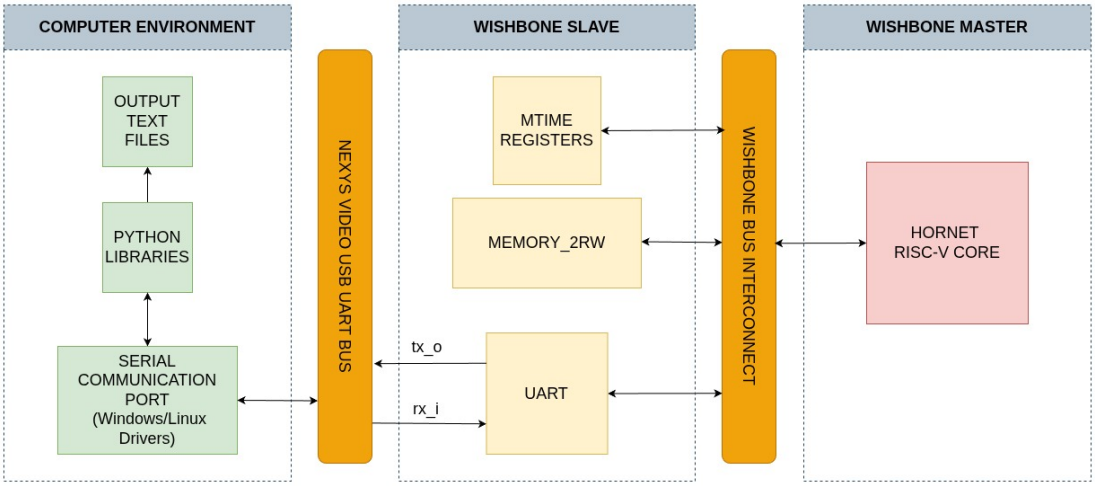

Figure 4.7 : Hardware Simulation Topology

Upon the completion of these preliminary tests, the environment—now supporting large memory, FPU operations, and UART communication—was ready for the main test: the MLP. Throughout all the main and preliminary tests, the topology shown in (Figure 4.7) is used.

4.3.1 Preliminary Tests

USB ports on computers provide users with several communication protocols, and one of them is the serial port. To develop projects around these serial ports, the Python library `pyserial` provides robust solutions. This library makes reading and writing the serial bus of the USB possible by simplifying the calculation and syncing the timing constraints, like baud rate and delay. Therefore, for all the tests in this section, Python codes are arranged around this library.

4.3.1.1 AES Encryption Test

The first preliminary test utilized the Advanced Encryption Standard (AES) [33]. AES is a widely used symmetric encryption algorithm that operates heavily on integer arithmetic (especially multiplication) and memory operations.

This test was selected to isolate and verify the UART and Wishbone Memory subsystems without engaging the Floating-Point Unit (FPU). A dedicated C firmware was developed to execute these functions. During implementation, a Vivado version dependency issue arose during memory synthesis. Consequently, the address width definition was altered from an exponential format (`RAM_DEPTH = 1 << ADDR_WIDTH`) to a logarithmic approach using the Verilog system function (`ADDR_WIDTH = $clog2(RAM_DEPTH)`).

4.3.1.2 MNIST Character Recognition Test

Following the integer-based AES test, the MNIST dataset [34] was employed to verify the Floating-Point Unit (FPU). MNIST is a standard benchmark for handwritten digit recognition. A small-scale Convolutional Neural Network (CNN) was implemented in C to classify these digits.

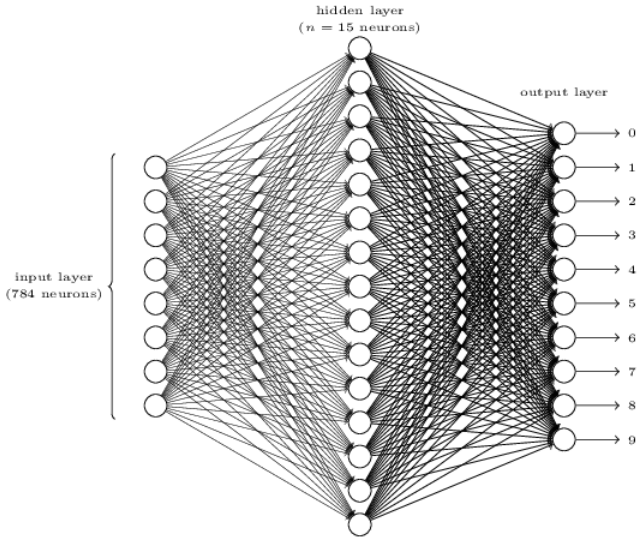

Figure 4.8 : MNIST Neuron Structure

This test served as a bridge between AES and MLP. Because the MNIST model (Figure 4.8) is relatively small, it fits easily within the Hornet processor's memory, allowing the verification focus to remain solely on the correctness of the FPU implementation.

During this phase, even though it is tested in simulation, the original built-in clock frequency of 40 MHz was found to cause timing failures post-implementation, likely due to the increased logic depth of the new FPU hardware. Consequently, the system clock frequency required reduction. Leveraging the fact that standard UART baud rates typically share integer multiple relationships (powers of two), the clock frequency was systematically stepped down by factors of two first to 20 MHz, and then to 10 MHz. Both frequencies were tested and confirmed to operate correctly. For the final verification phase, the system was standardized on the fastest stable frequency, 20 MHz.

4.3.2 Validation and Verification using Python Libraries

After successfully implementing the UART and interrupt drivers and verifying the C implementation of the MLP function, the design phase was complete. The final stage involved two custom Python scripts: one for transmitting the test data and receiving results in real-time, and another for the comprehensive comparison of the output files.

The first script, utilizing the `argparse`, `time`, and `sys` libraries, transmits the 22,543 input vectors from `inputs.txt` to the FPGA via UART. It displays the inference results in the terminal in real-time and writes the complete set of predictions to `FPGA_results.txt`.

```
...
[22535] true=0  pred=0  ->  [PASS]
[22536] true=0  pred=4  ->  [FAIL]
[22537] true=0  pred=0  ->  [PASS]
[22538] true=4  pred=4  ->  [PASS]
[22539] true=4  pred=4  ->  [PASS]
[22540] true=0  pred=0  ->  [PASS]
[22541] true=4  pred=4  ->  [PASS]
[22542] true=1  pred=1  ->  [PASS]

=== Summary ===
Total tested: 22543
Correct: 17666
Wrong:   4877
Accuracy: 78.37%
Total time: 2523.20 seconds
Results saved to FPGA results.txt
```

This test requires configuring parameters such as the baud rate and transmission delay. The full test suite completed in approximately 2,523.20 seconds (approx. 42 minutes and 3 seconds), achieving an overall accuracy of 78.37%.

The second script performs the detailed evaluation using `sklearn.metrics`. It compares three key files to validate the entire pipeline:

- `labels.txt`: The ground truth labels (correct answers) from NSL-KDD dataset.
- `label_results.txt`: The results from the local C simulation (software-only verification).
- `FPGA_results.txt`: The results obtained from the actual FPGA.

The results of these comparisons are summarized below.

4.3.2.1 Comparison 1: Local C Simulation vs. Ground Truth

This comparison verifies the logic of the C implementation against the established ground truth labels. The local C code achieved an accuracy of 78.53%, which closely tracks the baseline accuracy of 78.63% established by the original Python model. Other metrics (Table 4.1), however, display no difference.

Table 4.1 : Local C Simulation Performance

Class	Precision	Recall	F1-Score	Support
0 (DoS)	0.97	0.84	0.90	7,460
1 (Probe)	0.83	0.70	0.76	2,421
2 (R2L)	0.95	0.12	0.22	2,885
3 (U2R)	0.76	0.42	0.54	67
4 (Normal)	0.69	0.96	0.80	9,710
Weighted Avg	**0.83**	**0.79**	**0.75**	**22,543**

4.3.2.2 Comparison 2: FPGA Hardware vs. Ground Truth

The FPGA implementation achieved an accuracy of 78.37%. With a total execution time of approximately 40 minutes, the hardware demonstrates results nearly identical to those of the software-based Python model. Due to the single/double precision difference between RISC-V and regular x86 CPUs that run the C code, an expected difference is observed for accuracy and recall metrics, while others maintain the same results (Table 4.2).

Table 4.2 : FPGA Hardware Performance

Class	Precision	Recall	F1-Score	Support
0 (DoS)	0.96	0.84	0.90	7,460
1 (Probe)	0.85	0.68	0.75	2,421
2 (R2L)	0.92	0.12	0.21	2,885
3 (U2R)	0.64	0.27	0.38	67
4 (Normal)	0.69	0.97	0.80	9,710
Weighted Avg	**0.83**	**0.78**	**0.75**	**22,543**

4.3.2.3 Comparison 3: Hardware Fidelity (Local C vs. FPGA)

Finally, comparing the local C simulation directly to the FPGA output determines the hardware fidelity, or how accurately the FPGA reproduces the software logic. This comparison yielded an overall accuracy of 98.54%.

The minor deviation from the local C simulation was initially investigated as a potential data capture issue on the UART interface. However, tests run with varying baud rates produced identical results. If the error were caused by UART data handling, the results would likely fluctuate with transmission speed. This consistency suggests that the errors stem from the hardware itself, likely due to slight differences in floating-point handling or a systematic error caused by interrupt functions. This specific topic warrants investigation in future work.

Overall, the results indicate that the hardware implementation is a success, with negligible errors.

5. FINAL ADJUSTMENTS FOR HARDWARE ACCELERATOR

After achieving a successful implementation, the next step was to design a hardware accelerator to improve inference performance. However, previous model appears to be overly complicated for an accelerator design. In this chapter, we propose an optimized approach for both C and Python software optimization. Specifically, we apply a technique known as "Folding" to the Python training model, which effectively offers removing complex arithmetic operations from the final C/hardware inference code without trading accuracy.

5.1 The Problem: Batch Normalization

Standard Deep Learning models typically employ `Dense` → `Batch Normalization` → `Activation`. However, in the initial design (Figure 5.1, Top), this was not the case. So the C code below must change.

```
// L0: Dense (122->256) -> ReLU -> BN
dense_affine(x, INPUT_DIM, layer0_weights,
             layer0_biases, L0_OUT, z0);
for (int i=0; i<L0_OUT; ++i) a0[i] = relu(z0[i]);
bn_infer(a0, y0, bn0_gamma, bn0_beta, bn0_mean,
             bn0_var, bn0_eps, L0_OUT);

// L1: Dense (256->128) -> ReLU -> BN
dense_affine(y0, L0_OUT, layer1_weights,
             layer1_biases, L1_OUT, z1);
for (int i=0; i<L1_OUT; ++i) a1[i] = relu(z1[i]);
bn_infer(a1, y1, bn1_gamma, bn1_beta, bn1_mean,
             bn1_var, bn1_eps, L1_OUT);

// L2: Dense (128->64) -> ReLU -> BN
dense_affine(y1, L1_OUT, layer2_weights,
             layer2_biases, L2_OUT, z2);
for (int i=0; i<L2_OUT; ++i) a2[i] = relu(z2[i]);
bn_infer(a2, y2, bn2_gamma, bn2_beta, bn2_mean,
             bn2_var, bn2_eps, L2_OUT);

// L3: Dense (64->32) -> ReLU -> BN
dense_affine(y2, L2_OUT, layer3_weights,
             layer3_biases, L3_OUT, z3);
for (int i=0; i<L3_OUT; ++i) a3[i] = relu(z3[i]);
bn_infer(a3, y3, bn3_gamma, bn3_beta, bn3_mean,
             bn3_var, bn3_eps, L3_OUT);
```

```
// L4: Dense (32->5) -> Softmax
dense_affine(y3, L3_OUT, layer4_weights,
             layer4_biases, L4_OUT, logits);
softmax_stable(logits, L4_OUT, out_probs);
```

While Batch Normalization (BN) is essential during the training phase to stabilize convergence and prevent internal covariate shift, it poses significant challenges for hardware acceleration on edge devices.

