Implementing a fully connected neural network on an FPGA
Advanced Logic Design, 2024-2025

Nicolò Cecchin
nicolo.cecchin@studenti.unitn.it

Contents

1 Introduction

This project aims to explore the possibility of implementing a fully connected neural network in a hardware context, using an FPGA as the execution platform. The network was described in VHDL and developed using the Vivado environment.
The network was trained on the MNIST dataset using TensorFlow. The weights obtained after training were exported and then imported into the VHDL project, allowing inference to be performed directly on the FPGA board.

2 Problem description

The goal of this project is to implement a fully connected neural network on an FPGA device. This type of network consists of one or more layers in which each neuron is connected to all neurons in the next layer. Inference involves computing a weighted sum of the inputs for each neuron, followed by an activation function.
The network was trained in software using TensorFlow with the MNIST dataset for handwritten digit recognition. After training, the weights were saved and imported into the hardware project. The objective is to replicate the same inference process directly on the FPGA board, without relying on an external CPU.
To simplify the implementation of the required mathematical operations, and to maintain the same kind of operations done i sofware, floating-point representation was used. Specifically, Vivado-provided IP cores were employed for addition and multiplication. This choice reduced development complexity compared to a custom fixed-point implementation, though at the cost of higher resource usage.
To feed input data from a computer to the FPGA, a custom UART module was developed. It receives bytes via a serial connection and converts them into signals that can be processed by the network. This made it possible to test the design with new inputs in real time, facilitating the verification phase.
The main challenge of the project lies in properly coordinating the various hardware modules, managing data flow, timing, and control sequences through finite state machines.

3 General architecture

The neural network implemented in this project is of the *fully connected* (or dense) type, following a feed-forward structure. In this kind of architecture, each neuron in a given layer is connected to every neuron in the next layer, with no recurrent or convolutional connections.
The architecture chosen for the hardware implementation includes:

- **An input layer** consisting of 784 neurons (corresponding to the 28×28 pixels of the MNIST images);
- **One hidden layer** with 128 neurons;
- **An output layer** with 10 neurons, representing digits from 0 to 9.

Each neuron performs a weighted sum of its inputs, followed by an activation function. During training, the *ReLU* activation function was used for the hidden layers and *softmax* for the output layer. However, in the hardware implementation, the activation functions were simplified to reduce computational complexity. In particular, softmax is not used, a simple comparison to find the highest value is enough.
The network was trained using TensorFlow in a Python environment with the MNIST dataset. After training, the weights of each neural connection were exported in a compatible format and then loaded into memory on the FPGA platform, enabling real-time inference.
From a functional perspective, the hardware implementation can be divided into three main components:

- **Input module**: receives normalized pixel data and distributes it to the first-layer neurons;
- **Computation modules**: each neuron is implemented as a dedicated unit that computes the dot product between inputs and weights, followed by an activation function;
- **Output module**: gathers the results from the final layer and determines the predicted class.

The next chapter details the block scheme of these components, with a particular focus on modular organization, finite state machine used, and the data flow within the network.

4 Main blocks implementation

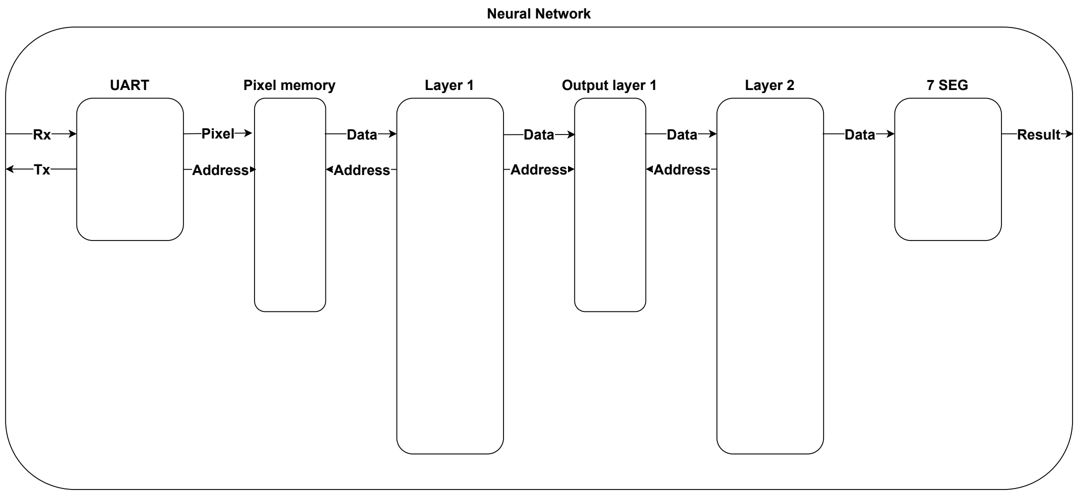
4.1 Neural network

4.1.1 Block description

This is the top-level block of the design, which contains the entire architecture of the neural network. It is composed of the following submodules:

- **UART Block**: receives the `Rx` signal and reconstructs a 32-bit value representing a single pixel of the input image.
- **Pixel Memory**: a dual-port RAM used to store the input image. It is controlled by two independent address signals: one for write operations and one for read operations.
- **Layer 1**: the hidden layer of the neural network. It generates the read address for the pixel memory and performs the first stage of inference.
- **Output Layer 1**: a dual-port RAM that stores the outputs of the hidden layer. Similar to the pixel memory, it uses separate address lines for read and write operations.
- **Layer 2**: the output layer of the network. It reads from the Output Layer 1 memory and computes the final classification outputs. It also generates the address signals for accessing Output Layer 1.
- **Comparator / 7-Segment Display Controller**: compares the outputs of the second layer to determine the most probable class. The result is used to drive the seven-segment display array.

4.1.2 Block diagram



4.1.3 Finite state machine

The control logic of the neural network is implemented using a Finite State Machine (FSM), which orchestrates the execution of the main blocks involved in the inference process. The FSM operates on five main states:

WAIT_IMAGE This is the idle state of the system and the initial state after reset. Upon entering this state, the system writes the IEEE 754 representation of the value `1.0` into the first position of the pixel memory. This value serves as a constant input used to multiply with the bias term of each neuron in the first layer.
Once the system begins receiving a new image over UART, the UART block starts assembling the 32-bit pixel values from the incoming `Rx` serial data. Each completed pixel is written to the

pixel memory, and the write address is incremented accordingly. When the transmission of the full image is completed (detected by monitoring the memory address), the FSM transitions to the `FIRST_LAYER` state.

