Deploying a CNN on PYNQ-Z2 for MNIST Digit Recognition

CSE 4602 Capstone Design Project Final Report

Submitted to Professor Hall and the Department of Computer Science & Engineering

Team Members

Alexander Wang, McKelvey School of Engineering, Washington University in St. Louis, w.yuanji@wustl.edu

Howard Hua, McKelvey School of Engineering, Washington University in St. Louis, h.howard@wustl.edu

Jade Gutierrez, McKelvey School of Engineering, Washington University in St. Louis, d.c.gutierrez@wustl.edu

Project Advisor

Michael Hall, McKelvey School of Engineering, Washington University in St. Louis, mhall24@wustl.edu

Date of Submission: May 1, 2026

Project Duration: January 2026 – May 2026

Abstract

This project implements a compact convolutional neural network for MNIST digit recognition [1] on a PYNQ-Z2 FPGA platform [2]. The design couples a Zynq processing system with custom AXI-Lite control logic and AXI-Stream datapaths so that the CPU can select a layer, load learned weights and biases, and stream an input image into the accelerator. Inside the FPGA, the CNN executes as a sequence of convolution, max-pooling, fully connected, and softmax stages managed by a runtime top module that routes weight streams to the appropriate layer. The final wrapper exposes a software-friendly control interface while preserving streaming throughput for inference. The resulting system demonstrates 99.094% accruacy on the python MNIST test cases and approximately 100% classification accuracy on our own hand-drawn digit tests through the PYNQ interface.

Contents

1 Introduction

1.1 Background and Motivation

Handwritten digit recognition is a standard benchmark for evaluating embedded machine learning systems because it combines a simple input domain with a meaningful end-to-end inference task. MNIST is especially popular because it is standardized, publicly available, and easy to compare across different models and hardware platforms. For this project, the goal was not only to train a CNN in software, but to deploy the model on FPGA hardware and support runtime control from the ARM processing system on a PYNQ-Z2 board. That combination creates a useful test case for accelerator design: the model is small enough to fit on a low-cost device, yet the system still needs a clean control path, a streaming data path, and a reliable method for loading parameters into the fabric.

The implementation in this repository follows that split. Software handles device interaction and weight delivery, while the FPGA implements the CNN datapath. The Vivado block design shows the Zynq processing system, AXI interconnects, DMA paths, a custom CNN wrapper, and the supporting BRAM-to-AXIS streaming logic used to move learned parameters into the accelerator.

1.2 Problem Statement

The problem addressed here is to classify MNIST digits on FPGA hardware using a design that fully relies on FPGA programmable logic for runtime computation, only using the ARM processing system for control and data management. The system must support three distinct data movements: control transactions over AXI-Lite, weight and bias loading over AXI-Stream, and image and classification streaming through the CNN pipeline. It must also preserve correct handshake behavior under backpressure so that the accelerator can stall safely when downstream logic is not ready.

1.3 Project Objectives and Requirements

The primary objective was to build a deployable MNIST CNN pipeline that could be controlled from the processor and run entirely in the programmable logic once the weights were loaded. A second objective was to make the design understandable and debuggable by separating the top-level control wrapper from the layer-level runtime modules. A third objective

was to document the protocol behavior clearly enough that software users and hardware reviewers can trace each transaction through the system.

Functional Requirements:

FR1 Accept AXI-Lite control writes to select a layer and start parameter loading.

FR2 Accept AXI-Stream weight and bias data for Conv1, Conv2, FC1, and FC2.

FR3 Accept an input image stream and return a streamed classification result.

Non-Functional Requirements:

NFR1 Preserve valid AXI handshake behavior under backpressure.

NFR2 Keep the design modular so layers can be inspected and tested independently.

NFR3 Fit the accelerator into a low-cost FPGA SoC workflow suitable for class-scale deployment.

2 System Design & Implementation

2.1 System Overview

Our MNIST number recognition core is first designed in Python using PyTorch, where we train a simple CNN architecture consisting of two convolutional layers followed by two fully connected layers. After training, we export the learned weights and biases into fixed-point format and save them as memory initialization files. These files are then used to load parameters into the FPGA accelerator at runtime (Figure 1).

Afterwards, we take the trained weights and biases and integrate them into the hardware design which uses a Zynq processing system to control a custom CNN accelerator through AXI-Lite and AXI-Stream interfaces. The Vivado block design shown in the project screenshot includes the processor system, AXI interconnects, DMA paths, the CNN wrapper, and a BRAM-to-AXIS component used to move learned parameters into the accelerator. At the top of the custom logic, `cnn_layers_axi_wrapper.v` exposes the processor-facing interfaces and forwards them to `mnist_simplecnn_axis_bram_wrapper.v`. That wrapper, in turn, instantiates `mnist_simplecnn_runtime_top.sv`, which sequences the actual CNN stages.

The runtime top implements the inference chain as Conv1 → Pool1 → Conv2 → Pool2 → FC1 → FC2 → Softmax as shown in Figure 2. The convolution and fully connected layers each support a load phase for weights and biases, followed by a collect-compute-stream cycle. The pooling and softmax stages are stream-oriented and operate directly on the outputs of the preceding layers.

2.2 Design Approach

The design is intentionally layered. The processor-facing wrapper owns protocol details and register semantics, while the runtime top manages CNN sequencing, and the individual layer modules focus on arithmetic and stream behavior. This separation made it easier to debug protocol issues without mixing them with layer math. It also allowed the same control path to handle all four parameterized load targets by selecting one layer at a time with a small register field.

The main trade-off was between simplicity and throughput. The implementation favors clarity and deterministic behavior over aggressive pipelining: each layer collects its input frame, computes locally, and then streams its output. That choice makes AXIS backpressure and completion signaling easy to reason about, which was important for integration with DMA and software control.

2.3 Implementation

The accelerator was implemented in SystemVerilog and Verilog inside Vivado projects that generate the programmable logic bitstream and hardware handoff files. The design is organized into individual CNN stages, top-level wrappers, and a Python control application on PYNQ that loads the bitstream, transfers parameters, and drives inference runs.

Debugging focused on protocol correctness first: verifying that AXI-Lite writes set the expected control register fields, that the load stream only advanced when the target layer asserted `tready`, and that each layer asserted `tlast` only on the final element of an output frame. Once the handshakes were stable, the pipeline was tested layer by layer and then as a full chained accelerator.

Tools and Technologies:

- **Vivado:** FPGA design and bitstream generation.
- **SystemVerilog/Verilog:** CNN layers, wrappers, and stream control.

- **Python on PYNQ:** Bitstream loading, AXI control, and inference testing.

2.4 Python CNN Training and Weight Export

Before implementing the CNN inference pipeline on FPGA, the network was first trained in Python using PyTorch. The purpose of this software stage was to obtain a set of trained convolutional and fully connected parameters that could later be converted into fixed-point format and loaded into the hardware accelerator.

The MNIST dataset was loaded from the original IDX image and label files. The image data were converted into NumPy arrays, normalized from the original pixel range $[0, 255]$ to $[0, 1]$, and reshaped into the PyTorch convolution format `NCHW`. Therefore, each MNIST image was represented as a tensor with shape `[1, 28, 28]`. The labels were converted into integer class indices for classification training.

The software model follows the same layer order as the hardware design: Conv1, Pool1, Conv2, Pool2, FC1, and FC2. Specifically, the first convolution layer maps the single-channel input image to 32 feature maps using a 3×3 kernel. After ReLU activation and 2×2 max-pooling, the second convolution layer maps 32 feature maps to 64 feature maps, again using a 3×3 kernel. A second max-pooling operation reduces the feature-map size, after which the result is flattened and passed through two fully connected layers. The first fully connected layer maps the flattened 64×5×5 feature vector to 64 hidden units, and the second fully connected layer maps the 64 hidden units to 10 output logits, corresponding to the ten digit classes.

The model was trained using cross-entropy loss and the Adam optimizer with a learning rate of 10^{-3}. The training set was split into a training subset and a validation subset, with 10% of the training data used for validation. The implementation used a batch size of 64 and trained the network for 5 epochs. After training, the model was evaluated on the MNIST test set to check that the software network produced reliable classification results before exporting its parameters for hardware use.

After training, the PyTorch `state_dict` was moved to the CPU and saved as a reference file. Then each layer's weights and biases were exported separately. The exported parameters include `conv1.weight`, `conv1.bias`, `conv2.weight`, `conv2.bias`, `fc1.weight`, `fc1.bias`, `fc2.weight`, and `fc2.bias`. To make these parameters usable by the SystemVerilog hardware modules, the floating-point tensors were quantized into signed 16-bit fixed-point values. The final trained accuracy of the software model on the MNIST test set was approximately

99.094%, which provided confidence that the exported parameters would yield good results when deployed on the FPGA.

