Digital Integrated Circuit Design
ECEN 4303
Final Project Report

Michael Thompson
A20155065

Abstract

This report covers the design and implementation of a three layer, hardware based multilayer perception (MLP) neural network trained with the purpose of identifying handwritten numbers ranging from 0-9 by reading in 784 grayscale pixel values. To avoid the complexity of a hardware based back propagation implementation, the network is first created and trained in software using the PyTorch package in Python. Once trained, the weights and biases are extracted into memory files for later use in the hardware network. To create the hardware network, ModelSim is used to write and simulate SystemVerilog files. A constraint placed on this project is no multiplication or addition sign may be used. Instead, a ripple adder and an array multiplier must be implemented for this purpose. In the end, a network that can successfully identify eight out of ten hand written numbers was created. The process of making this network instills a deep understanding on how a MLP neural works.

Contents

Introduction

This project is to make a hardware implementation of a neural network that can interpret hand written numbers from 0-9. This neural network will have 3 layers: input layer, one hidden layer and an output layer.

The input layer consists of 784 nodes which correlate to a pixel of a picture. Each node of the input layer takes in a grayscale value (0-1) from their pixel. The hidden layer consists of 50 nodes that each are connected to all 784 nodes of the input layer via edges. Each edge has a settable weight value. The value of each hidden layer node is the summation of each input layer node, multiplied by the edge weight that connects the node, plus the bias of the hidden layer node, all normalized by the sigma or another normalization function. Each hidden layer node is then connected to each output layer node by more edges. The value of the output layer is calculated in the same way as the hidden layer.

The weights and biases are set using the back propagation algorithm. A known answer is used to create a cost function: (output node – answer)^2. The back propagation algorithm then sets the weights to minimize the cost function by following the gradient of the cost. To train this network, an input picture is sent through the network and the backpropagation algorithm sets the weights to minimize the cost function.

To complete this project, the network must be created and trained in python to obtain the edge weights and node biases. Once the edge weights and node biases are obtained, the network can be implemented in hardware using those weights. It must be done in this way to avoid the extra complexity of training the hardware.

Software (Python)

The Python model was made to be representative of the final hardware model. It is much easier to train a software-based model and then hardcode the trained weights and biases in the final hardware design. This way, the complicated process of back propagation can be regulated to software only.

The model is made of three layers: input layer, hidden layer, output layer. The input layer consists of 784 inputs passed directly to the hidden layer. The hidden layer consists of 50 nodes and the output layer consists of 10 nodes. The forwarding between the hidden layer and the output passes through a rectified linear unit (ReLU) activation function.

The model is trained using the MNIST training set using 3 epochs. After 3 epochs of training, the model is then tested using MNIST's built-in test set. This test resulted in an accuracy of 97.6%.

Once the model is trained and tested, the state dictionary is then exported to a file for future manipulation. Using a separate python script, the state dictionary file is accessed and the weight and bias tensors are extracted. To make the weights and biases usable, each number is put through fxpmath's "Fxp" function. The "Fxp" function converts a value from decimal to floating point using given criteria. Due to the Verilog memory file specification, each file must be a ".mem" file containing binary or hex. To adhere to this, the "tohex" function is made in conjunction with python's "hex" function. The "tohex" function sign extends the number to the desired length and then passes the sign extended number into the hex function which returns a hex string in the form of "0xXXXX". The replace function is then used to strip the "0x".

During the testing process 16-bit, 24-bit and 32-bit numbers were tried and therefore each input, weight and bias file was tagged with its number length. Because the input, weights and biases range from -1 to 1, the need for large integer values was not needed. Therefore 16-bit was chosen with 5-bits for the integer value and 11-bits for the fraction value. Once each number is converted, it is printed to a file correlated to its specific tensor. This processes results in 4 files:

- hidden_weight_16.mem
- hidden_bias_16.mem
- output_weight_16.mem
- output_bias_16.mem

The values are now ready to be put into the hardware implementation.

