UNIVERSITY OF PISA

Master's Degree Artificial Intelligence and Data Engineering

Symbolic and Evolutionary Artificial Intelligence Course

Verilog Synthesis of the Inference Phase of an MLP

Work Group:
Arduini Luca
Lucchesi Filippo

ACADEMIC YEAR 2024/2025

Introduction

The primary objective of this project is to design, implement, and synthesize a dedicated hardware accelerator for neural network inference. Specifically, we focus on accelerating a Multi-Layer Perceptron (MLP) by translating a reference C++ software implementation into a Verilog model.

As a specific use case, the network architecture was trained to approximate the two-dimensional sinc (sinc2D) function. However, the resulting hardware is a generic and parameterizable MLP accelerator. This means it is not limited to a single task and can be leveraged to approximate any function by simply loading new weights trained on a different dataset.

This report documents the end-to-end process undertaken to achieve this goal. We will detail every critical step, beginning with the conversion of data representations from floating-point to a hardware-friendly fixed-point format within the C++ model. Subsequently, we describe the architectural design and Verilog implementation of each core component of the MLP, culminating in a complete, synthesizable hardware accelerator ready for deployment on an FPGA or ASIC.

The code is available on GitHub at: github.com/LucaArduini/MLP_Verilog

Contents

Chapter 1

Conversion of C++ MLP code from float to fixed-point arithmetic

When synthesizing Multi-Layer Perceptrons (MLPs) in Verilog for hardware implementation (like FPGAs or ASICs), fixed-point arithmetic is heavily favored over floating-point numbers. This is because implementing floating-point units (FPUs) requires dedicated hardware for things like:

- Handling the sign, exponent, and mantissa separately.
- Normalization and de-normalization.
- Alignment of decimal points (exponents) for addition/subtraction.
- Special case handling (NaN, infinity).

This translates to a significantly larger number of logic gates, lookup tables (LUTs), and registers on an FPGA, or a larger die area on an ASIC. Also, Floating Point operations are generally slower and have higher latency. The multiple steps involved (e.g., unpacking, aligning, calculating, normalizing, packing) often require multiple clock cycles, even in pipelined designs.

On the other hand, operations (addition, subtraction, multiplication) on fixed-point numbers can be implemented using standard integer arithmetic logic units (ALUs). The binary point is implicit and managed by the designer. This results in much simpler and smaller hardware. Integer operations are much faster and can often be completed in a single clock cycle, especially for addition and subtraction. Multiplication might take a few cycles or use dedicated DSP blocks (on FPGAs), but it's still generally faster than full FP multiplication. This leads to higher throughput for the MLP.

Due to all of the above reasons, we modified the original C++ implementation of the MLP (provided to us by Prof. Cococcioni) such that all operations are performed using fixed-point arithmetic instead of floating point numbers. This includes all matrix multiplications done during the forward phase and backward phase of the MLP training.

We then used that code as a baseline to design an FPGA-compliant MLP inference using Verilog, with the assistance of Prof. Baronti.

1.1 Implementation of the conversion

We made use of the CNL library (available at: github.com/johnmcfarlane/cnl) to implement the conversion. This library offers an automated method for declaring fixed-point types by specifying the binary point position and the underlying integer type.

To store the MLP weights, we chose a Q7.8 representation. This means that the range of representable numbers is [-128.0, +127.996].

```
#include "cnl/include/cnl/all.h"

////////////////////////////////////////////////
//            Fixed-Point parameters          //
////////////////////////////////////////////////

// Use one consistent constant for fractional bits
const int fractional_bits = 8;

// Define the fixed-point type using CNL library
// int16_t: underlying integer type (16 bits)
// cnl::power<-fractional_bits>: specifies the position of the binary point (8
    fractional bits)
using fixed_point_16 = cnl::scaled_integer<int16_t, cnl::power<-fractional_bits
    >>;
```

To store intermediate sums in matrix multiplications, we used a representation of double overall size: Q47.16.

```
// Define a wider accumulator type for intermediate sums in matrix
    multiplications
using temp_accumulator_fp = cnl::scaled_integer<int64_t, cnl::power<-
    fractional_bits*2>>; // e.g., Q47.16
```

We then modified the matrix multiplication functions to use this temporary accumulator, as shown in the following code.

```
// Modified to write to a temp_accumulator_fp destination C_ptr
void A_mult_B_wide_dest(const fixed_point_16* A_ptr, const fixed_point_16* B_ptr
    , temp_accumulator_fp* C_ptr,
                        int rigA, int colA, int colB) {
    for (int i = 0; i < rigA; i++) {
        for (int j = 0; j < colB; j++) {
            temp_accumulator_fp tmp_sum = 0.0;
            for (int k = 0; k < colA; k++) {
                tmp_sum += static_cast<temp_accumulator_fp>(A_ptr[i*colA+k]) *
    B_ptr[k*colB+j];
            }
            C_ptr[i*colB+j] = tmp_sum; // Store full precision sum
        }
    }
}
```

While the forward pass remained largely unchanged, some greater consideration had to be take in regards to the backward pass. During our testings, we found that the MLP was having trouble learning due to the saturation of the gradients when converting them from Q47.16 back to Q7.8. This in turn lead to the saturation of the weights after very few updates. To help mitigate this problem, we made two adjustments. The first one involved the manual clipping of the gradients used to update the hidden layer weight values (W1) and output layer weight values (W2). We use the parameter `grad_clip_abs_val_wide` to precisely define the clipping boundaries. After some experimenting, we have found 100 to be the best value.

```
void MLP_MSELIN_backprop(const array<fixed_point_16, elem> &y_true){

    //...

    // Step 6: Convert wide gradients to Q7.8 with clipping
    // This is crucial to prevent saturation in fixed-point representation
    const temp_accumulator_fp grad_clip_abs_val_wide = 100.0;

    for (int i = 0; i < n_hidden; ++i) {
        for (int j = 0; j < n_features+1; ++j) {
            temp_accumulator_fp val = dL_dW1_wide[i][j];
            if (val > grad_clip_abs_val_wide) val = grad_clip_abs_val_wide;
            else if (val < -grad_clip_abs_val_wide) val = -
    grad_clip_abs_val_wide;
            delta_W1_unscaled[i][j] = static_cast<fixed_point_16>(val);
        }
    }
    for (int i = 0; i < n_output; ++i) {
        for (int j = 0; j < n_hidden+1; ++j) {
            temp_accumulator_fp val = dL_dW2_wide[i][j];
            if (val > grad_clip_abs_val_wide) val = grad_clip_abs_val_wide;
            else if (val < -grad_clip_abs_val_wide) val = -
    grad_clip_abs_val_wide;
            delta_W2_unscaled[i][j] = static_cast<fixed_point_16>(val);
        }
    }

    //..

}
```

The second change we made was to limit the maximum possible update applied to the weights after each backpropagation. After some experimentation, we set `max_abs_final_update` to be equal to (2.0 / 256.0) , or 0.0078125.

```
void MLP_MSELIN_train(const array<array<fixed_point_16, n_features>, num_train>
    &x, const array<fixed_point_16, num_train> &y){

    //...

    const fixed_point_16 max_abs_final_update = fixed_point_16{2.0/256.0};

    array<array<fixed_point_16, n_features+1>, n_hidden> delta_W1;
    for (int i = 0; i < n_hidden; ++i) {
        for (int j = 0; j < n_features+1; ++j) {
            fixed_point_16 update_val = eta * delta_W1_unscaled[i][j];
            if (update_val > max_abs_final_update) update_val =
    max_abs_final_update;
```

```
            else if (update_val < -max_abs_final_update) update_val = -max_abs_final_update;
            delta_W1[i][j] = update_val;
        }
    }
    array<array<fixed_point_16, n_hidden+1>, n_output> delta_W2;
    for (int i = 0; i < n_output; ++i) {
        for (int j = 0; j < n_hidden+1; ++j) {
            fixed_point_16 update_val = eta * delta_W2_unscaled[i][j];
            if (update_val > max_abs_final_update) update_val = max_abs_final_update;
            else if (update_val < -max_abs_final_update) update_val = -max_abs_final_update;
            delta_W2[i][j] = update_val;
        }
    }

    //...

}
```

These adjustments led to a much stabler training, and an overall final accuracy that was comparable to the float version of the MLP.