FIRST_LAYER In this state, the first layer of the network performs its computations. At each clock cycle, the block reads a pixel value from the pixel memory by incrementing the read address. Each value is multiplied by its corresponding weight, and partial sums are accumulated. When all input values have been processed for all neurons, the state transitions to `SAVE_RESULTS`.

SAVE_RESULTS This state stores the outputs of the first layer into the dedicated dual-port memory. Before storing any result, the system writes the constant value `1.0` (in IEEE 754 format) at the first memory location, again to be used for the bias term of the second layer. The remaining outputs are processed one at a time: each is passed through a ReLU activation function, implemented simply by checking whether the most significant bit (the sign bit) is zero. If so, the value is stored; otherwise, zero is written. After all values are stored, the FSM advances to the `SECOND_LAYER` state.

SECOND_LAYER This state behaves similarly to `FIRST_LAYER`, but it operates on the memory containing the first layer's outputs. The second layer reads the data, performs its computations, and accumulates results for each output neuron. Once all calculations are complete, the FSM proceeds to the final state, `SHOW`.

SHOW In this state, the outputs of the second layer are compared using the dedicated comparator module, which identifies the index of the maximum value corresponding to the predicted digit. This index is then encoded and sent to the seven-segment display controller. Once the display is updated, the FSM returns to the `WAIT_IMAGE` state, ready to process a new image.

4.1.4 Main signals

Block	Signal	Description
External Interface	`clk100MHz`	100 MHz system clock
	`RsTx`	UART input (transmission)
	`RsRx`	UART output (reception)
	`seven_segments`	7-segment display driver (inout)
	`an`	Display digit enable
Layer 1 Memory	`address_in_1`	Write address for pixel memory
	`mem_in_1`	Input pixel data
	`write_en_mem_1`	Write enable (4-bit)
	`pixel`	Pixel received via UART
	`pixel_valid`	Pixel validity flag
	`i_1`	Write counter (784 + bias)
Layer 1	`neuron_input_layer_1`	Input to first layer
	`neuron_output_layer_1`	Outputs from 128 neurons
	`layer1_en`	Enable signal for layer 1
	`result_ready_layer1`	Completion signal from layer 1
	`results_layer_1`	Stored outputs of layer 1
Layer 2 Memory	`mem_in`	Input to memory between layers
	`write_en_mem_2`	Write enable for layer 2 memory
	`address_in_2`	Write address for layer 2 memory
	`i_2`	Counter for writing layer 2 input
Layer 2	`neuron_input_layer_2`	Input to layer 2
	`neuron_output_layer_2`	Outputs from 10 neurons
	`layer2_en`	Enable signal for layer 2
	`result_ready_layer2`	Completion signal from layer 2
	`results_layer_2`	Stored outputs of layer 2
Compare & Display	`compare_en`	Comparison start signal
	`compare_complete`	Comparison done signal
	`seg`	Output to 7-segment display
FSM	`present_state`	Current FSM state
	`next_state`	Next FSM state

Table 1: Main signals of the top-level neural network block

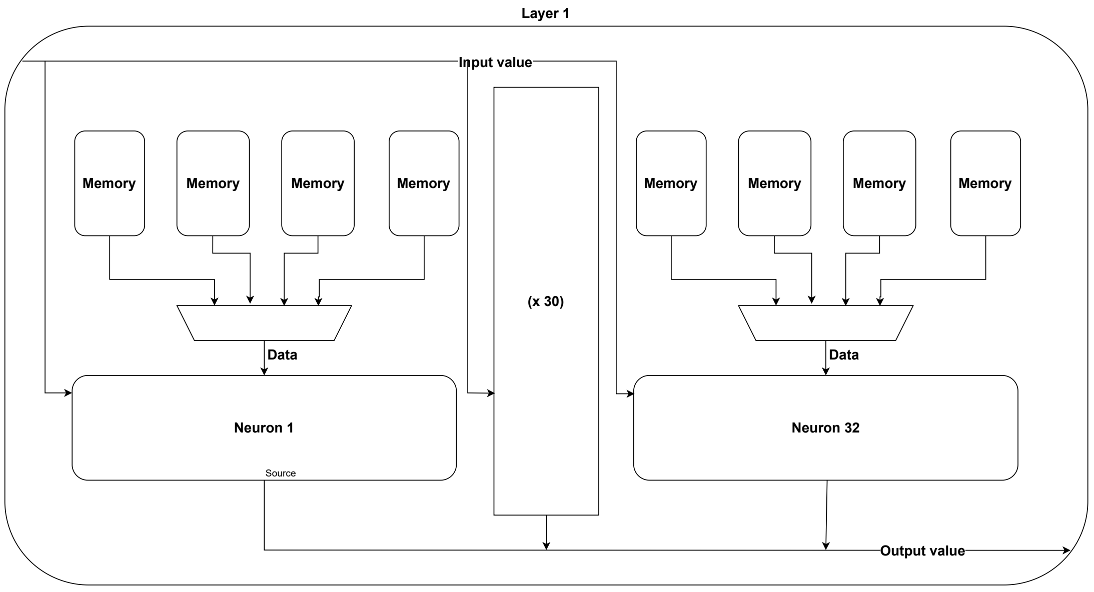
4.2 Layer 1

4.2.1 Block description

The Layer 1 module is composed of two main components: neurons and memories.

- **Neurons**: There are 32 physical neurons in the layer. Each neuron performs a multiply-and-accumulate (MAC) operation between the input pixel and the associated weight and bias.
- **Memories**: Each neuron is connected to 4 different weight memories (BRAM set ar single port ROM), for a total of 128 weight memories in the block. These memories are preloaded with values from `.coe` files, where the first value represents the bias and the remaining values are the weights for the corresponding neuron. There are 785 values in the memory.

4.2.2 Block diagram



4.2.3 Finite state machine

The behavior of Layer 1 is governed by a finite state machine compos med of five states:

IDLE This is the default state where the layer remains inactive. The FSM transitions out of this state only when the `layer1_en` signal is asserted.

RESET_ADDRESS In this state, the memory address counter is reset to zero. This is necessary because the same input must be processed four times, once for each memory block set. The transition to the next state occurs when the partial result ready signal is low, ensuring that it does not incorrectly trigger computation due to being held high from a previous operation.

START This state lasts for one clock cycle. It serves to initialize the neurons and allows internal signals to stabilize. This is needed to correctly transition the neurons into their compute state, ensuring proper timing alignment.

