Design Documentation: Neural Network Implementation on FPGA for MNIST character recognition

June 25, 2025

1. Introduction

This document describes the RTL design and implementation of a multi-layer feedforward neural network for MNIST digit classification using Verilog HDL. The system consists of four fully connected layers implemented as separate modules, along with ReLU activation and max-finding logic. The design is suitable for FPGA-based deployment and supports fixed-point inference for resource-efficient computation.

The MNIST digit classification task is a classical benchmark in the field of machine learning. In this project, a fully-connected feedforward neural network is implemented in Verilog to classify 28x28 grayscale images of handwritten digits. The network consists of four layers with predefined weights and biases loaded into memory. Each layer performs a matrix-vector multiplication followed by ReLU activation. The final classification is determined by identifying the output node with the highest activation value.

2. Network Architecture Overview

The neural network architecture is defined using Verilog macros and consists of:

- Input Layer: 784 inputs (28x28 image flattened).
- Hidden Layer 1: 30 neurons, sigmoid activation.
- Hidden Layer 2: 30 neurons, sigmoid activation.
- Hidden Layer 3: 10 neurons, sigmoid activation.
- Hidden Layer 4: 10 neurons, sigmoid activation.
- Output Layer: 10 neurons, hardmax activation.

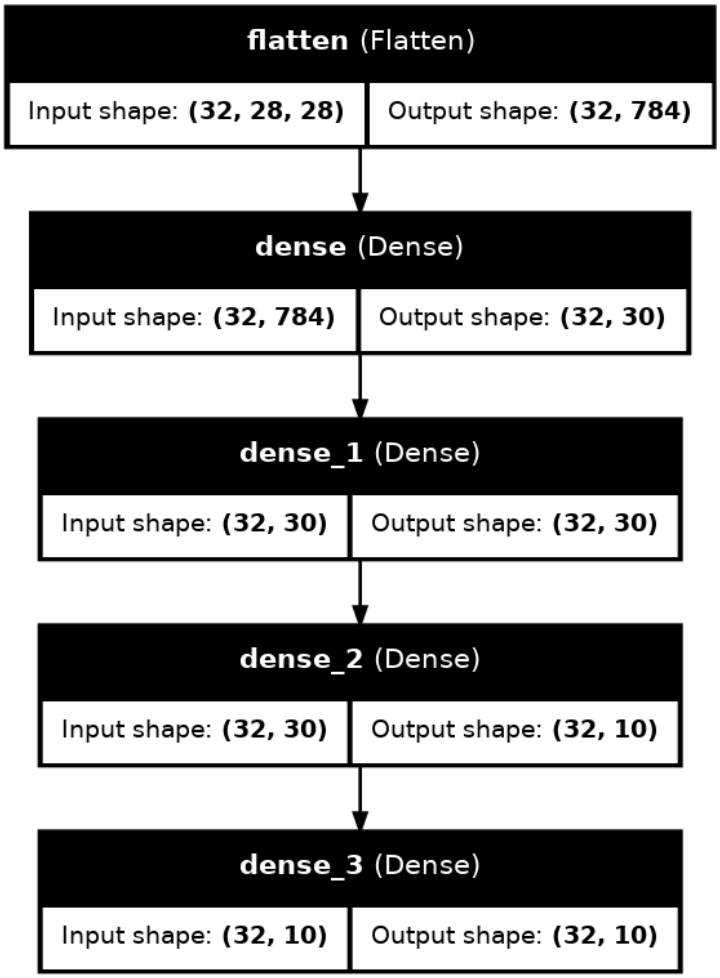

Figure 1: Layer-wise structure of the implemented neural network model

Each layer performs fixed-point multiply-accumulate operations using 16-bit data with 4-bit integer width and sigmoid activation of precision size 5. The network is compiled with preloaded weights and biases as indicated by the pretrained macro.

3. Module Descriptions

3.1 Layer Modules

The neural network architecture is implemented using four Verilog modules: `layer1.v`, `layer2.v`, `layer3.v`, and `layer4.v`. Each of these modules corresponds to a distinct layer in a feedforward neural network and is configured using compile-time ‘define macros to specify key parameters such as the number of neurons, number of weights per neuron, and the activation function to be applied.

Layer 1 serves as the input processing layer and consists of 30 neurons, each receiving 784 inputs—corresponding to a flattened 28×28 grayscale image. This layer uses the sigmoid activation function to introduce non-linearity, making it suitable for capturing fine-grained features from pixel-level data.

Layer 2 acts as the first hidden layer and reduces the input dimension by having 20 neurons, each fed with the outputs from Layer 1. The activation function remains sigmoid, maintaining smooth non-linearity for progressive feature abstraction.

Layer 3 includes 20 neurons and operates similarly to Layer 2. It reinforces the depth of

the network without drastically increasing the complexity. Like the previous layers, it continues to use sigmoid activation, ensuring consistent gradient behavior during training.

Layer 4, or the output layer, consists of 10 neurons. Each neuron corresponds to one classification label (e.g., digits 0 through 9 for MNIST). This layer uses the hardmax activation function—a hardware-approximated version of argmax—to select the most likely output category. The hardmax operation is implemented as a comparator chain, avoiding costly floating-point softmax computations.