To test the final network, pictures are needed. The software network was trained with MNIST's training set. MNIST also contains a testing set that is separate from the training set. The first "set" of the training set consists of 10 pictures which are perfect for testing the final network. The same python script that extracts the weights and biases also extracts these testing pictures and uses the same process to convert the values from decimal to fixed point hex and export them to individual mem files. In addition, the actual values of the pictures are exported into an "answers.txt" to be used to compare the output of the network.

The python code is in Appendix A and B.

Verilog

Overview

To implement the model in SystemVerilog, multiple modules must first be implemented. The most important of these modules are the multiplication, addition and activation function modules. Because of the restriction of not using "+" or "*" addition and multiplication modules must be created using gates. Once these modules are created, they can then be combined to make a neuron, which then can be used to make a layer, which can finally make a network.

Modules

Every module, with the exception of the full adder, is parameterized so that the entire network can easily change the size of numbers and how many bits of those numbers are fractional bits. To make these parameterizations work, multiplication and subtraction signs are used. This is justified as once the network is created, the parameterized values are constant and do not change as the network propagates.

Full Adder

The first module to implement is a full adder that can be linked together to make a ripple adder. The full adder consists of the standard design using:

$$Sum = Cin \oplus (A \oplus B)$$

and

$$Cout = AB + (A \oplus B)Cin.$$

The code, test-bench and test results are located in appendix C, D and E.

Adder

The adder is a ripple adder consisting of the full adders. A test-bench was created to confirm the operation of the adder. The code, test-bench and test results are located in appendix F, G and H.

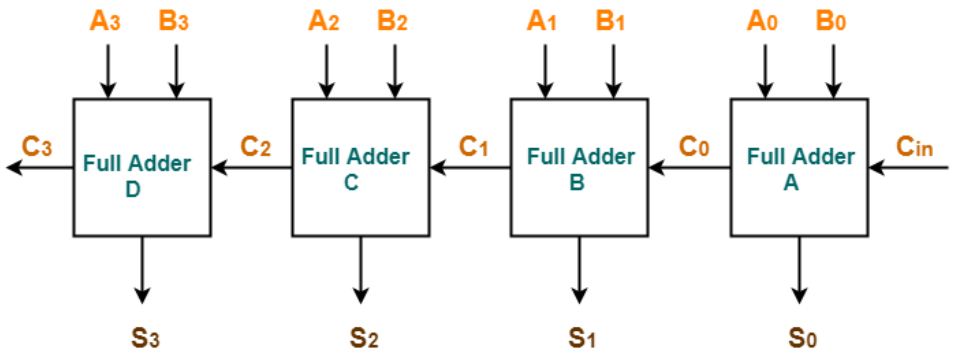


Figure 1: 4-bit Ripple Carry Adder

Multiplier

The multiplier is an array multiplier modeled after the figure below. However, the multiplier is expanded to allow for number size specification.

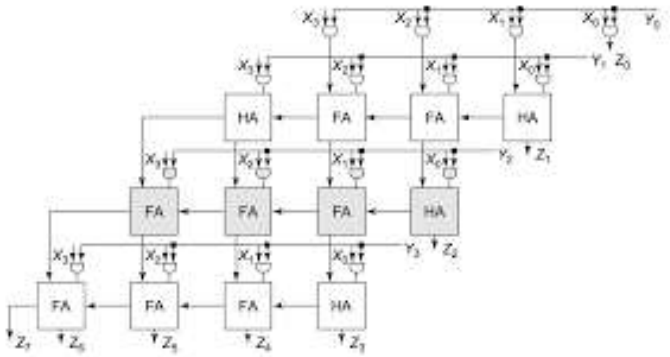


Figure 2: 4-bit Array Multiplier

This multiplier consists of multiple adders and a module that ANDs one bit to every bit in a number. This module is called bitAnd and can be found in appendix {}. To keep the fixed point at the correct location, the result of the multiplier is arithmetically shifted to the right the amount of bits that are fractional. A test-bench was created to confirm the operation of the multiplier.

The code, test-bench and test results are located in appendix I, J and K.

Activation

The ReLU activation module takes in a number and returns the same number if it is positive and 0 if the number is negative.