The fixed-point export uses 8 fractional bits. Therefore, each floating-point value x is converted according to

$$x_{\text{fixed}} = \text{round}(x \times 2^8).$$

The result is then clipped to the signed 16-bit range $[-32768, 32767]$ and stored as an `int16` value. For Verilog memory initialization and BRAM loading, each fixed-point value is written to a `.mem` file as a 16-bit hexadecimal word. This format matches the hardware data width used by the convolution and fully connected modules.

In addition to exporting the trained layer parameters, the Python script also exports one quantized sample input image and one reference software inference result. The sample input is saved in the same fixed-point `.mem` format as the weights, while the software output logits are saved as a NumPy file. These files are useful for hardware verification because the FPGA output can be compared against the Python model output using the same input image and trained parameters.

Figure 2 summarizes the complete Python-side workflow from dataset loading to hardware-ready parameter export.

2.5 Convolution Layer Design

The convolution layer is implemented as a parameterized AXI-Stream hardware module that supports both runtime parameter loading and fixed-point convolution computation. The module can be configured by input channel count, output channel count, input feature-map size, kernel size, data width, accumulator width, fractional-bit position, and optional ReLU activation. This allows the same RTL structure to be reused for different convolution stages in the CNN by changing parameters such as `IN_CH`, `OUT_CH`, `IN_H`, `IN_W`, and `K`. In this project, the layer is mainly used for 3×3 convolution in the MNIST CNN pipeline.

The design uses three internal block-RAM style arrays to store the input feature map, convolution weights, and bias values. The weight array stores all kernel coefficients in the order of output channel, input channel, kernel row, and kernel column, while the bias array stores one bias value for each output channel. Parameters can either be initialized from memory files through `$readmemh` when `STATIC_INIT` is enabled, or loaded at runtime through the dedicated parameter AXI-Stream interface. During runtime loading, the module first writes incoming stream words into the weight memory and then writes the remaining words into the

bias memory. When all `W_SIZE + B_SIZE` parameter words have been received, the module raises `load_done` for one cycle and moves to the input collection phase.

The convolution computation is controlled by a finite state machine with several major phases: parameter loading, input collection, bias initialization, multiply-accumulate computation, output write-back, and output streaming. Figure 3 shows the simplified FSM flow of the convolution layer.

During the input collection phase, the layer asserts `s_axis_tready` and stores each incoming input feature-map value into `input_fm`. After the full input frame has been collected, the module begins computing output pixels. For each output channel and output spatial location, the FSM first reads the corresponding bias value and initializes the accumulator as

$$acc = bias \ll FRAC\_BITS.$$

The left shift aligns the bias with the fixed-point accumulation format, because each input-weight multiplication produces a value with extra fractional bits.

The multiply-accumulate loop then iterates through all input channels and all kernel positions. For each kernel element, the module computes the input and weight addresses using the current output channel `oc`, output row `oh`, output column `ow`, input channel `ic`, kernel row `kh`, and kernel column `kw`. The corresponding input value and weight value are read from memory, multiplied, and added into the accumulator. Since the memories are synchronous, the design uses separate address, wait, multiply, and accumulation states to account for read latency and keep the timing behavior deterministic.

After all kernel elements for one output value have been accumulated, the result is shifted right by `FRAC_BITS` to return it to the layer output fixed-point format. The module then applies either signed saturation or ReLU followed by saturation. If `USE_RELU` is enabled, negative results are clipped to zero; otherwise, the signed value is preserved. The saturation logic prevents overflow when the accumulator result exceeds the representable `DATA_W`-bit signed range.

The output is sent through the AXI-Stream master interface. The module asserts `m_axis_tvalid` in the output phase and waits for `m_axis_tready` before advancing to the next output position. The `m_axis_tlast` signal is asserted only when the final output channel, final output row, and final output column have been reached. This ensures that the downstream layer receives a correctly framed output tensor. After the final output word is transferred, the convolution layer returns to `S_COLLECT` and is ready to process the next input frame.

Overall, this convolution design favors correctness, modularity, and predictable control over aggressive parallelism. It reuses one multiply-accumulate datapath across channels and kernel positions, which reduces hardware resource usage. Although this approach is not the fastest possible convolution architecture, it is well suited for this project because it simplifies AXI-Stream integration, supports runtime parameter loading, and keeps the computation sequence easy to debug on the PYNQ-Z2 platform.

2.6 Max-Pooling Layer Design

The max-pooling implementation is built around a compact two-state finite state machine that buffers an input feature map, computes 2x2 maxima across each channel, and then streams the downsampled result. The RTL implementation stores one input row in a small distributed RAM (`row_buffer`) and uses a few temporary registers (`odd_left`, `max_top`, `max_bot`, `max_val`) to form the 2x2 comparison window. Because pooling has no learned parameters and no multiply-accumulate datapath, the design emphasis is on correct indexing, minimal local storage, and strict handshake discipline so that downstream backpressure cannot corrupt the collected frame.

Handshake discipline is enforced by splitting behavior across two states (`S_RUN` and `S_SEND`). While in `S_RUN` the module accepts incoming AXI-Stream data (it asserts `s_axis_tready`) and updates internal buffers and spatial indices (`row`, `col`, `ch`). When a complete 2x2 pixel window is available the logic computes the pooled value and copies it into the output register `m_axis_tdata` but does not assert `m_axis_tvalid` yet. Instead the FSM transitions to `S_SEND`, which is the only state that presents a valid output beat and asserts `m_axis_tvalid`. In `S_SEND` the module waits for downstream `m_axis_tready` before committing the handshake; on a successful transfer it increments `out_count`, updates `m_axis_tlast` when `out_count` reaches the final output word (`OUT_WORDS - 1`), and returns to `S_RUN` to continue accepting input. This separation guarantees that input collection and output handshakes never overlap, which simplifies backpressure reasoning and prevents loss of acceptance-credit on the upstream stream.

The implementation uses distributed RAM buffering and careful indexing to achieve correctness and resource efficiency. A distributed RAM `row_buffer[0:INPUT_SIZE-1]` holds the current input row while the next row is streamed in, and spatial indices `row` and `col` iterate over the full input spatial domain (two rows per pooled output). Each pooled output corresponds to a 2x2 tile where even/odd row and column parity decides whether the design stores incoming values or performs comparisons. A temporary register `odd_left` holds the

left pixel of an odd column so that when the right pixel arrives, both horizontal comparisons (max_top and max_bot) can be formed and then reduced to a single pooled value max_val.

Output handshake discipline is maintained by driving s_axis_tready high only in the S_RUN state and asserting m_axis_tvalid only in S_SEND. The m_axis_tlast signal is derived from the condition out_count == OUT_WORDS - 1 so the downstream receiver sees a terminal beat on the final pooled element. The row buffer is annotated with (* ram_style = "distributed" *) for small feature-map widths to keep read/write latency single-cycle and placement predictable in the fabric.

2.7 Fully Connected Layer Design

The fully connected layer is implemented with a structure similar to the convolution layer, but the computation is organized as a matrix-vector multiplication instead of a sliding-window convolution. The module is parameterized by input vector size, output vector size, data width, accumulator width, fractional-bit position, and optional ReLU activation. This allows the same fully connected RTL design to be reused for both dense layers in the CNN by changing IN_SIZE and OUT_SIZE.

Internally, the layer contains three block-RAM style arrays: weight_reg, bias_reg, and input_vec_reg. The weight memory stores the dense weight matrix with size IN_SIZE * OUT_SIZE, the bias memory stores one bias value per output neuron, and the input vector memory stores the flattened feature vector received from the previous layer. Similar to the convolution layer, the fully connected layer supports two parameter initialization methods. If STATIC_INIT is enabled, weights and biases are loaded from memory files using $readmemh. Otherwise, the processor-side loading path streams parameters into the module at runtime. The incoming parameter stream first fills the weight memory and then fills the bias memory. After all parameter words have been loaded, load_done is asserted and the layer proceeds to collect input data.

The fully connected layer is also organized as a finite state machine with several major phases: parameter loading, input collection, bias initialization, dot-product computation, output write-back, and AXI-Stream output. Figure 4 shows the simplified FSM flow of the fully connected layer.

