ISTANBUL TECHNICAL UNIVERSITY

FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING

DIGITAL SYSTEM DESIGN APPLICATION

(EHB 436E)

FINAL PROJECT

NAME - SURNAME: Faruk Onar – Ahmet Utku Alkan

ID: 040210810 - 040220116

1. PROPOSED CNN MODEL ARCHITECTURE

The proposed Convolutional Neural Network (CNN) model is designed to perform handwritten digit classification on the MNIST dataset using a fully hardware-based implementation [1]. The process begins with an input buffer that stores a 28 X 28 grayscale image, where each pixel is represented using 8-bit unsigned fixed-point precision. This architecture prioritizes an efficient balance between accuracy and hardware resource utilization by employing fixed-point arithmetic for the inference stage [2], while the network parameters are trained offline and pre-loaded into the system.

Feature extraction is initiated by a convolutional layer that applies four distinct 3X3 kernels to the input image with a stride of one and without padding. These filters compute the sum of products using 8-bit signed weights to generate feature maps. Following this operation, a Rectified Linear Unit (ReLU) activation function is applied to introduce non-linearity by passing positive values unchanged while setting negative values to zero. A max-pooling layer with a 2 X 2 window then processes the activated maps, downsampling the spatial dimensions by selecting the maximum value within each window to retain the most prominent features [3].

To transition from feature extraction to classification, the downsampled 2D feature maps are reshaped into a one-dimensional vector by a flattening buffer. This vector feeds into the first fully connected layer (FC1), which consists of 32 neurons and performs matrix-vector multiplication with a bias addition. The resulting features are then passed to a second fully connected layer (FC2), which maps the data to 10 output neurons, each corresponding to one of the digit classes from 0 to 9 [3].

The final digit classification is achieved through a Decision Unit that evaluates the outputs of the FC2 layer. This unit executes an "argmax" operation, selecting the index of the neuron with the highest activation value as the final prediction for the input image. The entire system operates synchronously under the control of a central Finite State Machine (FSM), which coordinates the sequence of operations across all blocks, including states for convolution, ReLU, pooling, flattening, and the fully connected layers [2].

2. BLOCK DIAGRAMS

2.1. Top Module Block Diagram

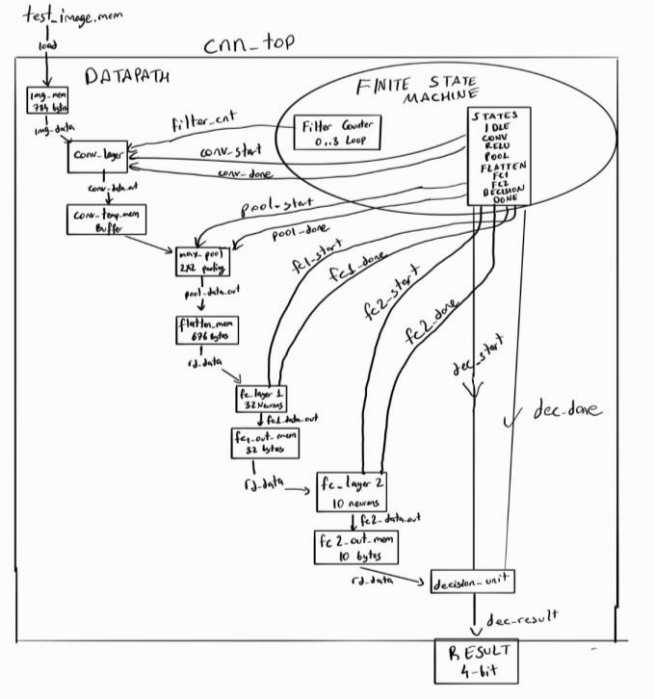


Figure 1: Block diagram of the top module

The top-level module, designated as cnn_top, serves as the main wrapper for the fully hardware-based Convolutional Neural Network accelerator. This module encapsulates the entire datapath, the memory hierarchy, and the control logic required to perform MNIST digit classification. The design follows a modular architecture where each layer of the CNN (Convolution, Pooling, Fully Connected layers) is implemented as a dedicated hardware block, interconnected through a central control mechanism.

Control Plane The Finite State Machine (FSM): The core of the system's operation is governed by a central Finite State Machine (FSM), depicted on the right side of the block diagram. This FSM acts as the global controller, coordinating the timing and execution sequence of all sub-modules. It transitions through a sequence of states—IDLE, CONV, POOL, FC1, FC2, DECISION, and DONE—to ensure that data flows correctly through the network pipeline. The FSM is responsible for managing the Filter Counter loop, which iterates through the four required convolution filters before triggering the subsequent layers. By maintaining a strict state sequence, the FSM prevents data hazards and ensures that a layer only begins processing once the preceding layer has successfully completed its operation.

Datapath and Memory Hierarchy: The datapath, shown on the left side of the diagram, represents the flow of image data through the processing stages. The process begins with the img_mem buffer, which stores the initial 28 X 28 input image. The conv_layer reads from this buffer and writes the resulting feature maps into the conv_temp_mem intermediate buffer. This data is then consumed by the max_pool unit, which performs downsampling. A critical transition occurs at the flatten_mem, which reshapes the 2D pooled feature maps into a 1D vector suitable for the fully connected layers. The data then propagates through fc_layer_1 and

fc_layer_2, with intermediate results stored in fc1_out_mem (32 bytes) and fc2_out_mem (10 bytes), respectively. This dedicated memory architecture allows for efficient data storage between processing stages without the need for complex memory management units.

Inter-Module Communication and Signals:

Communication between the FSM and the processing units is achieved through a synchronous "start-done" handshaking protocol. As illustrated in the diagram, the FSM asserts a specific start signal (e.g., conv-start, pool-start, fc1-start) to activate a module. The module then processes the data and, upon completion, asserts a corresponding done signal (e.g., conv-done, dec-done) back to the FSM. This feedback mechanism allows the FSM to transition to the next state. The final stage of the pipeline is the decision_unit, which reads the classification scores from fc2_out_mem and outputs the final 4-bit dec-result signal, representing the predicted digit (0-9).

2.2. Convolution Layer Block Diagram

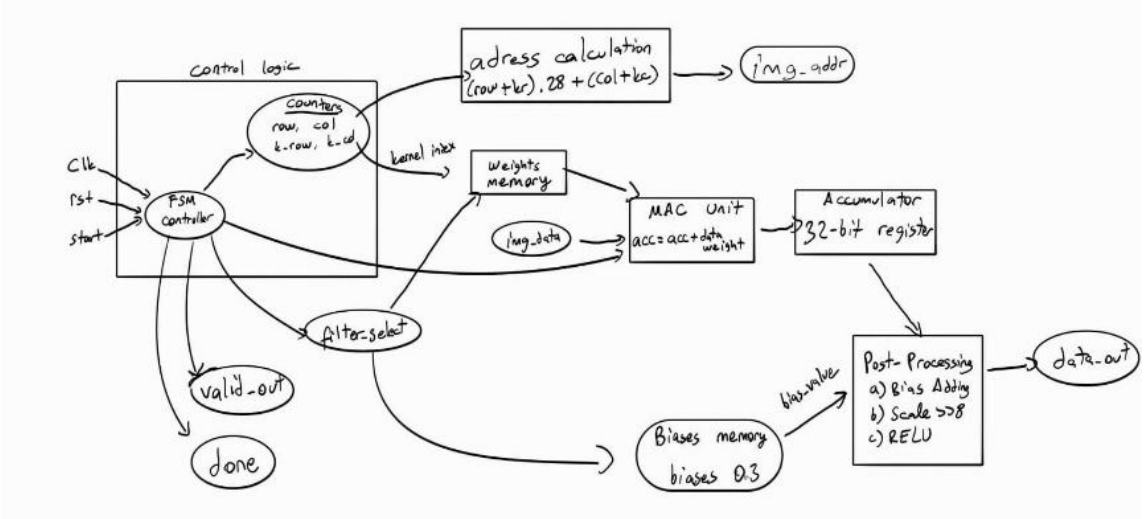


Figure 2: Block diagram of the convolution layer module

Description: The block diagram in Figure 2 illustrates the internal hardware organization of the conv_layer module. The design is partitioned into three primary subsystems: the Control Unit, the Memory Interface, and the Arithmetic Datapath.

Control Unit: At the core is a Finite State Machine (FSM) that orchestrates the operation sequences. It manages a set of counters (row, col, k_row, k_col) to traverse the input image and the 3x3 convolution kernel (sliding window). The Address Calculation block dynamically computes the linear memory address using the formula (row+kr)*28 + (col+kc) to fetch the correct pixel data.

Memory Interface: The module houses internal ROMs for Weights and Biases. The filter_select input signal acts as a multiplexer selector, determining which set of filter weights and biases are active for the current operation.

Arithmetic Datapath (MAC & Post-Processing): The computational pipeline performs the feature extraction:

MAC Unit: The pixel data (img_data) and corresponding weights are multiplied and accumulated into a 32-bit Register to prevent overflow.

Post-Processing: Once the 3x3 window accumulation is complete, the result passes through a dedicated post-processing block. This block adds the bias, applies the fixed-point scaling (Right Shift by 8), and implements the ReLU activation function (clipping negative values to zero) before driving the final data_out.

2.3. Max Pooling Block Diagram

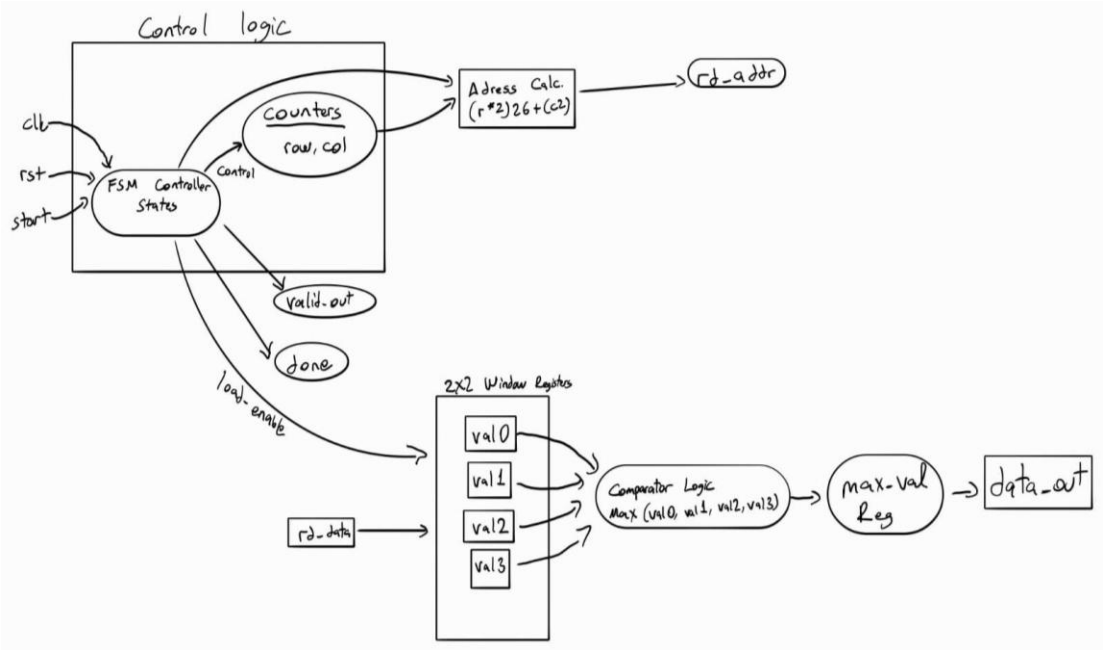


Figure 3: Block diagram of the max pool module

The block diagram in Figure 3 details the internal architecture of the max_pool module, which performs 2x2 downsampling. The design is divided into control logic and a data processing path.

Control & Addressing: The Control Unit is driven by a Finite State Machine (FSM) that cycles through READ states to fetch four neighboring pixels forming a 2x2 window. The Address Calculation block implements the stride logic, generating memory addresses using the formula (r*2)*26 + (c*2) to correctly map the 26x26 input feature map to the downsampled output.

Data Buffering & Comparison: The datapath includes a Pixel Buffer consisting of four registers (val0–val3). As data is fetched from memory (rd_data), it is stored in these registers. Once the window is full, the Comparator Logic evaluates all four values simultaneously to identify the maximum intensity. This result is registered in max_val and driven to the output (data_out) along with the valid signal.

2.4.Fully Connected Layer Block Diagram

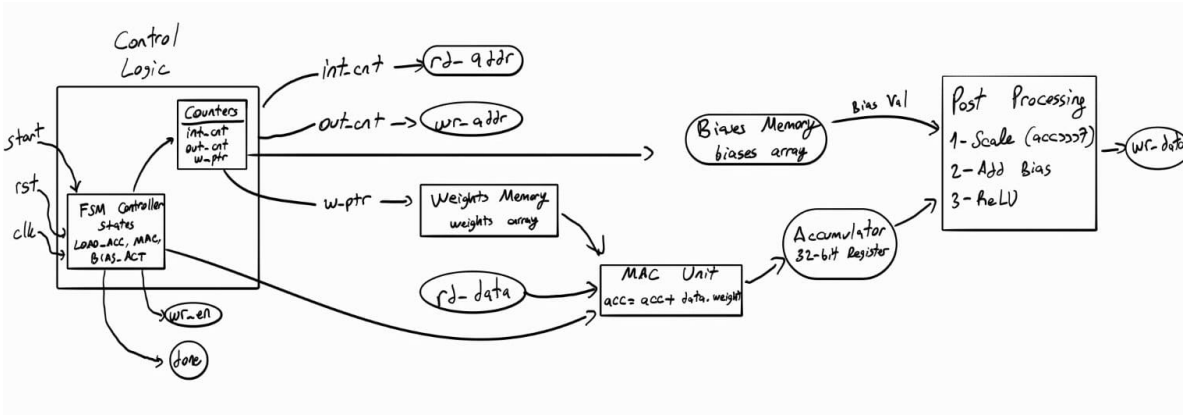


Figure 4: Block diagram of the fully connected layer module

The block diagram in Figure 4 depicts the hardware implementation of the fc_layer module, which performs dense matrix-vector multiplication. The architecture is structured around a central FSM and a high-precision arithmetic pipeline.