COMPUTE During this state, the memory address counter increments on each clock cycle, feeding both the input pixel and corresponding weight to the neurons. Once all addresses have been processed, the FSM waits for the neurons to complete their computation. When done, the partial result is stored. If fewer than four passes have been completed, the FSM returns to `RESET_ADDRESS`. Otherwise, it transitions to the final state.

FINISH : In this state, the final result of the layer is output to the next stage. After this, the FSM returns to the `IDLE` state, ready for a new input.

4.2.4 Main signals

Category	Signal	Description
External Interface	`clk`	System clock signal
	`layer_en`	Enables the layer's operation; triggers the FSM
	`neuron_input`	32-bit input vector broadcast to all neurons
	`address`	Shared address bus to access weights and input memory
	`result_ready`	Asserted when all 128 outputs are ready
	`neuron_output_array`	Final 128-element array with all neuron outputs
FSM Control	`present_state`	Current state of the FSM
	`i`	Index to select one of four 32-neuron groups
	`first`	Flag to delay memory access on first compute cycle
Neuron Control	`neuron_en`	Enables all 32 neurons for processing
	`neuron_rst`	Neuron reset signal (currently unused)
	`operation_en`	Controls neuron operation during computation
Memory Interface	`memory_out_array`	Full 128-element memory output array (weights/biases)
	`memory_out_small`	32-element subset used in the current cycle
Neuron Output	`neuron_output_small`	Outputs of the currently active 32 neurons
	`result_ready_small`	Readiness flags from the 32 neurons

Table 2: Main signals of the `layer1` module

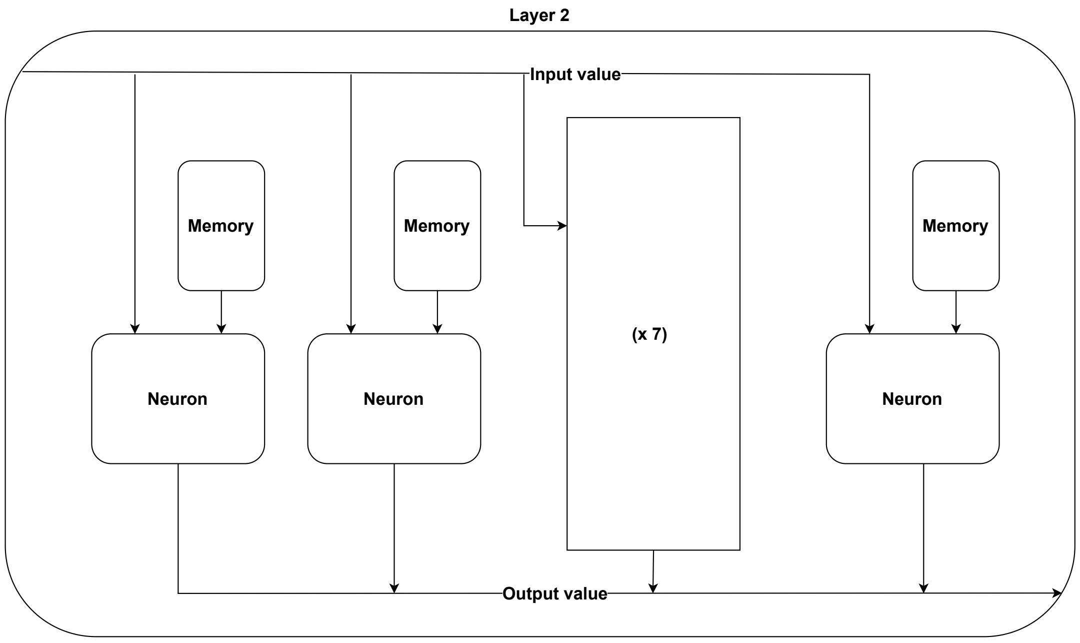
4.3 Layer 2

4.3.1 Block description

The `layer2` block implements the second layer of the neural network and consists of:

- **Neurons**: There are 10 physical neurons, each responsible for producing one of the final classification outputs.
- **Memories**: Each neuron is connected to a single memory (BRAM set ar single port ROM), pre-loaded with a `.coe` file containing the weights and the bias (stored as the first value). In this case there are 129 values in the memory.

4.3.2 Block diagram



4.3.3 Finite state machine

The finite state machine (FSM) that controls the `layer2` block is composed of four states:

IDLE This is the idle state, where the block remains inactive. All control signals are deactivated, and the address is reset to zero. When the `layer_en` signal is asserted, the FSM transitions to the `START` state to begin computation.

START This state is active for a single clock cycle. It enables the `neuron_en` signal, activating all neurons. The purpose of this state is to initialize the neurons and give them time to activate their internal functions. After one cycle, the FSM moves to the `COMPUTE` state.

COMPUTE This is the main computation state. The address is incremented at every clock cycle until the value reaches the predefined `max_address`. Each cycle provides all neurons with a new value from their respective memories. The 10 neurons operate in parallel and fetch their weights from 10 separate memories. When all neurons complete their computations (i.e., when `result_ready_array` equals `"1111111111"`), the FSM transitions to the `FINISH` state.

FINISH : In this state, the `result_ready` signal is asserted to indicate that the final output values are ready. The FSM remains in this state until the `layer_en` signal is deasserted, at which point it returns to the `IDLE` state.

4.3.4 Main signals

Category	Signal	Description
Inputs	`clk`	System clock signal used to synchronize FSM and neuron activity
	`layer_en`	Enables the FSM to start processing; transitions the FSM from `IDLE`
	`neuron_input`	Input value for all neurons, coming from the output of Layer 1 (post-ReLU)
Inout	`address`	Shared address used to sequentially read weights and biases from memory
Outputs	`result_ready`	Signals when the layer has finished processing and all outputs are ready
	`neuron_output_array`	Vector containing the outputs of the 10 neurons in the layer
Neuron Control	`neuron_en`	Enables all neurons during the compute phase
	`neuron_rst`	Optional reset line for neurons (currently unused)
	`operation_en`	Activates memory read and neuron computation during the `COMPUTE` state
FSM / Internal	`present_state`	Tracks the current FSM state (`IDLE`, `START`, `COMPUTE`, `FINISH`)
	`result_ready_array`	Per-neuron flag array; each bit is set when its corresponding neuron is done
	`memory_out_array`	Output from each neuron's memory block containing weights and bias
	`first`	Delays the first address increment by one cycle to align with MAC pipeline timing