1.2 Activation function and in-place computation

We used ReLU as the activation function of the hidden layer, due to its simplicity and ease of implementation in an FPGA. We also made some memory optimization by computing the ReLU and its gradient in-place, instead of using a support variable.

```
void MLP_relu_inplace(const array<array<fixed_point_16, elem>, n_hidden> &z, array<array<fixed_point_16, elem>, n_hidden> &relu_out){
    for (int i = 0; i < n_hidden; ++i) {
        for (int j = 0; j < elem; ++j) {
            relu_out[i][j] = max(fixed_point_16{0.0f}, z[i][j]);
        }
    }
}

void MLP_relu_gradient_inplace(const array<array<fixed_point_16, elem>, n_hidden> &Z, array<array<fixed_point_16, elem>, n_hidden> &reluGrad_out) {
    for (int i = 0; i < n_hidden; ++i) {
        for (int j = 0; j < elem; ++j) {
            reluGrad_out[i][j] = (Z[i][j] > 0.0f) ? fixed_point_16{1.0f} : fixed_point_16{0.0f};
        }
    }
}
```

1.3 Learning Rate

The learning rate was set to the minimum positive number representable in Q7.8, which is (1.0/256.0), or 0.00390625.

1.4 Accuracy comparison

The MLP we trained has a single hidden layer with 4 neurons and a single neuron in the output layer. It was tasked with approximating the following function:

$$10 * sinc(x_1) * sinc(x_2)$$

The model was trained for 500 epochs on a randomly generated dataset consisting of 22500 training patterns and 2250 test patterns.

Through experimentation with the clipping threshold and the maximum update allowed, the model trained using fixed-point precision reached similar performance to the float-based model, despite the severe lack in precision. Our results are shown in table 1.1. Figure 1.1 shows the side-by-side comparison of the true plot and our generated function using the `MLP_fixed` model.

Table 1.1: Mean Squared Error (MSE) Statistics over 10 Training Runs

	MLP_float		**MLP_fixed**	
Metric	**Training MSE**	**Test MSE**	**Training MSE**	**Test MSE**
Mean	0.8979	1.4934	0.6301	1.2497
Std Dev	0.6761	0.4760	0.0578	0.0573
Min	0.5331	1.2016	0.5956	1.2118
Max	2.7939	2.8206	0.7645	1.4074

Since in this experiment we obtained better results with `MLP_fixed`, we decided to save a set of weights generated by it to be used in the testing phase of the MLP we designed with Verilog. By looking at the generated plot, it is easy to see that the approximation is far from accurate; this is most likely due to the very small size of the model. To verify this claim, we did some further research, reported in chapter 4.

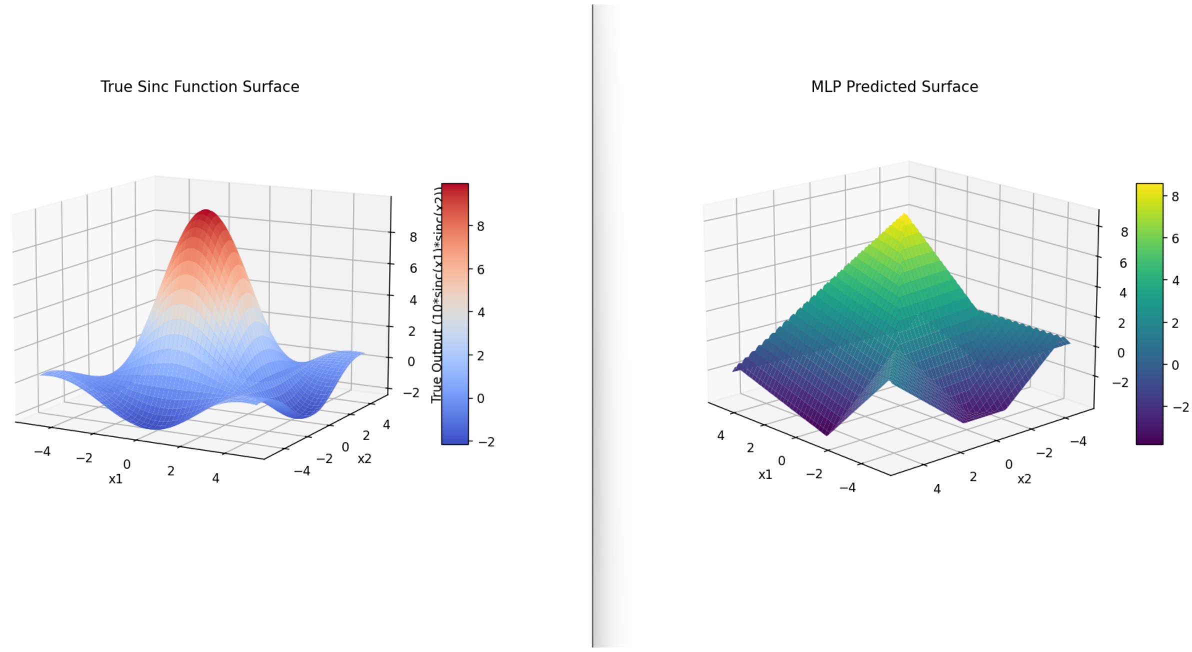

Figure 1.1: True $10 * sinc(x_1) * sinc(x_2)$ vs Learned Function with `MLP_fixed` model

Chapter 2

Synthesis of the inference of a simple MLP in Verilog

The next phase of the project involved the hardware implementation of the MLP inference process in Verilog. Our design is modular and centered around four key, parameterizable components.
At the core of our architecture are two fundamental units:

- **MAC** (Multiply-Accumulate) **Unit**: Performs the essential dot-product arithmetic for each neuron
- **Weight Memory**: A dedicated memory block for storing and accessing the network's weights

These units were then used to construct the network's layers:

- **Hidden Layer Module**: Integrates MAC and memory units to form the first computational layer
- **Output Layer Module**: Forms the final layer, processing the hidden layer's outputs to generate the network's result

These two layer modules are, in turn, instantiated within a **Top-Level Module**, which coordinates the entire inference process through a central state machine and manages external communication.

Throughout the design, a strong emphasis was placed on making every component parametric. This allows for straightforward scaling and modification, enabling the rapid assembly of different MLP architectures from these reusable building blocks.

2.1 MAC unit

The MAC unit is tasked with performing the multiplication and accumulation operations that are required when computing the pre-activation values Z1 and Z2:

$$Z1 = W1 * Ext(P^T)$$

$$Z2 = W2 * Ext(rA1)$$

where P is the input pattern and $rA1 = ReLU(Z1)$.

