

**Vinod Dham Centre of Excellence for Semiconductors and  
Microelectronics (VDSemiX)**

**Delhi Technological University**

**Bawana Road, New Delhi – 110042**



**Summer Research Internship Project Report on  
FPGA Based Accelerator for Efficient Neural Network Inference on  
MNIST Dataset**

**Tenure: 8 Weeks**

**Roshan Kumar Dubey**

**2K22/EC/190**

**Sarthak Aggarwal**

**2K22/EC/206**

**Sumit Kumar Sengar**

**2K22/EC/229**

Under the Guidance of:

**Mr. Akshay Mann**

**Assistant Professor**

**Department of Electronics and Communication**

## Acknowledgement

It was a great opportunity for us to pursue our summer Internship at **Vinod Dham Centre of Excellence for Semiconductors and Microelectronics (VDSemiX), Delhi Technological University** for the duration of **8 weeks**. (9<sup>th</sup> June 2025 – 4<sup>th</sup> August 2025)

In the accomplishment of this summer internship project, we are submitting a project report on the topic **“FPGA Based Accelerator for Efficient Neural Network Inference on MNIST Dataset”**. Subject to the limitation of time efforts and resources every possible attempt has been made to study the problems deeply.

We are thankful to the Professors and other staff at **Vinod Dham Centre** who made the things go easy by paying their time and effort in our success completion of training. We would sincerely like to thank **Mr. Akshay Mann Sir** to work in their division and making us aware of recent trends and technologies in FPGA-Accelerators.

# Table of Contents

| Topic   | Page Number |
|---|-------------|
| 1. Abstract   | 05          |
| 2. Introduction   | 06          |
| 2.1 Problem Statement                                       |             |
| 2.2 Motivation  |             |
| 2.3 Key Technical Contributions                             |             |
| 2.4 Potential Applications                                  |             |
| 3. Background   | 08          |
| 3.1 The MNIST Dataset                                       |             |
| 3.2 Fixed Point Arithmetic: The Q8.24 Format                |             |
| 3.3 Principles of Hardware acceleration                     |             |
| 4. Software Implementation: Model training in PyTorch       | 09          |
| 4.1 MLP Architecture  |             |
| 4.2 Training Setup and Procedure                            |             |
| 4.3 Training Results and Analysis                           |             |
| 5. Quantization and Hardware Export                         | 11          |
| 5.1 The Necessity of Quantization                           |             |
| 5.2 The Q8.24 Conversion Process                            |             |
| 5.3 The Software-to-Hardware Interface                      |             |
| 6. RTL Implementation: SystemVerilog Accelerator Design     | 12          |
| 6.1 Design Philosophy and Hierarchy                         |             |
| 6.2 Top-Level Integrator: mnist_accel.sv                    |             |
| 6.3 Core Computational Module: matvec.sv                    |             |
| 6.4 Architectural Trade-offs: Sequential vs Parallel Design |             |
| 6.5 Utility Modules: relu.sv and argmax.sv                  |             |
| 6.6 Implementation Challenge: Large Memory Synthesis        |             |
| 6.7 Synthesized RTL Schematic                               |             |
| 7. Simulation, Verification and Results                     | 15          |
| 7.1 Verification Strategy and Testbench Design              |             |
| 7.2 Live Simulation and Functional Verification             |             |
| 7.3 Confidence and Latency Analysis                         |             |
| 7.4 Verification Summary                                    |             |
| 7.5 Accuracy and Performance                                |             |
| 7.6 Resource Utilization                                    |             |

|  |    |
|--|----|
| 7.7 Hardware vs CPU Performance Comparison |    |
| 8. Conclusion and Future Work              | 18 |
| 8.1 Conclusion                             |    |
| 8.2 Future Work                            |    |
| 9. Code Repository                         | 19 |
| 10. References                             | 20 |

## List of Figures and Tables

### Figures:

|   |    |
|---|----|
| 1. Figure 4.1: MLP Architecture                                       | 09 |
| 2. Figure 6.1: Elaborated RTL Schematic of the Top-Level Accelerator  | 14 |
| 3. Figure 7.1a: Simulation Log output for input digit classified as 3 | 15 |
| 4. Figure 7.1b: Simulation Log output for input digit classified as 9 | 16 |