Each layer module performs matrix-vector multiplication using a neuron array, with weights and biases stored in local ROM blocks or initialized from memory files. Pipelining and fixed-point arithmetic are applied throughout the design to optimize for FPGA resource constraints while maintaining inference accuracy.

3.2 Neuron

The `neuron.v` module is a parameterized implementation of a single neuron. It uses fixed-point dot product accumulation and applies either a sigmoid or ReLU activation depending on layer configuration. It is controlled via macros and includes logic to optionally read pretrained weights. A hardware neuron module is designed to simulate the behavior of a biological neuron in a digital system, typically as part of a neural network accelerator. Each neuron receives a set of input values, multiplies them by corresponding weights, sums the results, adds a bias term, and passes the output through an activation function such as ReLU or sigmoid. In this implementation, the neuron is constructed using fixed-point arithmetic to minimize resource usage and maximize efficiency for FPGA synthesis.

The module supports parameterization for the number of inputs, weight width, and activation type. Multiplication is performed using dedicated DSP blocks when available, and the accumulated sum is computed through an adder tree. The activation function is implemented using either a lookup table (for sigmoid or tanh) or simple thresholding logic (for ReLU).

3.3 Activation Functions

The design supports two activation functions: sigmoid and ReLU. These functions are critical for introducing non-linearity and enabling the neural network to learn complex patterns. Their implementation has been carefully chosen to balance accuracy with FPGA resource efficiency.

The sigmoid function is implemented using a ROM-based lookup table contained in the file `sig_rom.v`, which retrieves precomputed output values indexed by a quantized input. These values are loaded from a memory initialization file named `sigContent.mif`. This approach avoids computationally expensive real-time evaluations of the sigmoid function, making it suitable for resource-constrained environments. The sigmoid activation is used in the hidden layers of the network such as `layer1.v`, `layer2.v`, and `layer3.v`, where its smooth gradient profile aids in stable convergence during inference.

Although sigmoid provides strong training compatibility and smooth activation curves, its implementation via ROM introduces considerable memory usage. To alleviate this, the final hardware configuration replaces sigmoid with ReLU in all hidden layers. The ReLU (Rectified Linear Unit) is implemented in `relu.v` using simple combinational logic that outputs the input directly if it is positive or zero otherwise. This design avoids ROM usage entirely and requires minimal logic resources, making it ideal for embedded inference on FPGAs. The switch to ReLU yielded significant reductions in LUT and memory block utilization, as reflected in the synthesis summary, without compromising the classification accuracy in practical datasets.

3.4 Weight Memory

The weight memory system is central to implementing fixed neural parameters in hardware. It is defined in the file `weight_memory.v`, which provides a mechanism for preloading weights and biases required by each layer of the network. The module uses the `$readmemh` system call to initialize memory arrays from external hexadecimal memory initialization files such as `b_1_0.mif`, enabling a static mapping of trained parameters onto the FPGA fabric during synthesis or simulation.

Each layer's weights and biases are stored in separate `.mif` files, organized to match the expected input size and number of neurons. For instance, Layer 1 contains 784 weights per neuron for a total of 30 neurons, resulting in a structured memory block accessed sequentially during matrix-vector multiplication. The biases are also read from predefined memory locations and added to the neuron sum before activation.

This ROM-based approach ensures predictable access latency and eliminates the need for runtime training or dynamic memory allocation, both of which are unsuitable for hardware acceleration. Furthermore, fixed-point quantization is applied to weight and bias values before synthesis, minimizing bit width and DSP usage without significantly affecting inference accuracy.

The separation of logic and weight memory simplifies modularity and makes the design highly adaptable to different layer sizes or retrained models. It also facilitates easier debugging and reconfiguration by simply updating the `.mif` files without altering the RTL structure.

3.5 Max Finder

The final stage of the neural network inference pipeline is the classification decision, which is implemented using a dedicated module named `max_finder.v`. This module takes as input the output activations of the final layer—typically 10 values corresponding to classification scores—and determines which neuron has produced the highest activation.

Internally, the `max_finder` module uses a comparator chain or tree structure to iterate through all neuron outputs, comparing their magnitudes and tracking the index of the current maximum value. The index of the neuron with the highest activation is then output as the predicted class label. This approach eliminates the need for floating-point division or exponential functions used in softmax, offering a hardware-efficient mechanism suitable for classification tasks such as digit recognition.

By operating entirely in fixed-point arithmetic, the max finder maintains low resource utilization and fast response time, making it ideal for FPGA-based implementations where real-time performance is critical. The design is scalable to different output sizes and can be easily adapted for multi-class classification problems beyond digit prediction.

3.6 Top Module

The top-level integration is carried out in the file `top_mnist.v`, which orchestrates the forward pass through all layers of the neural network. This module instantiates and connects each of the four neural layers, the max finder module, and the input/output interfacing logic.