Since we are working with Q7.8 fixed-point numbers, the product must be represented on 32 bits, in Q15.16 format. The accumulator is on double the number of bits (64), which gives us ample headroom for summing up multiple 32 bits numbers (the maximum number (N) of MAC sums that can be performed in the worst case is obtained with the following formula: $floor((2^{63} - 1)/(2^{31-1}))$; we obtain $2^{32} + 2$).

```
module MLP_mac #(
    parameter A_WIDTH = 16,   // Bit-width of input A
    parameter B_WIDTH = 16,   // Bit-width of input B
    parameter ACC_WIDTH = 64   // Bit-width of the result/accumulator
)(
    input clk,
    input start,       // Initializes acc = a * b
    input valid,       // Triggers accumulation: acc += a * b
    input signed [A_WIDTH-1:0] a,
    input signed [B_WIDTH-1:0] b,
    output signed [ACC_WIDTH-1:0] result
);

    reg signed [ACC_WIDTH-1:0] acc;
    wire signed [A_WIDTH + B_WIDTH - 1:0] product;
    wire signed [ACC_WIDTH-1:0] product_ext;

    assign product = a * b;

    // Explicit sign extension to ACC_WIDTH
    assign product_ext = {{(ACC_WIDTH - (A_WIDTH + B_WIDTH)){product[A_WIDTH + B_WIDTH - 1]}}, product};

    assign result = (acc >>> (A_WIDTH/2)); // Right shift to adjust the result

    always @(posedge clk) begin
        if (start) begin
            acc <= product_ext;
        end else if (valid) begin
            acc <= acc + product_ext;
        end
    end

endmodule
```

At any time, the accumulator value *acc* is equal to:

$$acc = \begin{cases} product\_ext & valid = -, start = 1 \\ acc + product\_ext & valid = 1, start = 0 \\ acc & valid = 0, start = 0 \end{cases}$$

If `start` is set, the previous accumulator value is substituted by the first extended product, thus removing the need to reset the content of the `acc` register. Notice that the result is returned on 64 bits; it will be scaled back to 16 bits by the layer module, which will be presented later. Also, the result is shifted right by 8 bits so that the decimal point is moved back to its original place.

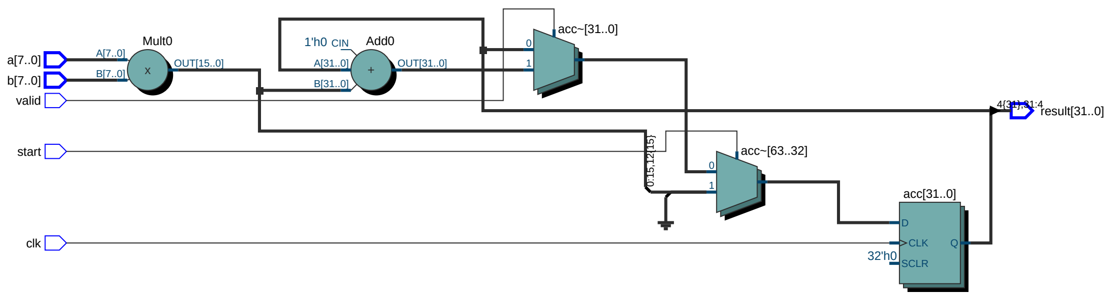

Figure 2.1: Register Transfer Level (RTL) of MAC unit.

2.2 Weight Memory

The `MLP_weight_mem` module provides a synchronous memory block for storing a layer's neural network weights. It implements a RAM structure with a single address port, often referred to as a Simple Dual-Port RAM. This configuration allows for simultaneous read and write operations.

The module features distinct behaviors for its read and write ports. The write operation is synchronous: when the `wr_en` signal is asserted, the input data (`wr_data`) is captured and stored at the location specified by `addr` on the rising edge of the clock. The read operation, however, is asynchronous, as it is described with a simple continuous assignment (*assign*). Consequently, the `rd_data` output updates immediately and combinationally in response to any change in the `addr` input. This design implies a "Read-Before-Write" behavior: if a read and a write occur at the same address in the same cycle, the read port will provide the value that was stored in memory before the write completes on the next clock edge.

```
module MLP_weight_mem #(
    parameter ADDR_WIDTH = $clog2(2+1), // log2(number of perceptrons)
    parameter DATA_WIDTH = 16           // Width of each weight (e.g., fixed-point)
)(
    input                   clk,        // Clock signal
    input                   rst,        // Reset signal
    input  [ADDR_WIDTH-1:0] addr,       // Address input for read/write operations
    input                   wr_en,      // Write enable signal
    input  [DATA_WIDTH-1:0] wr_data,    // Data to be written into the memory

    output [DATA_WIDTH-1:0] rd_data     // Data read from the memory at 'addr'
);

    // Memory declaration with M9K block usage
    // Internal memory (array of registers) with 2^ADDR_WIDTH locations, each
```

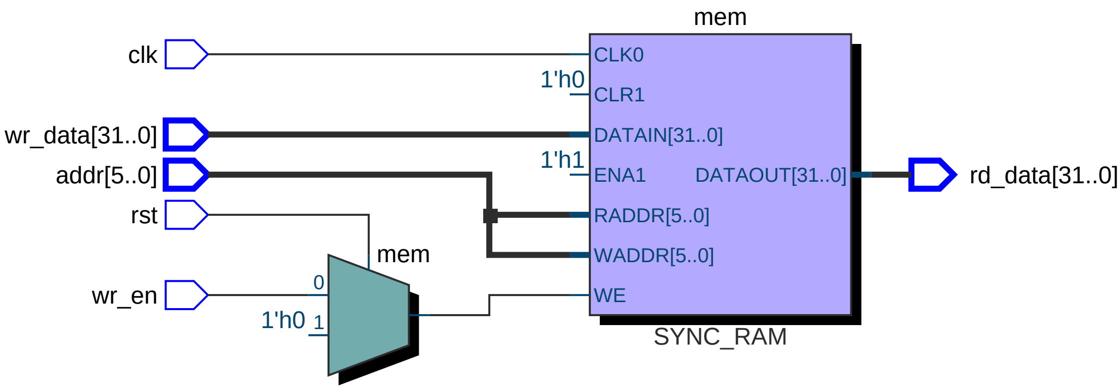

Figure 2.2: RTL of weight memory.

```
    location being DATA_WIDTH bits wide
    (* ramstyle = "M9K" *)
    reg [DATA_WIDTH-1:0] mem [0:(1 << ADDR_WIDTH)-1];

    always @(posedge clk) begin
        if (rst) begin
            //
            // Optional reset behavior (typically no init for BRAM)
            //
        end
        else if (wr_en) begin
            // If write enable is asserted, write 'wr_data' to the location
    specified by 'addr'
            mem[addr] <= wr_data;
        end
    end

  assign rd_data = mem[addr];

endmodule
```

The reset behaviour is not defined, since there is no default value that would be of interest to us in this use case.

2.3 Hidden Layer

The `MLP_layer_hidden` module represents a complete computational layer of the neural network. It is a highly parallel and parameterizable structure designed to compute the outputs for multiple neurons simultaneously. This architecture is a direct translation of the matrix-vector multiplication at the heart of a neural network layer into a parallel hardware implementation.