During S_COLLECT, the module asserts s_axis_tready and stores each incoming input word into input_vec_reg. Once IN_SIZE input values have been received, the FSM starts computing the first output neuron.

For each output neuron, the layer first reads the corresponding bias value and initializes the accumulator as

$$acc = bias \ll FRAC\_BITS.$$

Then the layer iterates over all input vector elements. For each input index, it reads one input value and one weight value from memory, multiplies them, and adds the product to the accumulator. The weight address is formed as `weight_base + in_idx`, where `weight_base` points to the beginning of the current output neuron's weight row. After one output neuron is completed, `weight_base` is increased by `IN_SIZE` so that the next neuron uses the next row of the weight matrix.

The arithmetic uses fixed-point representation. Since multiplying two `DATA_W`-bit fixed-point values produces additional fractional bits, the final accumulated result is shifted right by `FRAC_BITS` before being written to the output stream. The design includes the same signed saturation and optional ReLU logic used in the convolution layer. When `USE_RELU` is enabled, negative neuron outputs are clipped to zero; otherwise, the signed saturated value is directly streamed out. This makes the module suitable for both hidden fully connected layers, which usually use ReLU, and final classification layers, which may not require ReLU depending on the network configuration.

The output interface is AXI-Stream based. After each neuron result is produced, the module asserts `m_axis_tvalid` and waits until the downstream module asserts `m_axis_tready`. The `m_axis_tlast` signal is asserted when the final output neuron has been transferred. This guarantees that the following layer receives exactly one complete output vector per inference frame. After the final output value is streamed, the layer returns to the input collection state and can process the next flattened input vector.

This fully connected implementation uses a sequential dot-product architecture rather than computing all neurons in parallel. The main benefit is reduced resource usage, since one multiplier-accumulator datapath can be reused across all input elements and output neurons. This is appropriate for the PYNQ-Z2 implementation because the fully connected layers can require a large number of weights, and a fully parallel dense layer would consume significantly more DSP and logic resources. The chosen design therefore provides a practical balance between hardware cost, implementation clarity, and reliable AXI-Stream behavior.

2.8 Softmax Layer Design

The softmax layer is implemented as a lightweight AXI-Stream hardware module that converts a fixed-size vector of logits into normalized probability-like outputs. The design uses a four-state finite-state machine: `S_COLLECT`, `S_EXP`, `S_RECIP`, and `S_STREAM` (Figure 5). During collection, the module accepts `IN_SIZE` input logits from the AXI-Stream slave interface, stores them in an internal array, and simultaneously tracks the maximum logit value. The maximum is then subtracted from each logit before exponentiation, which improves numerical stability by preventing large positive exponent values. After this, the design computes approximate exponentials, accumulates their sum, looks up an approximate reciprocal of the sum, and finally streams each normalized output through the AXI-Stream master interface. The implementation uses fixed-point arithmetic, with exponent values and reciprocal values represented in Q0.16 format.

Instead of implementing a full exponential function and hardware divider, the design uses two lookup tables: one LUT for approximating $\exp(x_i - x_{\max})$ and another LUT for approximating $1/\sum_i \exp(x_i - x_{\max})$. The exponential LUT samples values from 0 to approximately -7.5 in steps of 0.5, while the reciprocal LUT maps the expected softmax denominator range, approximately $[1.0, \texttt{IN\_SIZE}]$, into 64 bins. This makes the design much cheaper than a mathematically exact softmax, since it avoids expensive Taylor-series exponentiation, iterative division, or large combinational arithmetic blocks. This style of approximation is also used in high-performance CORDIC-based designs, where iterative or table-assisted approximations are often used to replace costly exact mathematical operations in exchange for lower latency, fewer resources, and better hardware throughput.

The main drawback is that the result is only an approximation of true softmax. Accuracy may degrade when logits fall between LUT sample points, when the logit differences are smaller than the LUT resolution, when the denominator falls near bin boundaries in the reciprocal LUT, or when very small exponent terms are clipped into the last exponential bin. There may also be quantization error from fixed-point conversion, LUT rounding, and the final Q0.32 to Q0.16 shift.

However, in this project, we intentionally accepted this loss of precision to reduce overall flip-flop and LUT usage. By sacrificing some numerical accuracy, the design becomes more suitable for FPGA deployment where high performance, low resource usage, and deterministic streaming behavior are more important than exact floating-point softmax computation.

2.9 Control Wrapper Design

Finally, we utilized a FSM design to implement the control wrapper that interfaces with the processor. The wrapper exposes an AXI-Lite interface for control and an AXI-Stream interface for loading parameters. It maintains a register that selects which layer to load, and it generates load start signals based on AXI-Lite writes. The wrapper also routes the load stream to the appropriate layer based on the selected target, which keeps the design modular and allows the same control path to handle all four parameterized layers.

A specific design here is the BRAM to AXIS module, where we have a small block RAM that holds the weights and biases for a given layer, and the wrapper generates AXI-Stream traffic to move that data into the selected layer. The wrapper ensures that the load stream is only active when the target layer is ready to accept data, which prevents protocol violations and keeps the system stable under backpressure.

The control wrapper includes a compact BRAM-to-AXIS distributor whose role is to move weight and bias data stored in on-chip memory into the selected hardware layer over an AXI-Stream channel. As shown in Table 1, the distributor implements a small state machine that orchestrates BRAM reads, packetization, and AXIS handshakes.

`load_layer_sel`	target layer
00	Conv1
01	Conv2
10	FC1
11	FC2

Table 1: Load layer selection encoding

In operation, software writes `load_layer_sel` to pick a destination layer and pulses `load_start` to begin the transfer (Table 1). The distributor then reads BRAM words, presents them as AXI-Stream beats with correct `tvalid/tlast` behavior, and observes downstream `tready` to implement backpressure-safe transfers. Only the selected layer receives the load stream, and status registers/counters report progress and errors. The distributor supports full or partial frame loads, configurable burst sizes, and proper `tlast` generation so software can rely on an atomic, observable load operation.

2.10 Interactive GUI and PYNQ-Based FPGA Acceleration Layer

The CNN number recognition system employs a Flask-based REST API with an interactive web frontend that abstracts FPGA communication complexity. The GUI offers two input modes: image upload for batch processing and an interactive 28×28 grid canvas for real-time digit drawing. Users can draw black strokes (left-click) or erase white strokes (right-click) on a pixelated grid, with a live preview panel providing immediate visual feedback. Upon inference, the interface displays the predicted digit and a confidence score histogram showing model outputs for all ten digit classes (Figure 7).

The backend leverages the PYNQ framework to manage communication through three dedicated AXI DMA channels: one for loading trained weights into block RAM, one for streaming input images, and one for retrieving results. The `PynqCnnAccelerator` class encapsulates hardware interactions, handling buffer allocation, DMA synchronization, and error checking. The overlay automatically downloads the bitstream, resolves IP cores, and initializes weights at startup. Thread-safe operations and comprehensive timeout mechanisms prevent system hangs during hardware faults. REST API endpoints (`/process_image`, `/process_draw`, `/reload_weights`, `/health`) enable flexible integration, with input preprocessing in Q8.8 fixed-point format and JSON responses containing timing metadata and DMA statistics.

2.11 Overall Hardware Pipeline and Vivado Block Design

The overall FPGA system is built around a Zynq processing system connected to a custom CNN accelerator through AXI-Lite, AXI-Stream, AXI DMA, and AXI interconnect blocks. Figure 8 shows the final Vivado block design. The processing system is responsible for software control, DDR memory access, parameter preparation, and launching inference. The programmable logic side contains the CNN wrapper, DMA engines, AXI interconnects, reset logic, and interrupt connection logic.

The CNN accelerator itself is packaged inside the custom IP block `cnn_layers_bram_optimized`. This wrapper exposes three main streaming paths: an image input stream, a parameter loading stream, and an output stream. The image input stream sends one MNIST image into the CNN pipeline. The parameter loading stream sends trained weights and biases into the selected convolution or fully connected layer. The output stream returns the final classification result from the accelerator back to DDR memory.

A key design decision is to use separate DMA channels for image input, parameter loading, and result output. The image DMA, labeled `axi_dma_img`, is used only for inference input

data. It transfers the quantized MNIST image from DDR memory to the CNN wrapper through an AXI-Stream MM2S channel. The load DMA, labeled `axi_dma_load`, is used for runtime loading of weights and biases. It also uses an MM2S channel, but it is separated from the image DMA because parameter transfer and image transfer have different timing, different data sizes, and different control semantics. The output DMA, labeled `axi_dma_out`, uses an S2MM channel to write the accelerator output stream back into DDR memory.