The standard BN operation for a single neuron output z is defined as Equation 5.1.

$$y = \gamma \left(\frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}} \right) + \beta \tag{5.1}$$

Where:

- μ and σ^2 are the batch mean and variance (computed during training).
- γ and β are the learned scale and shift parameters.
- ε is a small constant for numerical stability.

From a hardware perspective, implementing Equation 5.1 directly is highly inefficient. It requires the hardware to perform a division and a square root operation for every single neuron in the layer. In digital logic design, division and square root units are computationally expensive, and can last multiple clock cycles to complete. Furthermore, implementing these operations in floating-point arithmetic requires complex FPU modules in a pipeline, which drastically complicates the accelerator design.

5.2 The Solution: Folding

To overcome these hardware limitations, we utilize a technique called `Folding`. This method exploits the fact that during the inference phase, the network weights are frozen. Consequently, the batch statistics (μ, σ) and the learnable parameters (γ, β) become constant values.

First of all, to implement folding, the order of the batch normalization must be set so that batch normalization and dense layer are next to each other. So in training:

```
# Define the MLP architecture
model = Sequential([
```

```
    Input(shape=(X_train_processed.shape[1],), name='input_layer'),

    # Layer 0
    Dense(256, use_bias=True, name='dense_0'),
    BatchNormalization(name='bn_0'),
    Activation('relu', name='act_0'),
    # Activation happens AFTER BN
    Dropout(0.4),

    # Layer 1
    Dense(128, use_bias=True, name='dense_1'),
    BatchNormalization(name='bn_1'),
    Activation('relu', name='act_1'),
    Dropout(0.3),

    # Layer 2
    Dense(64, use_bias=True, name='dense_2'),
    BatchNormalization(name='bn_2'),
    Activation('relu', name='act_2'),
    Dropout(0.2),

    # Layer 3
    Dense(32, use_bias=True, name='dense_3'),
    BatchNormalization(name='bn_3'),
    Activation('relu', name='act_3'),
    Dropout(0.1),

    # Output Layer (Softmax)
    Dense(num_classes, activation='softmax', name='dense_output')
])
```

After the fix of the model, the model can be fold.

Since the Dense layer operation ($z = \mathbf{w}x + b$) and the Batch Normalization operation are both linear transformations, they can be mathematically collapsed into a single linear operation. This allows us to "fold" the BN parameters directly into the weights and biases.

Substituting the Dense layer equation into the BN equation:

$$y = \gamma \left(\frac{(\mathbf{w}x + b) - \mu}{\sqrt{\sigma^2 + \varepsilon}} \right) + \beta \tag{5.2}$$

By rearranging the terms, we can isolate the coefficient for x and the constant terms:

$$y = \underbrace{\left(\frac{\gamma}{\sqrt{\sigma^2 + \varepsilon}} \mathbf{w} \right)}_{\mathbf{w}_{folded}} x + \underbrace{\left(\frac{\gamma}{\sqrt{\sigma^2 + \varepsilon}} (b - \mu) + \beta \right)}_{\mathbf{b}_{folded}} \tag{5.3}$$

This derivation yields the new "folded" parameters:

$$\mathbf{w}_{folded} = \frac{\gamma}{\sqrt{\sigma^2 + \varepsilon}} \cdot \mathbf{w} \tag{5.4}$$

$$\mathbf{b}_{folded} = \beta + \frac{\gamma(b - \mu)}{\sqrt{\sigma^2 + \varepsilon}} \tag{5.5}$$

By pre-calculating $\mathbf{w}_{folded}$ and $\mathbf{b}_{folded}$ in Python before exporting the model weights, the hardware inference structure simplifies to Equation 5.6.

$$y = \mathbf{w}_{folded} x + \mathbf{b}_{folded} \tag{5.6}$$

Python calculations and writing output for C/hardware:

```
def save_folded_c_parameters(model, filename="MLP_weights_folded.h"):
    """
    Performs Batch Normalization Folding:
    Merges BN parameters (gamma, beta, mean, var) into the preceding
    Dense layer's Weights and Biases.

    Formula:
      W_new = W_old * (gamma / sqrt(var + eps))
      b_new = (b_old - mean) * (gamma / sqrt(var + eps)) + beta
    """

    guard_name = filename.upper().replace(".", "_").replace("/", "_")
    output_lines = [
        f"#ifndef {guard_name}",
        f"#define {guard_name}",
        "\n// MLP FOLDED Weights (BN merged into Dense) -
        Generated by Python",
        "// format: float32",
        "\n"
    ]

    # Map pairs of (Dense, BN).
    layer_pairs = [
        ('dense_0', 'bn_0', 'layer0'),
        ('dense_1', 'bn_1', 'layer1'),
        ('dense_2', 'bn_2', 'layer2'),
        ('dense_3', 'bn_3', 'layer3')
    ]

    for i, (dense_name, bn_name, c_prefix) in enumerate(layer_pairs):
        print(f"Folding layer {i}: {dense_name} + {bn_name}...")

        # Get raw parameters
        dense_layer = model.get_layer(dense_name)
        bn_layer = model.get_layer(bn_name)

        # W shape: (input, output), b shape: (output,)
        W, b = dense_layer.get_weights()
```

```
        gamma, beta, mean, var = bn_layer.get_weights()
        epsilon = bn_layer.epsilon

        # --- CALCULATE FOLDING ---
        # 1. Calculate the scale factor: gamma / sqrt(var + epsilon)
        scale = gamma / np.sqrt(var + epsilon)

        # 2. Fold Weights: W_new = W * scale
        # We need to broadcast 'scale' across the rows of W
        W_new = W * scale

        # 3. Fold Biases: b_new = scale * (b - mean) + beta
        b_new = scale * (b - mean) + beta

        # --- WRITE TO STRING ---
        output_lines.append(
            f"// --- Layer {i} Folded ({dense_name} + {bn_name}) ---"
        )
        output_lines.append(
            format_array_to_c_style(W_new, f"{c_prefix}_weights")
        )
        output_lines.append(
            format_array_to_c_style(b_new, f"{c_prefix}_biases")
        )

    # --- Handle Output Layer (No Folding) ---
    print("Exporting output layer (no folding)...")
    out_layer = model.get_layer('dense_output')
    W_out, b_out = out_layer.get_weights()
    output_lines.append(f"// --- Layer 4 Output (No BN) ---")
    output_lines.append(format_array_to_c_style(W_out,
                        "layer4_weights"))
    output_lines.append(format_array_to_c_style(b_out,
                        "layer4_biases"))

    output_lines.append(f"#endif // {guard_name}")

    with open(filename, "w") as f:
        f.write("\n".join(output_lines))

    print(f"\n[SUCCESS] Folded parameters saved to: {filename}")

# Execute
save_folded_c_parameters(model, "MLP_weights_folded.h")
```

As illustrated in Figure 5.1 (Bottom), this eliminates the Batch Normalization layer entirely from the hardware data path. The FPGA no longer needs to perform square roots or divisions; it simply performs the standard Matrix-Vector Multiplication (MVM) using the updated weights. This can also be clearly seen from the Python model in Figure 5.2, too.

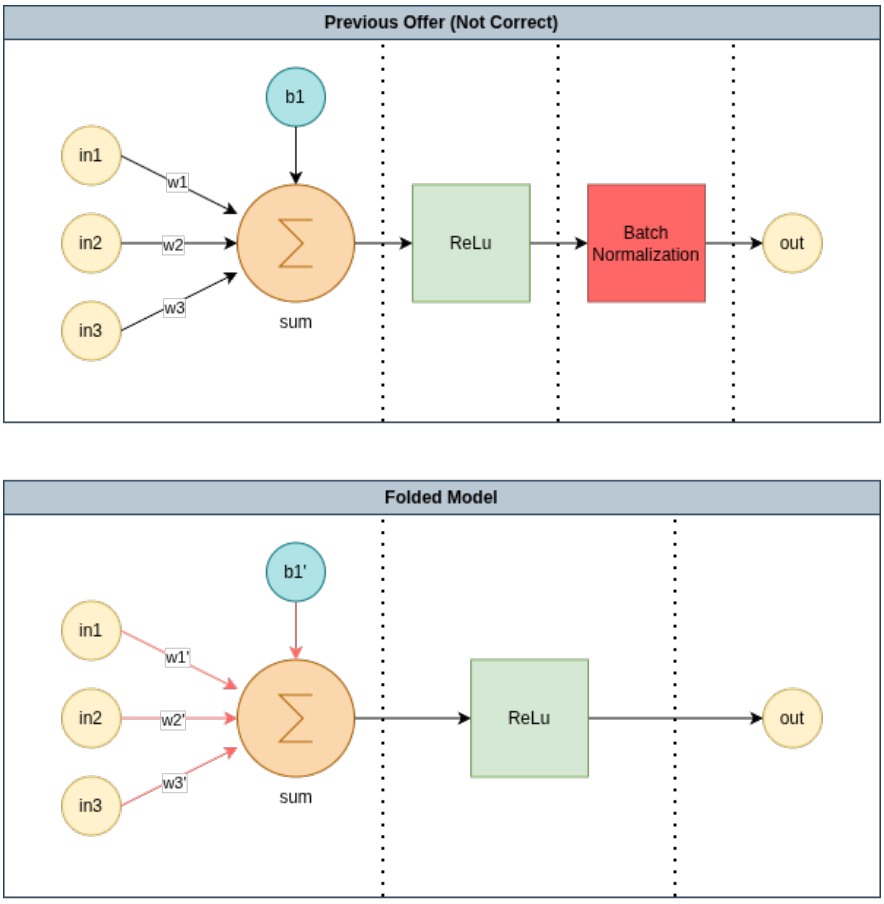

Figure 5.1 : Topology Comparison: Explicit Batch Normalization (Top) vs. Folded Architecture (Bottom). w1', w2', w3', and b1' refers to the new folded weights and biases.

Lastly, the last layer has a softmax step, which converts the output values into probabilities (between 0-1). However, for hardware implementation, this is not necessary. We achived the similar results by comparing each output. The largest value has the largest probability (argument of maxima or in short argmax), hence it is not required to compare after scaling. So the final function in C becomes:

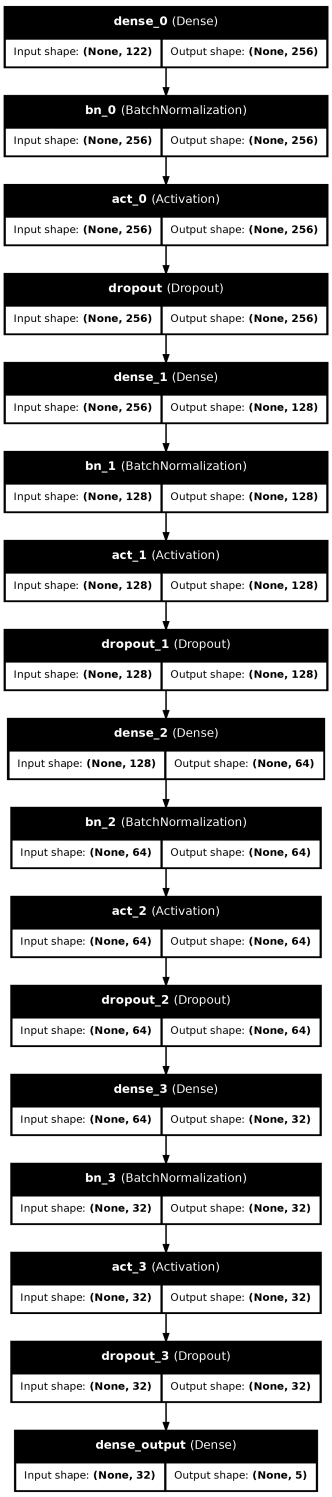

Figure 5.2 : Python Model of No Batch Model

```
int model_infer(const float *x)
{
    // Buffers
    float z0[L0_OUT];
    float z1[L1_OUT];
    float z2[L2_OUT];
    float z3[L3_OUT];
    float logits[L4_OUT];
```

```
    // --- Layer 0: Folded Dense -> ReLU ---
    dense_affine(x, INPUT_DIM, layer0_weights,
                layer0_biases, L0_OUT, z0);
    for (int i=0; i<L0_OUT; ++i) z0[i] = relu(z0[i]);

    // --- Layer 1: Folded Dense -> ReLU ---
    dense_affine(z0, L0_OUT, layer1_weights,
                layer1_biases, L1_OUT, z1);
    for (int i=0; i<L1_OUT; ++i) z1[i] = relu(z1[i]);

    // --- Layer 2: Folded Dense -> ReLU ---
    dense_affine(z1, L1_OUT, layer2_weights,
                layer2_biases, L2_OUT, z2);
    for (int i=0; i<L2_OUT; ++i) z2[i] = relu(z2[i]);

    // --- Layer 3: Folded Dense -> ReLU ---
    dense_affine(z2, L2_OUT, layer3_weights,
                layer3_biases, L3_OUT, z3);
    for (int i=0; i<L3_OUT; ++i) z3[i] = relu(z3[i]);

    // --- Layer 4: Output Dense (Logits) ---
    // Note: No ReLU here, No Softmax needed for Argmax
    dense_affine(z3, L3_OUT, layer4_weights, layer4_biases,
                 L4_OUT, logits);

    // --- Argmax ---
    int predicted = 0;
    float max_val = logits[0];
    for (int i = 1; i < L4_OUT; ++i) {
        if (logits[i] > max_val) {
            max_val = logits[i];
            predicted = i;
        }
    }

    return predicted;
}
```

5.3 Final Results

After the batch normalization folding step, both the software and hardware calculations have changed. In this section, the final results and hardware utilization/constraints are discussed. For the hardware and software results, the same metrics and codes as Chapter 4 are used.