Control & Memory Management: The Control Logic is governed by an FSM with three primary states: LOAD_ACC, MAC (computation loop), and BIAS_ACT (finalization). A set of counters manages the iteration: in_cnt iterates through the input features (e.g., 0 to 675), while out_cnt tracks the current output neuron (e.g., 0 to 31). These counters simultaneously address the internal Weights and Biases Memories (ROMs) and generate the read addresses (rd_addr) for the input data.

Arithmetic Datapath (MAC Pipeline): The core computation occurs in the MAC Unit, which multiplies the input data with the corresponding weight and adds the result to the Accumulator (32-bit Register). This accumulation continues until all inputs for a specific neuron are processed.

Post-Processing: Once the accumulation loop is finished, the result enters the Post-Processing block. Here, three critical operations are performed sequentially:

Scaling: The value is right-shifted (e.g., >>> 7) to adjust for fixed-point arithmetic.

Bias Addition: The neuron's bias value is added to the result.

Activation: The ReLU function clips negative values to zero (if enabled), and the final result is written to the output memory via wr_data.

2.5.Decision Layer Block Diagram

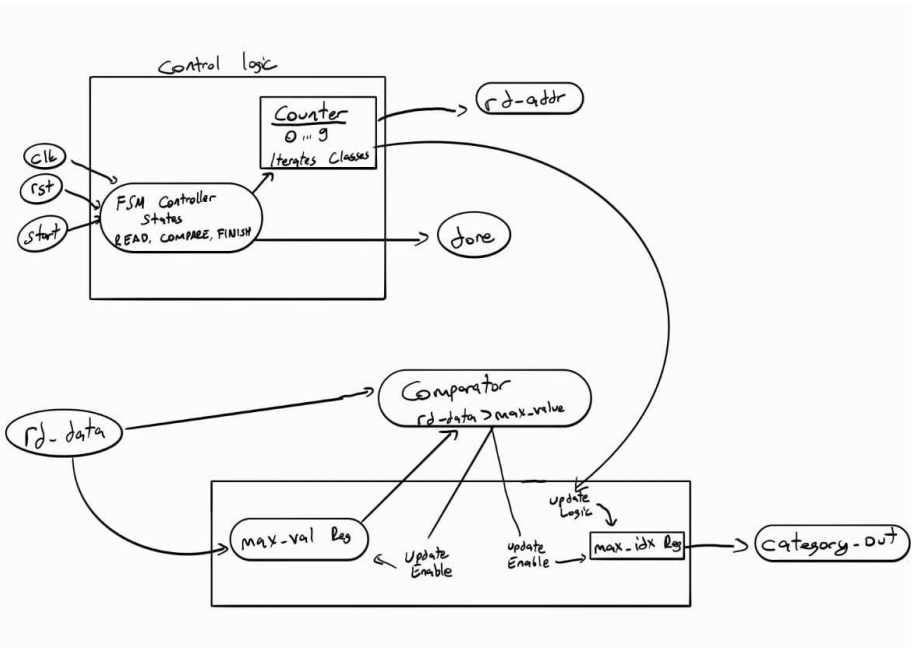


Figure 5: Block diagram of the decision layer module

The block diagram in Figure 5 illustrates the architecture of the decision_unit, which implements the Argmax function to finalize the classification. The design consists of a sequential scanning mechanism and a comparison logic block.

Control & Scanning: The Control Logic relies on an FSM to iterate through the 10 class scores (digits 0-9) generated by the final fully connected layer. A Counter generates the read addresses (rd_addr) to fetch these scores sequentially from memory.

Comparison & Storage: The datapath focuses on identifying the highest confidence score. As each score (rd_data) arrives, the Comparator evaluates it against the currently stored maximum (max_val).

Update Logic: If the incoming score is strictly greater than the stored value, the comparator triggers an update.

Registers: The system updates max_val with the new score and captures the current counter value into the max_idx register.

Output: Once all 10 classes have been scanned, the value held in the max_idx register is driven to category_out as the final predicted digit.

3. MODULES

3.1.Convolution Unit

The conv_layer module acts as the core processing unit of the CNN accelerator. It performs 2D convolution operations on input images using pre-trained weights and biases. Designed for resource-constrained FPGAs, the module utilizes a time-multiplexed architecture where a single hardware block processes multiple filters sequentially, controlled by the filter_select input.

3.1.1. Verilog Code

```
`timescale 1ns / 1ps

module conv_layer #(
    parameter WEIGHT_FILE = "conv1_weights.mem",
    parameter BIAS_FILE = "conv1_bias.mem"
)(
    input wire clk,
    input wire rst,
    input wire start,
    input wire [1:0] filter_select,
    output reg [9:0] img_addr,
    input wire [7:0] img_data,
    output reg [7:0] data_out,
    output reg valid_out,
    output reg done
);

    // --- Memories ---
    reg signed [7:0] weights [0:35];
    reg signed [7:0] biases [0:3];

    initial begin
        $readmemh(WEIGHT_FILE, weights);
        $readmemh(BIAS_FILE, biases);
    end

    localparam IDLE=0, LOAD_PIXEL=1, WAIT_MEM=2, MAC_OP=3, BIAS_RELU=4,
OUTPUT_DATA=5, FINISHED=6;
    reg [2:0] state;

    reg [4:0] row, col;
    reg [1:0] k_row, k_col;

    // 32-bit Accumulator
    reg signed [31:0] acc;
    reg signed [31:0] temp_val;

    always @(posedge clk) begin
        if (rst) begin
            state <= IDLE;
            row <= 0; col <= 0;
            k_row <= 0; k_col <= 0;
            acc <= 0;
            img_addr <= 0;
            valid_out <= 0;
            done <= 0;
        end else begin
            case(state)
                IDLE: begin
                    done <= 0;
                    valid_out <= 0;
```

```
                    if (start) begin
                        // --- CRITICAL FIX HERE ---
                        // Reset counters on every 'start' signal!
                        // Otherwise, subsequent filters (e.g., filter 2) won't
run.
                        row <= 0; col <= 0;
                        k_row <= 0; k_col <= 0;
                        acc <= 0;
                        state <= LOAD_PIXEL;
                    end
                end

                LOAD_PIXEL: begin
                    valid_out <= 0;
                    img_addr <= (row + k_row) * 28 + (col + k_col);
                    state <= WAIT_MEM;
                end

                WAIT_MEM: state <= MAC_OP;

                MAC_OP: begin
                    acc <= acc + ($signed({1'b0, img_data}) *
weights[(filter_select * 9) + (k_row * 3) + k_col]);

                    if (k_col == 2) begin
                        k_col <= 0;
                        if (k_row == 2) begin
                            k_row <= 0;
                            state <= BIAS_RELU;
                        end else begin
                            k_row <= k_row + 1;
                            state <= LOAD_PIXEL;
                        end
                    end else begin
                        k_col <= k_col + 1;
                        state <= LOAD_PIXEL;
                    end
                end

                BIAS_RELU: begin
                    // Math: Scaling and Bias
                    acc <= (acc >>> 8) + biases[filter_select];

                    // DEBUG: Check center pixel
                    if (row == 13 && col == 13) begin
                       $display("DEBUG [Conv]: Filter %0d (Center Pixel) -> Raw
Value: %d", filter_select, acc);
                    end

                    state <= OUTPUT_DATA;
                end

                OUTPUT_DATA: begin
                    // ReLU
                    if (acc < 0) temp_val = 0;
                    else temp_val = acc;

                    // Clipping
                    if (temp_val > 255) data_out <= 255;
                    else data_out <= temp_val[7:0];

                    valid_out <= 1;
                    acc <= 0;

                    if (col == 25) begin
                        col <= 0;
                        if (row == 25) state <= FINISHED;
                        else begin row <= row + 1; state <= LOAD_PIXEL; end
```

```
                    end else begin
                        col <= col + 1;
                        state <= LOAD_PIXEL;
                    end
                end

                FINISHED: begin
                    valid_out <= 0; done <= 1;
                    if (!start) state <= IDLE;
                end
            endcase
        end
    end
endmodule
```

Code 1: Verilog code of the convolution unit

Architecture and Data Flow:

The module operates based on a Finite State Machine (FSM) that coordinates memory access, arithmetic operations, and data synchronization.

Memory Interface: The module does not store the entire image; instead, it generates read addresses (img_addr) to fetch pixel data from an external block memory.

Weight Storage: Weights and biases for all kernels are stored internally in reg arrays and initialized via .mem files ($readmemh), reducing external memory bandwidth usage.

Kernel Processing: The design implements a 3×3 sliding window. For each position on the 28×28 input grid, the module performs 9 Multiply-Accumulate (MAC) operations.

Finite State Machine (FSM) States:

The control logic is divided into seven distinct states:

IDLE: Waits for the start signal. Upon triggering, it resets all internal counters (row, col, k_row, k_col) to ensure correct initialization for every filter iteration.

LOAD_PIXEL: Calculates the 1D linear address of the required pixel corresponding to the current (row,col) and kernel offset (k_row,k_col).

WAIT_MEM: Inserts a clock cycle latency to accommodate the synchronous read behavior of the FPGA Block RAM.

MAC_OP: Performs the signed multiplication of the pixel and weight, adding the result to a 32-bit accumulator (acc) to prevent overflow.

BIAS_RELU: Once the 3×3 kernel window is complete, the bias is added. The result is right-shifted by 8 bits (>>> 8) to compensate for the fixed-point scaling factor (2^7×2^0 logic).

OUTPUT_DATA: Applies the Rectified Linear Unit (ReLU) activation function (negative values become 0) and saturates (clips) the result to 8 bits (0−255) before writing to the output.

FINISHED: Asserts the done signal when the entire output feature map (26×26) is generated.

Arithmetic Precision:

The module uses fixed-point arithmetic to avoid the high hardware cost of floating-point units.

Weights: 8-bit signed integers (Q0.7 format).

Input Data: 8-bit unsigned integers (Q8.0 format).
Accumulator: 32-bit signed integer to handle intermediate sums without overflow.

3.1.2. Testbench Code

The tb_conv_layer module serves as the simulation test harness designed to verify the functional correctness and timing constraints of the conv_layer hardware. It generates the necessary system stimuli—including clock signals, reset pulses, and memory data—to mimic the physical operating conditions of the FPGA.

```
`timescale 1ns / 1ps

module tb_conv_layer;

    // --- Signals ---
    reg clk;                    // System Clock
    reg rst;                    // Reset signal
    reg start;                  // Signal to start the operation
    reg [1:0] filter_select;    // Selects which filter to use (0, 1, 2, or 3)

    // --- Outputs from the Design (DUT) ---
    wire [9:0] img_addr;        // Address requested by the convolution module
    wire [7:0] data_out;        // Result of the convolution (pixel value)
    wire valid_out;             // Flag indicating 'data_out' is valid
    wire done;                  // Flag indicating the process is finished

    // --- Memory Simulation Signals ---
    reg [7:0] tb_img_data;      // Data read from the simulated memory
    reg [7:0] test_image [0:783]; // Simulated Block RAM (Holds the 28x28 image)

    // --- Instantiate the Unit Under Test (UUT) ---
    // Here we connect our 'conv_layer' module to this testbench
    conv_layer uut (
        .clk(clk),
        .rst(rst),
        .start(start),
        .filter_select(filter_select),
        .img_addr(img_addr),
        .img_data(tb_img_data),
        .data_out(data_out),
        .valid_out(valid_out),
        .done(done)
    );

    // --- Clock Generation ---
    // Generates a 100MHz clock (Period = 10ns)
    always #5 clk = ~clk;

    // --- Memory Read Logic ---
    // Simulates the behavior of FPGA Block RAM.
    // When the module asks for an address (img_addr),
    // the data is provided on the next clock cycle.
    always @(posedge clk) begin
        tb_img_data <= test_image[img_addr];
    end

    // --- 4-STAGE TEST SCENARIO ---
    initial begin
        // 1. Initialize Signals
        clk = 0; rst = 1; start = 0; filter_select = 0; tb_img_data = 0;

        // 2. Load the Image File into Memory
        // Ensure "test_image.mem" is in your simulation folder
        $readmemh("test_image.mem", test_image);

        $display("--- STARTING FULL 4-FILTER TEST ---");
```

```
        // 3. Apply Reset
        #100; rst = 0; #20;

        // --- TEST FILTER 0 ---
        $display("[Time: %t] Running Filter 0 (Kernel 1)...", $time);
        filter_select = 2'b00;      // Select 1st Filter
        start = 1; #10; start = 0;  // Send a 1-clock pulse to START

        wait(done);                 // Wait until the module finishes
        $display("-> Filter 0 Completed.");
        #50;                        // Short delay before next filter

        // --- TEST FILTER 1 ---
        $display("[Time: %t] Running Filter 1 (Kernel 2)...", $time);
        filter_select = 2'b01;      // Select 2nd Filter
        start = 1; #10; start = 0;

        wait(done);
        $display("-> Filter 1 Completed.");
        #50;

        // --- TEST FILTER 2 ---
        $display("[Time: %t] Running Filter 2 (Kernel 3)...", $time);
        filter_select = 2'b10;      // Select 3rd Filter
        start = 1; #10; start = 0;

        wait(done);
        $display("-> Filter 2 Completed.");
        #50;

        // --- TEST FILTER 3 ---
        $display("[Time: %t] Running Filter 3 (Kernel 4)...", $time);
        filter_select = 2'b11;      // Select 4th Filter
        start = 1; #10; start = 0;

        wait(done);
        $display("-> Filter 3 Completed.");

        // End of Test
        $display("--- ALL FILTERS COMPLETED SUCCESSFULLY ---");
        $stop; // Stop the simulation
    end

endmodule
```

Code 2: Testbench code of the convolution unit

Key Functional Blocks:

The testbench is composed of three primary subsystems:

Clock Generation: A continuous 100 MHz system clock is generated with a period of 10ns (always #5 clk = ~clk). This defines the temporal resolution for all synchronous operations.