This separation makes the system easier to control and debug. If image input and parameter loading used the same DMA stream, the software would need to carefully multiplex two different kinds of packets and guarantee that the CNN wrapper interpreted them correctly. By using two separate input DMAs, the image path and the parameter-loading path are independent. The image DMA is only used when running inference, while the load DMA is only used when loading or updating layer parameters. This also avoids accidental mixing of input pixels and weight data on the same stream.

The transfer size of each DMA path is determined by the tensor size and the fixed-point data width. In this project, the hardware uses signed 16-bit fixed-point data, so each value occupies 2 bytes. The MNIST input image has size $1 \times 28 \times 28$, so the image DMA transfers 784 values, or 1568 bytes, per input image. The output layer produces 10 values, so the output DMA only needs to receive 20 bytes per inference result. The parameter-loading DMA transfers the weights and biases of the selected layer. Therefore, its transfer size depends on which layer is being loaded. Detailed memory weight file sizes and corresponding DMA transfer sizes are shown in Table 2.

Table 2: Main DMA transfer sizes using 16-bit fixed-point data

Transfer target	Number of values	Bytes per value	Transfer size
Input image	$1 \times 28 \times 28 = 784$	2	1568 bytes
Conv1 parameters	$32 \times 1 \times 3 \times 3 + 32 = 320$	2	640 bytes
Conv2 parameters	$64 \times 32 \times 3 \times 3 + 64 = 18496$	2	36992 bytes
FC1 parameters	$1600 \times 64 + 64 = 102464$	2	204928 bytes
FC2 parameters	$64 \times 10 + 10 = 650$	2	1300 bytes
Output logits	10	2	20 bytes

The AXI-Lite control path is connected through the AXI interconnect. It allows the Python program running on the processing system to configure DMA registers and the CNN wrapper control registers. For example, software can select which layer should receive the loading stream, start a parameter-loading transaction, start image inference, and check status signals.

The AXI memory interconnect connects the DMA engines to the Zynq processing system high-performance memory interface, allowing the DMA blocks to read input buffers from DDR and write output buffers back to DDR without requiring the CPU to manually move every data word.

The design also includes a processor system reset block and an interrupt concatenation block. The reset block generates synchronized reset signals for the AXI interconnects, DMA engines, and custom CNN wrapper. The interrupt concatenation block combines DMA interrupt signals and forwards them to the Zynq processing system. This allows the software to detect when DMA transactions have completed, although the design can also be controlled using polling during debugging.

Overall, the block design implements a clear dataflow structure. First, the Python control program prepares the fixed-point input image and trained parameters in DDR memory. Second, the load DMA streams weights and biases into the selected CNN layer when parameter loading is needed. Third, the image DMA streams the MNIST image into the CNN wrapper for inference. Finally, the output DMA receives the final CNN output and stores it back into DDR memory, where the Python program can read and interpret the predicted digit. This structure keeps computation inside the programmable logic while using the processing system for flexible control, memory management, and testing.

3 Evaluation & Results

3.1 Testing and Data Collection

Testing was performed at two levels. Initially, the individual CNN layers were simulated in their respective testbench and behavioral projects to verify weight loading, state transitions, and stream behavior. Later, the full runtime top was exercised in the connected AXIS pipeline to verify that the layers chained correctly and that the wrapper delivered weights to the intended target layer. The data used for testing consisted of simulated all-zero inputs and the exported parameter memories produced from the trained model.

The most important evaluation criteria were protocol correctness and functional inference flow: the accelerator had to accept AXI-Lite control writes, accept load-stream traffic only when the selected layer was ready, and return a complete output frame with a valid terminal `tlast`.

3.2 Results

3.2.1 Conv2d layer testbench simulation

Our testbench considers a single-channel 4×4 input feature map and a single 3×3 convolution kernel. The input feature map is streamed into the convolution layer in row-major order:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

The convolution kernel used in the testbench is an all-ones 3×3 kernel. Since the hardware uses fixed-point data with 8 fractional bits, the value 1.0 is represented as 256 in Q8.8 format. The bias is set to zero.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

With no padding and stride 1, the 4×4 input and 3×3 kernel produce a 2×2 output feature map. Since all kernel weights are equal to one and the bias is zero, each output element is the sum of the corresponding 3×3 input window:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \xrightarrow{\text{3x3 convolution}} \begin{bmatrix} 54 & 63 \\ 90 & 99 \end{bmatrix}$$

For example, the top-left output value is computed as

$$1 + 2 + 3 + 5 + 6 + 7 + 9 + 10 + 11 = 54.$$

Similarly, the top-right output is

$$2 + 3 + 4 + 6 + 7 + 8 + 10 + 11 + 12 = 63,$$

the bottom-left output is

$$5 + 6 + 7 + 9 + 10 + 11 + 13 + 14 + 15 = 90,$$

and the bottom-right output is

$$6 + 7 + 8 + 10 + 11 + 12 + 14 + 15 + 16 = 99.$$

The testbench first pulses `load_start` and streams the layer parameters through the loading AXI-Stream interface. It sends nine weight values, each equal to `16'sd256`, followed by one zero bias value. The final bias word is marked with `load_axis_tlast`. After the layer asserts `load_done`, the testbench streams the 16 input feature-map values through the input AXI-Stream interface and marks the final input word using `s_axis_tlast`.

As shown in Figures 9 and 10, the convolution layer correctly completes the parameter-loading phase, receives the streamed input feature map, performs the 3×3 multiply-accumulate operation, and produces the expected output values 54, 63, 90, and 99. The testbench also checks the AXI-Stream output handshake by monitoring `m_axis_tvalid`, `m_axis_tready`, and `m_axis_tlast`. The `tlast` signal is expected only on the final output value, which verifies that the layer outputs a correctly framed feature map.

While the testbench input is smaller than the full MNIST feature map, it verifies the core convolution operation, fixed-point weight loading, bias handling, output ordering, and AXI-Stream control behavior. Since the convolution module is parameterized by input channel count, output channel count, input size, kernel size, data width, and fractional-bit position, the same verified logic can be scaled and reused in the full CNN pipeline.

3.2.2 Max-Pooling layer testbench simulation

Our testbench considers the following 4×4 input feature map and a 2×2 non-overlapping max-pooling operation (stride 2):

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \xrightarrow{\text{2x2 max-pool}} \begin{bmatrix} 5 & 7 \\ 13 & 15 \end{bmatrix}$$

Each pooled element equals the maximum of its 2×2 tile; e.g., top-left $= \max(0, 1, 4, 5) = 5$, top-right $= \max(2, 3, 6, 7) = 7$, bottom-left $= \max(8, 9, 12, 13) = 13$, and bottom-right $= \max(10, 11, 14, 15) = 15$.

As shown in figures 11 and 12, the max-pooling layer correctly transitions between collection and send states, asserts valid only when the output data is ready, and produces the expected downsampled feature map values based on the input stream. While the testbench input is not the full 28×28 MNIST feature map, the logic is verified to perform the correct max-pooling operation on the provided 2×2 tile, and our layer also supports dynamic data width (size of overall matrix) and input size (size of kernel) when initialized. So this ensures correct functionality at runtime when deployed in the full CNN pipeline.

3.2.3 Fully Connected layer testbench simulation

Our testbench considers a fully connected layer with an input vector of size 4 and an output vector of size 3. The input vector streamed into the fully connected layer is

$$\mathbf{x} = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}^T.$$

The testbench uses three output neurons. The weights are stored in row-major order by output neuron. Since the hardware uses fixed-point data with 8 fractional bits, the value 1.0 is represented as 256 in Q8.8 format and the value -1.0 is represented as -256. In mathematical form, the testbench weight matrix is

$$W = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ -1 & 1 & -1 & 1 \end{bmatrix}.$$

The bias vector is

$$\mathbf{b} = \begin{bmatrix} 0 & 5 & -1 \end{bmatrix}^T.$$

Therefore, the expected fully connected output is computed as

$$\mathbf{y} = W\mathbf{x} + \mathbf{b}.$$

Substituting the testbench values gives

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ -1 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} + \begin{bmatrix} 0 \\ 5 \\ -1 \end{bmatrix} = \begin{bmatrix} 10 \\ 6 \\ 1 \end{bmatrix}.$$

For the first output neuron, the result is

$$1 \times 1 + 1 \times 2 + 1 \times 3 + 1 \times 4 + 0 = 10.$$

For the second output neuron, the result is

$$1 \times 1 + 0 \times 2 + 0 \times 3 + 0 \times 4 + 5 = 6.$$

For the third output neuron, the result is

$$(-1) \times 1 + 1 \times 2 + (-1) \times 3 + 1 \times 4 - 1 = 1.$$

The testbench first pulses `load_start` and streams the fully connected layer parameters through the loading AXI-Stream interface. It sends all 12 weight values first, followed by the 3 bias values. The final bias word is marked with `load_axis_tlast`. After the layer asserts `load_done`, the testbench streams the four input vector values through the input AXI-Stream interface and marks the final input word using `s_axis_tlast`.