5.3.1 Software Results

This subsection presents the performance metrics for both the Python and the C model. Finally, a comparison is made between the C simulation output and the ground truth labels.

5.3.1.1 Python Model Baseline

The Python model, with Batch Normalization layers folded into the dense weights, achieved an overall accuracy of 78.88%. The classification report (Table 5.1) insturates the results of the golden model for.

Table 5.1 : Python Model Classification Report (Folded)

Class	Precision	Recall	F1-Score	Support
0 (DoS)	0.97	0.85	0.90	7,460
1 (Probe)	0.82	0.70	0.76	2,421
2 (R2L)	0.93	0.12	0.21	2,885
3 (U2R)	0.47	0.25	0.33	67
4 (Normal)	0.70	0.97	0.81	9,710
Weighted Avg	**0.83**	**0.79**	**0.76**	**22,543**

Besides, the accuracy model accuracy and loss is illustrated in Figure 5.3 to show the convergence in the training steps. Epochs in the graphs are related with the backpropagation steps.

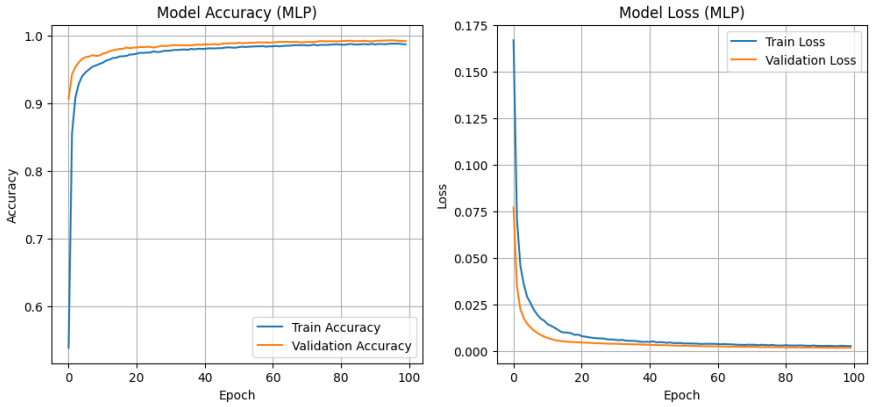

Figure 5.3 : Accuracy and Loss Models Throughout the Training

5.3.1.2 C Code Implementation

The standalone C implementation (`inference_light_noBatch.c`) was compiled using GCC and executed on the test dataset (with 22543 entries). The C model achieved an accuracy of 78.91%, which is virtually identical to the Python baseline,

confirming that the folding logic was correctly ported to C. Comparing with the first model, folded model is more accurate by 0.38%.

As it can be seen in Table 5.2 the F1-Score of critical classes did not change.

Table 5.2 : C Code Simulation Results (GCC)

Class	Precision	Recall	F1-Score	Support
0 (DoS)	0.97	0.85	0.90	7,460
1 (Probe)	0.82	0.70	0.75	2,421
2 (R2L)	0.93	0.12	0.21	2,885
3 (U2R)	0.47	0.25	0.33	67
4 (Normal)	0.70	0.97	0.81	9,710

Using the same `compare.py` script, the output file generated by the C simulation (`label_results.txt`) was validated against the dataset's ground truth (`labels.txt`). The validated accuracy is 78.53%.

5.3.2 Hardware Results

This subsection details the results obtained from the Vivado simulation and the physical FPGA tests via UART, followed by the verification comparisons.

5.3.2.1 Vivado Simulation

The design was synthesized and simulated using the Vivado Design Suite lasted ≈ 20.95 ms, and used 10.224 Gb of memory at peak. The simulation time of the folded model in Chapter 4 was 31.76 ms, so the model is improved by 34.04% interms of timing.

```
Success!
$finish called at time : 20951012500 ps
run: Time (s): cpu = 00:00:19 ; elapsed = 00:00:46
Memory (MB): peak = 10224.762 ; gain = 17.773
```

5.3.2.2 UART Test on FPGA

The complete test set was transmitted to the FPGA. The inference results were captured in real-time. For all the 22543 samples, the folded model completed the task in 2,163.46 seconds (≈ 36 minutes), achieving an accuracy of 78.87%. The previous model's accuracy was 78.37%, which means the design is enhanced both by the means of accuracy (+0.5%) and the means of simulation time (-14.26%).

```
[22539] true=4  pred=4  -> [PASS]
[22540] true=0  pred=0  -> [PASS]
[22541] true=4  pred=4  -> [PASS]
```

```
[22542] true=1  pred=1  -> [PASS]

=== Summary ===
Total tested: 22543
Correct: 17779
Wrong:   4764
Accuracy: 78.87%
Total time: 2163.46 seconds
Results saved to FPGA_results.txt
```

5.3.2.3 Comparison: Ground Truth vs. FPGA-UART

The results received from the FPGA (`FPGA_results.txt`) were compared against the ground truth labels. The hardware achieved a validated accuracy of 78.87%. Further details about the classes did not changed in the hardware (Table 5.3).

Table 5.3 : Validation: Ground Truth vs. FPGA Hardware

Class	Precision	Recall	F1-Score	Support
0 (DoS)	0.97	0.85	0.90	7,460
1 (Probe)	0.82	0.70	0.75	2,421
2 (R2L)	0.93	0.12	0.21	2,885
3 (U2R)	0.47	0.25	0.33	67
4 (Normal)	0.70	0.97	0.81	9,710
Weighted Avg	**0.83**	**0.79**	**0.76**	**22,543**

5.3.2.4 Hardware Fidelity: C-Model vs. FPGA

Finally, to measure the hardware fidelity, the FPGA results were compared directly against the C simulation results. The accuracy of this comparison is 98.35%.

5.3.3 Hardware Utilization and Constraints

In the hardware implementation, efficient utilization of the FPU, UART, Hornet core, and most critically, memory, is vital. Furthermore, both setup and hold timing constraints must be met to ensure reliable hardware operation.

To evaluate these metrics, the maximum achievable clock frequency was first established. Through iterative testing, this was found to be 26.250 MHz.

The timing summary for 26.25 MHz is presented in Figure 5.4.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.486 ns	Worst Hold Slack (WHS):	0.005 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	16217	Total Number of Endpoints:	16217	Total Number of Endpoints:	4613

All user specified timing constraints are met.

Figure 5.4 : Timing Summary.

Both setup and hold constraints are met; there are no timing violations.

Resource	Utilization	Available	Utilization %
LUT	10962	134600	8.14
FF	4351	269200	1.62
BRAM	128	365	35.07
IO	10	285	3.51
MMCM	1	10	10.00

Figure 5.5 : Utilization Summary 1.

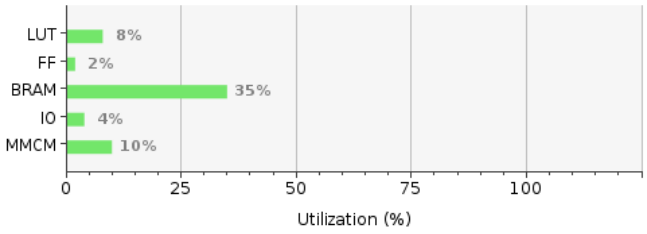

Figure 5.6 : Utilization Summary 2.

Name	Slice LUTs (134600)	Slice Registers (269200)	F7 Muxes (67300)	F8 Muxes (33650)	Slice (33650)	LUT as Logic (134600)	Block RAM Tile (365)	Bonded IOB (285)	BUFGCTRL (32)	MMCME2_ADV (10)
N fpga_top	10962	4351	704	84	3980	10962	128	10	2	1
uart0 (uart_wb)	61	47	0	0	27	61	0	0	0	0
mtime_regs (mtime_registers_wb)	257	195	0	0	117	257	0	0	0	0
memory (memory_2rw_wb)	78	3	0	0	52	78	128	0	0	0
loader0 (loader_wb)	107	38	0	0	55	107	0	0	0	0
core0 (core_wb)	10462	4029	704	84	3827	10462	0	0	0	0

Figure 5.7 : Utilization Detailed.

As shown in Figure 5.5, Figure 5.6, and Figure 5.7, the most critical resource in this project was indeed the memory (Block RAM - BRAM).

The power consumption analysis is shown in Figure 5.8.

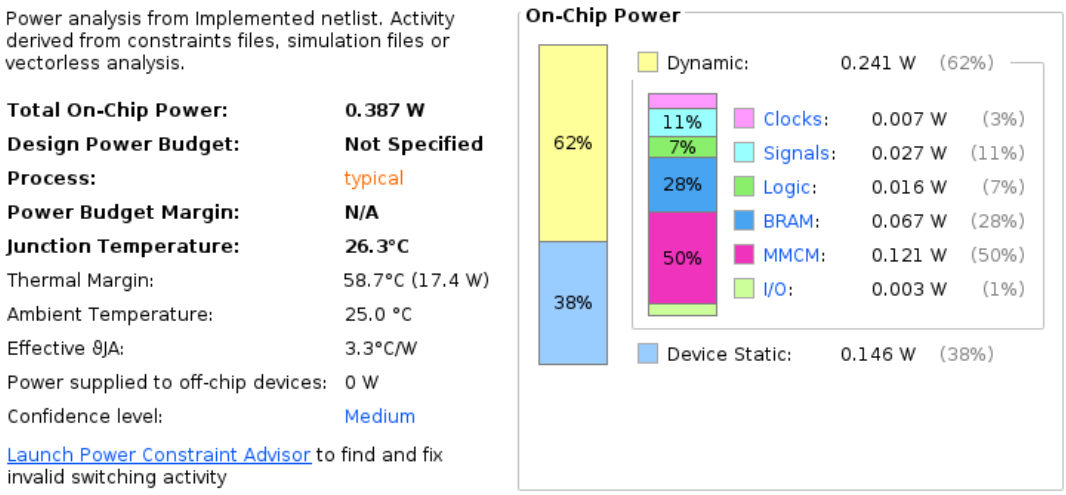

Figure 5.8 : Power Analysis Results.

6. HARDWARE ACCELERATOR DESIGN

The accelerator targets fully connected layers and speeds up inference on FP32 data through multiplication and addition operations. It also provides a memory mapped programming model for software control.

6.1 General Architecture

The design (Figure 6.1) is based on four fundamental strategies aimed at balancing memory requirements with increased processing power.