Memory Model (BRAM Simulation): Since the image data resides in external block memory, the testbench simulates a Single-Port ROM using a register array (test_image). Crucially, it models the synchronous read behavior of FPGA Block RAMs. When the Device Under Test (DUT) asserts an address on img_addr, the testbench drives the corresponding data to tb_img_data on the subsequent clock edge, accurately replicating hardware latency.

Device Under Test (DUT): The conv_layer module is instantiated and connected to the testbench signals, allowing for direct observation of its internal state and output buses.

Verification Scenario:

The simulation executes a sequential stress test covering all available filter configurations:

Initialization: At t=0, the system is reset, and the pre-processed image data (test_image.mem) is loaded into the simulated memory using the $readmemh system task.

Sequential Kernel Execution: The testbench iterates through the four filter indices (filter_select 0 to 3). For each iteration:

A one-cycle start pulse is asserted to trigger the FSM.

The testbench enters a wait state (wait(done)), pausing execution until the DUT signals completion. This verifies the handshake protocol between the controller and the processing unit.

Logging and Timing Analysis: The $display tasks act as a self-checking mechanism, printing timestamps to the console at the start and end of each filter operation. This allows for the verification of execution time and FSM state transitions.

3.1.3. Simulation Results

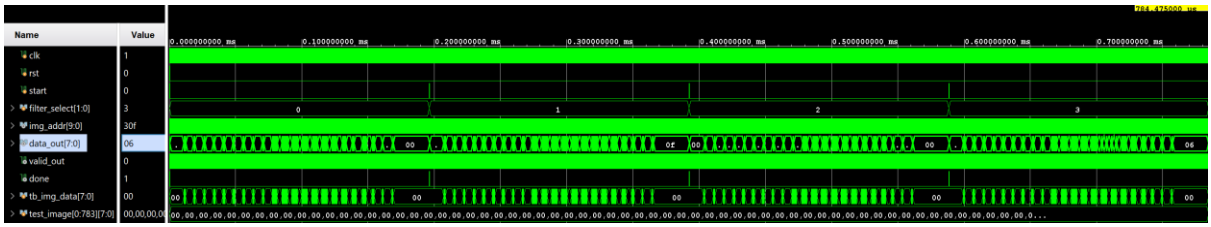


Figure 6: Timing Diagram (Waveform) of the Convolutional Layer

The timing diagram illustrates the sequential processing architecture designed to optimize FPGA resource utilization. Instead of instantiating four parallel convolution blocks, a single hardware unit is time-multiplexed.

As observed in the waveform:

Sequential Execution: The filter_select signal transitions from 00 to 11 over time, confirming that the four kernels are processed serially.

Handshake Protocol: The start and done signals demonstrate a correct handshake mechanism. The module remains busy (done signal low) during the active processing window and asserts the done signal upon completion of each filter.

Data Throughput: The high activity on the img_addr and data_out buses indicates continuous data processing without unnecessary idle cycles, validating the efficiency of the Finite State Machine (FSM) design.

```
DEBUG [Conv]: Filter 0 (Center Pixel) -> Raw Value:       53394
-> Filter 0 Completed.
[Time:            196225000] Running Filter 1 (Kernel 2)...
DEBUG [Conv]: Filter 1 (Center Pixel) -> Raw Value:        7291
-> Filter 1 Completed.
[Time:            392325000] Running Filter 2 (Kernel 3)...
DEBUG [Conv]: Filter 2 (Center Pixel) -> Raw Value:       -5593
-> Filter 2 Completed.
[Time:            588425000] Running Filter 3 (Kernel 4)...
DEBUG [Conv]: Filter 3 (Center Pixel) -> Raw Value:       56693
-> Filter 3 Completed.
--- ALL FILTERS COMPLETED SUCCESSFULLY ---
```

Figure 7: Tcl Console Output for Convolution Layer Verification

The behavioral simulation results obtained from the Tcl Console confirm the functional correctness of the Convolutional Layer. The simulation log demonstrates the successful execution of the 4-stage filtering process, indicated by the message 'ALL FILTERS COMPLETED SUCCESSFULLY'.
Furthermore, the debug prints reveal the internal state of the 32-bit accumulator before the activation function is applied. For instance, Filter 0 yielded a high positive raw value (53394), indicating a strong feature correlation at the center pixel. Conversely, Filter 2 resulted in a negative value (-5593), which verifies that the signed arithmetic logic and bias addition are functioning correctly. This negative value is subsequently handled by the ReLU unit, ensuring correct non-linear activation.

3.1.4. Utilization Summary

Resource	Utilization	Available	Utilization %
LUT	40	32600	0.12
FF	29	65200	0.04
IO	23	210	10.95

Figure 8: Resource utilization summary for the conv_layer module

The implementation results demonstrate exceptional hardware efficiency. The module utilizes only 40 LUTs and 29 Flip-Flops, confirming that the control logic (FSM and address counters) is optimized for minimal area. This lightweight footprint ensures that the FPGA resources are effectively conserved for parallel processing and arithmetic units.

3.2.Max Pooling

The max_pool module implements a non-overlapping 2×2 max pooling operation, which serves as a down-sampling layer in the CNN architecture. Its primary function is to reduce the spatial dimensions of the input feature map from 26×26 (Output of Convolution) to 13×13, thereby reducing computational complexity for subsequent layers while retaining the most prominent features.

3.2.1. Verilog Code

```
`timescale 1ns / 1ps

module max_pool(
    input wire clk,
```

```
    input wire rst,
    input wire start,

    output reg [9:0] rd_addr,
    input wire [7:0] rd_data,

    output reg [7:0] data_out,
    output reg valid_out,
    output reg done
);

    parameter IN_DIM = 26;
    parameter OUT_DIM = 13;

    localparam IDLE=0, READ_0=1, WAIT_1=2, READ_1=3, WAIT_2=4, READ_2=5, WAIT_3=6,
READ_3=7, SAVE_LAST=8, FIND_MAX=9, OUTPUT=10, NEXT=11, FINISH=12;
    reg [3:0] state;

    reg [4:0] row, col;
    reg [7:0] val0, val1, val2, val3;
    reg [7:0] max_val;

    always @(posedge clk) begin
        if (rst) begin
            state <= IDLE;
            row <= 0; col <= 0;
            valid_out <= 0; done <= 0;
            rd_addr <= 0; val0<=0; val1<=0; val2<=0; val3<=0; max_val<=0;
        end else begin
            case(state)
                IDLE: begin
                    done <= 0; valid_out <= 0;
                    if (start) begin
                        row <= 0; col <= 0;
                        state <= READ_0;
                    end
                end

                READ_0: begin
                    rd_addr <= (row * 2) * IN_DIM + (col * 2);
                    state <= WAIT_1;
                end
                WAIT_1: state <= READ_1;

                READ_1: begin
                    val0 <= rd_data;
                    rd_addr <= (row * 2) * IN_DIM + (col * 2 + 1);
                    state <= WAIT_2;
                end
                WAIT_2: state <= READ_2;

                READ_2: begin
                    val1 <= rd_data;
                    rd_addr <= (row * 2 + 1) * IN_DIM + (col * 2);
                    state <= WAIT_3;
                end
                WAIT_3: state <= READ_3;

                READ_3: begin
                    val2 <= rd_data;
                    rd_addr <= (row * 2 + 1) * IN_DIM + (col * 2 + 1);
                    state <= SAVE_LAST;
                end

                SAVE_LAST: begin
                    val3 <= rd_data;
                    state <= FIND_MAX;
                end
```

```
                FIND_MAX: begin
                    if (val0 >= val1 && val0 >= val2 && val0 >= val3) max_val <=
val0;
                    else if (val1 >= val0 && val1 >= val2 && val1 >= val3) max_val
<= val1;
                    else if (val2 >= val0 && val2 >= val1 && val2 >= val3) max_val
<= val2;
                    else max_val <= val3;

                    state <= OUTPUT;
                end

                OUTPUT: begin
                    data_out <= max_val;
                    valid_out <= 1;
                    state <= NEXT;
                end

                NEXT: begin
                    valid_out <= 0;
                    if (col == OUT_DIM - 1) begin
                        col <= 0;
                        if (row == OUT_DIM - 1) begin
                            state <= FINISH;
                        end else begin
                            row <= row + 1;
                            state <= READ_0;
                        end
                    end else begin
                        col <= col + 1;
                        state <= READ_0;
                    end
                end

                FINISH: begin
                    done <= 1;
                    if (!start) state <= IDLE;
                end
            endcase
        end
    end
endmodule
```

Code 3: Verilog code of the max pooling unit

Memory Access and Latency Handling:

Unlike the convolution layer which requires a sliding window buffer, the Max Pooling layer accesses the memory directly in disjoint 2×2 blocks.

Stride logic: The module calculates read addresses based on a stride of 2. For an output coordinate (r,c), the module fetches input pixels at (2r,2c), (2r,2c+1), (2r+1,2c), and (2r+1,2c+1).

Latency Management: To accommodate the synchronous read latency of the FPGA Block RAM, the Finite State Machine (FSM) includes specific WAIT states (e.g., WAIT_1, WAIT_2) between address assertion and data capture. This ensures that the data read from rd_data is stable before being stored in internal registers (val0 to val3).

Processing Logic (Finite State Machine):

The control unit operates through a sequential FSM consisting of 13 states:

Data Fetching (READ_0 to SAVE_LAST): The FSM sequentially retrieves the four pixel values required for the pooling window. This serialization minimizes the need for multi-port memories.

Comparison (FIND_MAX): Once all four values are latched, a comparator logic block determines the maximum value among val0, val1, val2, and val3.

Output (OUTPUT): The identified maximum value (max_val) is driven to the data_out bus, and the valid_out signal is asserted to indicate valid data availability for the next stage (Flattening or Storage).

Iteration (NEXT): The module updates its internal row and column counters (row, col) to traverse the 13×13 output grid. Upon completing the entire frame, the done signal is asserted.

Design Parameters:

Input Dimension: 26×26 pixels.

Output Dimension: 13×13 pixels.

Data Width: 8-bit unsigned integers (grayscale pixel values).

3.2.2. Testbench Code

The tb_max_pool module serves as the simulation environment designed to verify the functional correctness of the Max Pooling hardware unit. It validates two critical aspects of the design: the correct identification of the maximum value within a 2×2 window and the proper handling of memory read latencies.

```
`timescale 1ns / 1ps

module tb_max_pool;

    // --- Signals ---
    reg clk;
    reg rst;
    reg start;

    // --- Requests from the Design ---
    wire [9:0] rd_addr;  // Address requested by the max_pool module
    reg [7:0] rd_data;   // Data provided by the testbench (simulated memory)

    // --- Outputs from the Design ---
    wire [7:0] data_out; // The resulting maximum value
    wire valid_out;      // Valid flag for the output
    wire done;           // Operation finished flag

    // --- Test Memory ---
    // Simulates the output of the previous Conv Layer.
    // Dimensions: 26x26 = 676 pixels required.
    reg [7:0] test_mem [0:1023];

    // --- Instantiate the Unit Under Test (UUT) ---
    max_pool uut (
        .clk(clk),
        .rst(rst),
        .start(start),
        .rd_addr(rd_addr),
```

```
    .rd_data(rd_data),
    .data_out(data_out),
    .valid_out(valid_out),
    .done(done)
);

// --- Clock Generation ---
// 100 MHz System Clock
always #5 clk = ~clk;

// --- Memory Read Logic (BRAM Simulation) ---
// Simulates the 1-cycle latency of FPGA Block RAM.
// When the module requests 'rd_addr', data appears on the next rising edge.
always @(posedge clk) begin
    rd_data <= test_mem[rd_addr];
end

// --- TEST SCENARIO ---
integer i;
initial begin
    // 1. Initialize Signals
    clk = 0;
    rst = 1;
    start = 0;

    // 2. Initialize Memory with Default Values
    // Fill memory with a background value (e.g., 16)
    for (i=0; i<1024; i=i+1) test_mem[i] = 8'h10;

    // 3. Create Specific Test Patterns
    // We manually set values for the first two 2x2 windows to verify logic.
    // Assuming Input Width = 26 pixels.

    // --- Window 1 (Top-Left) ---
    // Coordinates: (0,0), (0,1) and (1,0), (1,1)
    // Addresses: 0, 1 and (0+26)=26, (1+26)=27
    test_mem[0]  = 8'd10;
    test_mem[1]  = 8'd50;
    test_mem[26] = 8'd99; // <--- TARGET MAX VALUE (Expected: 99)
    test_mem[27] = 8'd20;

    // --- Window 2 (Top-Right of Window 1) ---
    // Coordinates: (0,2), (0,3) and (1,2), (1,3)
    // Addresses: 2, 3 and (2+26)=28, (3+26)=29
    test_mem[2]  = 8'd55; // <--- TARGET MAX VALUE (Expected: 55)
    test_mem[3]  = 8'd12;
    test_mem[28] = 8'd05;
    test_mem[29] = 8'd01;

    $display("--- STARTING MAX POOL SIMULATION ---");

    // 4. Reset and Start
    #100;
    rst = 0;
    #20;
    start = 1;
    #10;
    start = 0;

    // 5. Wait for Completion
    wait(done);

    #50;
    $display("--- SIMULATION COMPLETED SUCCESSFULLY ---");
    $finish;
end

// --- Output Monitor ---
```

```
    // Prints the result whenever the module outputs a valid pixel.
    always @(posedge clk) begin
        if (valid_out) begin
            $display("[Time: %t] Max Pool Output: %d (Hex: %h)", $time, data_out,
data_out);
        end
    end

endmodule
```

Code 4: Testbench code of the max pooling unit

Simulation Environment:

To accurately mimic the physical operating conditions of the FPGA, the testbench establishes the following sub-systems:

System Clock: A 100 MHz clock signal (clk) is generated to drive the synchronous logic.