As shown in Figures 13 and 14, the fully connected layer correctly completes the parameter-loading phase, receives the streamed input vector, performs the matrix-vector multiply-accumulate operation, and produces the expected output values 10, 6, and 1. The testbench also verifies the AXI-Stream output handshake by monitoring `m_axis_tvalid`, `m_axis_tready`, and `m_axis_tlast`. The `tlast` signal is expected only on the final output neuron, which confirms that the layer outputs a correctly framed output vector.

Although the testbench uses a small 4-input, 3-output fully connected layer instead of the full MNIST dense layer, it verifies the core functionality of the module, including fixed-point weight loading, bias loading, dot-product accumulation, output ordering, and AXI-Stream control behavior. Since the fully connected module is parameterized by input size, output size, data width, accumulator width, fractional-bit position, and ReLU setting, the same verified logic can be scaled and reused for the full CNN pipeline.

3.2.4 Softmax layer testbench simulation

Referring to Figure 15 and 16, overall the softmax layer produces the expected outputs summing to an approximate of 1. However, we noticed that two adjacent input logits may produce identical softmax outputs because the proposed FPGA implementation approximates the exponential operation using a finite-resolution lookup table rather than evaluating the exponential function with full numerical precision. Although adjacent logits in the testbench are distinct in Q8.8 fixed-point representation, their difference may be smaller than the effective quantization interval used by the exponential lookup-table address generation. Consequently, two neighboring values can be mapped to the same LUT entry, yielding the same approximated exponential result. But in our later on-board tests, it is observed that such quantization-induced output collisions do not significantly affect the final classification results, as rarely in our testcases we create a tie between two different digit classes.

3.2.5 Overall wrapper layer simulation

Rather than testing the prediction accuracy for the overall CNN, we mainly tested the loading weights via BRAM-to-AXIS path and the inference flow through the runtime top. The testbench simulates a complete control write to select the Conv1 layer, streams in the corresponding weights and biases, and then feeds in an input image stream. The expected output is the final classification result after the input flows through all CNN stages.

As shown in figures 17 and 18, the MNIST layer correctly transitions between collection and send states, asserts valid only when the output data is ready, and produces the expected output values based on the input stream.

3.3 On-board testing

Additionally, we performed manual testing of 100 sample hand-written images to validate the system's performance under real-world conditions (Figure 19). Two of our teammates take turns writing digits via mouse on the simulated canvas, offering essentially two different writing styles for robustness. The combined results showcase an accuracy of 100%. As long as we write in close to normal fashion, the program can always recognize our intended digit. The on-board testing confirms that the accelerator can successfully process real input images and produce correct classifications, demonstrating the practical functionality of the design beyond simulation.

4 Discussion & Reflection

4.1 Technical Discussion

The implementation meets the central functional requirements by separating configuration, loading, and inference into protocol-visible stages. That separation is the main strength of the design: it makes the wrapper behavior understandable and keeps the layer code reusable. The main limitation is that the layers are intentionally sequential rather than deeply pipelined, so throughput is bounded by the collect-compute-stream cycle in each stage. For a class-scale FPGA project, that trade-off is reasonable because it reduced design risk and simplified integration.

4.2 Ethical Considerations

The direct ethical risk of this project is low because the system performs digit classification on a standard benchmark dataset. The broader consideration is responsible use of accelerator hardware: the design should be documented clearly enough that others can reproduce it and understand its behavior rather than treating the FPGA as a black box. The report and timing diagrams are part of that mitigation.

4.3 Project Reflection

The most important lesson from this project was that handshake discipline matters as much as arithmetic correctness. The accelerator only became dependable after the control wrapper, load stream, and layer state machines were aligned around the same readiness and completion rules. A future revision would likely introduce more pipelining and possibly a more compact parameter-loading protocol, but the current design provides a solid, understandable baseline.

A key lesson from this project is that reliable hardware acceleration depends not only on arithmetic correctness, but also on strict interface and handshake discipline. The modular structure of the system worked well because convolution, pooling, fully connected, softmax, and control logic could be tested separately before integration. At the same time, differences in coding style, FSM organization, reset behavior, and AXI-Stream assumptions between modules caused integration challenges, showing the importance of consistent coding conventions, clear interface definitions, and collaborative debugging using waveforms, console outputs, and Python reference results.

The current design provides a clear and functional baseline, but it still has limitations. The network is designed only for grayscale MNIST digit images and only classifies numerical digits from 0 to 9. It is also not robust for arbitrary digit images that do not follow the standard MNIST format, such as images with different sizes, colors, backgrounds, positions, or writing styles. In future work, we would improve input preprocessing, support more general image formats, and further optimize the convolution and fully connected layers through pipelining or parallelism.

5 Project Management

5.1 Deliverables

The completed deliverables are the Vivado FPGA project, the custom CNN wrapper and layer source code, the generated bitstream and hardware handoff files, and the Python-based control application used to load and run the accelerator on PYNQ-Z2. The project also includes layer-level and integration-level simulation artifacts for debugging and validation.

5.2 Timeline

The work progressed from model and layer validation into system integration, then into wrapper cleanup and final report preparation. The largest schedule risk was integration of the weight-loading path with the runtime top, since that required the control wrapper, the BRAM-to-AXIS path, and the layer-level load state machines to agree on the same start and completion behavior (Table 3).

Milestone	Planned Date	Actual Date	Status/Notes
M1	Early semester	Completed	Individual layers validated in simulation
M2	Mid semester	Completed	AXIS based image downscale on-board test and all individual layers passed simulation testbench
M3	Late semester	Completed	Final CNN system running on-board and final report prepared

Table 3: Planned vs. actual milestone timeline.

5.3 Team Contributions

Each team member contributed to a different part of the system: model and training work, HDL layer implementation and testing, and platform integration/reporting. The final design reflects that division by keeping the accelerator modular and separating the control wrapper from the per-layer logic (Table 4).

Team Member	Specific Contributions
Alexander Wang	Initial python development; conv2d, fully-connected layer development; overall system debugging and integration
Howard Hua	max-pooling, softmax layer development; AXI-DMA Stream debugging and implementation; overall system integration
Jade Gutierrez	GUI development; initial DMA streamline 2x+1 implementation

Table 4: Team member contributions.

6 Conclusion & Future Work

6.1 Conclusion

This project produced a working FPGA-oriented CNN architecture for MNIST digit recognition on PYNQ-Z2. The final system combines an AXI-Lite control interface, AXIS-based weight loading, and a streaming inference pipeline built from convolution, pooling, fully connected, and softmax stages, achieving close to 100% accuracy on both hand-drawn on-board tests and original MNIST test cases.

6.2 Future Work

Future improvements would focus on higher throughput and reduced latency. The most direct next step would be to pipeline the layer arithmetic more aggressively and reduce the number of cycles spent in collect-compute-stream transitions. Another extension would be to broaden the control interface so other classification tasks could be supported without modifying the hardware, such as by adding a more flexible parameter-loading protocol or supporting variable input sizes.

7 References

References

[1] Y. LeCun, C. Cortes, and C. J. C. Burges, "The MNIST Database of Handwritten Digits," 1998. [Online]. Available: http://yann.lecun.com/exdb/mnist/. Accessed: 2026.

[2] PYNQ Project, "PYNQ-Z2 Documentation and Python Overlay Workflow," accessed 2026.

8 Appendices

8.1 Additional Design Schematics

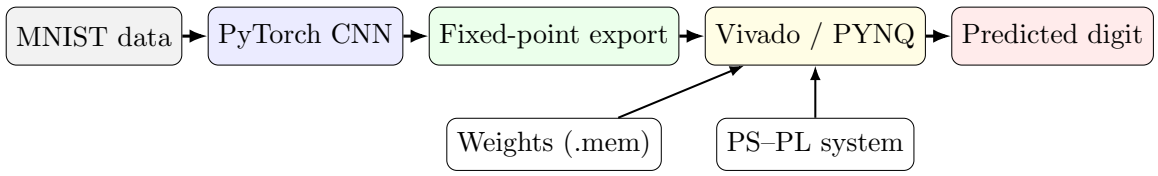

Figure 1: End-to-end workflow showing dataset, model training, fixed-point export, FPGA integration, and predicted output.