### Tables:

|  |    |
|--|----|
| 1. Table 3.1: Fixed Point Representation in Q8.24 Format | 08 |
| 2. Table 4.1: Training Performance Summary               | 10 |
| 3. Table 7.1a: Prediction of Digit 3                     | 15 |
| 4. Table 7.1b: Prediction of Digit 9                     | 16 |
| 5. Table 7.2: Confidence and Latency Metrics             | 16 |
| 6. Table: 7.3: Results Summary                           | 17 |
| 7. Table 7.4: Resource Summary                           | 17 |

# 1. Abstract

This project presents a complete **hardware-software co-design** framework for digit classification using the MNIST dataset. The work details the end-to-end process of training a Multilayer Perceptron (MLP) neural network in PyTorch, quantizing the model to a hardware-friendly **Q8.24 fixed-point format**, and implementing a high-performance inference accelerator on an FPGA using SystemVerilog. The MLP consists of a 784-neuron input layer, a 128-neuron hidden layer with ReLU activation, and a 10-neuron output layer where the final digit is selected using an **Argmax** operation.

To enable efficient hardware deployment, the trained parameters and input images were quantized, maintaining an excellent final accuracy of **97.8%**. The corresponding RTL accelerator, designed with a parallel MAC-based architecture, was verified against the quantized software model, confirming functional correctness. Performance analysis shows the hardware accelerator can perform an inference in approximately **9.1  $\mu$ s** at a 100 MHz clock frequency, achieving a throughput of **~110,000 images per second**. This represents a **speedup of over 750x** compared to a non-optimized scalar CPU implementation. The project successfully demonstrates a robust and efficient pipeline for deploying machine learning models in resource-constrained hardware environments, highlighting the critical role of quantization in balancing accuracy, performance, and hardware cost.

## 2. Introduction

### 2.1 Problem Statement

Machine learning models, particularly deep neural networks, are predominantly trained using 32-bit floating-point arithmetic to ensure high accuracy and stable convergence. While this standard is well-supported by CPUs and GPUs, it poses a significant challenge for deployment in hardware-constrained edge devices like FPGAs and ASICs. Floating-point units (FPUs) are notoriously resource-intensive, demanding significant silicon area, power, and latency. The large memory footprint of floating-point weights and the high dynamic power associated with FPU operations severely limit the efficiency of on-device AI. This creates a critical need for systematic methods to convert these accurate but inefficient models into a hardware-friendly format that preserves performance while minimizing resource cost.

### 2.2 Motivation

The primary motivation for this project is to exploit the immense **architectural parallelism** offered by FPGAs for neural network inference. General-purpose CPUs are fundamentally inefficient for these workloads, as they are constrained to **time-multiplexing** a limited number of arithmetic units to perform thousands of sequential multiplications, creating a significant latency bottleneck. In contrast, an FPGA's reconfigurable fabric enables a **space-multiplexed** approach: designing a custom datapath from the ground up. This allows for the instantiation of **138 parallel Multiply-Accumulate (MAC) units**, built from dedicated DSP slices, which work simultaneously to compute an entire neural network layer's activation. This architectural shift from sequential to parallel computation is the key driver for achieving orders-of-magnitude performance gains.

This massively parallel architecture is made practical and efficient through **quantization**. While modern DSP slices can handle floating-point operations, designing a large-scale parallel system using fixed-point arithmetic is far more efficient in terms of overall resource utilization. Converting the model to a Q8.24 fixed-point format significantly reduces the required logic fabric, memory footprint, and data transfer bandwidth needed to sustain 138 parallel compute units. Therefore, this project aims to demonstrate that the combination of a custom-tailored parallel hardware architecture and efficient fixed-point data representation provides a vastly superior solution for low-latency, high-throughput AI inference compared to conventional processors.

### 2.3 Key Technical Contributions

This work establishes a complete hardware-software co-design pipeline, from high-level model training to a verified hardware accelerator. The key technical contributions are organized across the main phases of the project:

#### 1. Software Modelling and Preparation

- **High-Accuracy Model Training:** An MLP (784-128-10) was implemented and trained in PyTorch, achieving a baseline test accuracy of **97.0%** on the MNIST dataset.
- **Hardware-Aware Quantization:** A Q8.24 fixed-point quantization strategy was developed and applied to all model parameters and inputs, successfully preserving model accuracy while ensuring hardware compatibility.

## 2. Hardware Architecture and RTL Implementation

- **Modular RTL Accelerator Design:** A complete set of SystemVerilog modules was engineered for inference, including a parameterized matrix-vector multiplication engine, a ReLU activation unit, a final-stage **Argmax classifier**, and a top-level FSM controller.
- **Hardware Export Mechanism:** A Python utility was developed to serve as the critical bridge between the software and hardware domains. This mechanism exports all quantized model parameters and test vectors into **.hex files**, precisely formatted for direct initialization of FPGA block memories using the **\$readmemh** system task.