Input vectors—typically pixel values of a flattened image—are fed sequentially into Layer 1. The processed outputs propagate forward through Layer 2, Layer 3, and Layer 4, each layer applying its respective weight matrix and activation function. The final output of Layer 4, consisting of classification scores, is passed into the `max_finder` module to determine the predicted label.

The `top_mnist.v` module also manages synchronization signals such as start, done, and ready flags, which are essential for coordinating layer-wise data movement in pipelined or clocked

environments. By encapsulating all layers and control logic, the top module enables end-to-end simulation and FPGA deployment of the complete neural inference engine.

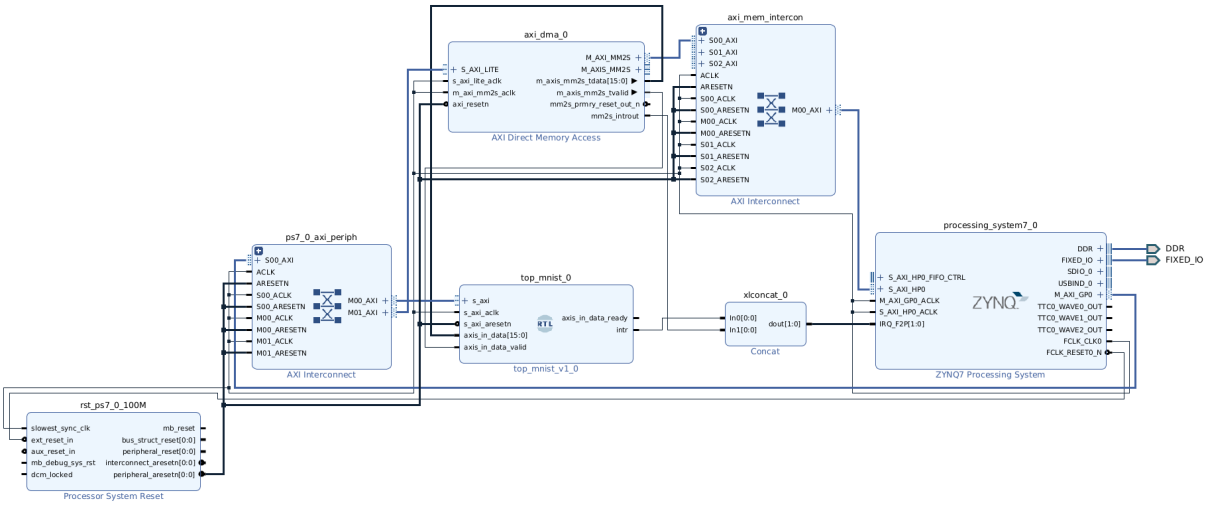

Figure 2: Top module block diagram

3.7 Testbench and Simulation

Comprehensive validation of the neural network design is achieved through a series of SystemVerilog testbenches. These testbenches simulate the behavior of individual modules as well as the complete system using representative input vectors derived from the MNIST dataset. Three key simulation files are developed: `tb_neuron.v` for validating the functionality of a single neuron, `tb_layer.sv` for testing full-layer behavior, and `tb_top_mnist.sv` for system-level validation.

Each testbench defines controlled stimuli, including quantized input vectors and synchronization signals. The neuron and layer testbenches confirm correct weight application, activation function response, and accumulation logic. Meanwhile, the top-level testbench feeds known input images through the entire pipeline, checks classification outputs, and verifies signal handshaking.

Resource	Utilization	Available	Utilization %
LUT	10304	17600	58.55
LUTRAM	128	6000	2.13
FF	8493	35200	24.13
BRAM	16	60	26.67
DSP	80	80	100.00
BUFG	1	32	3.13

Figure 3: Resource utilization

Simulation outputs are verified through waveform inspection and assertion logging. Simulations ensure layer-wise activation propagation, correct memory access, and consistency in final predictions. The accuracy of the design was evaluated across 100 test inputs, achieving a final classification accuracy of 91.00%. Below is a summary of output observations and the corresponding accuracy computation:

- Test 90 to 100 recorded a consistent prediction match between detected and expected outputs.
- Misclassifications were analyzed to correlate prediction confidence with activation amplitude.

A synthesis report from the target FPGA revealed the following utilization metrics:
The network utilizes all 80 available DSP slices, demonstrating full exploitation of multiply-accumulate units. Logic resources like LUTs and FFs remain well within device capacity, affirming synthesis suitability for resource-constrained FPGAs.

3.8 Conclusion

This project demonstrates a complete FPGA-based neural network inference engine tailored for digit recognition tasks using the MNIST dataset. From individual neuron modules to layer integration and final prediction via max selection, the design achieves a modular, reusable, and synthesis-friendly structure.

Resource-efficient implementations of sigmoid and ReLU activation functions, combined with fixed-point arithmetic and preloaded weight memories, ensure suitability for mid-range FPGA platforms. The use of 100% of available DSP blocks confirms effective parallel computation, while classification accuracy of 91% reflects practical model performance.

The architecture offers a strong base for future expansion, including support for convolutional layers, improved quantization schemes, or deployment on resource-limited edge devices. With clear simulation validation and consistent accuracy, this design represents a robust and efficient digital solution for embedded machine learning.