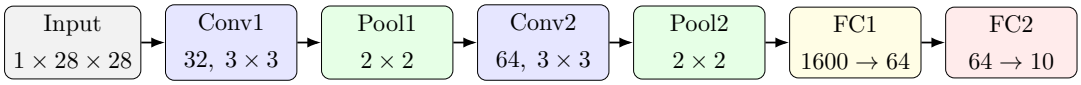

Figure 2: SimpleCNN architecture and major layer parameters used for MNIST classification.

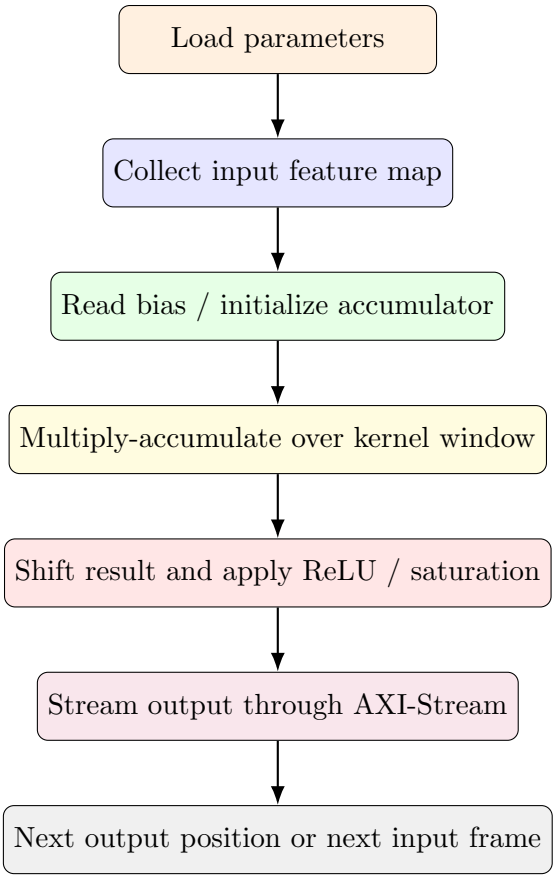

Figure 3: Simplified FSM flow of the convolution layer.

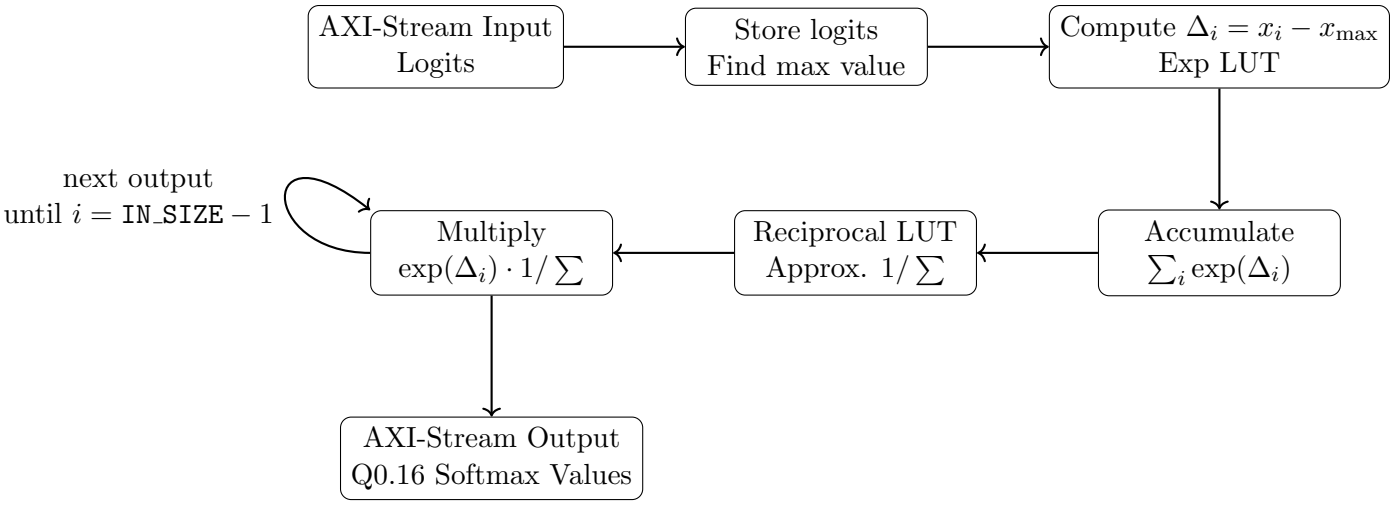

Figure 5: Main logic flow of the LUT-based fixed-point softmax layer.

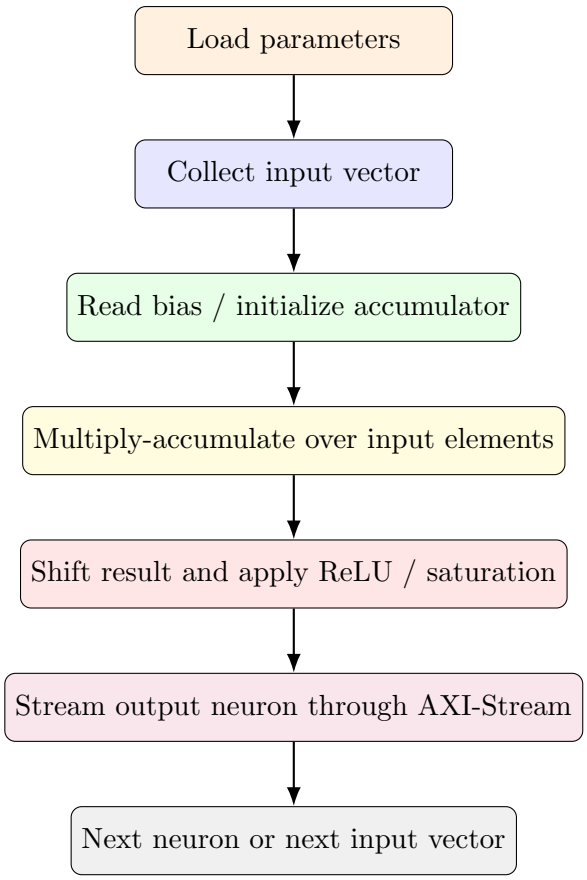

Figure 4: Simplified FSM flow of the fully connected layer.

Figure 6: BRAM-to-AXIS distributor inside the control wrapper: selects a target layer, reads BRAM, and packetizes AXI-Stream beats to the selected layer.

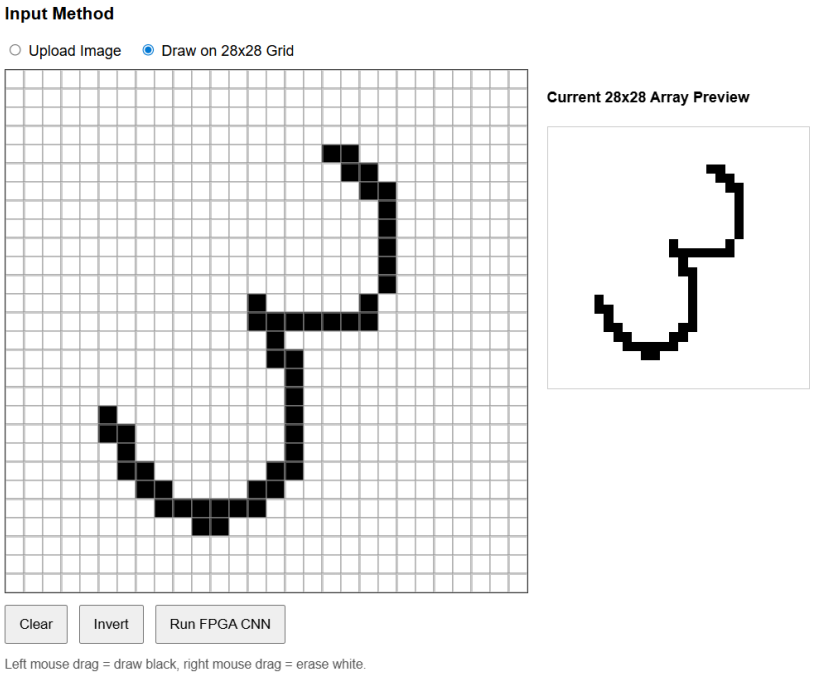

Figure 7: Interactive GUI for the CNN number recognition system.