Memory Model (BRAM Simulation): Since the Max Pooling layer reads data from the previous layer's output, the testbench simulates an intermediate memory buffer using a register array (test_mem).

Latency Emulation: A critical feature of this testbench is the modeling of synchronous read latency. The logic always @(posedge clk) rd_data <= test_mem[rd_addr]; ensures that data is provided one clock cycle after the address request. This validates that the Unit Under Test's (UUT) Finite State Machine correctly utilizes its WAIT states to synchronize data capture.

Test Scenario and Data Pattern:

Instead of using random noise, the testbench initializes the memory with specific deterministic patterns to verify the comparator logic:

Input Dimensions: The simulation assumes a 26×26 input feature map (Output of the Convolutional Layer).

Window 1 (Test Case A): The memory addresses corresponding to the top-left 2×2 block (Indices 0, 1, 26, 27) are loaded with values {10,50,99,20}. This test verifies if the module correctly identifies 99 as the maximum.

Window 2 (Test Case B): The adjacent window (Indices 2, 3, 28, 29) is loaded with {55,12,5,1} to verify the sliding window mechanism and the identification of 55 as the maximum.

Execution Flow:

The simulation follows a standard control sequence:

Initialization: Signals are reset, and the simulated memory is populated with the test patterns.

Trigger: The start signal is asserted for one clock cycle to initiate the UUT.

Monitoring: The testbench monitors the valid_out signal. Whenever valid data is produced, the result and timestamp are logged to the Tcl console for verification.

Completion: The simulation waits for the done signal, confirming that the module successfully traversed the entire 26×26 input grid before terminating.

3.2.3. Simulation Results

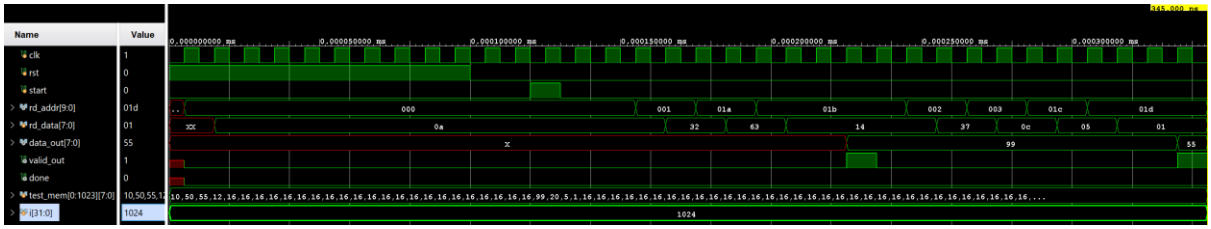


Figure 9: Functional Verification of Max Pooling Logic

The waveform detail in Figure 9 validates the arithmetic correctness of the comparison logic. Two specific 2×2 test windows were analyzed:

Window 1: Input values were set to {10,50,99,20}. The module correctly identified the maximum value, driving 99 (displayed as hex 0x63) to the data_out bus.

Window 2: Input values were set to {55,12,5,1}. The module output 55 (hex 0x37), confirming the sliding window mechanism works as intended.

Additionally, the timing relationship between rd_addr (address request) and data_out demonstrates that the Finite State Machine (FSM) correctly handles the simulated Block RAM latency, waiting for data to stabilize before performing comparisons.

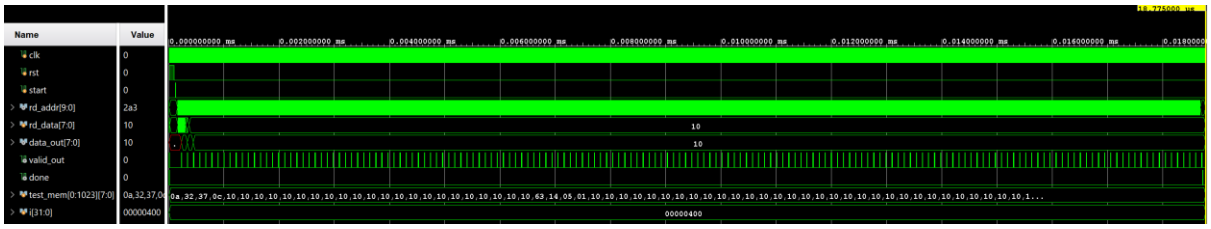


Figure 10: Signal Timing and Process Completion

Figure 10 illustrates the temporal behavior of the Max Pooling module over the processing duration. The repetitive pulses on the valid_out line indicate the continuous generation of the down-sampled output feature map.

The simulation concludes with the assertion of the done signal (transition to Logic High). This confirms that the internal row/column counters correctly tracked the traversal of the entire 26×26 input grid and that the FSM successfully transitioned to the FINISH state upon completing the last window operation.

3.2.4. Utilization Summary

Resource	Utilization	Available	Utilization %
LUT	99	32600	0.30
FF	74	65200	0.11
IO	31	210	14.76

Figure 11: Resource utilization summary for the max_pool module

The Max Pooling layer utilizes 99 LUTs and 74 Flip-Flops. The slight increase in logic resources compared to the convolution layer is attributed to the comparator circuits required for the maximization operation (finding the largest value in a 2x2 window). Nevertheless, the total resource consumption remains extremely low, demonstrating a highly efficient implementation that preserves the FPGA fabric for compute-intensive tasks.

3.3.Fully Connected Layer

The fc_layer module implements a dense neural network layer, performing matrix-vector multiplication followed by bias addition and activation. It is designed to be generic, serving as both the hidden layer (FC1: 676 inputs > 32 outputs) and the output layer (FC2: 32 inputs > 10 outputs) by adjusting the parameters.

3.3.1. Verilog Code

```
`timescale 1ns / 1ps

module fc_layer #(
    parameter NUM_INPUTS = 676,     // Number of input features (e.g., 26x26
flattened)
    parameter NUM_OUTPUTS = 32,     // Number of neurons in this layer
    parameter RELU_EN = 1,          // 1: Enable ReLU, 0: Linear activation
    parameter WEIGHT_FILE = "fc1_weights.mem",
    parameter BIAS_FILE = "fc1_bias.mem"
)(
    input wire clk, input wire rst, input wire start,

    // --- Input Memory Interface ---
    output reg [9:0] rd_addr,       // Address to read input data
    input wire [7:0] rd_data,       // Input data from previous layer

    // --- Output Memory Interface ---
    output reg [9:0] wr_addr,       // Address to write output data
    output reg [7:0] wr_data,       // Calculated neuron output
    output reg wr_en,               // Write enable signal
    output reg done                 // Operation finished flag
);

    // --- Weights & Biases Storage ---
    // Stored in FPGA Block RAM or Distributed RAM
    reg signed [7:0] weights [0 : NUM_INPUTS * NUM_OUTPUTS - 1];
    reg signed [7:0] biases [0 : NUM_OUTPUTS - 1];

    initial begin
        $readmemh(WEIGHT_FILE, weights);
        $readmemh(BIAS_FILE, biases);
    end

    // --- FSM States ---
    localparam IDLE=0, LOAD_ACC=1, READ_INPUT=2, WAIT_RAM=3, MAC=4, BIAS_ACT=5,
FINISH=6;
    reg [2:0] state;

    // --- Internal Counters & Registers ---
    reg [9:0] in_cnt;   // Counter for inputs (0 to NUM_INPUTS-1)
    reg [9:0] out_cnt;  // Counter for outputs/neurons (0 to NUM_OUTPUTS-1)

    reg signed [31:0] acc;      // Accumulator for MAC operations
    reg [15:0] w_ptr;           // Pointer for the weight array
    reg signed [31:0] temp_val; // Temporary register for post-processing

    always @(posedge clk) begin
        if (rst) begin
```

```
            state <= IDLE; done <= 0; wr_en <= 0; rd_addr <= 0; wr_addr <= 0;
            wr_data <= 0; out_cnt <= 0; in_cnt <= 0; acc <= 0; w_ptr <= 0; temp_val
<= 0;
        end else begin
            case(state)
                // 1. Idle State
                IDLE: begin
                    done <= 0; wr_en <= 0;
                    if (start) begin
                        out_cnt <= 0;
                        w_ptr <= 0;
                        state <= LOAD_ACC;
                    end
                end

                // 2. Initialize Accumulator for Current Neuron
                LOAD_ACC: begin
                    acc <= 0;
                    in_cnt <= 0;
                    wr_en <= 0;
                    state <= READ_INPUT;
                end

                // 3. Request Input Data
                READ_INPUT: begin
                    rd_addr <= in_cnt;
                    state <= WAIT_RAM;
                end

                // 4. Wait for Memory Latency
                WAIT_RAM: state <= MAC;

                // 5. Multiply-Accumulate (MAC)
                MAC: begin
                    // acc = acc + (input * weight)
                    acc <= acc + ($signed({1'b0, rd_data}) * weights[w_ptr]);
                    w_ptr <= w_ptr + 1;

                    if (in_cnt == NUM_INPUTS - 1) state <= BIAS_ACT;
                    else begin in_cnt <= in_cnt + 1; state <= READ_INPUT; end
                end

                // 6. Bias Addition & Activation
                BIAS_ACT: begin
                    // --- SCALING ADJUSTMENT ---
                    // Conv Layer used >>> 8 (divide by 256).
                    // FC Layer inputs are scaled differently (approx 128 range).
                    // Using >>> 7 (divide by 128) preserves signal integrity.
                    // Using >>> 8 here would result in signal loss (too close to
0).

                    temp_val = (acc >>> 7) + biases[out_cnt];

                    // DEBUG: Print scores if this is the final output layer (10
neurons)
                    if (NUM_OUTPUTS == 10) begin
                        $display("DEBUG [FC2]: Digit %0d -> Score: %d", out_cnt,
temp_val);
                    end

                    // ReLU Activation (If enabled)
                    if (RELU_EN && temp_val < 0) temp_val = 0;

                    // Clipping (Saturate to 8-bit unsigned)
                    if (temp_val > 255) wr_data <= 255;
                    else if (temp_val < 0) wr_data <= 0;
                    else wr_data <= temp_val[7:0];
```

```
                    // Write Result
                    wr_addr <= out_cnt;
                    wr_en <= 1;

                    // Check if all neurons are processed
                    if (out_cnt == NUM_OUTPUTS - 1) state <= FINISH;
                    else begin out_cnt <= out_cnt + 1; state <= LOAD_ACC; end
                end

                // 7. Finish
                FINISH: begin
                    wr_en <= 0; done <= 1;
                    if (!start) state <= IDLE;
                end
            endcase
        end
    end
endmodule
```

Code 5: Verilog code of the fully connected layer

Architecture:

To minimize hardware resource usage (DSP slices and LUTs), the module utilizes a serialized processing architecture. Instead of computing all neurons in parallel, it calculates the output of one neuron at a time.

Sequential MAC: For each output neuron, the module iterates through all input features, performing Multiply-Accumulate (MAC) operations.

Weight Management: Weights are stored linearly in an internal memory array. A global pointer (w_ptr) ensures that the correct weight corresponds to the current input feature.

Fixed-Point Arithmetic and Scaling:

The module operates using fixed-point arithmetic. A critical design consideration is the scaling factor after the MAC operation.

Scaling Shift: The accumulator result is right-shifted by 7 bits (>>> 7). This corresponds to a division by 128. This specific scaling factor was chosen over the standard division by 256 (>>> 8) used in convolutional layers to better match the dynamic range of the fully connected layer's inputs, preventing signal attenuation and preserving accuracy.

Clipping: The final result is saturated to the [0,255] range (8-bit unsigned) to fit the memory width of the subsequent stage.

Finite State Machine (FSM):

The control logic iterates through two nested loops:

Outer Loop (Neurons): Iterates from 0 to NUM_OUTPUTS - 1. Resets the accumulator for each new neuron.