The computational side of the system utilizes a vector based execution infrastructure containing 16 FP32 multiplication and addition units (MACs). This architecture enables the simultaneous updating of 16 output components and the parallel execution of computationally intensive operations.

To enable efficient parallel processing, the weight memory is divided into 16 separate memory banks. Instead of reading the weights separately for each MAC unit, the arithmetic unit can simultaneously read a 512 bit weight vector combining 16 32 bit weights. This bank structure speeds up data flow and prevents memory access from becoming a bottleneck in the parallel MAC structure.

Ping pong activation buffering ensures seamless data flow between layers. The activation memory consists of two logical memory banks. One bank stores input activations, while the other simultaneously stores outputs. Once a layer is complete, the roles of the memory banks are reversed. This approach reduces both latency and memory traffic because it eliminates the need to retrieve the output at the end of each layer and copy it back to the input.

Finally, the system organizes output generation using tile logic. Each tile represents an output block of 16 neurons, and the hardware is designed to compute this 16-neuron block in each iteration. If the output size of a layer exceeds 16, the output vector is split into multiple tiles and processed sequentially. This structure provides a workflow compatible with the hardware's native vector width. It simplifies the control logic and contributes to highly efficient resource utilization.

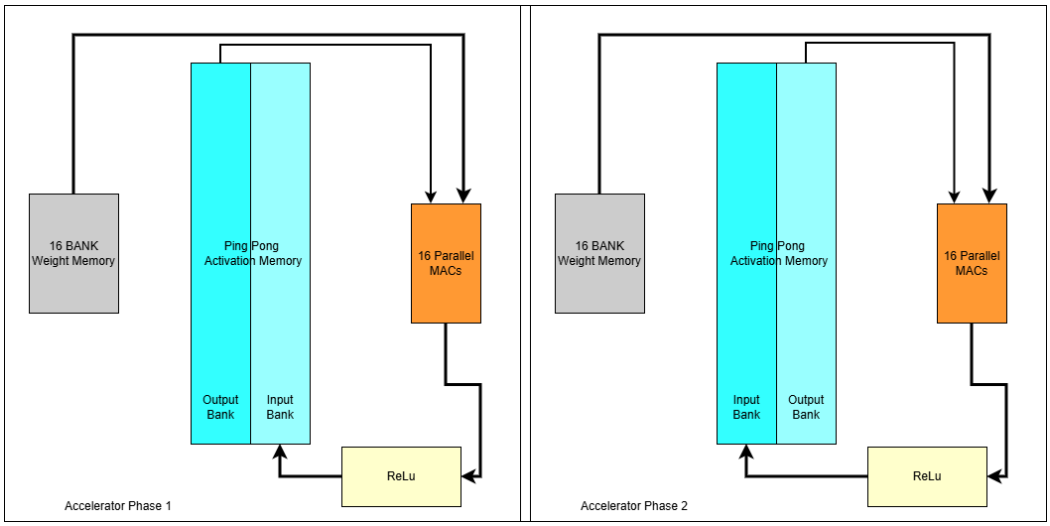

Figure 6.1 : High level block diagram of the accelerator architecture demonstrating dual phase memory access pattern.

6.2 RTL Decomposition and Interfaces

The hardware implementation is structured hierarchically to separate the bus protocol logic from the core computational datapath. This section describes the external interface and details the internal decomposition and internal module data flow.

6.2.1 External Interface

The accelerator integrates into the Hornet as a memory mapped Wishbone slave peripheral. Figure 6.2 shows the RTL system diagram created by Vivado, highlighting this access path.

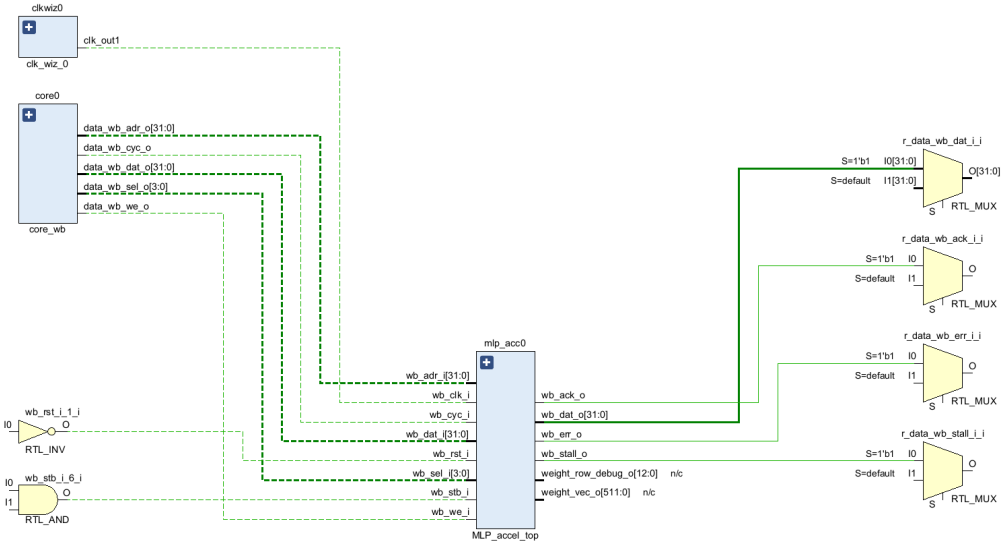

Figure 6.2 : System level integration schematic.

6.2.1.1 Wishbone handshake behavior.

The accelerator's Wishbone interface is implemented as a simple, nonblocking slave.

Accordingly, no wait states are added and the stall signal is permanently disabled, as it is assumed that the internal memory can always accept incoming requests.

For both read and write operations, the acknowledgment signal is generated as a registered version of the request strobe. This design results in a deterministic, constant response latency and simplifies timing analysis by aligning address decoding and read data multiplexing operations with a sequential handshake.

6.2.1.2 High level routing

The wishbone address is decoded to provide direct access to register field, weight memory window or activation memory window. Read data is returned to the CPU through a combinational multiplexing stage that selects the correct internal source based on the decode result.

6.2.2 Internal RTL Decomposition and Data Flow

. Internally, the accelerator consists of several submodules connected to each other with wide datapath. While the external data bus is 32 bits wide, the internal architecture uses a 512 bit data bus to sustain parallel MAC operations. Figure 6.3 shows these internal datapaths within the accelerator wrapper.

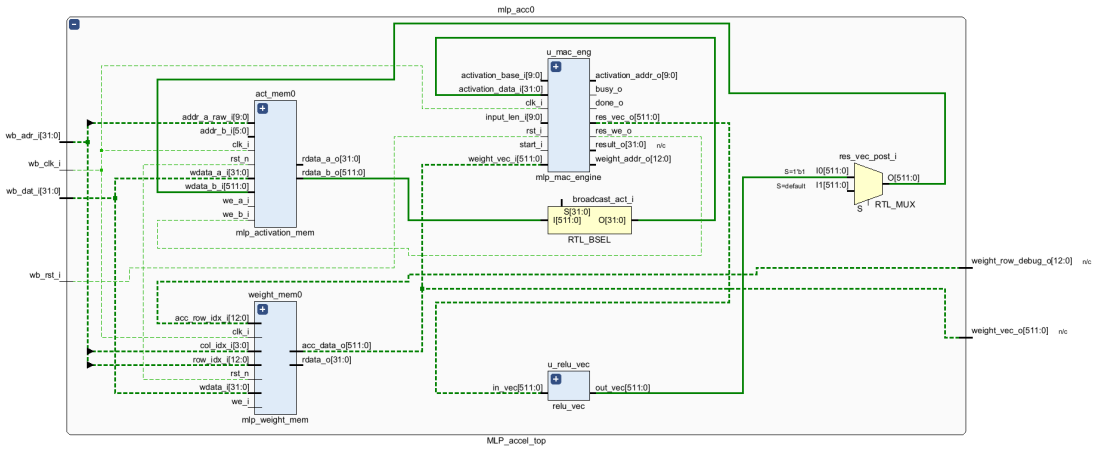

Figure 6.3 : Internal decomposition of the accelerator wrapper.

The responsibilities of the internal modules are summarized in Table 6.1.

Table 6.1 : Hierarchical breakdown of the MLP accelerator modules.

Module	Functionality & Responsibility
Accelerator wrapper	A high level wrapper with a Wishbone slave interface, address decoder, configuration registers, and central controller that controls the execution of layers and tiles.
Weight memory	Storage for model parameters. It bridges the width gap by accepting 32 bit CPU writes and reads while providing 512 bit vector reads to the MAC engine.
Activation memory	Dual purpose memory for input and output activations. It supports the logical ping pong buffering scheme for internal layer data flow and provides 512 bit row access to the MAC engine.
MAC engine	Computational core. It iterates over the input dimension and drives 16 parallel FP32 MAC lanes for each output tile.
ReLu	Vector post processing unit that applies ReLU to the 512 bit result vector before write back if enabled.

6.3 Memory Map and Address Decoding

The MLP accelerator can be accessed as a memory mapped peripheral with the Wishbone bus.

On the software side, same address space is used to load weights, program control, status registers and manage input vectors with an activation scratchpad. For clarity, the accelerator address range is organized into three windows which indicated by Table 6.2

Table 6.2 : Wishbone memory map of the MLP accelerator

Region	**Address range**
Weight memory	0x2000_0000 to 0x2004_FFFF
Register window	0x2005_0000 to 0x2005_FFFF
Activation memory	0x2006_0000 to 0x2006_0FFF

Address decoding is performed by checking which window the incoming wishbone address maps to. Accesses to the weight or activation windows are routed to the respective memory blocks, while accesses to the register window are handled by the accelerator's control and status register file.

6.3.1 Weight address translation

The weight window is word addressable from the core. Within the accelerator, the incoming wishbone address is interpreted as a two dimensional layout corresponding to the physical organization of the weight memory. Specifically, it is a split row and column index address. Each row corresponds to a 512 bit vector implemented as 16 parallel 32 bit lanes. The column index enables the core select one of these 16 lanes for read and write accesses, while the accelerator bus can access an entire row in parallel, creating a complete weight vector. These auxiliary signals are explicitly calculated in the accelerator wrapper.

6.3.2 Activation address translation

The activation window is also word addressable from the CPU. For accesses targeting the activation scratchpad, the raw word index is derived directly from the Wishbone address.

Internally, the activation memory is organized as 512 bit rows. Each row consisting of 16 parallel 32 bit lanes. Therefore, the word index is split into a row and a lane selector. The lower 4 bits select the column within a 512 bit row, while the upper bits select the row. This mapping allows software to access individual FP32 words, while the accelerator datapath can read and write entire 512 bit vectors efficiently.

6.4 Register Map and Programming Model

This section describes the accelerator's register file visible to the software and the intended programming model.

All registers are mapped to memory within the Wishbone address range of 0x2005_0000 to 0x2005_FFFF.

6.4.1 MLP control and layer configuration registers

The main execution mode is controlled by several control and status registers, as well as layer configuration array. Table 6.3 summarizes the main registers used during normal MLP execution.

Table 6.3 : MLP execution register map

Name	Address	Notes
REG_MLP_CTRL	0x2005_1020	Start and status register (busy, done, last output buffer).
REG_MLP_NUM_LAYERS	0x2005_1024	Number of layers (MAX LAYERS=8).
REG_MLP_LAYER[i]	0x2005_1100 + 4*i	Packed configuration word for layer *i*.

6.4.1.1 MLP_CTRL write semantics

Software controls execution by writing REG_MLP_CTRL. Only a small subset of bits have effects:

- **bit[0] (start_all):** writing a '1' issues a start request. Internally, this request is latched as a single cycle pulse to launch the MLP FSM.
- **bit[2] (done_clr):** writing a '1' clears the done flag after a completed run.

Other bits are ignored on write and are reserved for future extensions.

6.4.1.2 MLP_CTRL read semantics

Reading REG_MLP_CTRL provides the current execution status:

- **bit[1] (busy):** asserted while the accelerator is executing an MLP run.
- **bit[2] (done):** asserted when the run has finished; the bit remains set until cleared by writing done_clr (bit[2]).
- **bit[3] (last_output_buf):** indicates which ping pong buffer currently contains the most recent layer output.