Table 3: Main signals of the `layer2` module

4.4 Neuron

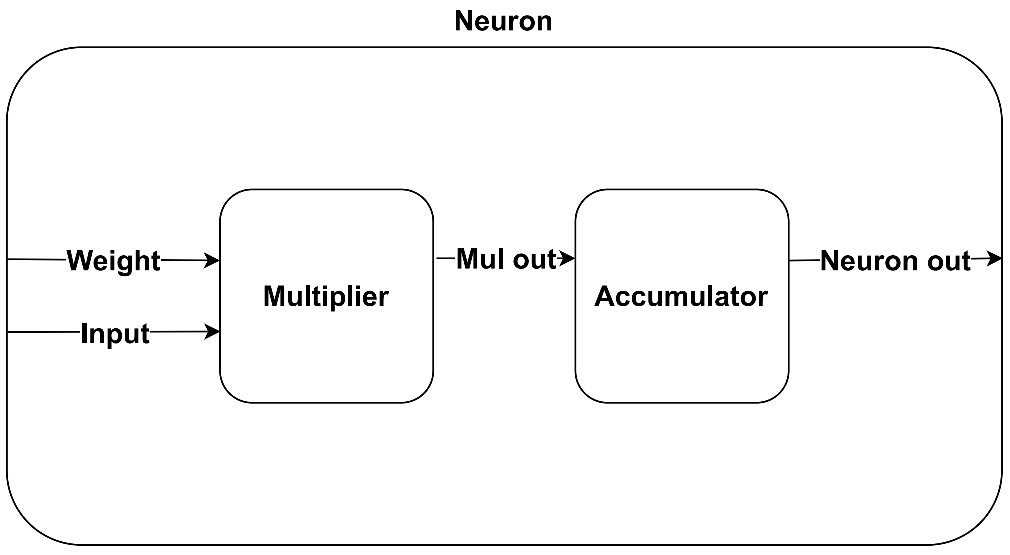
4.4.1 Block description

The neuron is the core computational unit of the layer. Each neuron receives one input value and its corresponding weight (read from memory, the first operation is always the bias), multiplies them together, and accumulates the result to compute the final output. The block is composed of the following submodules:

- **Multiplier**: The multiplier performs a floating-point multiplication between the input value and the corresponding weight. This module is implemented using a Vivado IP core with pipelined configuration. It has a fixed latency of 9 clock cycles to produce the result.
- **Accumulator**: The accumulator adds each partial product generated by the multiplier to a running sum. It continues accumulating until the last value is signaled. The accumulator is also pipelined and has a latency of 23 clock cycles per operation.

At the start of computation, the neuron receives the bias as the first value from memory and initializes the accumulator with it. Then, it sequentially processes each weight–input pair. Once all values have been processed, the final accumulated result is provided as the neuron's output.

4.4.2 Block diagram



4.4.3 Finite state machine

The FSM of the neuron consists of three states:

IDLE This is the initial state of the neuron. While in this state, the neuron remains inactive and resets internal values. It transitions to the **COMPUTE** state when the input signal `neuron_en` is asserted.

COMPUTE In this state, the neuron performs the Multiply-and-Accumulate (MAC) operations. The multiplier is enabled when `operation_en` is high. The result of the multiplication is passed to the accumulator.
The FSM monitors when the last multiplication result is expected. This condition is detected when `operation_en` is deasserted and no further valid data is being output from the multiplier (`mul_valid` = '0'). In that case, a signal `mul_last` is asserted and sent to the accumulator to indicate the end of the sequence.
The FSM transitions to the **FINISH** state once the accumulator signals that it has completed the operation by asserting `acc_last`.

FINISH This state indicates that the neuron has completed its computation. The output value from the accumulator is written to the output port, and the signal `result_ready` is asserted to notify external logic. The FSM returns to the **IDLE** state when `neuron_en` is deasserted.

4.4.4 Main signals

Category	Signal	Description
I/O Ports	`clk`	System clock signal driving the FSM and datapath
	`neuron_en`	Global enable signal for neuron activation
	`neuron_rst`	Reset signal (unused but declared for compatibility)
	`operation_en`	Triggers memory access and activates the multiplier
	`neuron_input`	Input value (from input layer or previous layer)
	`memory_out`	Weight or bias value read from memory
	`result_ready`	Asserted when the neuron finishes its computation
	`neuron_output`	Final accumulated result output (32-bit)
Data Signals	`mul_data`	Output of the multiplier, input to the accumulator
	`acc_data`	Final result from the accumulator
	`mul_out`	Raw result from multiplier (before being latched)
	`acc_out`	Raw result from accumulator (before being latched)
Control Signals	`mul_en`, `acc_en`	Enable multiplier / accumulator input respectively
	`a_ready`, `b_ready`	Multiplier handshake: ready to accept input A / B
	`c_ready`	Accumulator handshake: ready to accept input
	`mul_valid`, `acc_valid`	Signal valid output from multiplier / accumulator
	`mul_last`	Informs accumulator that current input is the last one
	`acc_last`	Indicates accumulator has finished processing
FSM / Internal	`present_state`	Current FSM state: `IDLE`, `COMPUTE`, or `FINISH`
	`last`	Flag used internally to detect when to assert `mul_last`

Table 4: Main signals of the `neuron_layer_1` module

4.5 Uart driver

4.5.1 Block description

The `uart_image_receiver` module handles the reception of a 32-bit pixel over a UART serial interface. Although it is not divided into instantiated hardware submodules, its operation can be logically separated into the following functional sections:

- **UART Reception FSM**: Detects the start of transmission, samples each bit of the incoming byte, and ensures proper timing using a configurable UART clock divider.
- **Byte-to-Pixel Assembler**: Stores each of the four incoming bytes in the correct section of a 32-bit register, reassembling the full pixel value.
- **Output Interface**: Signals the availability of a complete pixel through the `pixel_valid` signal and outputs the result on the `pixel` port.

The UART protocol implemented in this receiver uses the following configuration:

- **Baud rate:** 115200
- **Data bits:** 8
- **Start bits:** 1
- **Stop bits:** 1
- **Parity:** None

4.5.2 Finite state machine

The FSM controlling the module transitions through six states, each corresponding to a specific phase of UART communication:

IDLE Waits for the falling edge of the start bit (i.e., `rx = '0'`).

START Waits for half the bit duration to verify the start bit remains low (start-bit validation).

DATA Receives 8 bits of the UART frame one at a time, storing them into a temporary 8-bit register (`data_byte`).