The core computation for each neuron is a dot product. When analyzing this operation, a key pattern emerges: each individual input component (e.g., x_1) must be multiplied by an entire column of the layer's weight matrix, that is, the corresponding weight from every neuron. To exploit this inherent parallelism, we designed the hardware as shown in Figure 2.3. The `INPUT` value is broadcast on a shared bus to N_NEURONS identical processing lanes. Each lane, representing a single neuron, contains:

1. A dedicated **Weight Memory** (`MLP_weight_mem`), holding the unique weights and bias for that neuron
2. A **Multiply-Accumulate Unit** (`MLP_mac`), which performs the arithmetic

This structure allows the layer to process one component of the input vector across all neurons in a single clock cycle. For instance, in the case of four neurons, when the first input element x_1 is placed on the input bus, it is simultaneously multiplied by w_{11}, w_{21}, w_{31} and w_{41} in their respective MAC units. This parallel approach dramatically accelerates the dot product calculation.

After all input components have been processed and accumulated, a final, shared logic block applies the ReLU activation and clipping functions to all neuron outputs simultaneously, before registering the final results.

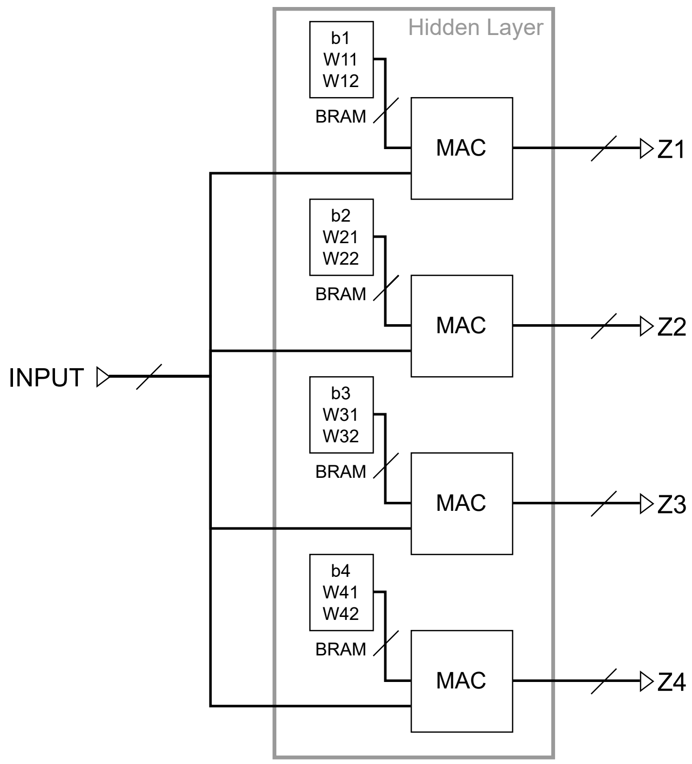

Figure 2.3: Schematic view of the hidden layer.

2.3.1 Operational Flow

The module operates in two distinct phases, orchestrated by an external controller (e.g., a top-level FSM).

1. **Weight Loading Phase (Initialization)**

   Before inference can begin, the weights for every neuron must be loaded into their respective memories. This is a sequential process managed by the external controller:

   - The external controller asserts the `wr_en` signal to enable writing.
   - It iterates through each neuron (`wr_row`) and each weight within that neuron (`wr_col`), providing `wr_weight` and pulsing `clk` to store all necessary weights.
   - During this phase, `start`, `valid`, and `relu_en` are typically kept low.

2. **Inference Phase (Forward Pass for one input vector)**

   Once the weights are loaded, the layer is ready to perform a forward pass. This phase processes a single input vector and requires *N_INPUTS + 1* clock cycles to complete.

   - The `input_value` and `input_index` are driven sequentially by the external controller, presenting each component of the layer's input vector one at a time.
   - **For the first input component (`input_index = 0`):**
     - `start` is asserted (along with `valid`).
     - Each `weight_mem_j` outputs `Weight_j0`.
     - Each `mac_j` computes `Input_0 * Weight_j0` and stores it in its accumulator. `mac_outputs[j]` now holds this initial product.
   - **For subsequent input components (`input_index = 1` to `N_INPUTS-1`):**
     - `start` is de-asserted. `valid` is asserted.
     - Each `weight_mem_j` outputs `Weight_ji` (for the current input `i`).
     - Each `mac_j` computes `Accumulator_j + Input_i * Weight_ji`. `mac_outputs[j]` is updated with the new sum.
   - **After all `N_INPUTS` have been processed:**
     - `mac_outputs[j]` for each neuron `j` now holds the complete dot product (pre-activation value).
     - `relu_en` is asserted for one clock cycle.
     - The `relu_clip_logic` block processes all `mac_outputs`, applies ReLU and clipping, and stores the final `OUT_WIDTH`-bit results into the `outputs_flat` register.

```
module MLP_layer_hidden #(
    parameter N_INPUTS     = 2+1,  // Number of inputs to the layer (and thus, weights per neuron)
    parameter N_NEURONS    = 4,    // Number of neurons in this layer
    parameter IN_WIDTH     = 16,   // Bit-width of each input value
    parameter WGT_WIDTH    = 16,   // Bit-width of each weight value
    parameter MAC_WIDTH    = 64,   // Bit-width of the accumulator in the MAC units
    parameter OUT_WIDTH    = 16    // Bit-width of each neuron's output (after ReLU/clipping)
)(
```

```
    input    clk,       // Clock signal

    // === Weight memory write interface ===
    // Allows external loading of weights into the neuron's weight memories.
    input wr_en,  // Global write enable for all weight memories
    input [WGT_WIDTH-1:0] wr_weight, // Weight data to be written
    input [$clog2(N_NEURONS)-1:0] wr_row,  // Selects which neuron's weight memory to write to (0 to N_NEURONS-1)
    input [$clog2(N_INPUTS)-1:0]  wr_col,  // Selects the specific weight address within the chosen neuron's memory (0 to N_INPUTS-1)


    // === Input and control signals ===
    input signed [IN_WIDTH-1:0] input_value,   // Current input value being processed
    input [$clog2(N_INPUTS)-1:0] input_index,   // Index of the current input_value (0 to N_INPUTS-1), used as read address for weight memories
    input start,  // Signal to initialize/reset all MAC accumulators (acc = input * weight, or 0 if first input is 0)
    input  valid, // Signal to trigger the accumulation step in all MAC units (acc += input * weight)
    input  relu_en, // Enable signal to register the MAC outputs through ReLU/clipping stage into `outputs_flat`

    // === Output vector ===
    // All neuron outputs are concatenated into a single flat vector.
    output reg [N_NEURONS*OUT_WIDTH-1:0] outputs_flat
);

    // === Constants ===
    // Maximum positive value representable by OUT_WIDTH bits (signed, but used for positive clipping)
    localparam signed [OUT_WIDTH-1:0] MAX_VAL = {1'b0, {(OUT_WIDTH-1){1'b1}}}; // e.g., 0111...1

    // === Weight memory outputs ===
    // Array of wires to hold the weight read from each neuron's dedicated weight memory.
    // `weight_out[j]` is the weight for the current `input_index` for neuron `j`.
    wire signed [WGT_WIDTH-1:0] weight_out [N_NEURONS-1:0];

    // === MAC outputs ===
    // Array of wires to hold the accumulated result from each neuron's MAC unit.
    wire signed [MAC_WIDTH-1:0] mac_outputs [N_NEURONS-1:0];

    // === Instantiate weight memories ===
    // Generates N_NEURONS instances of MLP_weight_mem, one for each neuron.
  genvar j; // Loop variable for generation
  generate
    for (j = 0; j < N_NEURONS; j = j + 1) begin : gen_weight_mem
      MLP_weight_mem #(
        .ADDR_WIDTH($clog2(N_INPUTS)), // Each memory stores N_INPUTS weights
        .DATA_WIDTH(WGT_WIDTH)
      ) weight_mem_j (
        .clk(clk),
        .rst(1'b0),
        .wr_en(wr_en && (wr_row == j)),
        .addr(wr_en ? wr_col : input_index),
        .wr_data(wr_weight),
        .rd_data(weight_out[j])
      );
    end
  endgenerate

  // === Connect weights to MACs ===
```

```
    // Generates N_NEURONS instances of MLP_mac, one for each neuron.
  generate
    for (j = 0; j < N_NEURONS; j = j + 1) begin : gen_mac
      MLP_mac #(
        .A_WIDTH(IN_WIDTH),
        .B_WIDTH(WGT_WIDTH),
        .ACC_WIDTH(MAC_WIDTH)
      ) mac_j (
        .clk(clk),
        .a(input_value),          // The current input_value is broadcasted to all MAC units.
        .b(weight_out[j]),        // The corresponding weight for neuron 'j' (from its weight memory).
        .valid(valid),            // Pass through valid signal
        .start(start),            // Pass through start signal
        .result(mac_outputs[j]) // Accumulated result for neuron j
      );
    end
  endgenerate

    // === ReLU + Clipping stage ===
    // This block registers the MAC outputs after applying ReLU and clipping, when 'relu_en' is asserted.
  always @(posedge clk) begin : relu_clip_logic
      integer n; // Loop variable for neurons
    if (relu_en) begin
      for (n = 0; n < N_NEURONS; n = n + 1) begin
        if (mac_outputs[n] < 0) begin       // ReLU: if negative, output 0
          outputs_flat[n*OUT_WIDTH +: OUT_WIDTH] <= 0;
        end
                else if (mac_outputs[n] > MAX_VAL) begin  // Clipping: if greater than MAX_VAL, output MAX_VAL
                // Note: This comparison implicitly extends MAX_VAL to MAC_WIDTH for comparison.
          outputs_flat[n*OUT_WIDTH +: OUT_WIDTH] <= MAX_VAL;
        end else begin         // Otherwise, output the (truncated) MAC result
          outputs_flat[n*OUT_WIDTH +: OUT_WIDTH] <= mac_outputs[n][OUT_WIDTH-1:0];
        end
      end
    end
        // If relu_en is not asserted, outputs_flat retains its previous value.
  end

endmodule
```