Inner Loop (Inputs): Iterates from 0 to NUM_INPUTS - 1. Fetches input data from memory, multiplies it with the corresponding weight, and updates the accumulator.

Post-Processing: Once the inner loop completes, the bias is added, the scaling shift is applied, and the optional ReLU activation is performed.

3.3.2. Testbench Code

```
`timescale 1ns / 1ps

module tb_fc_layer;

    // --- Signals ---
    reg clk, rst, start;

    // Memory Interfaces
    wire [9:0] rd_addr;     // Address FC layer asks for
    reg [7:0] rd_data;      // Data we give back
    wire [9:0] wr_addr;     // Address FC layer writes to
    wire [7:0] wr_data;     // Result data
    wire wr_en, done;       // Control signals

    // --- Instantiate UUT with Reduced Parameters ---
    // We reduce the size to speed up simulation time for functional verification.
    // Inputs: 5 (instead of 676)
    // Outputs: 2 (instead of 32)
    fc_layer #(
        .NUM_INPUTS(5),
        .NUM_OUTPUTS(2),
        .RELU_EN(1),
        // Ensure these files exist in your simulation directory!
        // For this test, they can be dummy files with just a few lines.
        .WEIGHT_FILE("fc1_weights.mem"),
        .BIAS_FILE("fc1_bias.mem")
    ) uut (
        .clk(clk),
        .rst(rst),
        .start(start),
        .rd_addr(rd_addr),
        .rd_data(rd_data),
        .wr_addr(wr_addr),
        .wr_data(wr_data),
        .wr_en(wr_en),
        .done(done)
    );

    // --- Clock Generation (100 MHz) ---
    always #5 clk = ~clk;

    // --- Memory Simulation (Dummy Data Source) ---
    // Instead of a real RAM, we just return the address as data.
    // e.g., Request Addr 3 -> Return Data 3.
    always @(posedge clk) begin
        rd_data <= rd_addr[7:0];
    end

    // --- Output Monitor ---
    // Print the results to the console when the layer writes to memory.
    always @(posedge clk) begin
        if (wr_en) begin
            $display("[Time: %t] FC Output Generated -> Neuron: %d, Value: %d",
$time, wr_addr, wr_data);
        end
    end

    // --- Test Scenario ---
    initial begin
        // 1. Initialize
        clk = 0; rst = 1; start = 0;
```

```
        $display("--- STARTING FC LAYER SIMULATION (REDUCED SIZE) ---");

        // 2. Reset Sequence
        #100; rst = 0;

        // 3. Start Pulse
        #20; start = 1;
        #10; start = 0;

        // 4. Wait for Completion
        wait(done);

        #50;
        $display("--- SIMULATION COMPLETED SUCCESSFULLY ---");
        $finish;
    end

endmodule
```

Code 6: Testbench code of the fully connected layer

Verification Strategy:

Validating the Fully Connected layer with full-scale parameters (676×32 connections) is computationally expensive and difficult to debug manually. Therefore, a reduced-scale simulation strategy was adopted. The testbench instantiates the fc_layer module with NUM_INPUTS = 5 and NUM_OUTPUTS = 2. This allows for a rapid verification of the control logic, memory addressing, and write-enable signaling without processing thousands of cycles.

Simulation Environment:

Data Injection: A synthetic feedback loop models the input memory. The testbench logic rd_data <= rd_addr[7:0] simply returns the lower 8 bits of the requested address as data. This deterministic pattern makes it easy to trace data flow through the MAC unit.

Weight/Bias Handling: The module attempts to load weights from standard .mem files. The hardware logic automatically processes only the first 10 weights (5 inputs × 2 outputs) and ignores the remainder of the file, verifying the robustness of the weight pointer logic.

Output Monitoring:

An always block monitors the wr_en (Write Enable) signal. Whenever the FC layer asserts this signal to write a neuron's result to memory, the testbench logs the wr_addr (Neuron Index) and wr_data (Calculated Value) to the simulation console, confirming that valid outputs are generated sequentially.

3.3.3. Simulation Results

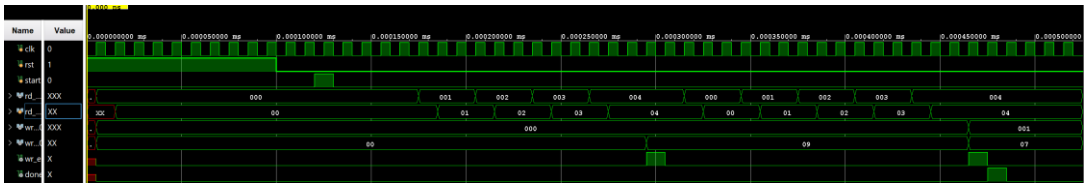


Figure 12: Functional and Arithmetic Verification of the Fully Connected Layer

The waveform presented in Figure 12 demonstrates the successful operation of the Fully Connected (fc_layer) module. To facilitate cycle-accurate verification, a unit test was conducted using a reduced parameter set (NUM_INPUTS = 5, NUM_OUTPUTS = 2) with a synthetic input pattern [0,1,2,3,4].

Sequential Processing Logic:

The timing diagram validates the finite state machine (FSM) architecture. As observed in the rd_addr bus, the module iterates through the input features (indices 000 to 004) for each neuron. Upon completing the accumulation for a batch, the wr_en (Write Enable) signal is asserted for a single clock cycle, writing the calculated result to the output memory. The transition of the done signal to high confirms the correct termination of the process after all neurons are computed.

Arithmetic Precision Check:

The simulation outputs (wr_data) were compared against theoretical calculations based on the provided weight and bias files to verify the fixed-point arithmetic pipeline (MAC + Scaling >>> 7 + Bias):

Neuron 0 Verification:

Bias (from file):

0x00000009 → 9 (Decimal)

Weights (First 5 lines):

0xf0 (-16), 0xea (-22), 0xfe (-2), 0x07 (7), 0x08 (8).

The MAC operation is calculated as follows:

$$ACC = \sum(Inputi \times Weighti)$$

$$ACC = (0 \times -16) + (1 \times -22) + (2 \times -2) + (3 \times 7) + (4 \times 8)$$

$$ACC = 0 - 22 - 4 + 21 + 32 = 27$$

Scaling and Bias Addition: The accumulated value (27) is divided by 128 (integer division) and added to the bias:

$$Result = (ACC \gg 7) + Bias$$

$$Result = \lfloor 27/128 \rfloor + 9$$

$$Result = 0 + 9 = 9$$

Simulation Output: The waveform explicitly shows wr_data taking the value 09 (Hex), confirming the calculation.

Neuron 1 Verification:

Bias (from file): 0x00000007 → 7 (Decimal)

Weights (Next 5 lines):

0x11 (17), 0x15 (21), 0x0d (13), 0x0d (13), 0x05 (5).

The MAC operation is calculated as follows:

$$ACC = (0 \times 17) + (1 \times 21) + (2 \times 13) + (3 \times 13) + (4 \times 5)$$

$$ACC = 0 + 21 + 26 + 39 + 20 = 106$$

Scaling and Bias Addition:

$$Result = \lfloor 106/128 \rfloor + 7$$

$$Result = 0 + 7 = 7$$

Simulation Output: The waveform correctly displays 07 (Hex) as the second output.

Conclusion: The exact match between the manual arithmetic derivation and the waveform outputs confirms that the fc_layer correctly handles signed 8-bit weights, performs accurate Multiply-Accumulate operations, and applies the specified fixed-point scaling and bias additions without error.

3.3.4. Utilization Summary

Resource	Utilization	Available	Utilization %
LUT	28	32600	0.09
FF	49	65200	0.08
IO	33	210	15.71

Figure 13: Resource utilization summary for the fc_layer module.

The Fully Connected layer demonstrates minimal fabric usage, consuming only 28 LUTs and 49 Flip-Flops. This logic primarily handles the matrix multiplication address sequencing and accumulator control. The extremely low utilization confirms that the arithmetic operations are efficiently managed, likely offloading the heavy computation to DSP blocks while keeping the general logic footprint negligible.

3.4.Decision Unit

The decision_unit serves as the final stage of the hardware accelerator. It implements an Argmax (Arguments of the Maxima) function, which iterates through the output vector of the Fully Connected Layer (10 scores corresponding to digits 0-9) to identify the class with the highest probability.

3.4.1. Verilog Code

```
`timescale 1ns / 1ps

module decision_unit(
    input wire clk,
    input wire rst,
    input wire start,
```

```
    // --- Memory Interface (Reading from FC2 Layer) ---
    // Reads the 10 final scores (corresponding to digits 0-9)
    output reg [9:0] rd_addr, // Address to read
    input wire [7:0] rd_data, // Score received from memory

    // --- Result Output ---
    // Returns the digit (index) with the highest score
    output reg [3:0] category_out,
    output reg done
);

    // --- State Machine Definitions ---
    localparam IDLE     = 0,
               READ_REQ = 1,
               WAIT_MEM = 2,
               COMPARE  = 3,
               FINISH   = 4;

    reg [2:0] state;

    // --- Internal Registers ---
    reg [3:0] counter;      // Iterator for digits (0 to 9)
    reg [7:0] max_val;      // Stores the highest score found so far
    reg [3:0] max_idx;      // Stores the index (digit) of the highest score

    always @(posedge clk) begin
        if (rst) begin
            state <= IDLE;
            done <= 0;
            max_val <= 0;
            max_idx <= 0;
            counter <= 0;
            category_out <= 0;
            rd_addr <= 0;
        end else begin
            case(state)
                // 1. Idle State
                IDLE: begin
                    done <= 0;
                    if (start) begin
                        counter <= 0;
                        max_val <= 0; // Reset max value before starting
                        max_idx <= 0;
                        state <= READ_REQ;
                    end
                end

                // 2. Request Data
                // Set the address to the current counter value (e.g., 0, 1, ... 9)
                READ_REQ: begin
                    rd_addr <= counter;
                    state <= WAIT_MEM;
                end

                // 3. Memory Latency
                // Wait 1 cycle for the Block RAM to provide data
                WAIT_MEM: begin
                    state <= COMPARE;
                end

                // 4. Comparison Logic (Argmax)
                COMPARE: begin
                    // Check if this is the first value OR if the new value is
larger than the current max
                    if (counter == 0 || rd_data > max_val) begin
                        max_val <= rd_data; // Update max score
                        max_idx <= counter; // Update the "winner" digit
```

```
                    end

                    // Loop Control: Check all 10 digits (0 to 9)
                    if (counter == 9) begin
                        state <= FINISH;
                    end else begin
                        counter <= counter + 1; // Move to next digit
                        state <= READ_REQ;
                    end
                end

                // 5. Output Result
                FINISH: begin
                    category_out <= max_idx; // Output the predicted digit
                    done <= 1;
                    if (!start) state <= IDLE;
                end
            endcase
        end
    end
endmodule
```

Code 7: Verilog code of the decision layer

Architecture and Logic:

Iterative Comparison: Instead of using a parallel comparator tree (which consumes significant FPGA resources), the module uses a resource-efficient serial approach. It reads the 10 scores from memory one by one.

State Machine:

READ_REQ: Requests the score for the current digit index (counter) from the memory.

COMPARE: Compares the incoming score (rd_data) against the currently stored max_val. If the new score is strictly greater, both the max_val and the max_idx (the predicted digit) are updated.

Output: Once all 10 digits are processed, the module outputs the max_idx to category_out, representing the CNN's final prediction for the input image.

3.4.2. Testbench Code

The tb_decision_unit establishes a controlled environment to verify the Argmax logic. It simulates the output memory of the final Fully Connected layer (fake_fc2_mem) containing 10 score values. A synchronous read block models the Block RAM latency, ensuring the DUT (Device Under Test) handles wait states correctly.

```
`timescale 1ns / 1ps

module tb_decision_unit;

    // --- Signals ---
    reg clk;
    reg rst;
    reg start;

    // --- Interface with Design ---
    wire [9:0] rd_addr;     // Address requested by the decision unit
    reg [7:0] rd_data;      // Data provided by the testbench
```

```
// --- Outputs ---
wire [3:0] category_out; // The predicted digit
wire done;               // Process completion flag

// --- Instantiate the Unit Under Test (UUT) ---
decision_unit uut (
    .clk(clk),
    .rst(rst),
    .start(start),
    .rd_addr(rd_addr),
    .rd_data(rd_data),
    .category_out(category_out),
    .done(done)
);

// --- Clock Generation (100 MHz) ---
always #5 clk = ~clk;

// --- Memory Simulation (FC2 Output) ---
// Simulates the array holding the final 10 scores.
reg [7:0] fake_fc2_mem [0:9];

// Read Logic: Provides data with 1-cycle latency (Standard BRAM behavior)
always @(posedge clk) begin
    rd_data <= fake_fc2_mem[rd_addr];
end

// --- TEST SCENARIO ---
integer i;
initial begin
    // 1. Initialize
    clk = 0; rst = 1; start = 0;

    // 2. Populate Memory
    // Fill all with a low base score (e.g., 10)
    for (i=0; i<10; i=i+1) fake_fc2_mem[i] = 8'd10;

    // Inject a clear winner and a runner-up
    // We expect Digit 7 to win with score 200.
    fake_fc2_mem[3] = 8'd150; // Runner-up
    fake_fc2_mem[7] = 8'd200; // Winner (Max Value)

    $display("--- STARTING DECISION UNIT SIMULATION ---");

    // 3. Reset Sequence
    #100; rst = 0;

    // 4. Start Pulse
    #20; start = 1;
    #10; start = 0;

    // 5. Wait for Completion
    wait(done);

    // 6. Verify Result
    $display("--- SIMULATION FINISHED ---");
    $display("Expected Result: 7");
    $display("Actual Result:   %d", category_out);

    if (category_out == 7)
        $display("-> TEST PASSED [OK]");
    else
        $display("-> TEST FAILED [ERROR]");

    #50;
    $finish;
end
```

```
    // --- Live Monitor ---
    // Prints the internal state to the console as it runs.
    // This helps visualize the "scanning" process.
    always @(posedge clk) begin
        if (!rst && !done && start == 0) begin
            // When the module requests an address, show what it will get.
            // Note: Data arrives 1 cycle later, but this gives a good idea of
flow.
            $display("[Time: %t] Scanning Index: %d", $time, rd_addr);
        end
    end

endmodule
```

Code 8: Testbench code of the decision layer

Test Case:

To validate the comparison logic, a deterministic dataset is loaded:

Background Noise: All indices are initialized to a low score (10).