The code, test-bench and test results are located in appendix L, M and N.

Integration

With the modules complete, the network can be assembled.

Node

The first building block of the network is a neuron (referred to as a node in this report). The general design is shown below with only three inputs.

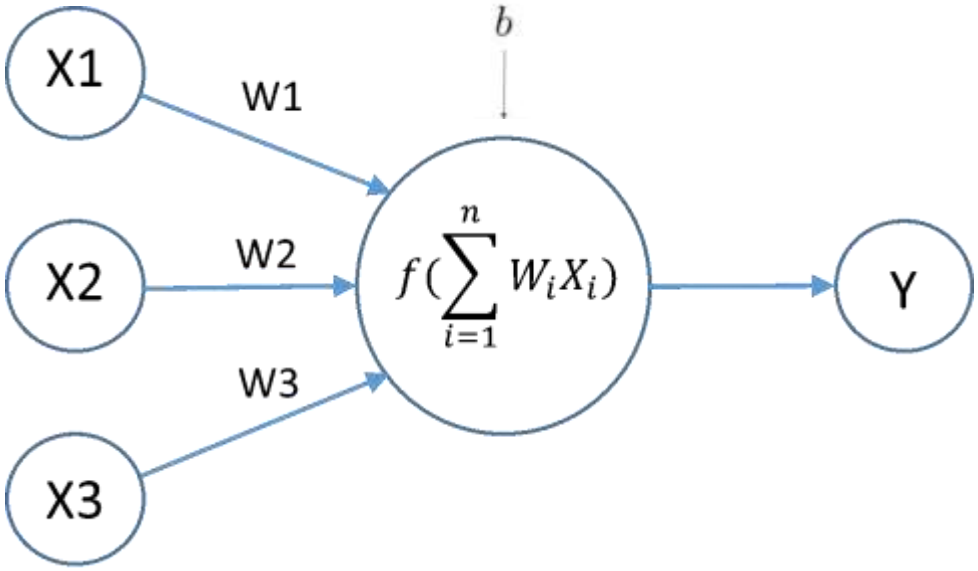


Figure 3: Node

The two major processes in a node are the multiplication of weights (W) with the inputs (X) and then the adding of all weights together with the bias (b). The resulting value is then passed to the output (Y).

There are two types of nodes for this particular network: hidden layer node and output layer node. The hidden layer nodes will take in the hand written value in the form of 784 grayscale pixels. Therefore, this node has 784 inputs. The hidden layer outputs 50 values to the output layer, resulting in each output node taking in 50 inputs. To simplify the system, the node is parameterized with the amount of inputs it needs to take in. This results in the same node file being used for both types of nodes just with different parameter values.

Originally, the node was designed as a purely combinational module using a multiplier for each input and a tree adder to sum all weighted inputs and the bias. However, with the adders being ripple adders and the multipliers being array multipliers, a combinational version will very quickly take up over 25G of RAM resulting in the crashing of ModelSim.

To solve this problem, a sequential version was designed using one multiplier and three adders. Each input is multiplied with its weight. This product is then added to a sum register that is constantly fed back into the same adder (right).

To determine how many additions, need to be done, a counter is required. This counter is an accumulator like the lower adder (right) but

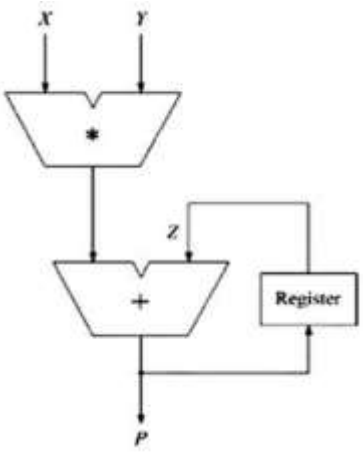


Figure 4: Multiplier into Accumulator

instead of the multiplier going into the adder, a one is added. This results in a counter that increments on every clock cycle. The final adder is used to add the bias to the sum before it is passed to the output.