2.4 Output Layer

The `MLP_layer_output` module shares the same fundamental architecture as the hidden layer module. The key distinction lies in the final output stage. While the hidden layer applies a ReLU activation function, the output layer typically does not. Instead, its purpose is to produce the final inference value. Therefore, this module replaces the ReLU logic with a saturation (or clipping) mechanism. This ensures that the final result is safely clipped to fit within the specified `OUT_WIDTH` without overflow.

For brevity's sake, the code of the Output Layer is not reported, as it is mostly identical to the hidden layer.

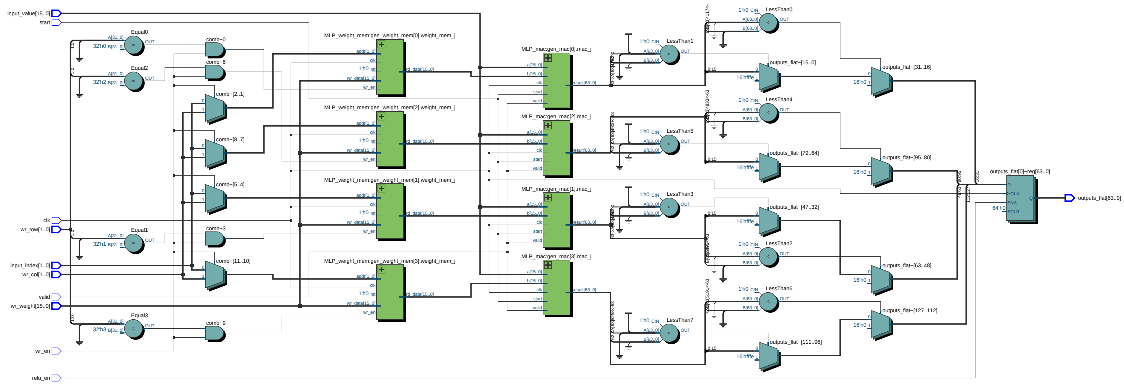

Figure 2.4: RTL of Hidden Layer.

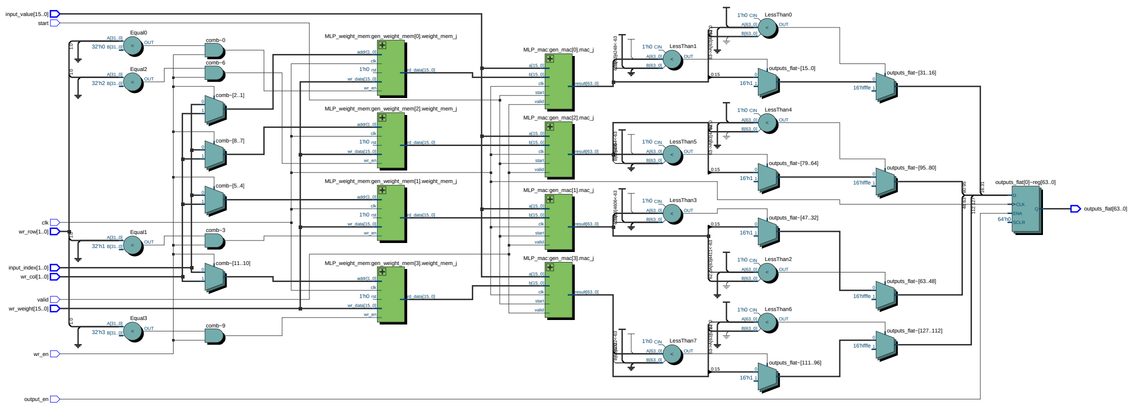

Figure 2.5: RTL of Output Layer.

2.5 Putting it all together: MLP with one hidden layer, single output

The final `mlp` module serves as the top-level controller for the entire hardware accelerator, integrating the computational layer modules and orchestrating the entire inference process. It acts as a master controller, managing the data flow and operational sequence via a central Finite State Machine (FSM). The `MLP_layer_hidden` and `MLP_layer_output` modules are instantiated as slave components that perform the core neural network computations upon command.

Communication with an external system (like a CPU) is handled through a simplified, memory-mapped interface inspired by Avalon-MM. An external controller can use this interface to write inputs and weights, read outputs and status, and control the execution flow. The primary interface signals include `addr` to select a target register, `writedata` and `readdata` for data transfer, and `write_en` to initiate a write operation.

The module exposes four addressable locations:

- `ADDR_CTRL`: to read/write in the `mlp`'s control register;
- `ADDR_INPUT`: to write the input values inside the `mlp`'s internal registries;
- `ADDR_WEIGHT`: to write weight values inside the weight memories;
- `ADDR_OUTPUT`: to read the final output.