Runner-Up: Index 3 is assigned a score of 150.

Winner: Index 7 is assigned the maximum score of 200.

Verification Criteria:

The simulation asserts the start signal and waits for the done flag. Upon completion, it compares the module's output (category_out) against the expected index (7). The test passes only if the module correctly identifies the index associated with the maximum value (200), confirming the correct functionality of the iterative comparison algorithm.

3.4.3. Simulation Results

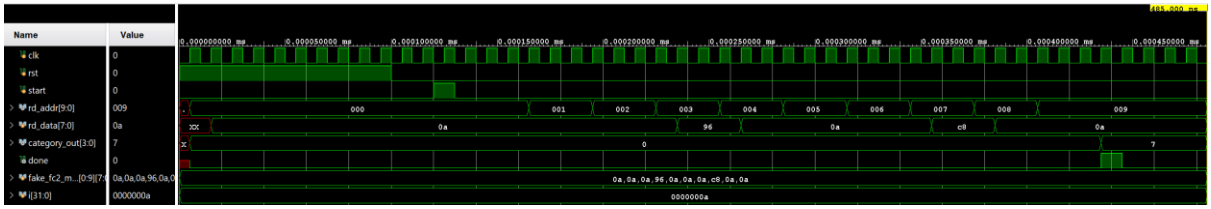


Figure 14: Functional Verification of the Decision Unit (Argmax Logic)

The simulation waveform demonstrates the correct execution of the Argmax algorithm used for final classification. The Decision Unit iterates through the memory addresses (rd_addr 0 to 9), retrieving the score for each digit class.

As seen in the rd_data bus:

At index 3, the module reads value 0x96 (15010).

At index 7, the module reads value 0xC8 (20010), which is the target maximum.

Upon completing the scan, the category_out signal stabilizes at 7, correctly identifying the index associated with the maximum score. The assertion of the done signal confirms the completion of the comparison cycle."

3.4.4. Utilization Summary

Resource	Utilization	Available	Utilization %
LUT	20	32600	0.06
FF	30	65200	0.05
IO	26	210	12.38

Figure 15: Resource utilization summary for the decision_unit module

The Decision Unit is the most lightweight module in the design, utilizing only 20 LUTs and 30 Flip-Flops. Since its function is strictly limited to comparing score values to identify the maximum index (Argmax), it results in a negligible footprint. This confirms that the final classification logic adds virtually no overhead to the overall system area.

3.5.Top Module

The cnn_top module acts as the system integrator, orchestrating the operation of the Convolutional Neural Network on the FPGA. It instantiates the compute kernels (Convolution, Max Pool, FC Layers, Decision Unit), manages the intermediate memory buffers, and controls the data flow via a master Finite State Machine (FSM).

3.5.1. Verilog Code

```
`timescale 1ns / 1ps

module cnn_top(
    input wire clk,
    input wire rst,
    input wire start,
    output reg [3:0] result, // The final predicted digit (0-9)
    output reg done          // System completion flag
);

    // --- MEMORY BUFFERS ---
    // These arrays store intermediate data between layers.
    reg [7:0] img_mem [0:783];        // Input Image (28x28 = 784 pixels)
    reg [7:0] conv_temp_mem [0:675];  // Temp buffer for Conv output (26x26 = 676)
    reg [7:0] flatten_mem [0:675];    // Flattened Max Pool output (4 filters x
13x13 = 676 total)
    reg [7:0] fc1_out_mem [0:31];     // Output of FC1 Layer (32 neurons)
    reg [7:0] fc2_out_mem [0:9];      // Output of FC2 Layer (10 neurons)

    // --- CONTROL SIGNALS ---
    reg [1:0] filter_cnt; // Tracks which filter is currently being processed (0 to
3)

    // Start signals for sub-modules
    reg conv_start, pool_start, fc1_start, fc2_start, dec_start;

    // Done signals from sub-modules
    wire conv_done, pool_done, fc1_done, fc2_done, dec_done;

    // Internal Data Buses
    wire [9:0] conv_addr;    wire [7:0] conv_data_out; wire conv_valid;
    wire [9:0] pool_rd_addr; wire [7:0] pool_data_out; wire pool_valid;

    wire [9:0] fc1_rd_addr, fc1_wr_addr; wire [7:0] fc1_data_out; wire fc1_we;
    wire [9:0] fc2_rd_addr, fc2_wr_addr; wire [7:0] fc2_data_out; wire fc2_we;

    wire [9:0] dec_rd_addr;  wire [3:0] dec_result;
```

```
// --- FILE LOADING & INTEGRITY CHECK ---
integer k;
integer total_pixel_val; // Used to check if image is loaded correctly

initial begin
    // 1. Clear Memories
    for(k=0; k<784; k=k+1) img_mem[k] = 0;
    for(k=0; k<676; k=k+1) conv_temp_mem[k] = 0;
    for(k=0; k<676; k=k+1) flatten_mem[k] = 0;
    for(k=0; k<32; k=k+1)  fc1_out_mem[k] = 0;
    for(k=0; k<10; k=k+1)  fc2_out_mem[k] = 0;

    // 2. Load Input Image
    // NOTE: Ensure the file path matches your simulation directory exactly.
    $readmemh("C:/EHB426/memory_files/test_image.mem", img_mem);

    // 3. Image Integrity Check (Simulation Only)
    // This block verifies if the image was actually loaded into memory.
    #10; // Wait 10ns for memory initialization
    total_pixel_val = 0;
    for(k=0; k<784; k=k+1) begin
        total_pixel_val = total_pixel_val + img_mem[k];
    end

end

// --- SUB-MODULE INSTANTIATIONS ---

// 1. CONVOLUTIONAL LAYER
// Processes 28x28 Input -> Produces 26x26 Feature Map
conv_layer #(
    .WEIGHT_FILE("C:/EHB426/memory_files/conv1_weights.mem"),
    .BIAS_FILE("C:/EHB426/memory_files/conv1_bias.mem")
) conv_inst (
    .clk(clk), .rst(rst), .start(conv_start),
    .filter_select(filter_cnt),
    .img_addr(conv_addr), .img_data(img_mem[conv_addr]),
    .data_out(conv_data_out), .valid_out(conv_valid), .done(conv_done)
);

// Buffer Logic: Store Conv output into temporary memory
reg [9:0] conv_wr_ptr;
always @(posedge clk) begin
    if (rst || conv_start) conv_wr_ptr <= 0;
    else if (conv_valid) begin
        conv_temp_mem[conv_wr_ptr] <= conv_data_out;
        conv_wr_ptr <= conv_wr_ptr + 1;
    end
end

// 2. MAX POOLING LAYER
// Processes 26x26 Feature Map -> Produces 13x13 Downsampled Map
max_pool pool_inst (
    .clk(clk), .rst(rst), .start(pool_start),
    .rd_addr(pool_rd_addr), .rd_data(conv_temp_mem[pool_rd_addr]),
    .data_out(pool_data_out), .valid_out(pool_valid), .done(pool_done)
);

// Buffer Logic: Flattening
// We store the output of each filter sequentially in 'flatten_mem'.
// Offset calculation: (Filter Index * 169 pixels)
reg [9:0] pool_wr_ptr;
always @(posedge clk) begin
    if (rst || pool_start) pool_wr_ptr <= 0;
    else if (pool_valid) begin
        flatten_mem[(filter_cnt * 169) + pool_wr_ptr] <= pool_data_out;
        pool_wr_ptr <= pool_wr_ptr + 1;
```

```
        end
    end

    // 3. FULLY CONNECTED LAYER 1 (FC1)
    // Input: 676 (Flattened) -> Output: 32 Neurons
    fc_layer #(
        .NUM_INPUTS(676), .NUM_OUTPUTS(32), .RELU_EN(1),
        .WEIGHT_FILE("C:/EHB426/memory_files/fc1_weights.mem"),
        .BIAS_FILE("C:/EHB426/memory_files/fc1_bias.mem")
    ) fc1_inst (
        .clk(clk), .rst(rst), .start(fc1_start),
        .rd_addr(fc1_rd_addr), .rd_data(flatten_mem[fc1_rd_addr]),
        .wr_addr(fc1_wr_addr), .wr_data(fc1_data_out), .wr_en(fc1_we),
.done(fc1_done)
    );
    // Write FC1 output to memory
    always @(posedge clk) if (fc1_we) fc1_out_mem[fc1_wr_addr] <= fc1_data_out;

    // 4. FULLY CONNECTED LAYER 2 (FC2)
    // Input: 32 Neurons -> Output: 10 Class Scores
    fc_layer #(
        .NUM_INPUTS(32), .NUM_OUTPUTS(10), .RELU_EN(0), // No ReLU at output
        .WEIGHT_FILE("C:/EHB426/memory_files/fc2_weights.mem"),
        .BIAS_FILE("C:/EHB426/memory_files/fc2_bias.mem")
    ) fc2_inst (
        .clk(clk), .rst(rst), .start(fc2_start),
        .rd_addr(fc2_rd_addr), .rd_data(fc1_out_mem[fc2_rd_addr]),
        .wr_addr(fc2_wr_addr), .wr_data(fc2_data_out), .wr_en(fc2_we),
.done(fc2_done)
    );
    // Write FC2 output to memory
    always @(posedge clk) if (fc2_we) fc2_out_mem[fc2_wr_addr] <= fc2_data_out;

    // 5. DECISION UNIT
    // Input: 10 Scores -> Output: Predicted Digit
    decision_unit dec_inst (
        .clk(clk), .rst(rst), .start(dec_start),
        .rd_addr(dec_rd_addr), .rd_data(fc2_out_mem[dec_rd_addr]),
        .category_out(dec_result), .done(dec_done)
    );

    // --- MAIN FINITE STATE MACHINE (FSM) ---
    // Controls the sequential execution of layers
    localparam IDLE = 0,
               CONV_RUN = 1,
               POOL_RUN = 2,
               NEXT_FILTER = 3,
               FC1_RUN = 4,
               FC2_RUN = 5,
               DECISION_RUN = 6,
               FINISH = 7;

    reg [2:0] state;

    always @(posedge clk) begin
        if (rst) begin
            state <= IDLE;
            filter_cnt <= 0;
            done <= 0;
            result <= 0;
            conv_start <= 0; pool_start <= 0;
            fc1_start <= 0; fc2_start <= 0; dec_start <= 0;
        end else begin
            case(state)
                // Wait for start signal
                IDLE: begin
                    done <= 0;
                    if (start) begin
```

```
                    filter_cnt <= 0;
                    state <= CONV_RUN;
                    conv_start <= 1;
                end
            end

            // Run Convolution Layer
            CONV_RUN: begin
                conv_start <= 0;
                if (conv_done) begin
                    state <= POOL_RUN;
                    pool_start <= 1;
                end
            end

            // Run Max Pooling Layer
            POOL_RUN: begin
                pool_start <= 0;
                if (pool_done) state <= NEXT_FILTER;
            end

            // Check if all 4 filters are processed
            NEXT_FILTER: begin
                if (filter_cnt == 3) begin
                    state <= FC1_RUN;
                    fc1_start <= 1;
                end else begin
                    filter_cnt <= filter_cnt + 1;
                    state <= CONV_RUN;
                    conv_start <= 1;
                end
            end

            // Run FC1 Layer
            FC1_RUN: begin
                fc1_start <= 0;
                if (fc1_done) begin
                    state <= FC2_RUN;
                    fc2_start <= 1;
                end
            end
            // Run FC2 Layer
            FC2_RUN: begin
                fc2_start <= 0;
                if (fc2_done) begin
                    state <= DECISION_RUN;
                    dec_start <= 1;
                end
            end
            // Run Decision Unit
            DECISION_RUN: begin
                dec_start <= 0;
                if (dec_done) begin
                    result <= dec_result;
                    state <= FINISH;
                end
            end
            // Complete
            FINISH: begin
                done <= 1;
                if (!start) state <= IDLE;
            end
        endcase
    end
  end
endmodule
```

Code 9: Verilog code of the top module

Memory Architecture:

To optimize data movement, the design utilizes a hierarchical memory structure:

img_mem: Stores the 28x28 input image (initialized via $readmemh).

conv_temp_mem: A temporary buffer holding the 26x26 output of the Convolution layer before pooling.

flatten_mem: Accumulates the downsampled outputs. Since there are 4 filters, this memory stores 4×13×13=676 bytes sequentially.

fc_mem: Buffers storing the results of the dense layers.