6.4.2 Layer configuration word format

Each layer configuration register packs the parameters required by the layer scheduler:

- **input_dimension:** bits [9:0]
- **output_dimension:** bits [25:16]
- **activation_type:** bits [29:28] (0 = linear, 1 = ReLU)

6.5 Weight Memory Architecture

The weight memory is implemented as 16 independent banks of 32 bit RAM (Figure 6.4) that together form a single 512 bit row vector. These banks allow the accelerator bus to fetch a full 16 lane weight vector with a single row access, while also supporting simple 32 bit word read and write operations by the CPU.

The main RTL parameters are:

- RAM DEPTH = 5120 rows
- VECTOR WIDTH = 512 bits

The resulting capacity calculation is shown in the Equation 6.1.

$$5120 \times 16 \times 4 \text{ bytes} = 327{,}680 \text{ bytes} \tag{6.1}$$

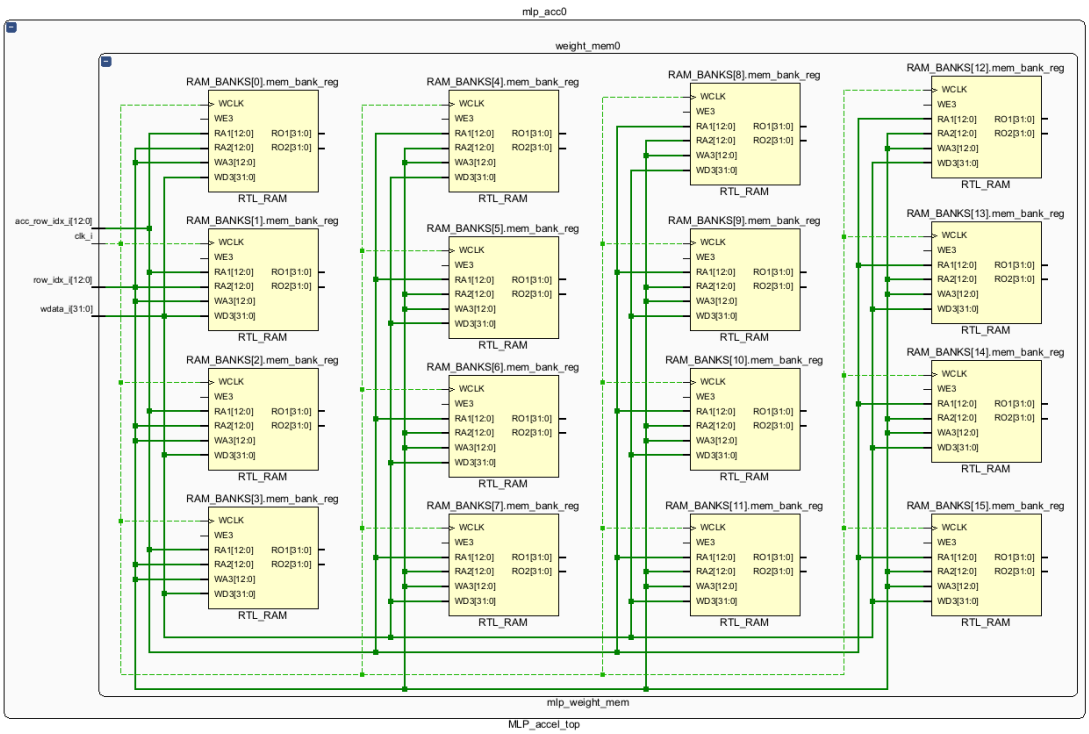

Figure 6.4 : 16 bank structure of weight memory.

6.5.1 CPU Port

Software accesses the weight memory as 32-bit words, targeting a single bank. To match the synchronous RAM read latency, the column index is registered by one cycle. This ensures the readback multiplexer (Figure 6.5) aligns with the valid data output.

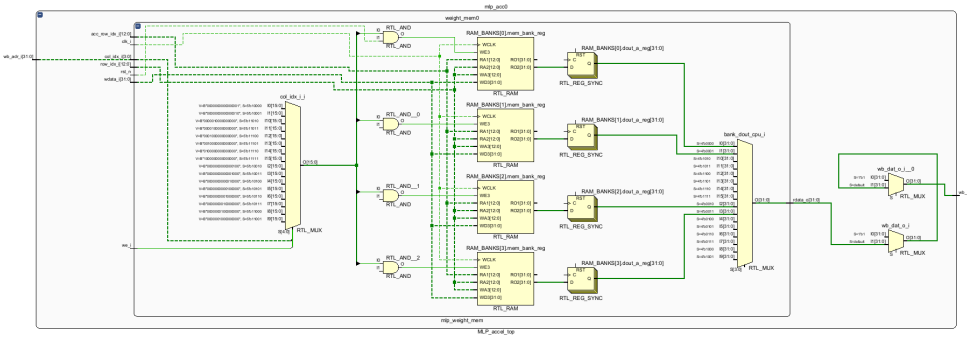

Figure 6.5 : Access structure over the first four banks by core.

6.5.2 Accelerator Port

During execution, the MAC engine (Figure 6.6) reads a complete 512-bit row vector from a selected address in a single access. The returned vector contains 16 FP32 weights, one for each lane, corresponding to an output tile.

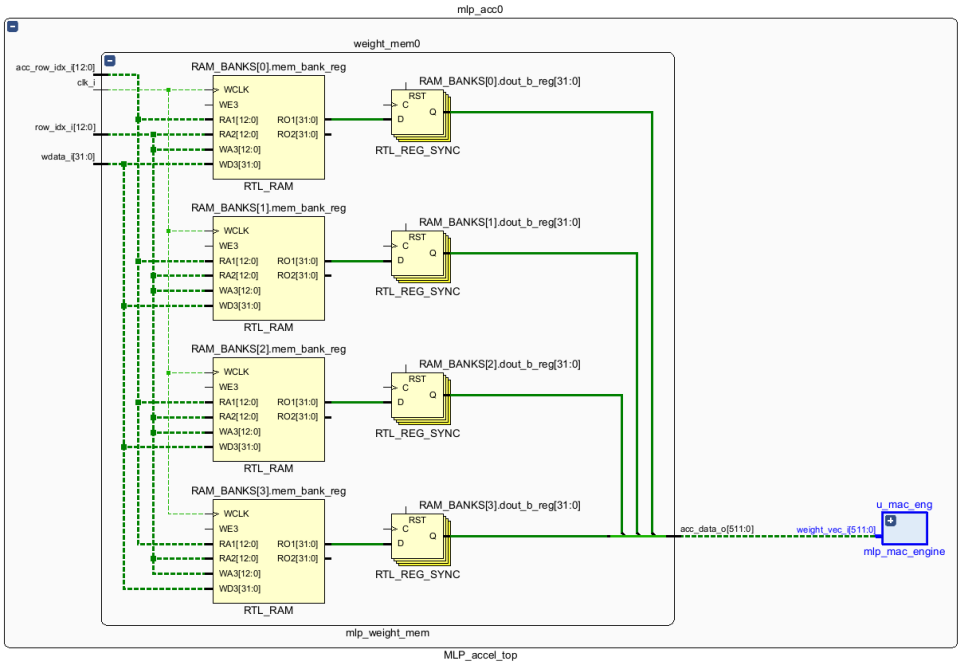

Figure 6.6 : Access structure over the first four banks by MAC engine.

6.5.3 Weight Layout In Memory For Tiled Fully Connected Layers

For a dense layer outputs are partitioned into tiles of 16 neurons. The number of tiles are calculated as it shown in the Equation 6.2.

$$\text{number of tiles} = \left\lceil \frac{\text{output dimension}}{16} \right\rceil \tag{6.2}$$

Weights are stored row and major with respect to the input index. For tile t and input index i, one memory row (Equation 6.3) stores a 16-lane weight vector, and lane j of that row contains the weight of $W[i][16t + j], \quad j \in \{0, \ldots, 15\}$.

$$\text{row} = \text{layer base} + t \cdot \text{input dimension} + i \tag{6.3}$$

If output dimension is not a multiple of 16, the final tile contains unused lanes. These lanes are treated as padding and should be ignored by software (or equivalently filled with zero in memory).

6.6 Activation Memory Architecture

Activations are stored in a small scratchpad. The buffer memory is divided into 16 parallel 32-bit banks and 64 rows. Each row thus forms a 512-bit vector containing 16 FP32 activation values. With this organization, the total capacity is $16 \times 64 \times 4$ bytes $=$ 4096 bytes.

From the CPU side, the activation window is accessed as a linear array of 32-bit words. From the accelerator side, the datapath reads and writes full 512-bit rows, which matches the 16 lane parallel MAC structure.

6.6.1 Ping Pong Buffering Across Layers

To avoid data movement overhead, the 64 rows are split into two buffers. Buffer A occupies of rows 0-31, and Buffer B occupies rows 32-63.

The MLP controller toggles the ping flag so that one buffer serves as the input activation stream while the other buffer receives the computed outputs. After each layer completes, the roles swap. At the end of the network, the status bit reports which buffer currently contains the final output vector.

6.6.2 Activation Addressing and Broadcast to the MAC Engine

During execution, the MAC engine generates a linear word address for the input activation stream. This word address is mapped onto the banked organization by splitting it into a row and a lane selector.

The corresponding 512-bit row is read from the activation memory, the selected 32-bit lane is extracted, and that FP32 value is broadcast to all 16 MAC lanes for the current cycle. This scheme (Figure 6.7) preserves a simple linear addressing model for software and for the MAC engine, while still enabling efficient 512-bit row accesses in the datapath.

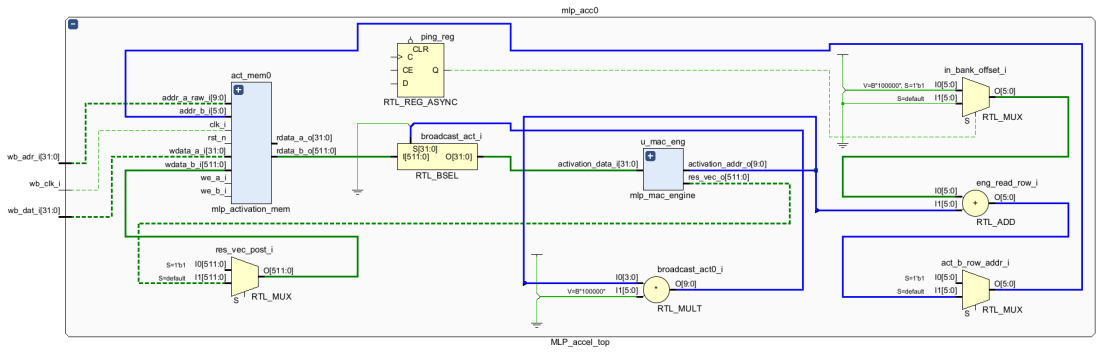

Figure 6.7 : Activation addressing by MAC engine.

6.6.3 Output Storage Layout

Each tile produces 16 outputs in parallel. At the end of a tile computation, the engine writes the result vector as one 512-bit row into the current output buffer. As a result, a layer output vector is stored sequentially in tile order. The next layer can then consume this output as a linear word stream using the same addressing convention as the input activations.

6.7 MAC Engine

MAC engine evaluates a tiled dot product, producing 16 outputs which means one tile in parallel. For a given tile, the output vector is computed as shown in the Equation 6.4.

$$y_{\text{tile}} = \sum_{i=0}^{\texttt{input dimension}-1} x_i \cdot w_{i,\text{tile}} \tag{6.4}$$

where $w_{i,\text{tile}}$ is a 16 lane FP32 weight vector and x_i is a scalar FP32 activation value broadcast to all lanes for that iteration.

6.7.1 Parallel MAC Lanes and Accumulation

Sixteen identical MAC lanes operate in parallel (Figure 6.8). Each lane contains an FP32 multiply and a FP32 adds feeding an accumulator register. At the beginning of a tile computation, the accumulators are cleared. On each iteration, the lane updates its accumulator with the new partial sum.