The control register has 4 useful bits:

- `CTRL_RUN_BIT`: starts the computation;
- `CTRL_DONE_BIT`: signals that the computation is done;
- `CTRL_INTERRUPT_BIT`: enables an interrupt signal when the computation is done;
- `CTRL_SET_LAYER_BIT`: sets the layer in which we are currently writing weights.

We won't show here the entire MLP Verilog code, as it is hundreds of lines long, but we will describe in detail the operational flow.

I. Configuration/Initialization Phase

This phase is driven by an external controller via the Avalon-MM interface, primarily when the FSM is in the `IDLE` state.

A **System Reset (`rst = 1`):**

1. The FSM transitions to `START`, then immediately to `IDLE`.
2. The `ctrl` register is cleared (e.g., `run` and `done` bits are 0).
3. `input_regs[0]` (bias for the hidden layer) is initialized to `1` (in Q7.8 format). The `input_index` (for writing inputs) is initialized.

4. Internal indices for weight loading (`hidden_weight_row`, `hidden_weight_col`, `output_weight_row`, `output_weight_col`) are reset.

B **Loading Input Values (while FSM is in `IDLE`):**

1. The external controller writes to address `ADDR_INPUT` (`2'd1`).
2. If `write_en` is high, `wr_input` becomes active.
3. On the next positive clock edge, `writedata[IN_WIDTH-1:0]` is stored into `input_regs [input_index]`.
4. The internal `input_index` automatically increments to point to the next input slot.
5. The external controller repeats this for all `N_INPUTS` actual input values. The first value written by the user goes into `input_regs[1]`, as `input_regs[0]` is the hardcoded bias.

C **Loading Weights (while FSM is in `IDLE`):**

1. The external controller first writes to `ADDR_CTRL` (`2'd0`) to set or clear `ctrl[CTRL_SET _LAYER_BIT]` to select the target layer:
   - `ctrl[CTRL_SET_LAYER_BIT] = 0`: Target hidden layer (layer 0).
   - `ctrl[CTRL_SET_LAYER_BIT] = 1`: Target output layer (layer 1).
2. The external controller then writes to address `ADDR_WEIGHT` (`2'd2`).
3. If `write_en` is high, `wr_weight` becomes active.
4. On the next positive clock edge, `writedata[WGT_WIDTH-1:0]` is passed as the weight data to the selected layer's weight memory interface.
5. The internal weight address indices (`hidden_weight_row/col` or `output_weight _row/col`) for the selected layer automatically increment to allow sequential filling of its weight matrix (column by column, then row by row).
6. The external controller repeats this process for all weights of the selected layer, then optionally changes `CTRL_SET_LAYER_BIT` and repeats for the other layer.

D **Setting up for Run (Optional, via `ADDR_CTRL`):**

1. The external controller can set `ctrl[CTRL_INTERRUPT_BIT]` to enable an interrupt (`irq`) when the computation is done.
2. The external controller can write to `writedata[31:16]` (which updates `out_sel`) to pre-select which output neuron's value will be available on `readdata` when reading from `ADDR_OUTPUT`. For `N_OUTPUT=1`, `out_sel` is effectively 0.

II. Inference/Computation Phase

This phase is triggered by the external controller setting the `run` bit.

A. **Start Computation:**

1. The external controller writes to `ADDR_CTRL` (`2'd0`), setting `ctrl[CTRL_RUN_BIT]` to `1`.
2. The FSM transitions from `IDLE` to `PRE_RUN_LAYER0`.

B. **Hidden Layer Computation (Layer 0):**

1. **State `PRE_RUN_LAYER0` (1 clock cycle):**
   - `input_index` (for reading from `input_regs`) is reset to `0`.
   - FSM transitions to `RUN_LAYER0`.
2. **State `RUN_LAYER0` (`N_INPUTS_HIDDEN` clock cycles):**
   - `input_index` increments from `0` to `N_INPUTS_HIDDEN - 1`.
   - For each cycle:
     - `valid_layer_0` is asserted.
     - If `input_index == 0`, `start_layer_0` is asserted (to initialize MACs in `layer0`).
     - The `layer0` module (`MLP_layer_hidden`) processes `input_regs[input_index]` and its corresponding weights, performing a Multiply-Accumulate (MAC) operation.
   - When `input_index` completes its sequence, FSM transitions to `RUN_ReLU0`.
3. **State `RUN_ReLU0` (1 clock cycle):**
   - `relu_en_hidden` is asserted.
   - The `layer0` module applies its ReLU activation and clipping to the accumulated MAC results and stores them internally in `hidden_layer_outputs_flat`.
   - FSM transitions to `PRE_RUN_LAYER1`.

C. **Output Layer Computation (Layer 1):**

1. **State `PRE_RUN_LAYER1` (1 clock cycle):**
   - `hidden_index` (input index for `layer1`) is reset to `0`.
   - FSM transitions to `RUN_LAYER1`.
2. **State `RUN_LAYER1` (`N_INPUTS_OUTPUT` clock cycles):**
   - `hidden_index` increments from `0` to `N_INPUTS_OUTPUT - 1`.
   - For each cycle:
     - `valid_layer_1` is asserted.
     - If `hidden_index == 0`, `start_layer_1` is asserted.
     - The input to `layer1` is taken from `output_layer_inputs_flat` (which is `hidden_layer_outputs_flat` concatenated with a '1' (in Q7.8 format) to represent the bias. The current slice is selected by `hidden_index`.
     - The `layer1` module (`MLP_layer_output`) processes this input and its corresponding weight, performing MAC operations.

- When `hidden_index` completes its sequence, FSM transitions to `RUN_CLIP1`.

3. **State `RUN_CLIP1` (1 clock cycle):**
   - `clip_en_output` (acting as `output_en` for `layer1`) is asserted.
   - The `layer1` module applies its clipping and stores the final results in `output_layer_outputs_flat`.
   - FSM transitions to `DONE`.

D. **Computation Complete:**

1. **State `DONE`:**
   - `ctrl[CTRL_RUN_BIT]` is cleared.
   - `ctrl[CTRL_DONE_BIT]` is set to `1`.
   - If `ctrl[CTRL_INTERRUPT_BIT]` was enabled, the `irq` signal is asserted (`irq = ctrl[2] && ctrl[1]`).
   - The module remains in this state until a `restart` condition.

III. Reading Results and Restarting the Process

1. The external controller can detect completion by:
   - Polling `ctrl[CTRL_DONE_BIT]` by reading from `ADDR_CTRL`.
   - Responding to the `irq` signal generated by the MLP module (if enabled).

2. The external controller reads the MLP output(s) by accessing `ADDR_OUTPUT` (`2'd3`). The `readdata` will contain the output of the neuron selected by `out_sel`. If `N_OUTPUT > 1`, the controller may need to update `out_sel` (via `ADDR_CTRL`) and read `ADDR_OUTPUT` multiple times.

3. To start a new computation:
   - The external controller writes to `ADDR_CTRL`. If this write occurs while `ctrl[CTRL_DONE_BIT]` is high, the `restart` signal is asserted (`restart = wr_ctrl && ctrl[CTRL_DONE_BIT]`).
   - This `restart` causes the FSM to transition from `DONE` to `START` (and then to `IDLE`).
   - In the `START` state, `ctrl[CTRL_DONE_BIT]` and `ctrl[CTRL_RUN_BIT]` are cleared.
   - The external controller can then load new input values (if necessary), new weights (if necessary), and set `ctrl[CTRL_RUN_BIT]` again to initiate the next inference.