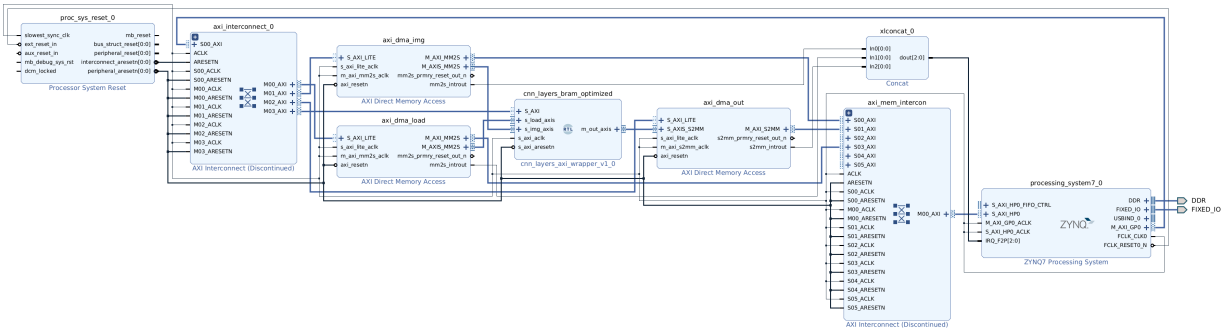

Figure 8: Vivado block design of the CNN accelerator system.

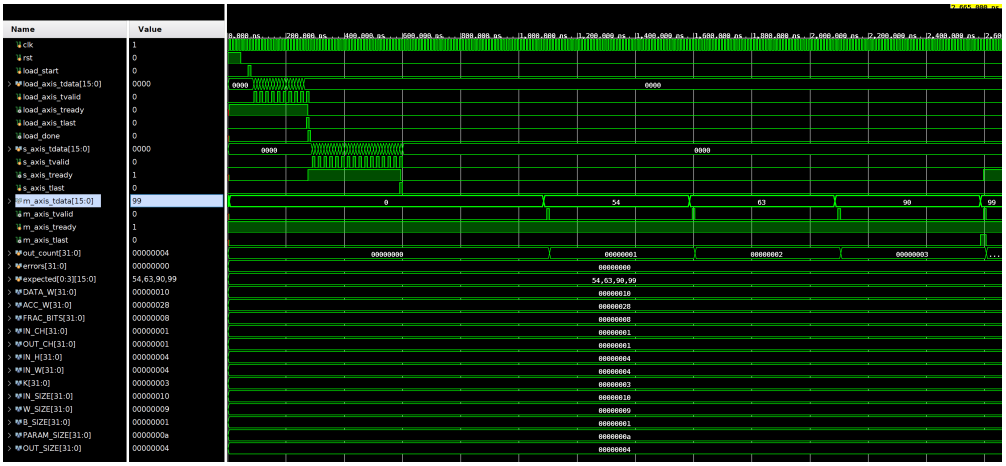

Figure 9: Simulation waveform of conv2d testbench.

```
CONV OUT[0] = 54, expected 54, tlast=0
CONV OUT[1] = 63, expected 63, tlast=0
CONV OUT[2] = 90, expected 90, tlast=0
CONV OUT[3] = 99, expected 99, tlast=1
PASS: tb_conv2d_layer_axis_bram
```

Figure 10: Tcl console output showing the expected conv2d output values.

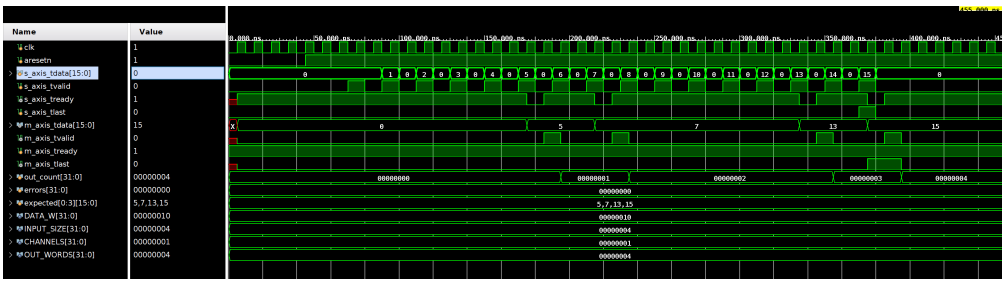

Figure 11: Simulation waveform of max pool testbench.

```
POOL OUT[0] = 5, expected 5, tlast=0
POOL OUT[1] = 7, expected 7, tlast=0
POOL OUT[2] = 13, expected 13, tlast=0
POOL OUT[3] = 15, expected 15, tlast=1
PASS: tb_maxpool_layer_axis_bram
```

Figure 12: Tcl console output showing the expected max-pooled output values.

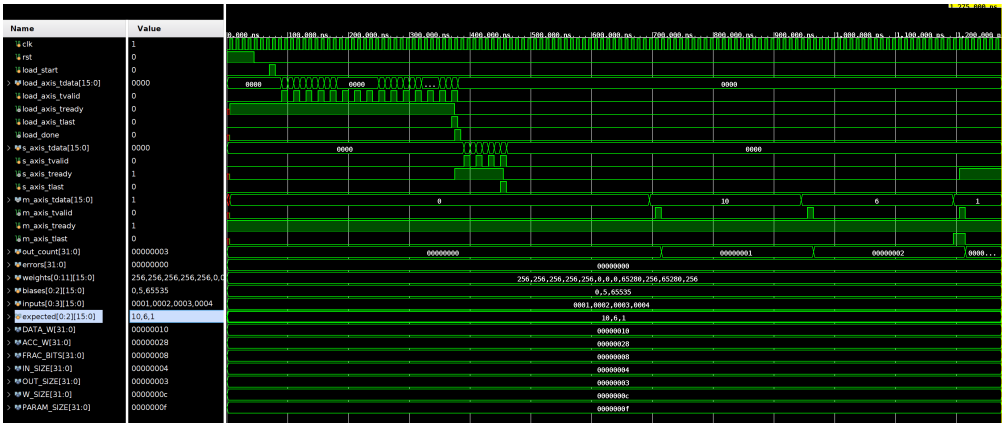

Figure 13: Simulation waveform of fully connected testbench.

```
run all
FC OUT[2] = 1, expected 1, tlast=1
PASS: tb_fully_connected_layer_axis_bram
```

Figure 14: Tcl console output showing the expected fully connected output values.

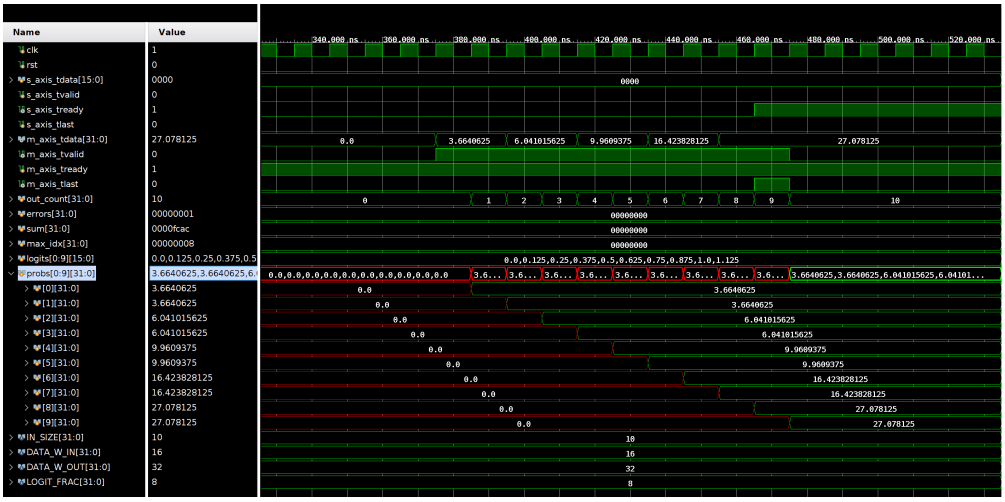

Figure 15: Simulation waveform of softmax testbench.

```
SOFTMAX OUT[0] = 1876 / 65536 = 0.028625, tlast=0
SOFTMAX OUT[1] = 1876 / 65536 = 0.028625, tlast=0
SOFTMAX OUT[2] = 3093 / 65536 = 0.047195, tlast=0
SOFTMAX OUT[3] = 3093 / 65536 = 0.047195, tlast=0
SOFTMAX OUT[4] = 5100 / 65536 = 0.077820, tlast=0
SOFTMAX OUT[5] = 5100 / 65536 = 0.077820, tlast=0
SOFTMAX OUT[6] = 8409 / 65536 = 0.128311, tlast=0
SOFTMAX OUT[7] = 8409 / 65536 = 0.128311, tlast=0
SOFTMAX OUT[8] = 13864 / 65536 = 0.211548, tlast=0
SOFTMAX OUT[9] = 13864 / 65536 = 0.211548, tlast=1
SOFTMAX SUM = 64684 / 65536 = 0.987000, max_idx=8
Error: softmax max probability is not on largest input logit
```

Figure 16: Tcl console output showing the expected softmax output values.