STOP Waits for the stop bit duration after all data bits have been received.

ADD Transfers the received byte into the appropriate section of the 32-bit pixel register (`pixel_reg`) based on the current byte count. Once all four bytes are received, transitions to `SEND`.

SEND Signals that a valid pixel has been received by asserting `pixel_valid` and outputting `pixel`.

4.5.3 Main signals

Category	**Signal**	**Description**
Communication Interface	`clk`	System clock that drives the FSM and internal counters
	`rx`	UART receive line for serial input from the host
	`tx`	UART transmit line (currently unused)
	`pixel`	32-bit pixel assembled from four UART bytes
	`pixel_valid`	High for one clock when `pixel` output is ready
UART Timing Control	`clk_counter`	Counts clock cycles to sample UART bits properly
	`CLK_PER_BIT`	Constant that determines bit duration (based on baud rate)
Data Assembly	`data_byte`	Holds a full 8-bit UART byte once received
	`pixel_reg`	Accumulates 4 bytes into one 32-bit pixel
	`current_bit`	Tracks the current bit being received (0–7)
	`current_byte`	Tracks the current byte index (0–3) within the pixel

Table 5: Main signals of the `uart_image_receiver` module

4.6 Comparator

4.6.1 Block description

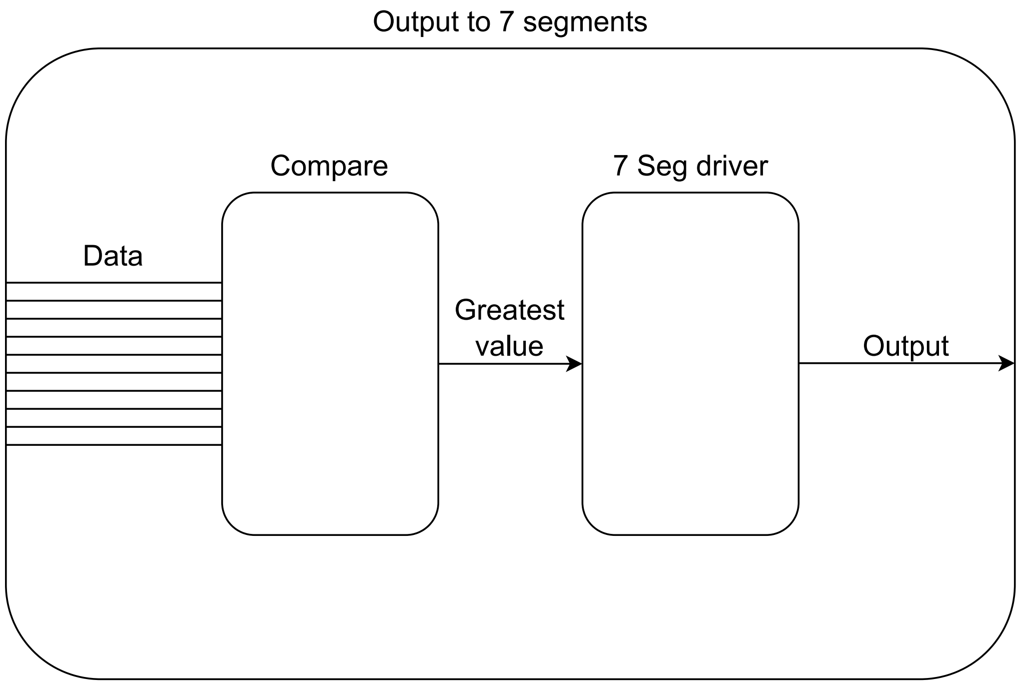
This module is responsible for comparing the outputs of the neural network and displaying the index of the maximum value on a 7-segment display (the output of the neural network). It consists of two main components:

- **Floating Point Comparator:** The comparison operation is performed using a Vivado IP core configured in floating-point mode with comparison functionality. The module iteratively compares the network outputs two at a time, updating an internal index whenever a larger value is found. This process continues until all 10 values have been checked, identifying the digit with the highest output. In this case the floating-point IP has a latency of 3 clock cycles, however it has not been used in pipeline mode because to perform the next operation the result of the previous is required.

- **7-Segment Display Driver:** Once the maximum value's index is determined, this subcomponent maps the corresponding digit (0–9) to its encoded 7-segment display representation. The final output is a static 8-bit vector driving a common anode 7-segment display, where each bit controls a specific segment.

The overall module is triggered by a `compare_en` signal and produces two outputs: a signal (`compare_complete`) indicating the end of the operation, and an 8-bit output (`seven_segments`) encoding the digit corresponding to the network's most confident prediction.

4.6.2 Block diagram



4.6.3 Finite state machine

The FSM controlling the `compare_digit` module transitions through five states, each corresponding to a specific phase of the comparison and display process:

IDLE Initial state where internal counters are reset, the 7-segment output is turned off (`"11111111"`), and the module waits for `compare_en = '1'` to start the comparison process.

VALUES Selects two adjacent input values from the `digits` array and loads them into the inputs of the comparison IP core. If the current value is greater than the previously stored maximum, updates the index of the maximum.

START Enables the comparison IP by asserting `op_en` and increments the comparison index `i`. Then transitions to the `COMPARE` state.

COMPARE Waits for the IP core to signal a valid result by asserting `res_tvalid`. Once the result is ready, stores it in the internal signal `result` and returns to `VALUES` to continue comparing the next pair of values.

FINISH After all comparisons are complete, sets the 7-segment output to the digit corresponding to the index of the maximum value. Also asserts `compare_complete` to indicate that the process is finished, before transitioning back to `IDLE`.

4.6.4 Main signals

Category	Signal	Description
Interface Signals	`clk`	System clock for FSM and synchronous logic
	`digits`	10-element array of 32-bit values to compare
	`compare_en`	Enables the comparison process
	`seven_segments`	Output to the 7-segment display showing the maximum index
	`compare_complete`	Signals the end of the comparison process
FP Comparator Interface	`a_data`, `b_data`	Operands fed into the floating-point comparator
	`a_tready`, `b_tready`	IP handshake signals: ready for input A and B
	`op_en`	Enables the comparison operation
	`res_tvalid`	Indicates that comparison output is valid
	`result_data`	8-bit result (LSB indicates outcome of comparison)
FSM and Control Signals	`present_state`, `next_state`	FSM state tracking
	`i`	Iteration index over the `digits` array
	`index`	Stores current index of the maximum value found
	`compare_finished`	Internal signal signaling comparison end
	`result`	Holds last comparison outcome (1 if `a > b`)

Table 6: Main signals of the `compare_digit` module

5 Accuracy Evaluation

To assess the effectiveness of the implemented neural network, we evaluate its classification accuracy both in software and in hardware. Additionally, we measure the level of agreement between the two, using the same trained weights in both environments.