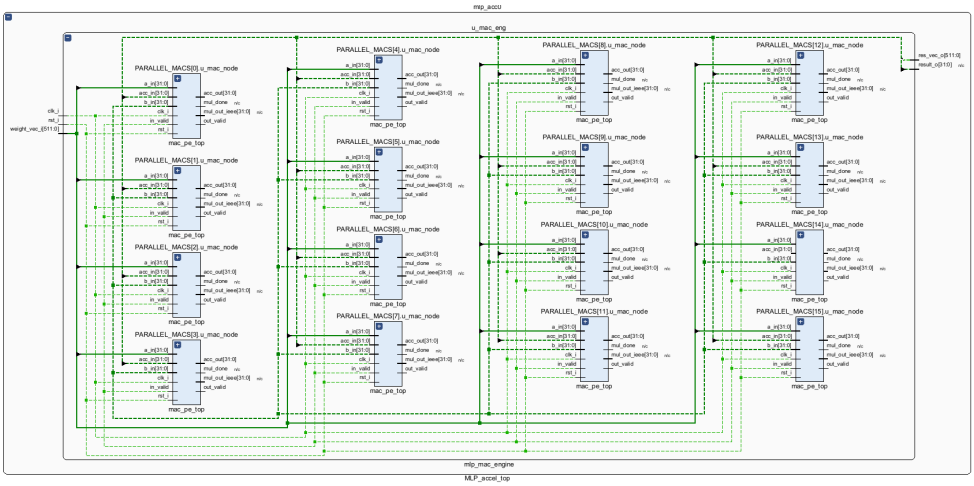

Figure 6.8 : 16 MAC units structure

6.7.2 MAC Lane Micro Architecture

Each of the 16 parallel lanes is coupled with a local accumulator register logic. The design follows an iterative accumulation model. For iteration k ($i = k$), lane j updates its partial sum as Equation 6.5.

$$\mathtt{acc}_j^{(k+1)} = \mathtt{acc}_j^{(k)} + x_k \cdot w_{k,\text{tile}}[j] \tag{6.5}$$

Here, x_k is the broadcast FP32 activation and $w_{k,\text{tile}}[j]$ is the unique FP32 weight for that lane.

The MAC lane architecture is shown in Figure 6.9.

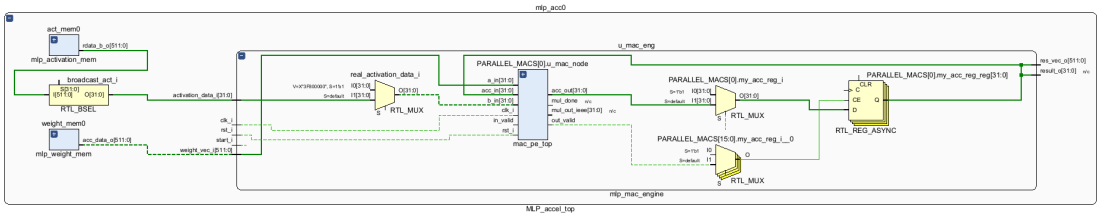

Figure 6.9 : MAC lane architecture

6.7.2.1 Feedback Loop

The micro architecture relies on an explicit feedback loop. The current value of the accumulator register is fed back into the accumulator input port of the PE. The PE performs the FP32 multiplication and addition, producing a new candidate sum at accumulator output.

6.7.2.2 Processing Element Overview

The PE block encapsulates the FP32 multiply and FP32 add datapath. Its output is pipelined, and the output valid strobe indicates when the accumulator output carries a valid updated sum (Figure 6.10).

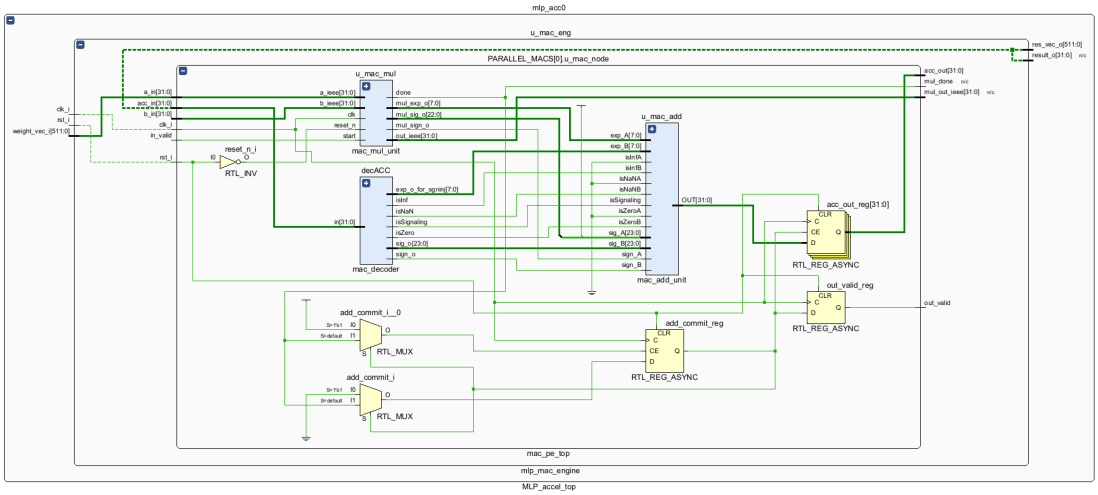

Figure 6.10 : Processing element

6.7.2.3 Valid driven Register Update

Since the FP32 units internally have multi cycle latency, the accumulator register is not updated on every clock cycle. Instead, the update is gated by the output valid strobe generated by the PE. The register captures the value from accumulator output only when output valid is asserted. This mechanism allows the control FSM to operate agnostic of the specific pipeline depth of the arithmetic units. Table 6.4 shows the signals and corresponding RTL port.

Table 6.4 : Signal mapping for the MAC Processing Element (PE).

Signal Description	**RTL Port**	**Role in Micro-architecture**
Weight Input	`a_in`	Receives the FP32 weight word specific to this lane.
Activation Input	`b_in`	Receives the broadcast FP32 activation value.
Accumulator Feed	`acc_in`	Input for the current partial sum, fed back from the external register.
Next Accumulator	`acc_out`	Output carrying the updated partial sum (`acc_in` + (`a_in` · `b_in`)).
Output Valid	`out_valid`	Strobe indicating that the result on `acc_out` is valid and should be written to the register.

6.7.3 Bias Handling

Bias is implemented by treating it as an extra input dimension. On the final iteration of the dot product, the engine injects a constant activation value of 1.0f regardless of the contents of the activation memory.

Accordingly, software must include the bias weights as the last weight row of each tile, and program input dimension to include this bias term.

6.7.4 Post-processing: Vector ReLU

After accumulation, the 16-lane result vector can optionally pass through a ReLU stage depending on the current layer configuration. When the activation type is set to ReLU, each lane is clamped to zero if it is negative. Otherwise, the raw accumulated result is forwarded unchanged. This selection is implemented as a simple mux in the top module, and the ReLu block (Figure 6.11) performs lane-wise clamping.

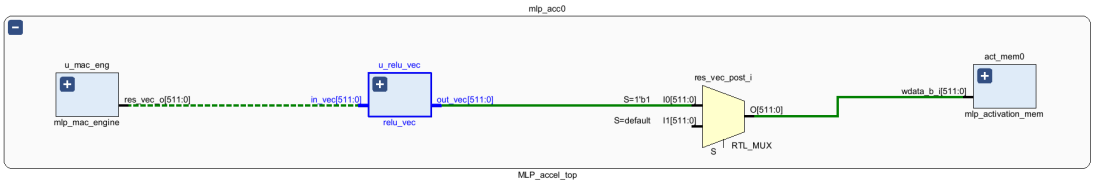

Figure 6.11 : ReLu connection scheme

6.7.5 Iterative Control FSM

The MAC engine is driven by a compact FSM that iterates over the input dimension and manages four states. The states are explained in the Table 6.5.

Table 6.5 : MAC engine FSM states and per-state actions

State	Main purpose	Actions in the state
`S_IDLE`	Wait for a new tile computation request	Monitors start bit. When asserted, clears the lane accumulators, resets the iteration counter, and prepares to start the first input step.
`S_ADDR`	Prepare addresses for the current iteration	Computes and updates weight address and activation address for the current counter value. Addresses are held stable so the synchronous memories can produce valid data for the next phase.
`S_ISSUE`	Launch one MAC iteration	Asserts the internal one cycle strobe `mac_issue` to mark the current activation, weight pair as valid input to the PE lanes. This state typically lasts one clock cycle.
`S_WAIT`	Wait for PE completion and accumulate	Waits until the PE lanes assert output valid. When valid, adds the partial products into the lane accumulators, increments the iteration counter, and either returns to `S_ADDR` for the next input or finishes the tile after the last input is consumed.

At the end of the final iteration, the accumulated 16 lane vector is made available as result vector and the engine asserts its completion indication to the higher level controller.

6.7.6 Post Synthesis Resource Utilization and Timing Summary

To quantify the hardware cost of the proposed accelerator, Vivado post-synthesis reports were used to extract resource utilization and an initial timing estimate for the target device. These results are reported after synthesis. Therefore, they should be interpreted as estimates. In particular, the final achievable frequency and critical path may change after implementation due to routing delays and physical placement effects.

6.7.6.1 Resource utilization

Figure 6.12 and Figure 6.13 summarizes the main FPGA resources consumed by the accelerator included design.

Resource	Estimation	Available	Utilization %
LUT	93282	134600	69.30
FF	39872	269200	14.81
BRAM	256	365	70.14
DSP	48	740	6.49
IO	9	285	3.16

Figure 6.12 : Post synthesis resource utilization.

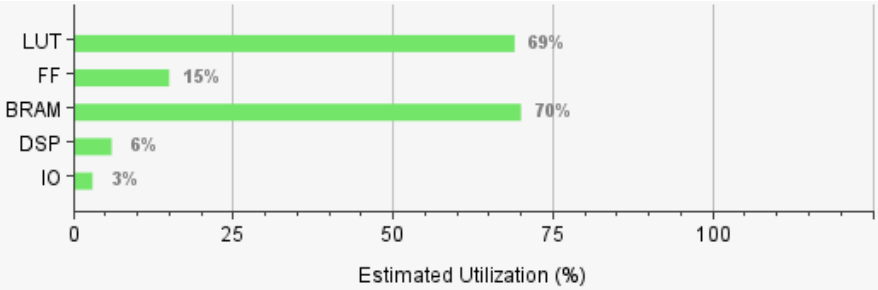

Figure 6.13 : Post synthesis resource utilization percentage.

6.7.6.2 Timing estimate

A preliminary timing check was performed with a target clock period of 100 ns. Table 6.6 reports the post-synthesis slack values.

Table 6.6 : Post synthesis timing summary

Metric	Value
Target period (ns)	100.000 ns
WNS (ns)	42.042 ns

A target period of 100 ns corresponds to a 10 MHz clock constraint. The post synthesis setup slack is positive WNS = +42.042 ns, indicating that the longest path meets the setup requirement with margin. An approximate critical path delay can be inferred as Equation 6.6, thereby the maximum frequency is calculated as Equation 6.7.

$$T_{\text{crit}} = T_{\text{target}} - \text{WNS} = 100\ \text{ns} - 42.042\ \text{ns} = 57.958\ \text{ns} \tag{6.6}$$

$$f_{\max} = \frac{1}{T_{\text{crit}}} = \frac{1}{57.958 \times 10^{-9}} \approx 17.3\ \text{MHz}. \tag{6.7}$$

6.7.6.3 Power estimate

Vivado power analysis (Figure 6.14) was used to obtain an initial power estimate after synthesis.