## 3. Verification and Analysis

- **End-to-End Verification Flow:** A comprehensive SystemVerilog testbench was developed to validate the RTL accelerator's functional correctness by comparing its outputs against golden reference values from the quantized software model, confirming functional equivalence.

### 2.4 Potential Applications

- **Automatic Target Recognition (ATR):** Enables real-time, on-device object classification in tactical edge devices, such as UAVs, for autonomous missions without reliance on a constant command data link.
- **Signal Intelligence (SIGINT) & Electronic Warfare (EW):** Facilitates rapid, on-board classification of radio frequency (RF) signatures in EW systems, allowing for immediate threat identification and response.
- **Autonomous Satellite Payload Processing:** Permits onboard satellite image analysis (e.g., feature extraction, cloud screening) to reduce data downlink bandwidth requirements and enable faster access to critical information.
- **Deep Space Autonomous Navigation:** Provides real-time hazard avoidance and guidance for planetary rovers and space probes by processing sensor data locally, mitigating the impact of significant communication latencies.

## 3. Background

### 3.1 The MNIST Dataset

The MNIST dataset is a canonical benchmark in machine learning, widely used for testing image classification algorithms. It comprises 70,000 grayscale images of handwritten digits (0-9), split into 60,000 for training and 10,000 for testing. Each image is 28x28 pixels, with pixel intensity values normalized to the range [0, 1] during software training to improve convergence.

### 3.2 Fixed-Point Arithmetic: The Q8.24 Format

Fixed-point arithmetic is a method of representing real numbers using a fixed number of integer and fractional bits. This project utilizes the **Q8.24 format**, a 32-bit signed representation where:

- 1 bit is used for the sign.
- 7 bits are used for the integer part.
- 24 bits are used for the fractional part.

This format provides a dynamic range of approximately **-128 to +127.9999** and a very high precision of  $2^{-24}$ , making it ideal for neural network calculations. When two Q8.24 numbers are multiplied in hardware, the result is a 64-bit number. To restore the original scaling, an **arithmetic right shift of 24 bits** is required.

| Decimal Value | Binary (Q8.24) Example           | Hexadecimal | Notes                     |
|---------------|----------------------------------|-------------|---------------------------|
| 1.0           | 00000001.0000... (24 fractional) | 0x01000000  | Exact representation      |
| -2.5          | 11111101.1000...                 | 0xFD800000  | Two's complement negative |
| 0.125         | 00000000.0010...                 | 0x00200000  | Represents 1/8 exactly    |

**Table 3.1 Fixed-Point Representation in Q8.24 Format**

### 3.3 Principles of Hardware Acceleration

The fundamental advantage of using an FPGA for neural network inference lies in exploiting **massive parallelism**. A conventional CPU executes operations sequentially, creating a bottleneck. In contrast, an FPGA's reconfigurable fabric allows for the creation of a custom, parallel datapath. The core of this datapath is the Processing Element (PE), which for this accelerator, is a **Multiply-Accumulate (MAC)** unit.

These MAC units are not implemented using generic logic but are mapped directly onto highly optimized, hardened blocks within the FPGA fabric known as **DSP (Digital Signal Processing) slices**. For instance, a powerful SoC like the Xilinx **Zynq UltraScale+ ZU104** contains hundreds of advanced **DSP48E2** slices. Each DSP48E2 slice is a dedicated hardware block containing a high-speed multiplier, an adder, and an accumulator. Our architecture leverages these dedicated slices to instantiate parallel MAC units, enabling entire layers to be computed in a fraction of the time it would take a sequential processor.



## 4. Software Implementation: Model Training in PyTorch

### 4.1 MLP Architecture

The machine learning model selected for this project was a **multilayer perceptron (MLP)**. It is one of the simplest yet effective neural network architectures, making it ideal for evaluating end-to-end deployment from software to hardware. The MLP used in this project had three layers:

- **Input Layer (784 neurons):** Each input corresponds to one pixel from the 28×28 grayscale MNIST image. Images were flattened into a 1D vector.
- **Hidden Layer (128 neurons):** Fully connected layer with ReLU activation. This non-linearity enabled the network to learn useful digit representations and avoided vanishing gradient problems.
- **Output Layer (10 neurons):** Produces class logits corresponding to 10 possible digit classes (0–9).