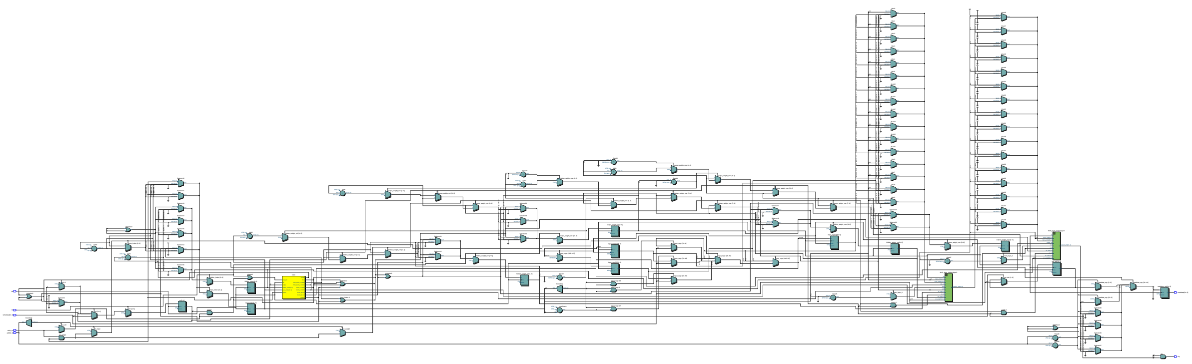

Figure 2.6: RTL of the Multi-Layer Perceptron.

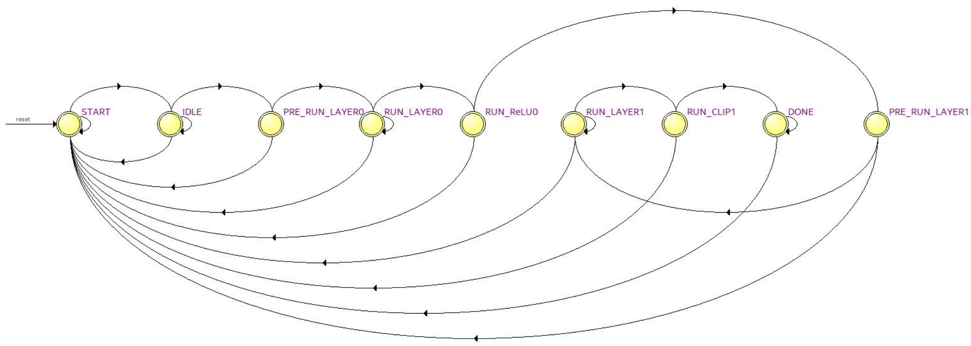

Figure 2.7: State transition diagram.

Chapter 3

Testbenches

During the design of the MLP network, the use of several testbenches was essential to verify the correct behavior of the system. Specifically, three of them focus on individual components of the MLP, testing all their functionalities and potential edge cases, while the final testbench evaluates the entire model by performing a full forward pass.

3.1 Weight memory testbench

This testbench applies a series of stimuli to the DUT (Device Under Test) inputs (`addr`, `wr_data`, `wr_en`) to verify the correct functionality of the memory module by testing its basic operations. It simulates memory usage through a sequence of write and read tests to ensure robustness, including the evaluation of edge cases. In specific, the tested operations are:

- writing data to specific addresses,
- reading data back from those addresses and checking for correctness,
- ensuring data persistence (data remains stored),
- checking boundary conditions (e.g., maximum address),
- verifying that no writes occur when the write enable (`wr_en`) signal is de-asserted,
- observing the behavior when reading from uninitialized memory locations.

All test cases completed successfully, confirming the correct operation of the module.

3.2 MAC testbench

This testbench is designed to verify the correct functionality of the Multiply-Accumulate (MAC) unit, `MLP_mac`. It applies a sequence of stimuli to the DUT inputs (`start`, `valid`, `a`, `b`) to simulate a multi-step accumulation process. The testbench is self-checking: it computes the expected result in parallel and compares it with the module's output at each cycle to ensure correctness.

The test sequence begins by verifying the initialization mechanism. When the first operation is triggered via the `start` signal, the testbench checks that the output corresponds solely to the product of the initial inputs. This confirms that the accumulator is correctly initialized without any residual value from a previous state. The verification then continues with three subsequent input pairs, driven by the `valid` signal. At each of these steps, the testbench confirms that the MAC unit correctly performs accumulation, ensuring the new product is added to the previously stored result. To ensure robust arithmetic handling, the simulation uses both positive and negative signed numbers throughout this process.

All test cases completed successfully, confirming the correct functionality of the multiply-accumulate unit.

3.3 Hidden layer testbench

This testbench aims to verify the complete functionality of the integrated hidden layer module, `MLP_layer_hidden`. It validates the entire operational flow, from loading the weights into the layer's weight memories to processing a full input vector and generating the final output. The testbench is self-checking: a SystemVerilog function, `calculate_expected_output_behavioral`, computes the expected output in parallel, allowing a direct comparison with the hardware results at the end of each computation cycle.

The test sequence is structured to cover several key scenarios. It begins by loading an initial set of weights into the layer's internal memories, simulating the module's configuration phase. Subsequently, it processes a complete input vector by streaming its components sequentially. Once the entire vector has been processed, the testbench triggers the activation function and output registration with the `relu_en` signal, then verifies the result. To ensure the module can handle multiple independent computations, a second test is performed using a new input vector but with the same weights, confirming that the `start` signal correctly resets the internal accumulators. Finally, the testbench demonstrates the layer's reconfigurability by loading a new set of weights and processing another input vector, specifically targeting edge cases such as ReLU activation and output value clipping.

Once again, all tests had a positive outcome.

3.4 MLP Inference testbench (top-level module)

This final testbench validates the entire MLP architecture by simulating a complete forward pass. It interacts with the main module by writing commands and data to specific addresses, closely mirroring how a processor would control the hardware accelerator in a real system. The test is data-driven, reading pre-defined inputs and weights from external files to configure the network.

The test sequence begins by loading all necessary data into the MLP: first the input vector, followed by the weights for both the hidden and output layers. This is done by sending the data to the correct internal addresses. Once the MLP is configured, the testbench starts the computation by setting a `run` bit in a control register. It then waits for the hardware to finish its calculations, which is indicated by a `done` signal. In parallel, the SystemVerilog function `calculate_mlp_behavioral` calculates the expected final result using the exact same inputs and weights. Upon completion, the testbench reads the output from the DUT and compares it against the value from the software model.

The successful comparison for the provided test case validates the correctness of the entire integrated design, from data input and weight loading to the final computational result.

Chapter 4

Further Studies

4.1 Impact of Increasing Neurons in the Hidden Layer

After verifying that our MLP worked correctly, we wanted to test the impact of adding more neurons to the hidden layer. We tried 10, 25 and 50 neurons. In Table 4.1 are shown the results in terms of training and test error.

It's interesting to note that for 10 neurons there was a large reduction in training error, of around 0.40, and a smaller reduction in test error, around 0.10. It's also worth noting that 25 and 50 neurons provided no meaningful improvement in the reduction of the error and the models took significantly longer to train. From figures 4.1 to 4.3 we can see that the model is able to create a more complex plot (i.e. with more intersecting hyperplanes) as the number of neurons increases.