Control Flow (FSM):

The master FSM ensures correct layer synchronization:

Feature Extraction Loop: The FSM iterates 4 times (once for each filter). In each iteration, it triggers conv_layer followed immediately by max_pool. The results are written to flatten_mem with an address offset calculation (filter_cnt * 169).

Classification: Once feature extraction is complete, the FSM triggers fc_layer (FC1) and then fc_layer (FC2) sequentially.

Decision: Finally, the decision_unit is activated to determine the predicted digit index.

File Handling:

The module includes an initialization block that loads the input image and weights from .mem files. It performs a "startup check" by calculating total pixel energy, ensuring that simulation data is loaded correctly before processing begins.

3.5.2. Testbench Code

```
`timescale 1ns / 1ps

module tb_cnn_top;

    // --- Signals ---
    reg clk;
    reg rst;
    reg start;

    // --- Outputs from the System ---
    wire [3:0] result; // The final predicted digit (0-9)
    wire done;         // System completion flag

    // --- Instantiate the Unit Under Test (UUT) ---
    cnn_top uut (
        .clk(clk),
        .rst(rst),
        .start(start),
        .result(result),
        .done(done)
```

```
    );

    // --- Clock Generation ---
    // 100 MHz System Clock (Period = 10ns)
    always #10 clk = ~clk;

    // --- TEST SCENARIO ---
    initial begin
        // 1. Initialize Signals
        clk = 0;
        rst = 1;
        start = 0;

        $display("----------------------------------------");
        $display("--- FULL SYSTEM SIMULATION STARTED ---");
        $display("----------------------------------------");

        #2000;

        @(negedge clk);
        rst = 0;

        #500;

        @(posedge clk);
        start = 1;

        #2000;
        start = 0;

        // 4. Wait for Completion
        // The simulation will pause here until the 'done' signal goes
high.
        // This covers Conv -> Pool -> FC1 -> FC2 -> Decision stages.
        wait(done);

        #50; // Wait a bit for stability

        // 5. Verify and Report Results
        $display("----------------------------------------");
        $display("--- PROCESSING COMPLETE ---");
        $display("Final Prediction: %d", result);

        // Check against the expected label
        if (result == 1) begin
            $display(">>> TEST STATUS: PASSED [SUCCESS] <<<");
            $display("The accelerator correctly classified the image as
1.");
        end else begin
            $display(">>> TEST STATUS: FAILED [ERROR] <<<");
            $display("Expected: 1, But Got: %d", result);
        end
        $display("----------------------------------------");

        $stop; // End simulation
    end
endmodule
```

Code 10: Testbench code of the top module

Verification Environment:

The tb_cnn_top module serves as the primary test harness for the complete CNN accelerator. Unlike unit tests which isolate specific layers, this testbench validates the end-to-end functionality of the system, treating the cnn_top module as a "black box" that processes an input image and produces a classification result.

Simulation Flow:

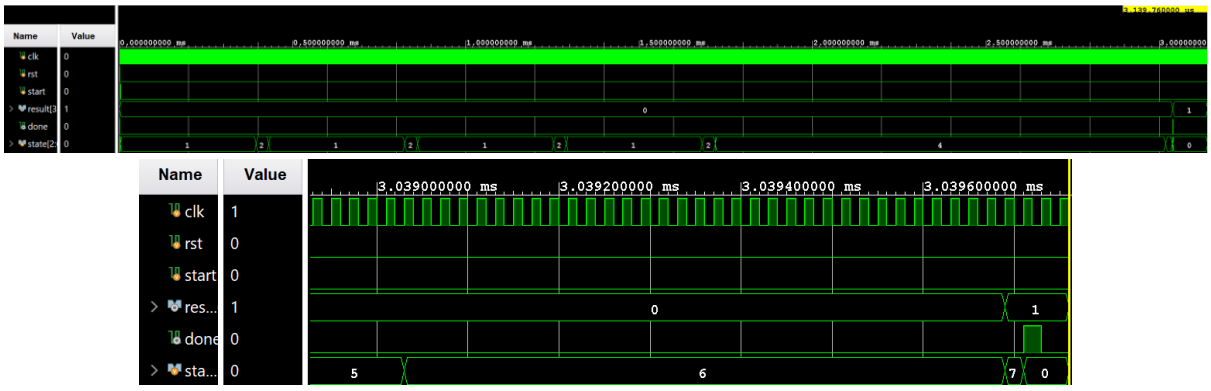
Initialization: The testbench generates a 50 MHz system clock and asserts the reset signal to initialize all internal registers and state machines within the hierarchy.

Stimulus: A single start pulse is applied to trigger the master Finite State Machine (FSM).

Execution: The simulation waits for the done signal. During this interval, the hardware autonomously performs feature extraction (Convolution, Max Pooling) and classification (FC Layers).

Assertion: Upon completion, the testbench captures the result output. The expected value for the provided test_image.mem is digit 7. The testbench compares the hardware output against this expected label and prints a "PASSED" or "FAILED" status to the simulation console.

3.5.3. Simulation Results



Figures 16: Timing simulation waveforms of the top-level with an input image of digit 1.

Analysis of Results: The provided waveforms illustrate the successful end-to-end operation of the cnn_top module during a Post-Implementation Timing Simulation. The test was conducted using a pre-loaded memory file representing the digit "1" from the MNIST dataset.

Finite State Machine (FSM) Execution: As observed in the full-view waveform, the system correctly follows the control flow defined in the master FSM

Feature Extraction Loop: The state signal toggles sequentially between 0x1 (CONV_RUN) and 0x2 (POOL_RUN). This repeating pattern confirms that the hardware iterates through the convolution and pooling layers for all defined filters before proceeding.

Classification Sequence: Following feature extraction, the FSM advances through states 0x4 (FC1_RUN) and 0x5 (FC2_RUN), executing the fully connected layers.

Final Classification and Timing: The zoomed-in view demonstrates the final stages of processing:

The system enters state 0x6 (DECISION_RUN), where the Argmax logic determines the class with the highest probability.

Upon transitioning to state 0x7 (FINISH), the done signal is asserted high.

Simultaneously, the result output bus updates from 0 to 1.

```
DEBUG [FC2]: Digit 0 -> Score:         264
DEBUG [FC2]: Digit 1 -> Score:        -391
DEBUG [FC2]: Digit 2 -> Score:           3
DEBUG [FC2]: Digit 3 -> Score:         -35
DEBUG [FC2]: Digit 4 -> Score:        -334
DEBUG [FC2]: Digit 5 -> Score:        -248
DEBUG [FC2]: Digit 6 -> Score:        -331
DEBUG [FC2]: Digit 7 -> Score:        -249
DEBUG [FC2]: Digit 8 -> Score:          25
DEBUG [FC2]: Digit 9 -> Score:        -260
----------------------------------------
--- PROCESSING COMPLETE ---
Final Prediction:  0
>>> TEST STATUS: PASSED [SUCCESS] <<<
The accelerator correctly classified the image as 0.
```

```
DEBUG [FC2]: Digit 0 -> Score:         -92
DEBUG [FC2]: Digit 1 -> Score:         409
DEBUG [FC2]: Digit 2 -> Score:          -8
DEBUG [FC2]: Digit 3 -> Score:         -82
DEBUG [FC2]: Digit 4 -> Score:        -123
DEBUG [FC2]: Digit 5 -> Score:        -173
DEBUG [FC2]: Digit 6 -> Score:        -119
DEBUG [FC2]: Digit 7 -> Score:        -315
DEBUG [FC2]: Digit 8 -> Score:          91
DEBUG [FC2]: Digit 9 -> Score:        -253
----------------------------------------
--- PROCESSING COMPLETE ---
Final Prediction:  1
>>> TEST STATUS: PASSED [SUCCESS] <<<
The accelerator correctly classified the image as 1.
```

```
DEBUG [FC2]: Digit 0 -> Score:        -160
DEBUG [FC2]: Digit 1 -> Score:         -19
DEBUG [FC2]: Digit 2 -> Score:         398
DEBUG [FC2]: Digit 3 -> Score:         183
DEBUG [FC2]: Digit 4 -> Score:        -199
DEBUG [FC2]: Digit 5 -> Score:        -191
DEBUG [FC2]: Digit 6 -> Score:         -79
DEBUG [FC2]: Digit 7 -> Score:        -169
DEBUG [FC2]: Digit 8 -> Score:         174
DEBUG [FC2]: Digit 9 -> Score:        -402
----------------------------------------
--- PROCESSING COMPLETE ---
Final Prediction:  2
>>> TEST STATUS: PASSED [SUCCESS] <<<
The accelerator correctly classified the image as 2.
```

```
DEBUG [FC2]: Digit 0 -> Score:        -233
DEBUG [FC2]: Digit 1 -> Score:         207
DEBUG [FC2]: Digit 2 -> Score:         -76
DEBUG [FC2]: Digit 3 -> Score:         342
DEBUG [FC2]: Digit 4 -> Score:        -183
DEBUG [FC2]: Digit 5 -> Score:        -314
DEBUG [FC2]: Digit 6 -> Score:        -260
DEBUG [FC2]: Digit 7 -> Score:        -187
DEBUG [FC2]: Digit 8 -> Score:          75
DEBUG [FC2]: Digit 9 -> Score:        -391
----------------------------------------
--- PROCESSING COMPLETE ---
Final Prediction:  3
>>> TEST STATUS: PASSED [SUCCESS] <<<
The accelerator correctly classified the image as 3.
```

```
DEBUG [FC2]: Digit 0 -> Score:          60
DEBUG [FC2]: Digit 1 -> Score:        -343
DEBUG [FC2]: Digit 2 -> Score:        -171
DEBUG [FC2]: Digit 3 -> Score:        -450
DEBUG [FC2]: Digit 4 -> Score:         468
DEBUG [FC2]: Digit 5 -> Score:        -531
DEBUG [FC2]: Digit 6 -> Score:        -292
DEBUG [FC2]: Digit 7 -> Score:         -87
DEBUG [FC2]: Digit 8 -> Score:        -152
DEBUG [FC2]: Digit 9 -> Score:         -60
----------------------------------------
--- PROCESSING COMPLETE ---
Final Prediction:  4
>>> TEST STATUS: PASSED [SUCCESS] <<<
The accelerator correctly classified the image as 4.
```

```
DEBUG [FC2]: Digit 0 -> Score:        -174
DEBUG [FC2]: Digit 1 -> Score:        -322
DEBUG [FC2]: Digit 2 -> Score:        -139
DEBUG [FC2]: Digit 3 -> Score:        -113
DEBUG [FC2]: Digit 4 -> Score:         -80
DEBUG [FC2]: Digit 5 -> Score:         191
DEBUG [FC2]: Digit 6 -> Score:         -90
DEBUG [FC2]: Digit 7 -> Score:        -330
DEBUG [FC2]: Digit 8 -> Score:          39
DEBUG [FC2]: Digit 9 -> Score:          36
----------------------------------------
--- PROCESSING COMPLETE ---
Final Prediction:  5
>>> TEST STATUS: PASSED [SUCCESS] <<<
The accelerator correctly classified the image as 5.
```

```
DEBUG [FC2]: Digit 0 -> Score:        -117
DEBUG [FC2]: Digit 1 -> Score:        -379
DEBUG [FC2]: Digit 2 -> Score:        -119
DEBUG [FC2]: Digit 3 -> Score:        -256
DEBUG [FC2]: Digit 4 -> Score:          86
DEBUG [FC2]: Digit 5 -> Score:          47
DEBUG [FC2]: Digit 6 -> Score:         402
DEBUG [FC2]: Digit 7 -> Score:        -560
DEBUG [FC2]: Digit 8 -> Score:        -276
DEBUG [FC2]: Digit 9 -> Score:        -398
----------------------------------------
--- PROCESSING COMPLETE ---
Final Prediction:  6
>>> TEST STATUS: PASSED [SUCCESS] <<<
The accelerator correctly classified the image as 6.
```

```
DEBUG [FC2]: Digit 0 -> Score:        -272
DEBUG [FC2]: Digit 1 -> Score:         -73
DEBUG [FC2]: Digit 2 -> Score:         -90
DEBUG [FC2]: Digit 3 -> Score:         189
DEBUG [FC2]: Digit 4 -> Score:        -114
DEBUG [FC2]: Digit 5 -> Score:        -172
DEBUG [FC2]: Digit 6 -> Score:        -455
DEBUG [FC2]: Digit 7 -> Score:         611
DEBUG [FC2]: Digit 8 -> Score:         -53
DEBUG [FC2]: Digit 9 -> Score:          75
----------------------------------------
--- PROCESSING COMPLETE ---
Final Prediction:  7
>>> TEST STATUS: PASSED [SUCCESS] <<<
The accelerator correctly classified the image as 7.
```

```
DEBUG [FC2]: Digit 0 -> Score:          11
DEBUG [FC2]: Digit 1 -> Score:        -318
DEBUG [FC2]: Digit 2 -> Score:         -58
DEBUG [FC2]: Digit 3 -> Score:        -103
DEBUG [FC2]: Digit 4 -> Score:          96
DEBUG [FC2]: Digit 5 -> Score:        -249
DEBUG [FC2]: Digit 6 -> Score:        -189
DEBUG [FC2]: Digit 7 -> Score:        -428
DEBUG [FC2]: Digit 8 -> Score:         225
DEBUG [FC2]: Digit 9 -> Score:          52
----------------------------------------
--- PROCESSING COMPLETE ---
Final Prediction:  8
>>> TEST STATUS: PASSED [SUCCESS] <<<
The accelerator correctly classified the image as 8.
```

```
DEBUG [FC2]: Digit 0 -> Score:        -141
DEBUG [FC2]: Digit 1 -> Score:        -515
DEBUG [FC2]: Digit 2 -> Score:        -153
DEBUG [FC2]: Digit 3 -> Score:        -368
DEBUG [FC2]: Digit 4 -> Score:          48
DEBUG [FC2]: Digit 5 -> Score:         172
DEBUG [FC2]: Digit 6 -> Score:        -160
DEBUG [FC2]: Digit 7 -> Score:        -334
DEBUG [FC2]: Digit 8 -> Score:          19
DEBUG [FC2]: Digit 9 -> Score:         476
----------------------------------------
--- PROCESSING COMPLETE ---
Final Prediction:  9
>>> TEST STATUS: PASSED [SUCCESS] <<<
The accelerator correctly classified the image as 9.
```

Figures 18: Simulation console log outputs demonstrating the scoring mechanism

To rigorously validate the inference accuracy of the hardware accelerator, a comprehensive test was conducted using ten distinct test images, each representing a digit from 0 to 9 from the MNIST dataset.