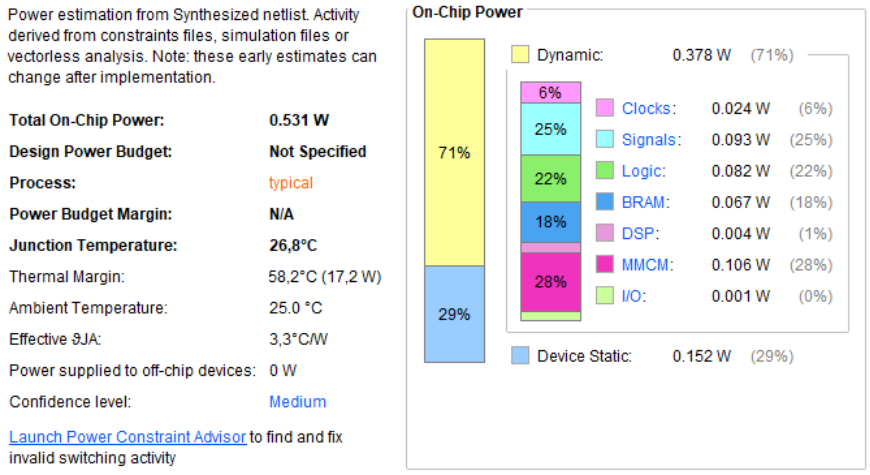

Figure 6.14 : Post synthesis power estimate.

6.7.7 Weight Packing for the Accelerator

The accelerator expects weights to be stored in a 16-lane 512-bit tiled layout, while the golden model exports each layer as a flat 1D weight array and a separate bias vector. Therefore, a Python script is used to convert the software representation into the exact linear memory order required by the hardware.

The script Parses layer weights and layer biases from a C header and reshapes the weights into a 2D matrix. Then folds the bias vector as an additional last row. Finally pads the output dimension to a multiple of 16 and packs the matrix tile by tile into a linear FP32 stream that matches the accelerator's row vector read order. Bias folding is implemented by appending b as an extra row.

```
W_fused = np.vstack([W, b.reshape(1, -1)])
```

Tiling and packing are done by slicing each 16-output tile and flattening it

```
tile_block = W_fused[:, col_start:col_end]   # (in+1) x 16
packed_data.extend(tile_block.flatten())
```

This preprocessing step is necessary because the hardware fetches weights as 512-bit rows and computes 16 outputs in parallel; the output dimension must be tile-aligned, and bias is applied by treating it as an extra input dimension in the hardware datapath.

6.7.8 Software Driver and Inference Flow

To run inference on the MLP accelerator, a lightweight C driver is used. The driver provides a small API on top of the memory-mapped Wishbone interface. It loads the pre-packed weights, programs each layer configuration registers, writes the input vector into activation memory, starts the accelerator, and finally reads back the output logits. Softmax is computed on the CPU to keep the hardware datapath focused on dense layers.

6.7.8.1 Memory-mapped Access Registers

The driver defines base addresses and convenience macros for register and memory accesses

```
#define MLP_BASE_ADDR    0x20000000u
#define MLP_REG_BASE_OFF 0x00050000u
#define MLP_ACT_RAM_OFF  0x00060000u
#define REG32(addr) (*(volatile uint32_t *)(addr))
#define ADDR_REG(off) (MLP_BASE_ADDR + MLP_REG_BASE_OFF + off)
#define ADDR_ACT(idx) (MLP_BASE_ADDR + MLP_ACT_RAM_OFF + (idx * 4))
```

This hides address arithmetic and makes the API calls directly correspond to the hardware memory map.

6.7.8.2 Layer Programming Model

Each layer is configured by writing a packed configuration word into `REG_MLP_LAYER[i]`. The driver applies the accelerator's bias convention by itself.

```
uint32_t hw_in_dim = in_dim + 1; // Bias term
cfg_val |= (hw_in_dim & 0x3FF);
cfg_val |= ((out_dim & 0x3FF) << 16);
cfg_val |= ((act_type & 0x3) << 28);
```

6.7.8.3 Weight Loading

Weights are included from the generated header and copied into the weight memory window as a contiguous FP32 stream. The software maintains a running destination offset so that layers are placed back-to-back in memory.

```
mlp_load_weights(weights_l0_hw, L0_HW_SIZE, w_off);
w_off += L0_HW_SIZE;
...
mlp_load_weights(weights_l4_hw, L4_HW_SIZE, w_off);
w_off += L4_HW_SIZE;
```

6.7.8.4 Input Staging and Bias Injection

The input vector is written to activation memory starting at word index 0. To match the hardware bias as an extra input convention, the driver appends a constant 1.0 at the end of the input stream.

```
for (i = 0; i < length; i++) ram_ptr[i] = input_data[i];
ram_ptr[length] = 1.0f; // Bias input
```

6.7.8.5 Start, Polling and Completion

Inference is launched by writing the start bit to REG_MLP_CTRL and then polling the done bit. After completion, the driver marks it as done to prepare for the next run.

```
REG32(ADDR_REG(REG_CTRL_OFF)) = 1;          // start
while (!(REG32(ADDR_REG(REG_CTRL_OFF)) & (1 << 2))) { } // wait done
REG32(ADDR_REG(REG_CTRL_OFF)) = (1 << 2);  // clear done
```

6.7.8.6 Reading Results and Ping Pong Buffer Selection

Because the activation scratchpad uses ping pong buffering, the final output location varies and which reported in REG_MLP_CTRL. The driver selects the correct half bank and reads FP32 values. For the last layer, the accelerator still computes a 16 lane tile internally, but software only consumes the meaningful logits.

```
uint32_t last_buf = (status >> 3) & 0x1;
uint32_t read_offset = last_buf ? ACT_BANK_SIZE_WORDS : 0;
for (i = 0; i < length; i++) output_buf[i] = ram_ptr[read_offset + i];
```

6.7.8.7 CPU Side Post Processing

The driver computes softmax on the CPU from the readed logits and then choose biggest logit to obtain the predicted class. This keeps the accelerator focused on the dense layer workload while still providing end to end classification in software.

6.8 Simulation Results and Functional Validation

This section reports RTL simulation results for the proposed accelerator and compares them against a software baseline. The evaluation focuses on functional correctness with respect to a golden model and latency measured in clock cycles.

6.8.1 Experimental Setup

All results are obtained from Vivado RTL simulation by running a bare metal inference program. The same input vector is applied to both the software baseline and the accelerator-assisted flow.

6.8.2 Measurement Methodology

Performance is reported in terms of execution time, calculated based on a fixed system clock frequency of 40 MHz. For the accelerator path, the measurement window is defined by the duration of the mlp_busy signal, which means inference time.

6.8.3 Software baseline

As a baseline, the MLP is evaluated in software using the same network parameters. The baseline latency is measured from the beginning of the first layer computation to the production of the final logits.

6.8.4 Accelerator-assisted Inference Flow

Accelerator-assisted execution follows a defined workflow that begins with loading weights into the weight memory window and programming the layer configuration registers. Subsequently, the input vector is written to the activation memory, and the inference engine is triggered with the REG_MLP_CTRL register. Once the operation completes, the resulting logits are retrieved from the activation memory, while the final Softmax calculation is offloaded to the CPU as a post-processing step

6.8.5 Cycle Breakdown and Speedup

We evaluate the performance improvement by comparing the accelerator-assisted execution against the pure software baseline established earlier in Section 5.3.2.1.

For the hardware-accelerated approach, the total inference time is derived from the RTL simulation waveform shown in Figure 6.15. Note that the one-time weight loading overhead and programming configuration registers are excluded from this inference metric, as they are amortized over continuous operation. The overall speedup is computed as shown in the Equation 6.8.

$$\text{Speedup} = \frac{t_{\text{SW}}}{t_{\text{HW}}} \tag{6.8}$$

Substituting the measured values, where the software baseline required approximately 20.5 ms and the accelerator-assisted inference completed in 707.8 μs.

$$\text{Speedup} = \frac{20.5\ \text{ms}}{707.8\ \mu\text{s}} = 28.96\times \tag{6.9}$$

As shown in the Equation 6.9, the accelerator provides a speedup of approximately $28.96\times$ compared to the pure software implementation.

6.8.6 Accelerator Assisted Waveform

Figure 6.15 shows the accelerator operating time range during inference.

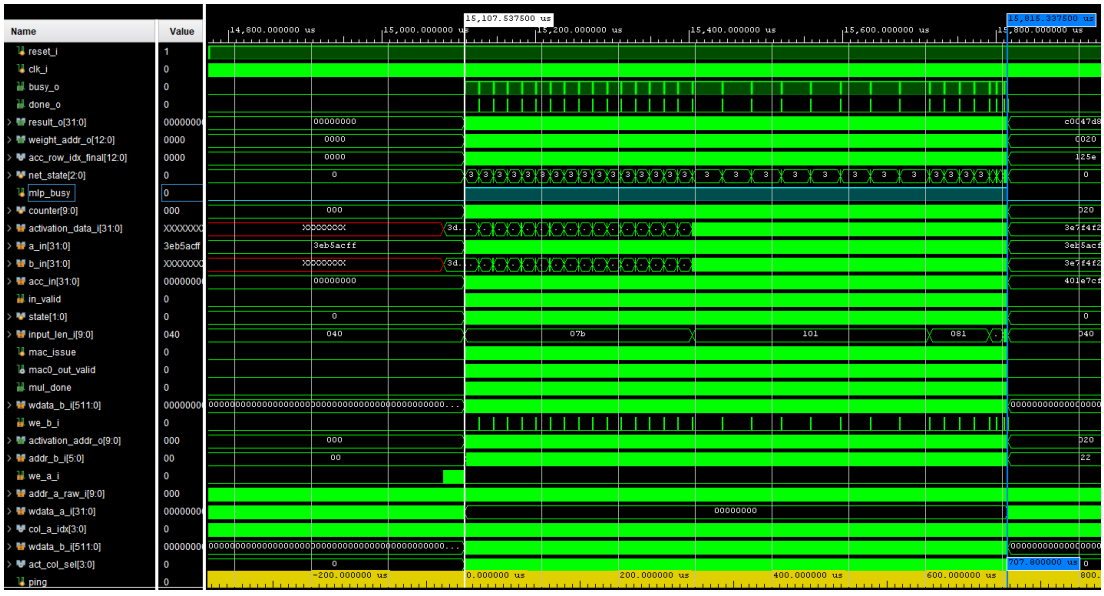

Figure 6.15 : Inference with accelerator

6.8.7 Functional Correctness

Correctness is validated by comparing the accelerator-produced logits against the golden model for the same input vector. The maximum absolute error is reported as shown in the Equation 6.10.

$$\varepsilon_{\max} = \max_i \left| y_i^{\text{golden}} - y_i^{\text{hw}} \right|. \tag{6.10}$$

In our experiments, the observed error remained within 5×10^{-7}.

6.8.8 Power and Energy Efficiency Estimates

To evaluate the energy efficiency of the proposed architecture, power consumption was estimated using the Xilinx Vivado Power Analysis tool based on the post-synthesis netlist. The analysis assumes a clock frequency of 40 MHz and typical switching activity for the targeted FPGA device.

Figure 6.14 summarizes the power report results. Using the measured execution time and the estimated total on-chip power, the energy per inference is calculated as shown in Equation 6.11.

$$E_{\text{inference}} = P_{\text{total}} \times t_{\text{HW}} \tag{6.11}$$

For the proposed architecture, the total on-chip power is reported as 0.531 W. With a measured inference latency of 707.8 μs, the energy per inference is calculated as shown in Equation 6.12.

$$E_{\text{proposed}} = 0.531\,\text{W} \times 0.7078\,\text{ms} \approx 0.376\,\text{mJ} \tag{6.12}$$

To assess the improvement, this result is compared with the previous iteration of the design. As shown in Figure 5.8, the previous design exhibited a lower power consumption of 0.387 W. However, it had a significantly longer inference time of 20.95 ms. Calculating the energy efficiency for the previous design is as shown in Equation 6.13.

$$E_{\text{prev}} = 0.387\,\text{W} \times 20.95\,\text{ms} \approx 8.11\,\text{mJ} \tag{6.13}$$

Comparing these figures demonstrates that the proposed accelerator achieves approximately a 21.57$\times$ increase in energy efficiency compared to the previous implementation.