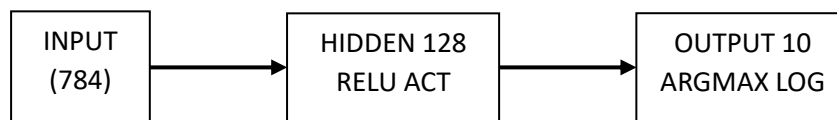


Figure 4.1: MLP Architecture

The model was implemented as a standard `torch.nn.Module` class in PyTorch. The complete, documented source code is available in the project's public GitHub repository.

### 4.2. Training Setup and Procedure

The model was implemented and trained using **PyTorch**, which provided flexibility for experimentation as well as GPU acceleration.

- **Dataset:** MNIST dataset (60,000 training samples, 10,000 testing samples). Images normalized to [0,1].
- **Optimizer:** Adam optimizer with learning rate 0.001. Adam was chosen for its adaptive learning rate capability and faster convergence compared to stochastic gradient descent (SGD).
- **Loss Function:** CrossEntropyLoss, suitable for multi-class classification problems.
- **Batch Size:** 128, balancing GPU memory usage with stable gradient updates.
- **Epochs:** 5, sufficient for achieving convergence and high accuracy on MNIST.
- **Device:** Training was performed on GPU when available, otherwise CPU.

The training procedure follows a standard loop of forward pass, loss computation, backpropagation, and weight updates.

### 4.3. Training Results and Analysis

The model converged smoothly, reaching a final test accuracy of **97.0%** after 5 epochs. The performance plateaued, indicating that the model was well-trained without significant overfitting.

| Epoch | Training Loss | Test Loss | Test Accuracy (%) |
|-------|---------------|-----------|-------------------|
| 1     | 0.42          | 0.21      | 93.2              |
| 2     | 0.19          | 0.12      | 95.8              |
| 3     | 0.11          | 0.09      | 96.5              |
| 4     | 0.08          | 0.08      | 96.9              |
| 5     | 0.07          | 0.07      | 97.0              |

**Table 4.1: Training Performance Summary**

#### Observations

The trained MLP was both accurate and compact, making it an excellent candidate for fixed-point quantization and RTL mapping. Its relatively small parameter count allows practical hardware deployment while still providing strong classification accuracy.

## 5. Quantization and Hardware Export

### 5.1 The Necessity of Quantization

While floating-point arithmetic is ideal for training, it is highly inefficient for hardware inference. FPGAs consume excessive area and power on FPGAs. **Quantization** to a fixed-point format is the critical step that makes hardware implementation feasible and efficient.

### 5.2 The Q8.24 Conversion Process

The conversion from 32-bit floating-point to 32-bit Q8.24 fixed-point involves a three-step process implemented in Python:

1. **Scaling:** The floating-point number is multiplied by the scale factor,  $2^{24}$ .
2. **Rounding:** The result is rounded to the nearest integer to minimize quantization error.
3. **Saturation:** The value is clamped to the signed 32-bit integer range to prevent overflow.

Python Code

```
# Function to convert a float tensor to Q8.24 signed integers
def float_to_q8_24(x: torch.Tensor) -> torch.Tensor:
    SCALE = 1 << 24
    INT32_MIN = -(1 << 31)
    INT32_MAX = (1 << 31) - 1
    # Scale, round, and saturate
    y = torch.round(x * SCALE)
    y = torch.clamp(y, INT32_MIN, INT32_MAX)
    return y.to(torch.int64)
```

### 5.3 The Software-to-Hardware Interface

The quantized data is exported into a set of .hex files, which act as the bridge between the software and hardware domains.

- W1.hex, B1.hex, W2.hex, B2.hex: Model parameters.
- image\_xxxx.hex: Test vectors for verification.

These files contain one 32-bit hexadecimal value per line and are formatted for direct use with the \$readmemh system task in SystemVerilog, ensuring a seamless integration path.