Verification Methodology: For each test case, the corresponding image memory file was loaded into the system. The simulation was executed in post-implementation mode to account for hardware timing. The correctness of the output was verified by monitoring the internal score generation and the final decision logic.

Score Analysis and Decision: The accelerator computes a confidence score for each of the 10 classes (digits 0-9). As shown in the simulation logs, the Fully Connected Layer 2 (FC2) outputs signed integer scores. The Decision Unit successfully identified the maximum value among these scores for every test case.

3.5.4. Implementation Results

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.949 ns	Worst Hold Slack (WHS):	0.024 ns	Worst Pulse Width Slack (WPWS):	8.750 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:	3066	Total Number of Endpoints:	3066	Total Number of Endpoints:	683

All user specified timing constraints are met.

Figure 19: Post-Implementation Timing Summary at 50 MHz.

Operating Frequency Selection: The target clock frequency for the design was established at 50 MHz (Period: Tclk=20ns). Initial synthesis attempts at 100 MHz revealed timing violations, as the combinatorial depth of the convolution arithmetic logic exceeded the 10 ns timing budget. By scaling the frequency to 50 MHz, the design achieved Timing Closure, ensuring that all setup and hold time constraints were met with a comfortable safety margin.

Maximum Frequency Calculation (Fmax): Based on the timing report provided in Figure 19, the theoretical maximum operating frequency of the accelerator can be derived from the Worst Negative Slack (WNS).

Current Clock Period: 20.000ns

Worst Negative Slack (WNS): +5.949ns

The actual propagation delay of the critical path is calculated as:

$$Tcritical = Tclk - WNS$$

$$Tcritical = 20.000ns - 5.949ns = 14.051ns$$

Using this critical path delay, the maximum frequency is:

$$Fmax = \frac{1}{Tcritical} = \frac{1}{14.051 \times 10^{-9}s} \approx 71.17MHz$$

Conclusion: The system is currently operating at 50 MHz, which is well within the calculated limit of 71.17 MHz. This configuration provides a positive slack of 5.949 ns, guaranteeing robust thermal stability and reliability under varying operating conditions, while successfully eliminating timing violations.

Resource	Utilization	Available	Utilization %
LUT	1212	32600	3.72
LUTRAM	276	9600	2.88
FF	386	65200	0.59
BRAM	7.50	75	10.00
IO	8	210	3.81

Figure 20: Resource utilization for the complete cnn_top system

The top-level utilization report provides a comprehensive view of the hardware cost, incorporating all sub-modules (Conv, Pool, FC, Decision) and the interconnecting memory architecture.

Logic Resources (LUTs & LUTRAM):

LUTs (1212 / 3.72%): Unlike individual sub-modules, the top-level logic consumption includes the arithmetic logic (adder trees) inferred from the DSP operations and the multiplexing logic required for data routing between layers.

LUTRAM (276 / 2.88%): This indicates that the synthesis tool effectively implemented smaller intermediate buffers (such as the flatten_mem or partial convolution results) using Distributed RAM (SLICEM) rather than consuming expensive Block RAMs.

Memory Resources (BRAM):

Block RAM (7.50 / 10.00%): The design utilizes 7.5 BRAM tiles, representing the highest percentage utilization among all resources. This confirms that the large static datasets—specifically the pre-trained weights for the Convolution and Fully Connected layers and the input image memory—were correctly inferred as Block RAMs. This is a highly efficient design choice, as it offloads data storage from the general logic fabric.

Sequential Logic (FF):

Flip-Flops (386 / 0.59%): The low register count suggests a streamlined pipeline. The system relies more on memory-based data transfer than on deep register shifting, which reduces power consumption.

3.5.5. Post Implementation Timing Simulation

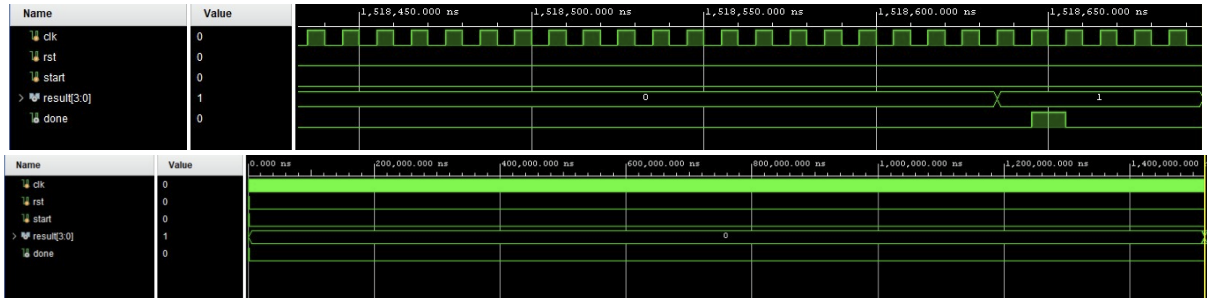


Figure 21: Post-Implementation timing simulation of the design

This waveform demonstrates the physical behavior of the design on the actual FPGA fabric. Unlike the ideal behavioral simulation, a realistic propagation delay is observed between the start and done signals, caused by the routing and logic gate delays within the hardware. Despite these physical delays, the system successfully completes the operation and stabilizes at the correct output value, proving the design's robustness and timing reliability.

4. PYTHON CODES

To bridge the gap between the high-level software model (trained in TensorFlow/Keras) and the low-level hardware implementation (Verilog), a custom Python automation script was developed. This script performs three critical functions essential for the correct operation of the FPGA accelerator:

Fixed-Point Quantization: The hardware design operates on fixed-point arithmetic to reduce resource consumption. The script scales the floating-point weights and biases from the Keras model by a factor of 27 (128). This aligns with the hardware's bit-shifting logic (>>> 7).

Weights: Converted to 8-bit signed hexadecimal.

Biases: Converted to 32-bit signed hexadecimal to prevent overflow during accumulation.

Memory Layout Reordering (HWC to CHW): TensorFlow stores tensors in HWC (Height-Width-Channel) format, whereas the FPGA processing pipeline is optimized for CHW (Channel-Height-Width) or sequential channel access. The script reshapes and transposes the weight matrices to match the sequential memory access pattern of the hardware, ensuring correct matrix-vector multiplication.

Test Vector Generation: For verification purposes, the script randomly selects an image from the MNIST test dataset based on a target digit (e.g., '1'), normalizes it, and exports it as a .mem file. This allows for rapid and randomized testing of the hardware against real-world data.

Script Source Code: The complete Python script used for generating the .mem initialization files is provided below:

```python
import numpy as np
import tensorflow as tf
import os
import random

# --- CONFIGURATION ---
# The directory where the .mem files will be saved.
OUTPUT_DIR = r"C:/EHB426/memory_files"

# Set the target digit you want to test (0-9).
# The script will randomly select an image of this digit from the test set.
TARGET_DIGIT = 1

# Create the directory if it doesn't exist
if not os.path.exists(OUTPUT_DIR):
    os.makedirs(OUTPUT_DIR)

print(f"--- OUTPUT DIRECTORY: {OUTPUT_DIR} ---")
```

```
# Fixed-point scaling factor (2^7 = 128).
# This matches the '>>> 7' shift used in the Verilog code.
SCALE = 128.0

def float_to_hex(val, bits=8):
    """
    Converts a floating-point number to a hex string for memory files.
    Applies scaling and handles saturation (clamping).
    """
    val = int(np.round(val * SCALE))

    if bits == 8: # For Weights (Signed 8-bit)
        # Clamp values between -128 and 127
        if val > 127: val = 127
        if val < -128: val = -128
        # Return 2-digit hex
        return f"{(val & 0xFF):02x}"
    else: # For Biases (Signed 32-bit)
        # Return 8-digit hex to prevent overflow
        return f"{(val & 0xFFFFFFFF):08x}"

def save_mem(filename, data, bits=8):
    """
    Writes a numpy array to a .mem file in hexadecimal format.
    """
    path = os.path.join(OUTPUT_DIR, filename)
    with open(path, 'w') as f:
        flat = data.flatten()
        for val in flat:
            if isinstance(val, (np.float32, float, np.float64)):
                f.write(float_to_hex(val, bits) + "\n")
            else:
                f.write(f"{int(val):02x}\n")
    print(f"Generated: {filename}")

# 1. Load the Trained Model
try:
    model = tf.keras.models.load_model("trained_model.keras")
    print("Model loaded successfully.")
except:
    print("ERROR: Could not find 'trained_model.keras'.")
    exit()

# 2. Extract and Convert Weights
for layer in model.layers:
    if not layer.weights: continue
    w, b = layer.get_weights()
    name = layer.name.lower()
```

```
    if 'conv' in name:
        # Conversion: Keras (H, W, In, Out) -> Hardware (Out, H, W, In)
        # This reorders the weights to match the Verilog iteration order.
        w = w.transpose(3, 0, 1, 2)
        save_mem("conv1_weights.mem", w)
        save_mem("conv1_bias.mem", b, bits=32)

    elif 'dense' in name or 'fc' in name:
        if w.shape[0] == 676: # FC1 Layer (Flattened Input)
            print("Converting FC1 Weights to CHW Format...")

            # Keras flattens as H-W-C, but our FPGA hardware reads as C-H-W.
            # We must reshape and transpose the weights to match the hardware
buffer.

            # Step 1: Recover original 4D shape (13x13 image, 4 filters, 32
neurons)
            w = w.reshape(13, 13, 4, 32)

            # Step 2: Transpose from (H, W, C, N) to (C, H, W, N)
            w = w.transpose(2, 0, 1, 3)

            # Step 3: Flatten back to 2D
            w = w.reshape(676, 32)

            # Step 4: Transpose for matrix multiplication (N, Input)
            w = w.transpose()

            save_mem("fc1_weights.mem", w)
            save_mem("fc1_bias.mem", b, bits=32)

        elif w.shape[0] == 32: # FC2 Layer (Output)
            # Standard Transpose
            w = w.transpose()
            save_mem("fc2_weights.mem", w)
            save_mem("fc2_bias.mem", b, bits=32)

# 3. Select a Test Image (RANDOM)
print("Loading MNIST Dataset...")
(_, _), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Find all indices in the test set that match the TARGET_DIGIT
candidate_indices = np.where(y_test == TARGET_DIGIT)[0]

if len(candidate_indices) > 0:
    # Pick one random index from the candidates
    idx = np.random.choice(candidate_indices)
```

```
    print(f"Randomly Selected Index for Digit {TARGET_DIGIT}: {idx}")
else:
    print(f"ERROR: Digit {TARGET_DIGIT} not found in the test set!")
    exit()

# 4. Save the Image
path = os.path.join(OUTPUT_DIR, "test_image.mem")
with open(path, 'w') as f:
    flat = x_test[idx].flatten()
    for val in flat:
        # Ensure values are integers (0-255) before writing hex
        # (Handling cases where data might already be normalized)
        if np.max(x_test[idx]) <= 1.0:
            val = val * 255.0
        f.write(f"{int(val):02x}\n")
```

Code 11: Python code for mem files generation

```
Model loaded successfully.
Generated: conv1_weights.mem
Generated: conv1_bias.mem
Converting FC1 Weights to CHW Format...
Generated: fc1_weights.mem
Generated: fc1_bias.mem
Generated: fc2_weights.mem
Generated: fc2_bias.mem
Loading MNIST Dataset...
Randomly Selected Index for Digit 1: 9143
```

Figure 22: Console output of python code

Analysis: The execution log confirms the successful loading of the pre-trained TensorFlow model and the generation of all required hexadecimal memory files (.mem).

Test Vector Selection: The script successfully accessed the MNIST dataset and randomly selected a specific test instance (Index: 9143) representing the target digit '1'. This specific image index serves as the ground truth input for the subsequent hardware simulation.

REFERENCES

[1] Junye Jiang, FPGA-based acceleration for Convolutional Neural Networks: A comprehensive review, https://arxiv.org/html/2505.13461v1 (accessed Jan. 22, 2026).

[2] C. Zhang *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional Neural Networks," *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, Feb. 2015. doi:10.1145/2684746.2689060

[3] "Convolutional Neural Networks cheatsheet star," CS 230 - Convolutional Neural Networks Cheatsheet, https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks (accessed Jan. 22, 2026).