7. REALISTIC CONSTRAINTS AND CONCLUSIONS

7.1 Practical Applications of This Project

The primary practical application of this project is the deployment of a lightweight Intrusion Detection System (IDS) on edge devices. By integrating a custom AI accelerator with a RISC-V core, this system can be utilized in Internet of Things (IoT) gateways, industrial controllers, and autonomous embedded systems where low-latency, localized threat detection is required without relying on cloud connectivity.

7.2 Realistic Constraints

In reality, network applications require fast communication protocols such as Ethernet, which can go up to Gbps data transfer speeds. However, in this project, a significantly slower UART communication protocol is used with a 26.25 MHz open-source core. For a real-world realization, both a faster CPU and a faster communication standard are required.

All the calculations done in this project are floating-point calculations. In the digital world, no matter how much they are accelerated, floating-point calculations take a long time and have a possibility of error.

Furthermore, even with the memory optimization throughout the project, the memory is still a large unit inside the SoC. This might lead to significant violations in floor planning, place & route, and timing closure steps in ASIC design.

Specific to the accelerator architecture, the hardware resources impose hard limits on the neural network topology. The control logic currently supports a maximum network depth of 8 layers. Due to the activation memory sizing used for the ping pong buffering strategy, the width of any single layer cannot exceed 1024 neurons. Additionally, the dedicated weight memory capacity is fixed at 320 KB, capping the total model size at approximately 81,920 parameters. Finally, to fully utilize the 16 lane MAC architecture,

layer dimensions ideally require padding to multiples of 16, which may introduce slight overhead for non aligned topologies.

7.3 Social, Environmental, and Economic Impact

By enabling the detection and classification of attacks directly on the observed network node, this design provides a viable and faster security option for the rapidly growing IoT market.

Also, by achieving such results at the edge of the network, the power required for the data transfers between the servers and IoT devices can be avoided completely.

7.4 Cost Analysis

This project is accomplished with the Nexys Video FPGA that was provided by the Electric and Electronics Faculty of Istanbul Technical Univercity. Xilinx Vivado Design Suite under the academic Student License is utilized, incurring no additional software costs.

7.5 Standards

IEEE 1364-2005 standard for Verilog HDL, the RISC-V User-Level ISA V2.2 specification for the RV32IMF core architecture, the IEEE 754-2008 standard for single-precision floating-point arithmetic, the C99 standard for C programming, and the standard UART protocol for serial communication are used for this project.

7.6 Health and Safety Concerns

There are no health or safety concerns.

7.7 Future Work and Recommendations

In this project, the data transfer has been the main bottleneck. To avoid this, in future designs, an AXI Stream bus can be used rather than the Wishbone. Also, using a viable Bootloader can offer both significant memory efficiency, shorter initialization, and a weight/bias update mechanism from a cloud server.

In terms of neural networks, this project only utilized the, most basic neural network design with slight optimizations to prove any DNN model can be implemented on an edge device. However, there exist various networks models and functions. Especially for IDS, some Machine Learning methods have been proven to be better.

REFERENCES

[1] **Khraisat, A.**, **Gondal, I.**, **Vamplew, P. and Kamruzzaman, J.** Survey of intrusion detection systems: techniques, datasets and challenges, *Cybersecurity*, *2*(1), 1–22.

[2] **Palekar, S. and Narkhede, N.** 32-Bit RISC processor with floating point unit for DSP applications, *2016 IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, IEEE, pp.2062–2066.

[3] **Jeya, P.G.**, **Ravichandran, M. and Ravichandran, C.** Efficient classifier for R2L and U2R attacks, *International Journal of Computer Applications*, *45*(21), 28–32.

[4] **Hanzaleturgut**, (2025), RF, NSL-KDD, Multi-class, 2025, https://www.kaggle.com/code/hanzaleturgut/rf-nsl-kdd-multi-class-2025, (Accessed: Nov. 9, 2025).

[5] **Nielsen, M.A.** *Neural networks and deep learning*, volume 25, Determination press San Francisco, CA, USA.

[6] **Elster, A.D.**, (2024), How does back-propagation work?, Questions and Answers in MRI, https://mriquestions.com/back-propagation.html, (Accessed: Dec. 25, 2025).

[7] **Alfaro-Gómez, R.A. and Alvarado-Lagunes, J.** RISC-V Floating Point Unit.

[8] **Nandland**, (2024), UART Serial Port Module, https://nandland.com/uart-serial-port-module/, (Accessed: Oct. 31, 2025).

[9] **Digilent**, (2024), Nexys Video FPGA Board Reference Manual, https://digilent.com/reference/programmable-logic/nexys-video/reference-manual, (Accessed: Nov. 5, 2025).

[10] **Y. Tekin, A. T. Özkan**, (2025), HORNET-RV32IMF For AI Applications Repository, https://github.com/GSTL-ITU/HORNET-RV32IMF-For-AI-Applications, (Accessed: Dec. 29, 2025).

[11] **Askarihemmat, M.**, **Wagner, S.**, **Bilaniuk, O.**, **Hariri, Y.**, **Savaria, Y. and David, J.P.** BARVINN: Arbitrary precision DNN accelerator controlled by a RISC-V CPU, *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, pp.483–489.

[12] **Thieu, G.B.**, **Gesper, S.**, **Payá-Vayá, G.**, **Riggers, C.**, **Renke, O.**, **Fiedler, T.**, **Marten, J.**, **Stuckenberg, T.**, **Blume, H.**, **Weis, C.** *et al.* ZuSE Ki-Avf:

application-specific AI processor for intelligent sensor signal processing in autonomous driving, *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, pp.1–6.

[13] **Chander, V.N. and Varghese, K.** A soft risc-v vector processor for edge-ai, *2022 35th International Conference on VLSI Design and 2022 21st International Conference on Embedded Systems (VLSID)*, IEEE, pp.263–268.

[14] **Vinayakumar, R.**, **Alazab, M.**, **Soman, K.P.**, **Poornachandran, P.**, **Al-Nemrat, A. and Venkatraman, S.** Deep learning approach for intelligent intrusion detection system, *IEEE access*, *7*, 41525–41550.

[15] **Mishra, P.**, **Varadharajan, V.**, **Tupakula, U. and Pilli, E.S.** A detailed investigation and analysis of using machine learning techniques for intrusion detection, *IEEE communications surveys & tutorials*, *21*(1), 686–728.

[16] **Goodfellow, I.**, **Bengio, Y.**, **Courville, A. and Bengio, Y.** *Deep learning*, volume 1, MIT press Cambridge.

[17] **Lin, T.Y.**, **Goyal, P.**, **Girshick, R.**, **He, K. and Dollár, P.** Focal loss for dense object detection, *Proceedings of the IEEE international conference on computer vision*, pp.2980–2988.

[18] **Kingma, D.P.** Adam: A method for stochastic optimization, *arXiv preprint arXiv:1412.6980*.

[19] **Srivastava, N.**, **Hinton, G.**, **Krizhevsky, A.**, **Sutskever, I. and Salakhutdinov, R.** Dropout: a simple way to prevent neural networks from overfitting, *The journal of machine learning research*, *15*(1), 1929–1958.

[20] **Hennessy, J.L. and Patterson, D.A.** *Computer architecture: a quantitative approach*, Elsevier.

[21] **RISC-V Foundation** The RISC-V Instruction Set Manual, Volume I: User-Level ISA, **Technical Report**, RISC-V Foundation, `https://riscv.org/technical/specifications/`, version 2.2 (Accessed: Dec. 28, 2025).

[22] **Paldurai, K. and Hariharan, K.** FPGA implementation of delay optimized single precision floating point multiplier, *2015 International Conference on Advanced Computing and Communication Systems*, IEEE, pp.1–5.

[23] **Arish, S. and Sharma, R.** Run-time-reconfigurable multi-precision floating-point matrix multiplier intellectual property core on FPGA, *Circuits, Systems, and Signal Processing*, *36*(3), 998–1026.

[24] **Sunesh, N. and Sathishkumar, P.** Design and implementation of fast floating point multiplier unit, *2015 International conference on VLSI systems, architecture, technology and applications (VLSI-SATA)*, IEEE, pp.1–5.

[25] **Jain, A.**, **Dash, B.**, **Panda, A.K. and Suresh, M.** FPGA design of a fast 32-bit floating point multiplier unit, *2012 International Conference on Devices, Circuits and Systems (ICDCS)*, IEEE, pp.545–547.

[26] **Tozlu, Y. and Yılmaz, Y.**, (2021), Design and Implementation of a 32-bit RISC-V Core, https://web.itu.edu.tr/~orssi/thesis/2021/YavuzTozlu_bit.pdf, (Accessed: Oct. 31, 2025).

[27] **Daysal, S. and Tuzcu, M.**, (2022), Extending the Instruction Set of RISC-V Processor for Floating Point Arithmetic, https://web.itu.edu.tr/~orssi/thesis/2022/SalihDaysal_bit.pdf, (Accessed: Oct. 31, 2025).

[28] **Ozden, M.K. and Eroglu, D.Z.**, (2025), Design and Testing of a Highly-Adaptable RISC-V Verification Environment, http://web.itu.edu.tr/~orssi/thesis/2025/MeteOzden_bit.pdf, (Accessed: Oct. 31, 2025).

[29] **RISC-V-AI**, (2025), HORNET RISC-V AI Implementation Repository, https://github.com/RISC-V-AI/HORNET/tree/main/HornetRISC-V_AI, (Accessed: Nov. 9, 2025).

[30] **Tavallaee, M.**, **Bagheri, E.**, **Lu, W. and Ghorbani, A.A.** A detailed analysis of the KDD CUP 99 data set, *2009 IEEE symposium on computational intelligence for security and defense applications*, Ieee, pp.1–6.

[31] **Lin, T.Y.**, **Goyal, P.**, **Girshick, R.**, **He, K. and Dollár, P.** Focal Loss for Dense Object Detection, *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp.2980–2988.

[32] **riscv-collab**, (2025), RISC-V GNU Toolchain Repository, https://github.com/riscv-collab/riscv-gnu-toolchain, (Accessed: Nov. 9, 2025).

[33] **Primitives, A.** Advanced Encryption Standard (AES)(FIPS-197).

[34] **Deng, L.** The mnist database of handwritten digit images for machine learning research [best of the web], *IEEE signal processing magazine*, *29*(6), 141–142.

[35] **RISC-V Foundation** The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, **Technical Report**, RISC-V Foundation, https://riscv.org/technical/specifications/, version 1.12 (Accessed: Dec. 28, 2025).

[36] **Kaggle**, (2019), NSL-KDD Dataset, https://www.kaggle.com/datasets/hassan06/nslkdd, (Accessed: Nov. 9, 2025).

Name Surname: Yusuf Tekin

Place and Date of Birth: Tekirdag, 06/10/2002

E-Mail: tekiny20@itu.edu.tr

EDUCATION:

- **B.Sc.:** 2026 (expected), Istanbul Technical University, Faculty of Electrical and Electronics Engineering, Electronics and Communication Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- YTU Prototype Workshop - Elec. and Prototyping Intern. *Jun 2023 - Jul 2023*
- AIF Informatics and Consultancy - Data Analyst Intern. *Jun 2024 - Jul 2024*
- TUBITAK BILGEM - Intern. *Aug 2025 - Sep 2025*
- TUBITAK BILGEM - Undergraduate Researcher. *Sep 2025 - Jan 2026*

Name Surname: Ahmet Tolga Özkan

Place and Date of Birth: Muğla, 02/10/2000

E-Mail: ozkanah19@itu.edu.tr

EDUCATION:

- **B.Sc.:** 2026 (expected), Istanbul Technical University, Faculty of Electrical and Electronics Engineering, Electronics and Communication Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- Analog Devices - Embedded Software Intern. *Aug 2024 - Sep 2024*
- Ekin Smart City - Electronics Intern. *Oct 2024 - Nov 2024*
- Baykar Technologies - Embedded Software Intern. *Mar 2025 - May 2025*
- Baykar Technologies - Embedded Software Intern. *Jun 2025 - Aug 2025*
- Baykar Technologies - Embedded Software Developer. *Nov 2025 - Jan 2026*