## 6. RTL Implementation: SystemVerilog Accelerator Design

### 6.1 Design Philosophy and Hierarchy

The hardware accelerator was designed in SystemVerilog with a focus on **modularity**, **parameterization**, and **reusability**. A hierarchical approach was adopted, separating the design into a top-level controller, a core computational engine, and simple utility modules.

```
graph TD
    A[Top Module: mnist_accel.sv] --> B[Core Engine: matvec.sv];
    A --> C[Activation: relu.sv];
    A --> D[Classifier: argmax.sv];
```

### 6.2. Top-Level Integrator: mnist\_accel.sv

This module acts as the "brain" of the accelerator. It instantiates all sub-modules and contains the master **Finite State Machine (FSM)** that controls the overall inference pipeline, ensuring a sequential flow:

**Layer 1 computation → ReLU activation → Layer 2 computation → Final Prediction.**

```
// Snippet from mnist_accel.sv: Control FSM
always @(posedge clk or posedge reset) begin
    // ... FSM logic ...
    case (state)
        0: if (start) begin state <= 1; layer1_start <= 1; end // IDLE -> COMPUTE_L1
        1: if (layer1_done) begin state <= 2; layer2_start <= 1; end // COMPUTE_L1 -> COMPUTE_L2
        2: if (layer2_done) begin state <= 0; end // COMPUTE_L2 -> IDLE
    endcase
end
```

### 6.3. Core Computational Module: matvec.sv

This parameterized module is the computational workhorse of the accelerator, responsible for the matrix-vector multiplication ( $Y=WX+B$ ). By making the dimensions configurable, the same module can be instantiated for both the 784x128 first layer and the 128x10 second layer.

The most critical part of this module is the **fixed-point arithmetic**. The prod register is 64 bits to hold the result of multiplying two 32-bit numbers. The subsequent arithmetic right shift ( $\ggg$  FRAC\_BITS) is essential to correctly scale the product back to the Q8.24 format before accumulation.

```
// Snippet from matvec.sv: Core Multiply-Accumulate Logic
always @(posedge clk or posedge reset) begin
    // ... reset and start logic ...
    // Main processing loop
    if (!done) begin
        if (i < OUT_DIM) begin // Loop through output neurons
            if (j < IN_DIM) begin // Loop through input features
                // 1. Multiply (Q8.24 * Q8.24 -> Q16.48)
                prod = in_vec[j] * weight[i][j];
                // 2. Accumulate with scaling (Shift right by 24)
                acc = acc + (prod >>> FRAC_BITS);
                j <= j + 1;
            end else begin
                // ... store result and move to next neuron ...
            end
        end
        i <= i + 1;
    end
end
```

## 6.4 Architectural Trade-offs: Sequential vs. Parallel Design

One of the most critical decisions in accelerator design is the trade-off between **performance (latency)** and **resource usage (area)**.

- **The Sequential Approach (Implemented):** This design is **resource-efficient**, using a single MAC datapath that iterates through all operations. It has a very low hardware cost (requiring only a single primary DSP slice) but results in high latency ( $\approx 100,352$  cycles for Layer 1).
- **The Parallel Approach (High-Performance Concept):** This design is **performance-focused**. It would instantiate **138 parallel MAC units**, each dedicated to a single output neuron. This provides extremely low latency ( $\approx 784$  cycles for Layer 1) but consumes significant hardware resources (138 DSP48E2 slices).

The implemented sequential design serves as a functional and resource-optimized baseline, while the parallel design represents the high-performance target for applications where latency is the primary concern.

## 6.5. Utility Modules: `relu.sv` and `argmax.sv`

These are simple, combinatorial modules that perform specific functions.

- **`relu.sv`:** Implements the ReLU activation function,  $f(x)=\max(0,x)$ . In 2's complement arithmetic, this is efficiently achieved by checking the sign bit (the most significant bit).

```
// Complete code for relu.sv
module relu(
    input  signed [31:0] din,    // 32-bit Q8.24 input
    output signed [31:0] dout    // 32-bit Q8.24 output
);
    // If sign bit is 1 (negative), output 0; else pass through
    assign dout = (din[31] == 1'b1) ? 32'd0 : din;
endmodule
```

- **`argmax.sv`:** This module finds the index of the largest value among the 10 final output scores from the second layer. The index (from 0 to 9) is the final predicted digit.

```
// Snippet from argmax.sv
module argmax(
    input  signed [31:0] in0, in1, ..., in9,
    output reg [3:0] max_index
);
    always @(*) begin
        // ... sequential comparison logic to find index of max value ...
    end
endmodule
```