5.1 Software Accuracy

The network was first tested in a pure software environment, using floating-point arithmetic and the original training parameters. The classification accuracy over the test dataset was:

Software Accuracy: 54.00%

The confusion matrix in Figure 1 highlights the distribution of predictions across classes.

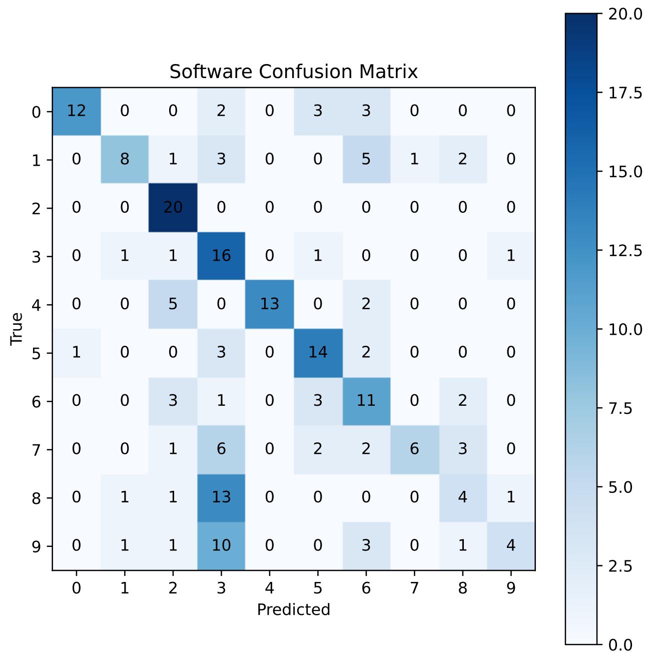


Figure 1: Confusion matrix - Software implementation

5.2 Hardware Accuracy

Using the same trained weights, the network was deployed in hardware and tested with input data streamed via UART. The classification accuracy observed in hardware was:

Hardware Accuracy: 53.50%

This slight drop in accuracy can be attributed to quantization errors, limited precision in internal signals, and possible timing issues in real-time operation. The confusion matrix is reported in Figure 2.

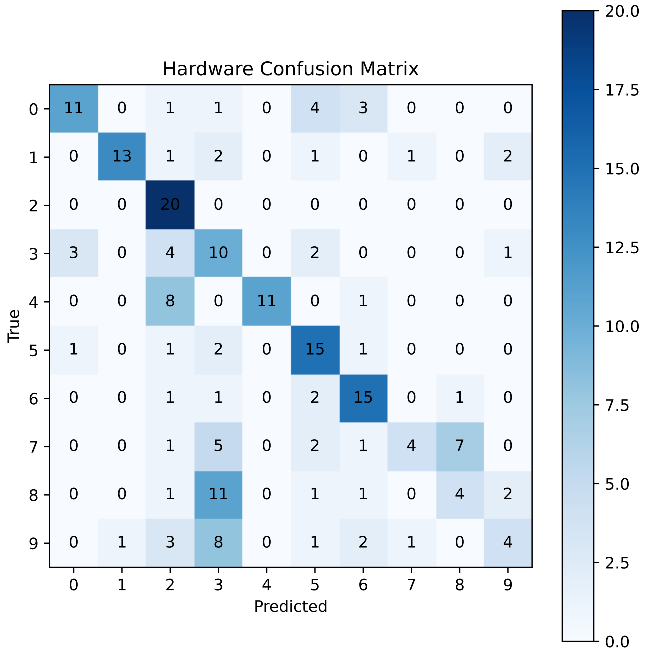


Figure 2: Confusion matrix - Hardware implementation

5.3 Software vs Hardware Agreement

To quantify the consistency between the software model and the hardware inference engine, we computed the agreement rate across predictions:

SW vs HW Agreement Rate: 72.50%

This metric measures the percentage of test samples for which software and hardware produced the same classification result. Figure 3 shows the confusion matrix comparing software and hardware outputs directly (i.e., ground truth is ignored).

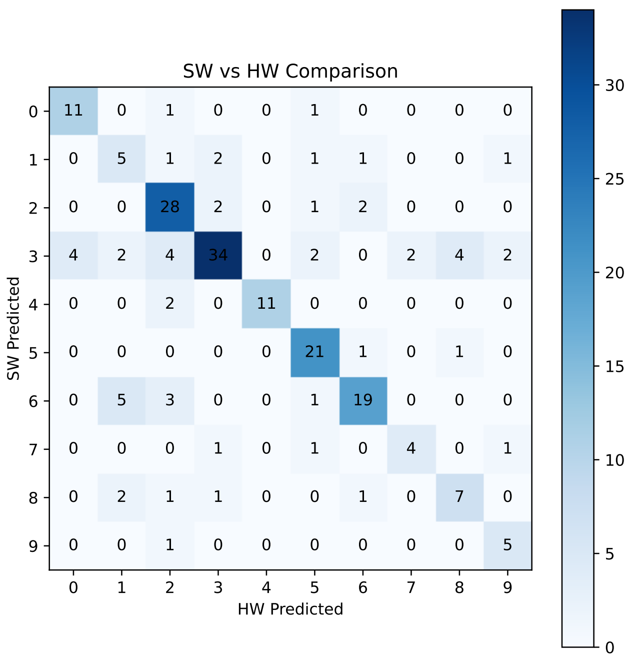


Figure 3: SW vs HW prediction agreement

5.4 Testing Conditions and Error Analysis

The results presented above were obtained by testing the model using a custom Python application that allows the user to draw digits manually and send the resulting images to the hardware system via UART. A total of 20 samples per digit were tested, leading to 200 test cases overall.
Unlike standard MNIST testing, this method introduces variation in handwriting style, line thickness, and centering, which may differ significantly from the training distribution. As a result, overall accuracy is lower compared to evaluations on MNIST data.
Despite this, the network demonstrated robust behavior:

- Digits **7, 8, and 9** were the most challenging to classify, with accuracies around **25%** in both software and hardware. This suggests that the network struggles to generalize these digits when drawn in a freer style.
- Digit **2** was classified correctly in **100%** of the cases by both implementations, indicating strong robustness for this class.