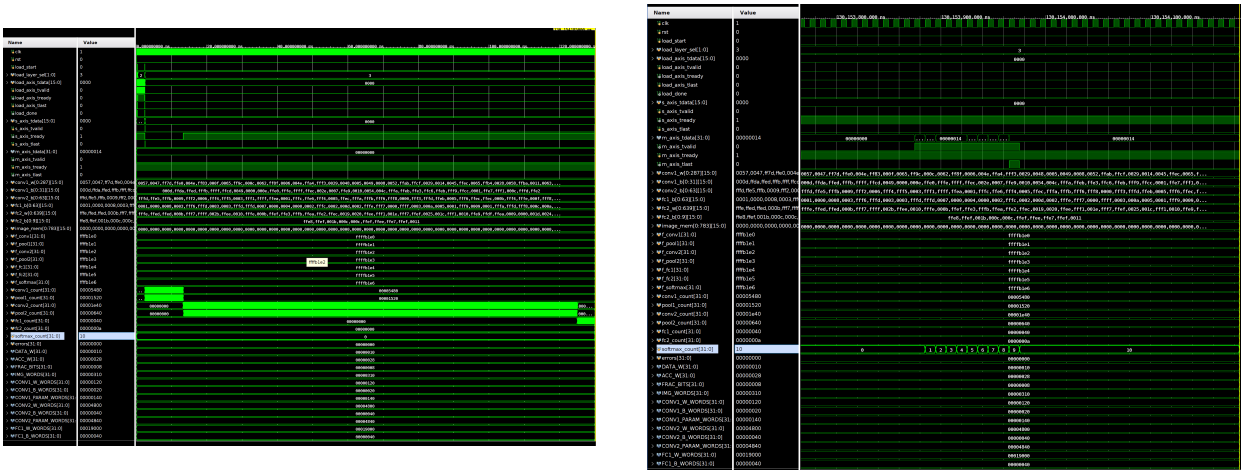

Figure 17: Overall simulation waveforms from the MNIST testbench (left) and zoomed final stages (right).

```
run all
Loaded CONV1 at t=6560000
Start loading CONV2: 18496 words at t=6585000
Loaded CONV2 at t=376520000
Start loading FC1: 102464 words at t=376545000
Loaded FC1 at t=2425840000
Start loading FC2: 650 words at t=2425865000
Loaded FC2 at t=2438880000
Start image stream: 784 words at t=2438905000
End image stream at t=2454580000
SOFTMAX[0] = 20 last=0 t=130153885000
SOFTMAX[1] = 1839 last=0 t=130153895000
SOFTMAX[2] = 20 last=0 t=130153905000
SOFTMAX[3] = 20 last=0 t=130153915000
SOFTMAX[4] = 20 last=0 t=130153925000
SOFTMAX[5] = 13594 last=0 t=130153935000
SOFTMAX[6] = 20 last=0 t=130153945000
SOFTMAX[7] = 36954 last=0 t=130153955000
SOFTMAX[8] = 13594 last=0 t=130153965000
SOFTMAX[9] = 20 last=1 t=130153975000
Finished full CNN at t=130153985000
CONV1 count OK: 21632
POOL1 count OK: 5408
CONV2 count OK: 7744
POOL2 count OK: 1600
FC1 count OK: 64
FC2 count OK: 10
SOFTMAX count OK: 10
PASS: tb_mnist_simplecnn_axis_bram_wrapper
$finish called at time : 130154185 ns : File "/home/
```

Figure 18: Tcl console output showing the expected MNIST output values.

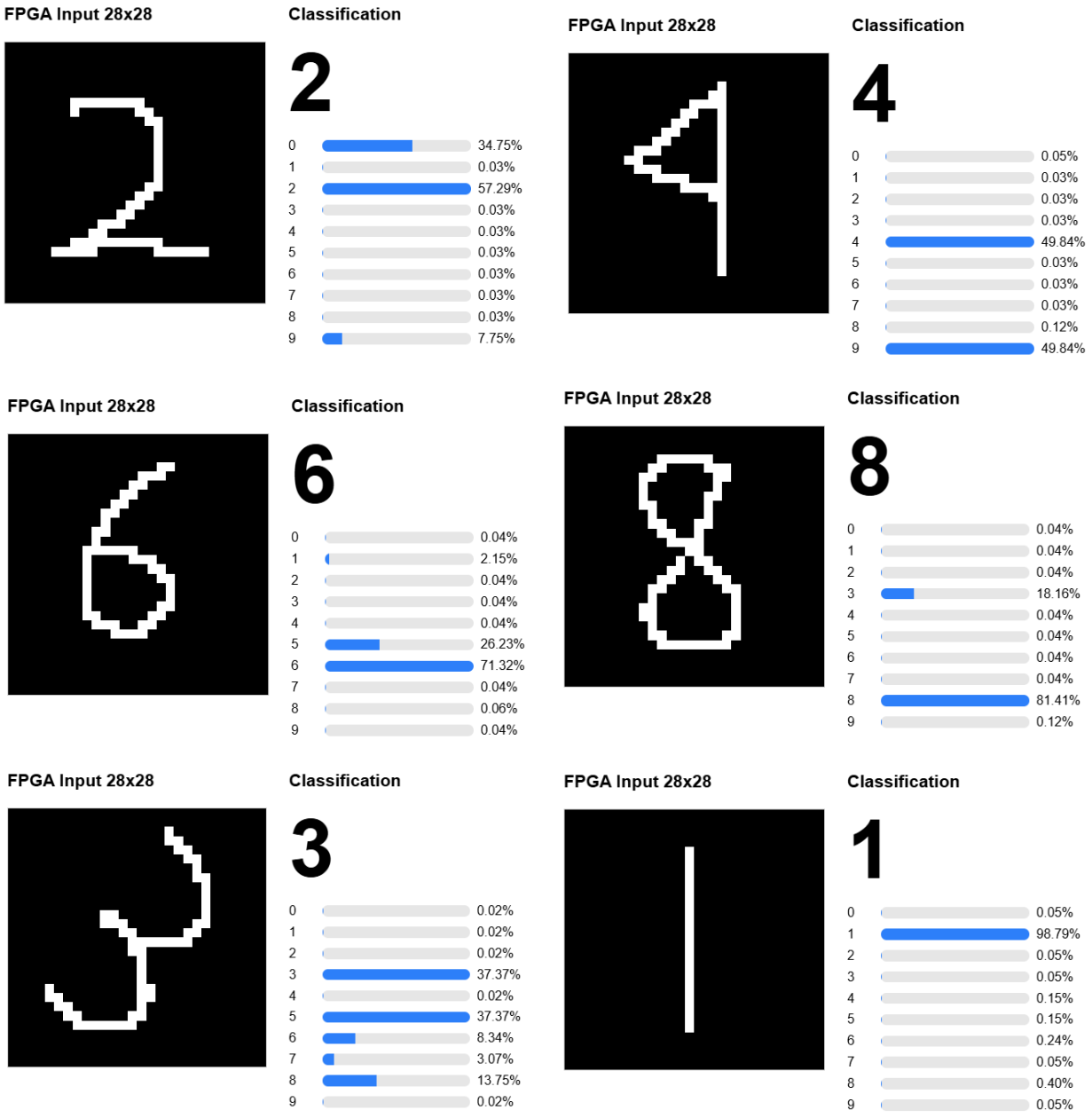

Figure 19: Selected on-board test images captured from the running PYNQ-Z2 system.

8.2 Source Code / Repository Link

The complete source code and project files are available at the following GitHub repository: https://github.com/WashU-CSE4602-SP26/cse4602-team-alex-howard-jade. Currently the repository is private, but it can be made public upon request for review.