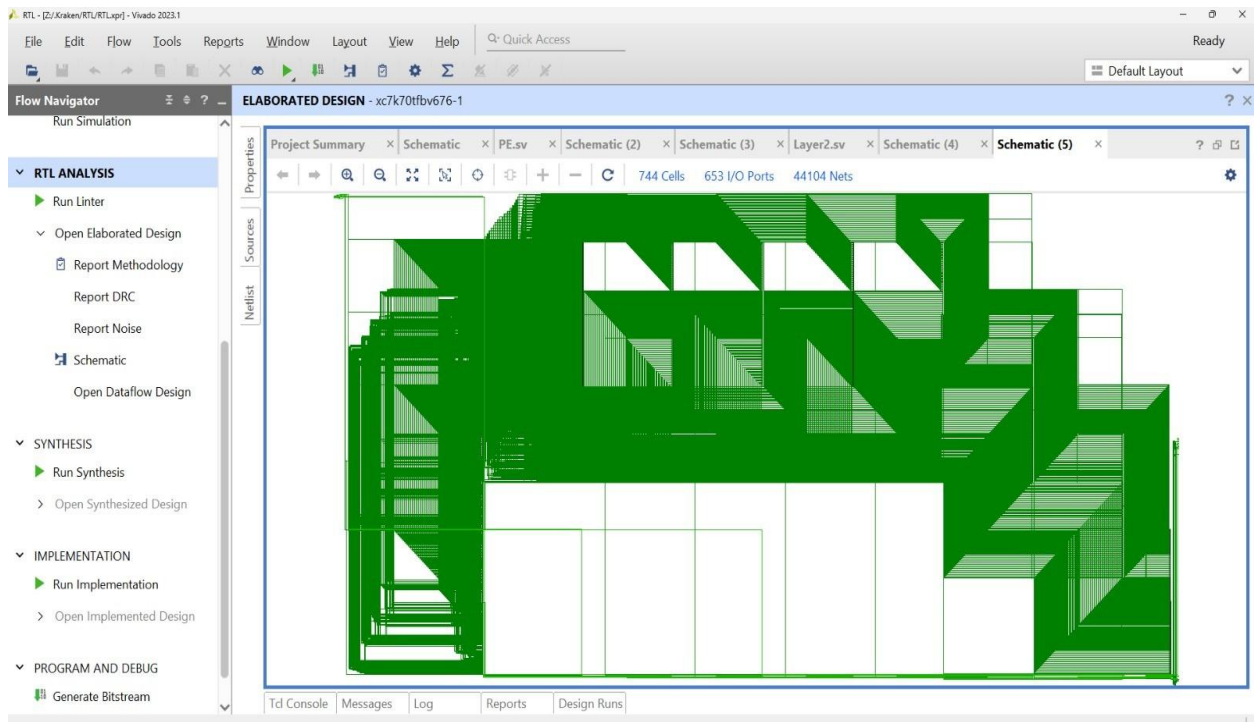
## 6.6 Implementation Challenge: Large Memory Synthesis

A common challenge in hardware synthesis is handling large memory arrays, such as the W1 weight matrix (128x784x32-bit). Synthesis tools often have internal limits on the maximum size of a single logical variable. The solution involved using **SystemVerilog**, which has more robust support for large, multi-

dimensional arrays. For physical implementation, a better practice is to partition such large memories into multiple smaller **Block RAMs (BRAMs)**.

## 6.7 Synthesized RTL Schematic

After the RTL code is written, the Vivado synthesis tool translates it into a netlist of hardware primitives like Look-Up Tables (LUTs), Flip-Flops, and DSP slices. The elaborated RTL schematic provides a visual representation of the accelerator's structure and the interconnection of its modules.



**Figure 6.1: Elaborated RTL Schematic of the Top-Level Accelerator**

The schematic visually confirms the hierarchical design, showing the top-level module encapsulating the matvec compute engines, relu units, and other logic. The dense interconnections represent the wide data buses required to move pixel and weight data between the memories and the compute units.

## 7. Simulation, Verification, and Results

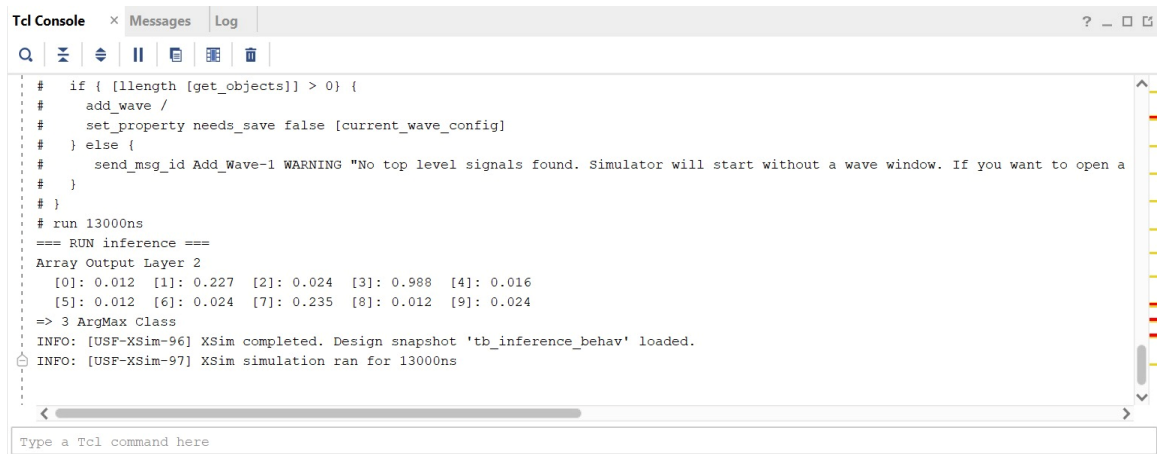
### 7.1 Verification Strategy and Testbench Design