These observations suggest that although hardware precision has an effect, a major source of error comes from input variability, emphasizing the importance of dataset alignment between training and testing phases.

6 Timing comparison

6.1 Execution Time Comparison

In this section, we estimate the number of clock cycles required for the hardware implementation to complete a single inference and compare the result to the execution time of the equivalent software model.

Hardware Timing Breakdown

Assuming a system clock of 100 MHz (i.e., 10 ns per cycle), the total execution time can be estimated by summing the clock cycles for each component:

- **UART Transmission:** 784 pixels $\times$ 32 bits $=$ 25088 bits. At 115200 baud, this requires:

$$\frac{25088}{115200} \approx 0.218 \text{ s} \approx 218 \text{ ms}$$

  This is the main bottleneck of the system.

- **Neuron computation:** each neuron performs 785 MAC operations. With a 9-stage multiplier and a 23-stage accumulator (pipelined), each new result takes 1 cycle per input $+$ latency overhead (constant). Thus, a single neuron completes in approximately:

$$785 + 9 + 23 = 817 \text{ cycles}$$

- **Layer 1:** The layer processes 128 neurons in 4 groups of 32, reusing the 32 physical units. FSM adds 4 extra cycles per group:

$$4 \times (817 + 4) = 3284 \text{ cycles}$$

- **Output Memory (Layer 1 $\rightarrow$ Layer 2):** 129 values saved $\rightarrow$ 129 cycles.
- **Layer 2:** All 10 neurons run in parallel:

$$817 + 3 = 820 \text{ cycles}$$

- **Comparison and Display:** The comparator performs 9 pairwise operations with 2-stage compare pipeline and 2 FSM cycles:

$$(3 + 2) \times 9 + 2 = 47 \text{ cycles}$$

Total Clock Cycles (excluding UART):

$$3284 + 129 + 820 + 47 = \boxed{4280 \text{ cycles}}$$

$$\text{Execution time at 100MHz} \approx 4280 \times 10\,\text{ns} = \boxed{42.8\,\mu\text{s}}$$

Including UART:

$$\boxed{218\,\text{ms} + 42.8\,\mu\text{s}} \approx 218\,\text{ms} \quad \text{(dominated by UART)}$$

Software Execution Time

The same inference performed using a Python implementation (the network built with the same weights) on CPU takes between:

$$\boxed{0.1 - 0.3\,\text{ms}}$$

Discussion

Although the hardware pipeline itself is significantly faster (˜43 μs), the UART transmission introduces a large overhead.
The actual compute part on hardware (that is the main part of the project) is **2–7$\times$ faster** than the software version, demonstrating the benefit of hardware acceleration—particularly when communication is not a bottleneck.

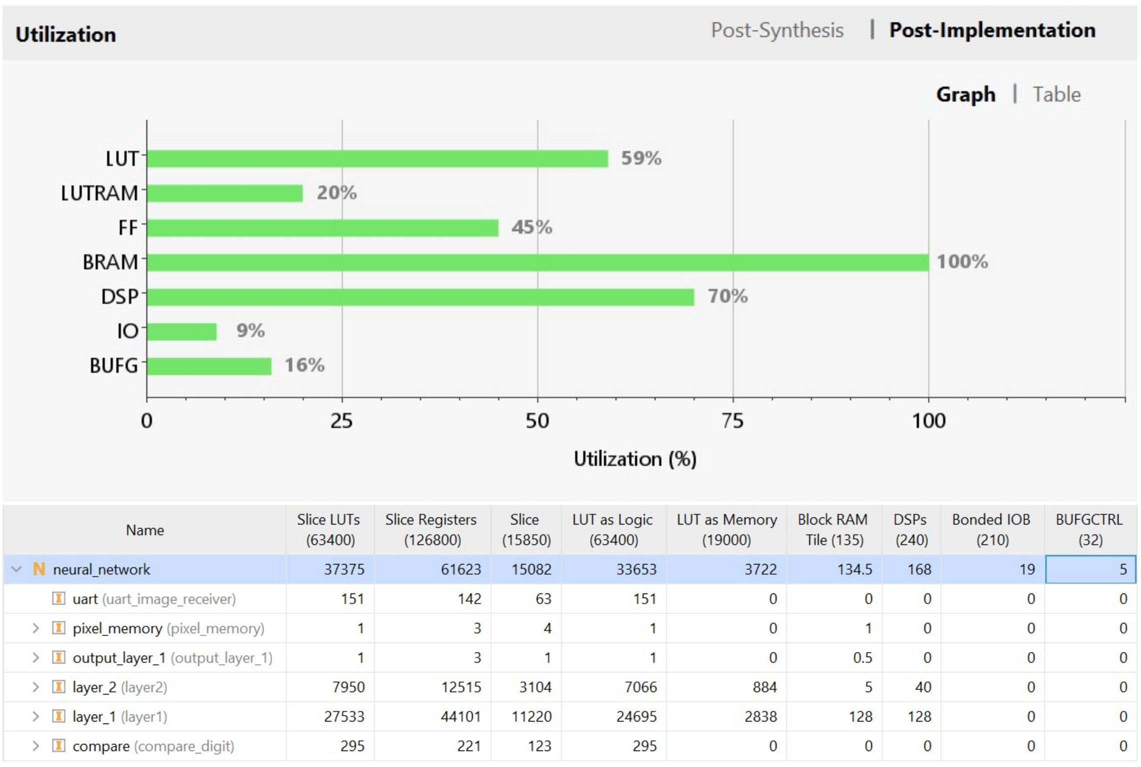
7 Performance and resource utilization

This section focuses on the on board performance of the circuit, showing Vivado results for area, timing and power utilization

7.1 Area Utilization

The resource utilization of the implemented neural network is presented in Figure 4. The design demonstrates intensive use of FPGA resources while leaving room for additional functionality:

- **BRAM** utilization is nearly maximized at 99.63% (134.5 out of 135 blocks), indicating intensive memory usage characteristic of neural network implementations.
- **DSP** blocks are utilized at 70% (168 out of 240), showing significant but not exhaustive use of hardware multipliers.
- **LUT** usage stands at 58.95% (37,375 out of 63,400), while **FF** utilization is 45.37% (57,527 out of 126,800).
- Lower utilization is seen in **LUTRAM** (19.59%), **IO** (9.05%), and **BUFG** (15.63%).