Table 4.1: Mean Squared Error (MSE) Statistics over 5 Runs

	10 Neurons		**25 Neurons**		**50 Neurons**	
Metric	**Training MSE**	**Test MSE**	**Training MSE**	**Test MSE**	**Training MSE**	**Test MSE**
Mean	0.3861	1.1277	0.4410	1.1069	0.6639	1.1316
Std Dev	0.0948	0.1261	0.0949	0.1018	0.1719	0.0490
Min	0.2687	0.9743	0.2968	1.042	0.3605	1.0721
Max	0.4881	1.3162	0.5383	1.285	0.7689	1.1892

4.2 Future developments of the project

Even though our design is perfectly compatible with an FPGA, we were only able to simulate via software its behaviour. The next step would be to load this design on a real FPGA. The possibilities of developing hardware accelerators for neural networks are limitless: an idea could be to train a model on the MNIST dataset, load the model onto an FPGA equipped with a touch screen, and run on-the-fly digit recognition.

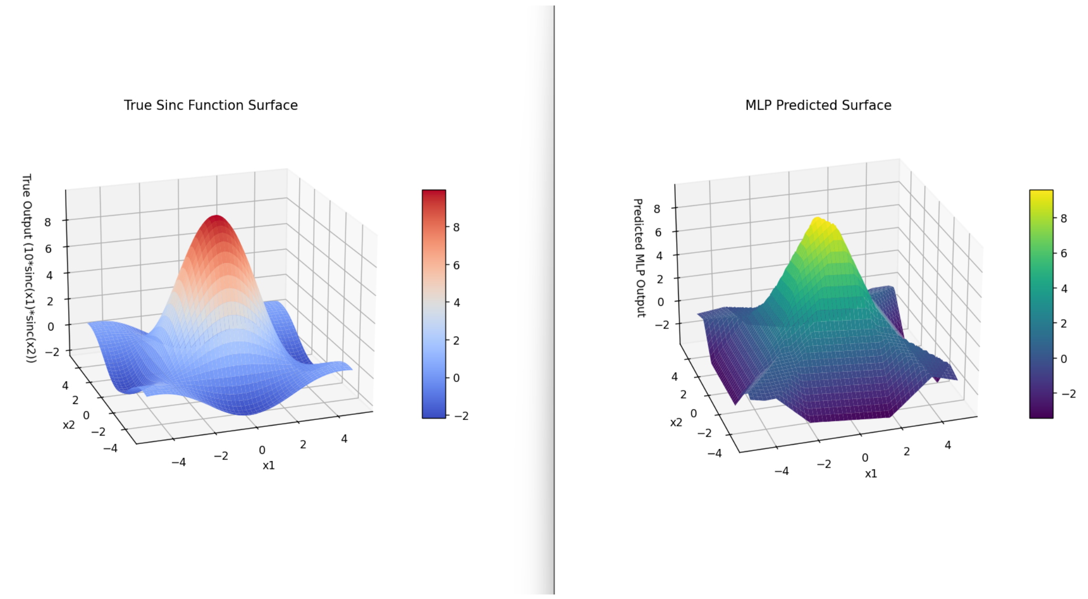

Figure 4.1: Plot Comparison - 10 Neurons.

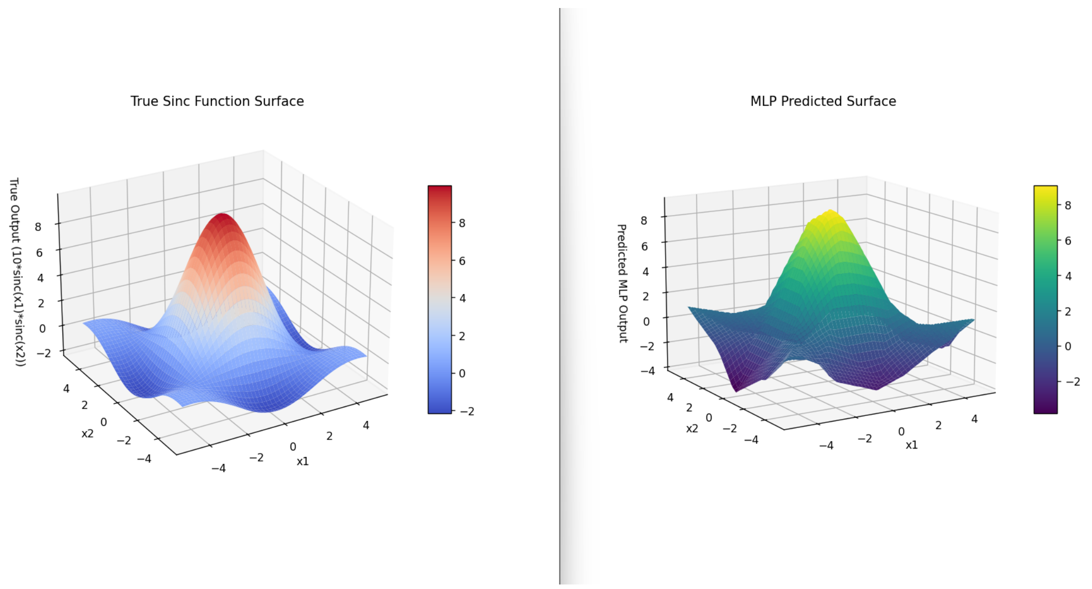

Figure 4.2: Plot Comparison - 25 Neurons.

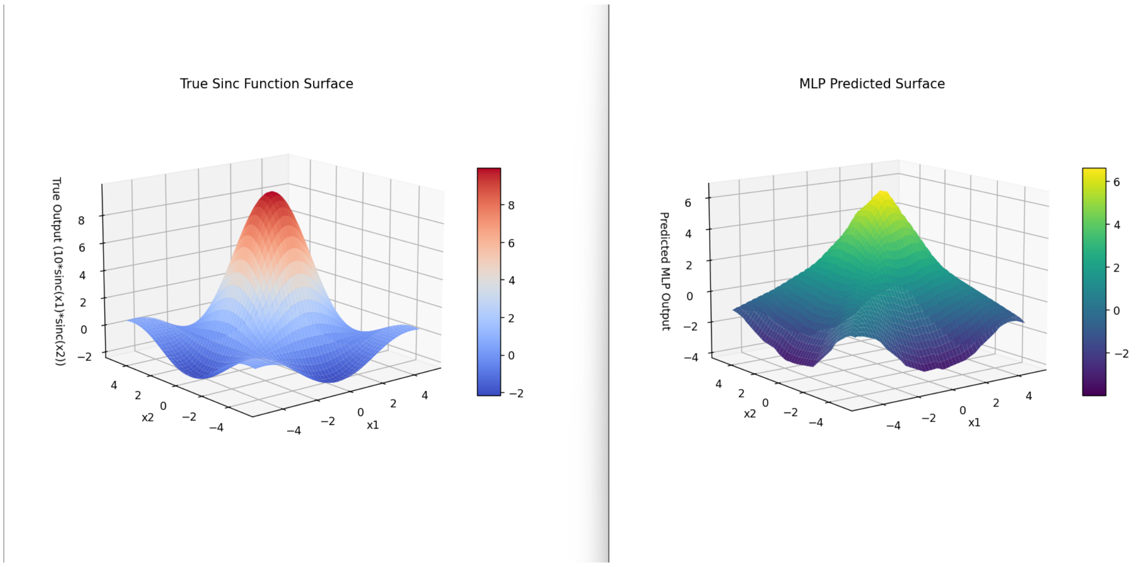

Figure 4.3: Plot Comparison - 50 Neurons.