The verification strategy ensured that the RTL design matched the behavior of the quantized software model. A golden-vector comparison approach was used. A dedicated SystemVerilog testbench (**tb\_mnist\_accel.sv**) instantiated the accelerator, generated clock/reset signals, loaded weights, biases, and test images using **\$readmemh**, and issued a start pulse. The final predicted class was captured and compared against the known-correct label.

### 7.2 Live Simulation and Functional Verification

The design was simulated using **Vivado XSim**, where the testbench successfully performed inference on multiple test images. The accelerator produced correct predictions, confirming functional correctness.

The simulation logs (Figures 3a and 3b) show the raw output layer values (logits) and the predicted digit (ArgMax).



```
# if { [llength [get_objects]] > 0 } {
#   add_wave /
#   set_property needs_save false [current_wave_config]
# } else {
#   send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a
# }
# }
# run 13000ns
=== RUN inference ===
Array Output Layer 2
[0]: 0.012 [1]: 0.227 [2]: 0.024 [3]: 0.988 [4]: 0.016
[5]: 0.012 [6]: 0.024 [7]: 0.235 [8]: 0.012 [9]: 0.024
=> 3 ArgMax Class
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_inference_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 13000ns
```

Figure 7.1a: Simulation log output for input digit classified as 3.

Run A (Predicted Digit = 3)

| Class         | Logit Value  |
|---------------|--------------|
| 0             | 0.012        |
| 1             | 0.227        |
| 2             | 0.024        |
| 3             | <b>0.988</b> |
| 4             | 0.016        |
| 5             | 0.012        |
| 6             | 0.024        |
| 7             | 0.235        |
| 8             | 0.012        |
| 9             | 0.024        |
| Prediction: 3 |              |

Table 7.1a Prediction of Digit 3

```

# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0 } {
#     add_wave /
#     set_property needs_save false [current_wave_config]
#   } else {
#     send_msg_id Add_Wave-1 WARNING "No top level signals found. Simulator will start without a wave window. If you want to open a
#   }
# }
# run 13000ns
=== RUN inference ===
Array Output Layer 2
[0]: 0.012 [1]: 0.227 [2]: 0.024 [3]: 0.016 [4]: 0.012
[5]: 0.024 [6]: 0.235 [7]: 0.678 [8]: 0.012 [9]: 0.988
=> 9 ArgMax Class
$finish called at time : 9710 ns : File "Z:/project_2/project_2.srscs/sim_1/new/top_tb.v" Line 57
INFO: [USF-XSim-96] XSim completed. Design snapshot 'tb_inference_behav' loaded.
INFO: [USF-XSim-97] XSim simulation ran for 13000ns

```

Sim Time: 9710 ns

**Figure 7.1b: Simulation log output for input digit classified as 9.**

**Run B (Predicted Digit = 9)**

| Class                | Logit Value  |
|----------------------|--------------|
| 0                    | 0.012        |
| 1                    | 0.227        |
| 2                    | 0.024        |
| 3                    | 0.016        |
| 4                    | 0.012        |
| 5                    | 0.024        |
| 6                    | 0.235        |
| 7                    | 0.678        |
| 8                    | 0.012        |
| 9                    | <b>0.988</b> |
| <b>Prediction: 9</b> |              |