This design does a multiplication every clock cycle which means that after the amount of inputs worth of clock cycles, the output will be ready. Because of this a ready signal is triggered after that amount of clock cycles. The final aspect of the node is the start signal. With the nodes being chained together, the node should not start calculating until the previous node's output is ready. Linking the previous node's ready signal to the current node's start signal solves this problem.

The code, test-bench and test results for the node are in appendix O, P, Q and R.

Layer

With a node implemented, a layer consisting of multiple nodes can be implemented. The layer is parameterized with the amount of inputs and outputs to the node. The amount of nodes is the same as the output size. Each node takes in the same inputs but each node gets unique weights and biases. Each layer takes in all the weights and biases and then pass one bias and one set of weights, the size of the input to each node. The output of the layer consists of the output of each of the nodes, which are each passed through the ReLU activation function. This includes the output layer, because we don't care about negative values when looking at the output. Because each node in a layer takes the same amount of clock cycles to output, the ready signal from one node can be passed out of the layer as the layer's ready signal. Similarly, the start signal that comes into the layer is connected to each individual node so that the nodes all start at the same time.

The code, test-bench and test results for the layer are in appendix S, T, U and V.

Network

The network consists of two layers. The first layer is parameterized with an in size of 784 and an out size of 50. The second layer is parameterized with an in size of 50 and out size of 10. These are the hidden and output layers. The weights and biases gathered from the software network are in files separated by layer. Therefore, the network can load each set of weights and biases into memory that can be passed to each layer. The hidden layer's ready signal and output are connected directly to the output layer's input and start signal ports respectively. The output's output and ready signal are passed to the networks output and ready signal. The network itself takes in the input and a start signal. This input and start signal are passed directly into the hidden layer.

The code and test-bench for the network are in appendix W and X. The results are figure 5 in the following section.

Complete Network

With this design, the input and start signal are passed into the network which proceeds to process the input in the hidden layer. Once the hidden layer is done processing, the output is passed into the output layer to be processed. Once the output layer is done processing, the network outputs a ready signal and the ten values of the output.

The following simulation results show the output of the network when a set of ten pictures that the system did not train with are inputted.

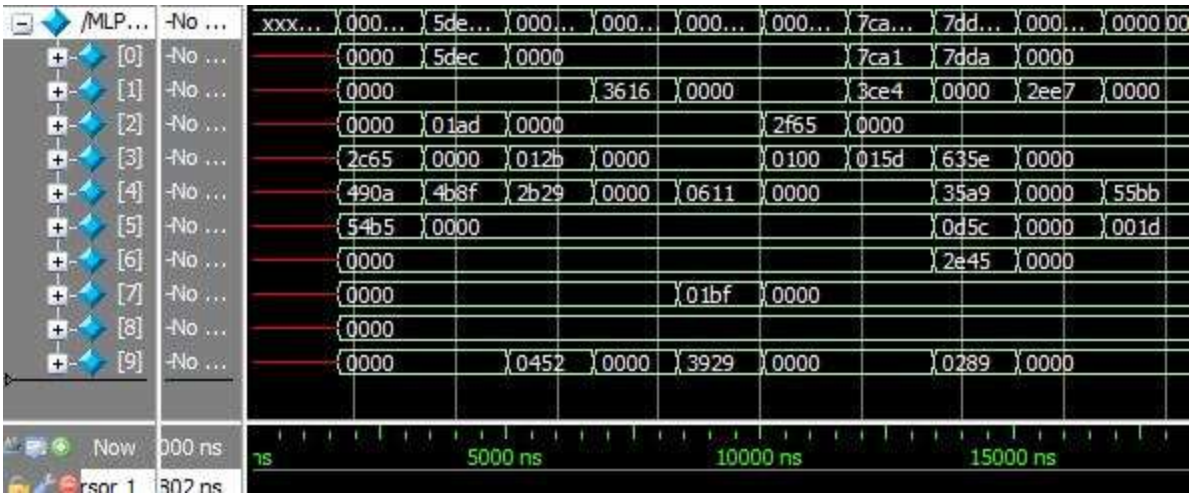


Figure 5: Multilayer Perception Unit Neural Network Results

The system shows its 0-9 prediction by the largest number in the output set. For example, the first picture's output with the largest value is 5. This means the system thinks that the picture is of a 5.