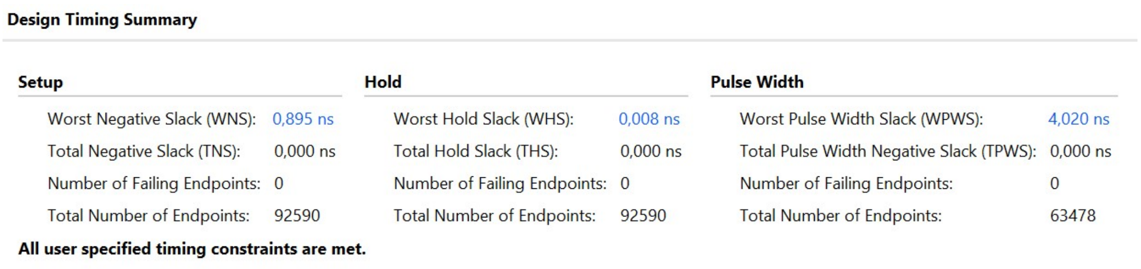
Name	Slice LUTs (63400)	Slice Registers (126800)	Slice (15850)	LUT as Logic (63400)	LUT as Memory (19000)	Block RAM Tile (135)	DSPs (240)	Bonded IOB (210)	BUFGCTRL (32)
neural_network	37375	61623	15082	33653	3722	134.5	168	19	5
uart (uart_image_receiver)	151	142	63	151	0	0	0	0	0
pixel_memory (pixel_memory)	1	3	4	1	0	1	0	0	0
output_layer_1 (output_layer_1)	1	3	1	1	0	0.5	0	0	0
layer_2 (layer2)	7950	12515	3104	7066	884	5	40	0	0
layer_1 (layer1)	27533	44101	11220	24695	2838	128	128	0	0
compare (compare_digit)	295	221	123	295	0	0	0	0	0

Figure 4: Resource utilization showing both graphical and tabular representations of FPGA resource consumption.

7.2 Timing Analysis

The implemented design meets all timing constraints, as verified by Vivado's static timing analysis 5. The worst negative slack (WNS) of **0.895 ns** indicates that the most critical path still has sufficient margin at the target clock frequency of 100 MHz (10 ns period).
Key timing metrics include:

- **Setup Time Analysis:** No setup violations were reported (TNS = 0.000 ns) across all 92,590 endpoints.
- **Hold Time Analysis:** No hold violations occurred (WHS = 0.008 ns) with zero failing endpoints.
- **Pulse Width:** The minimum pulse width constraint is comfortably satisfied with a slack of 4.020 ns.



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,895 ns	Worst Hold Slack (WHS):	0,008 ns	Worst Pulse Width Slack (WPWS):	4,020 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	92590	Total Number of Endpoints:	92590	Total Number of Endpoints:	63478

All user specified timing constraints are met.

Figure 5: Timing summary showing all constraints are met with positive slack values.

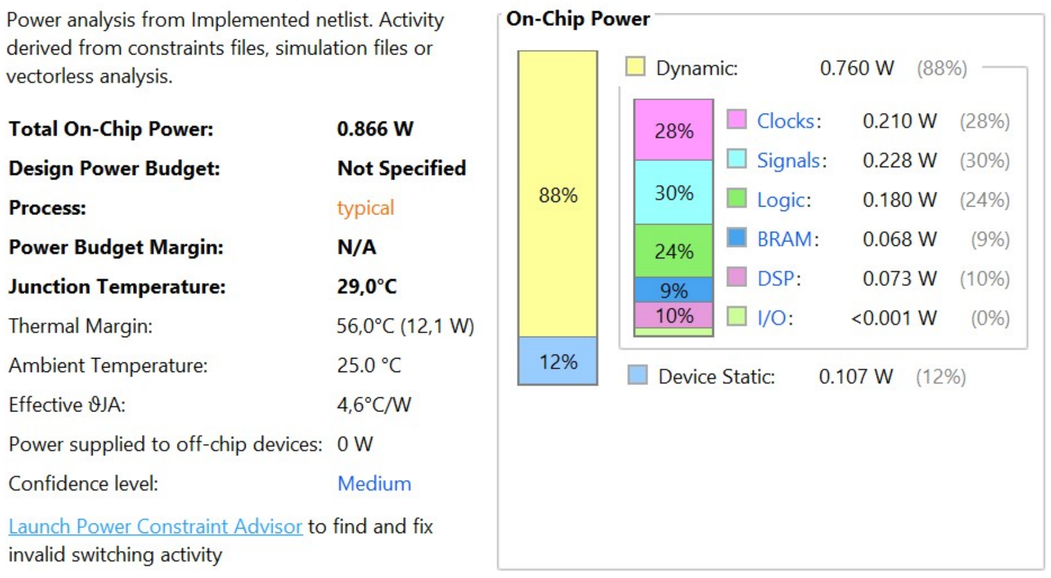
These results confirm the design's temporal reliability and suitability for operation at the specified system clock frequency.

7.3 Power Consumption

The power analysis (6) reveals a total on-chip power consumption of **0.866 W** with the following distribution:

- **Dynamic Power:** 0.760 W (88% of total)
  - Clock network: 0.210 W (28% of dynamic)
  - Signals: 0.228 W (30%)
  - Logic: 0.180 W (24%)
  - BRAM: 0.068 W (9%)
  - DSP: 0.073 W (10%)
- **Static Power:** 0.107 W (12% of total)

The thermal analysis shows a junction temperature of 29.0°C with a substantial thermal margin of 56.0°C, indicating excellent thermal characteristics for the design.



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.866 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**29,0°C**
Thermal Margin:	56,0°C (12,1 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	4,6°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	0.760 W	(88%)
Clocks:	0.210 W	(28%)
Signals:	0.228 W	(30%)
Logic:	0.180 W	(24%)
BRAM:	0.068 W	(9%)
DSP:	0.073 W	(10%)
I/O:	<0.001 W	(0%)
Device Static:	0.107 W	(12%)

Figure 6: Power distribution showing dynamic and static power components with detailed breakdown.

8 Conclusions

This project presented the implementation of a fully connected neural network for handwritten digit recognition on an FPGA. The system was designed using a modular architecture, with each block optimized for hardware execution. Performance evaluation demonstrated that the network is capable of classifying user-drawn digits with reasonable accuracy, closely matching its software version.

Timing analysis confirmed that the design meets all real-time constraints, and resource utilization remained within acceptable limits, confirming the feasibility of deploying such a neural model on an FPGA. Power analysis further supported the efficiency of the implementation. Overall, this work proves that FPGA-based neural networks are a viable and flexible solution for embedded machine learning applications.