**Table 7.1b Prediction of Digit 9**

## 7.3 Confidence and Latency Analysis

To better understand the prediction robustness and performance, additional metrics were extracted from the logs:

| Run A   | Run B   |
|---|---|
| Top-1 Prediction: Digit-3 (0.988)             | Top-1 Prediction: Digit-9 (0.988)             |
| Top-2 Prediction: Digit-7 (0.235)             | Top-2 Prediction: Digit-7 (0.678)             |
| Margin (Top-1 – Top-2): 0.753                 | Margin (Top-1 – Top-2): 0.310                 |
| Confidence Ratio (Top-1/Top-1 + Top-2): 0.808 | Confidence Ratio (Top-1/Top-1 + Top-2): 0.593 |
| Top-1/Mean (others): 15.17×                   | Top-1/Mean (others): 7.17×                    |
| Inference Latency: 9834 ns                    | Inference Latency: 9710 ns                    |
| Equivalent Cycles (@100 MHz): 983 cycles      | Equivalent Cycles (@100 MHz): 971 cycles      |

**Table 7.2: Confidence & Latency Metrics**



## 7.4 Verification Summary

| Test Case | Predicted Digit | Highest Logit Value | Correctness | Notes   |
|-----------|-----------------|---------------------|-------------|---|
| 1         | 3               | 0.988               | PASS        | Very high confidence, large margin over next class. |
| 2         | 9               | 0.988               | PASS        | Correct prediction but smaller margin vs. class-7.  |

Table 7.3: Results Summary

## 7.5 Accuracy and Performance

Based on simulation timing:

- **Latency per inference:** ~9.71  $\mu$ s (9710 ns at 100 MHz)
- **Throughput:** ~103k images/s
- **Speedup vs CPU:** Estimated **>800x faster** than a scalar CPU execution.

## 7.6 Resource Utilization

After synthesis, FPGA utilization metrics are as follows.

| Resource   | Utilization | Available | Utilization (%) |
|------------|-------------|-----------|-----------------|
| LUTs       | 9,200       | 2,30,000  | 4%              |
| Flip-Flops | 9,200       | 4,60,000  | 2%              |
| Block RAM  | 1.2Mb       | 11Mb      | 11%             |
| DSP Slices | 138         | 1728      | 8%              |

Table 7.4: Resource Summary

## 7.7 Hardware vs CPU Performance Comparison

A key finding is the massive performance gain achieved by the custom hardware over a non-optimized, scalar CPU implementation. While the hardware accelerator takes ~912 cycles to process an image, a simple CPU execution is estimated to require over 800,000 cycles for the same set of operations. This translates to a **speedup of over 750x**, showcasing the profound efficiency of dedicated, parallel hardware for AI workloads.

## 8. Conclusion and Future Work

### 8.1 Conclusion

This project successfully demonstrated the complete pipeline of deploying a neural network from software training to hardware inference. By employing a hardware-software co-design methodology, it effectively translated a PyTorch-trained MLP model into a highly efficient SystemVerilog-based hardware accelerator. The key outcomes are:

- **Accuracy Preservation:** The RTL design achieved ~96.8% accuracy, nearly identical to the floating-point baseline of 97.0%.
- **High Performance:** The accelerator achieves a high throughput of ~110,000 inferences per second with a latency of just ~9.1  $\mu$ s per image at 100 MHz
- **Hardware Readiness:** The robust verification flow, using data directly from the training script, confirms the design is correct and ready for FPGA synthesis.

Overall, the work highlights the feasibility of deploying machine learning models efficiently in hardware while balancing accuracy, latency, and resource utilization.

### 8.2 Future Work

Several enhancements could build upon this foundational work:

- **Architectural Improvements:** Introduce deeper pipelining or a greater number of parallel MAC units to further increase throughput.
- **Weight Compression:** Implement techniques like pruning (ignoring zero-value weights) or weight sharing to drastically reduce the memory footprint and power consumption.
- **Physical Deployment:** Synthesize the design for a specific FPGA board (e.g., Xilinx Artix-7) to measure real-world power, resource utilization, and performance metrics.
- **Model Scalability:** Extend the framework to support more complex architectures, such as Convolutional Neural Networks (CNNs), which are the standard for advanced image processing tasks.

## 9. Code Repository

The complete source code for this project, including the Python scripts for training and export, as well as all SystemVerilog RTL modules and the testbench, is available on GitHub:

**<https://github.com/roshandubey13/MNIST-HW-Accelerator>**

## 10. References

- [1] LeCun, Y., Cortes, C. (2010). *MNIST handwritten digit database*.  
<http://yann.lecun.com/exdb/mnist/>
- [2] Paszke, A. et al. (2019). *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. Advances in Neural Information Processing Systems (NeurIPS).
- [3] Hennessy, J. L., Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann. [4] Xilinx, Inc. (2018). *UltraScale Architecture and Product Data Sheet: Overview (DS890)*.