The table below shows the interpreted results and the actual picture values.

Picture number	1	2	3	4	5	6	7	8	9	10
Picture Value	5	0	4	1	9	2	1	3	1	4
Network's Best Prediction	5	0	4	1	9	2	0	0	1	4
Network's Second Best Prediction	4	4	9	N/A	4	3	1	3	N/A	5

Figure 6: Table of Multilayer Perception Neural Network interpreted results and actual picture values

Testing this network with a set of ten pictures that the system trained yields an 80% accuracy. However, when looking at the failed predictions, the second highest prediction is correct. This shows that the general trend of predictions is as desired even if some final predictions are off.

Conclusion

Throughout this process of making a network, I went from knowing what a neural network does to understanding how each little part functions. I had to use the knowledge I learned from ECEN 4303 to make the adders and multipliers. In addition, I had to do in depth research on how a node works and how nodes are connected to make layers and how layers are connected to make a network. In the end I saw the versatility of parameterized nodes and layers. If the nodes and layers could be any size, then any MLP network could be created using these modules. Anything, from a single-layer network to a 100-layer network, can easily be made using my modules. I learned a lot from this project.

References

Singhal, Akshay. "Akshay Singhal." Gate Vidyalay, Publisher Name Gate Vidyalay Publisher Logo, 29 Aug. 2019, https://www.gatevidyalay.com/ripple-carry-adder/.

ESE570, Spring 2018 HW 7: 4x4 Array Multiplier Thursday ... https://www.seas.upenn.edu/~ese570/spring2018/handouts/hw7_mult.pdf.

Anomaly, Aiden, et al. "Deep Neural Network's Precision for Image Recognition, Float or Double?" Stack Overflow, 1 Jan. 1965, https://stackoverflow.com/questions/40537503/deep-neural-networks-precision-for-image-recognition-float-or-double.

Puttam, Krishnaprasad, et al. "Implementation of Low Power and High Speed Multiplier-Accumulator Using SPST Adder and Verilog." Semantic Scholar, 1 Jan. 1970, https://www.semanticscholar.org/paper/Implementation-of-Low-Power-and-High-Speed-Using-Puttam-Rao/188f9a6924b88f8851e063555d1bb9e6e2cbfab5.

Appendices

Appendix A: Python MLP

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import transforms, datasets
import torch.optim as optim
import pickle

train = datasets.MNIST("", train=True, download=True, transform =
transforms.Compose([transforms.ToTensor()]))

test = datasets.MNIST("", train=True, download=False, transform =
transforms.Compose([transforms.ToTensor()]))

trainset = torch.utils.data.DataLoader(train, batch_size=10, shuffle=True)
testset = torch.utils.data.DataLoader(test, batch_size=10, shuffle=True)

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(784, 50)
        self.output = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(self.hidden(x))
        x = self.output(x)

        return F.log_softmax(x, dim=1)

net = Net()
X = torch.rand((28,28))
X = X.view(-1, 28*28)

output = net(X)

optimizer = optim.Adam(net.parameters(), lr=0.001)

EPOCHS = 3

for epoch in range(EPOCHS):
    for data in trainset:
        #data is a batch of featuresets and lavels
        X, y = data
        net.zero_grad()
```

```
        output = net(X.view(-1, 28*28))
        loss = F.nll_loss(output, y)
        loss.backward()
        optimizer.step()
    print(loss)

correct = 0
total = 0
with torch.no_grad():
    for data in testset:
        X, y = data
        output = net(X.view(-1, 784))
        for idx, i in enumerate(output):
            if torch.argmax(i) == y[idx]:
                correct += 1
            total += 1
print("Accuracy: ", round(correct/total,3))

print()

print("Model's state_dict:")
for param_tensor in net.state_dict():
    print(param_tensor, "\t", net.state_dict()[param_tensor].size())

with open('MLP.state_dict','wb') as stateFile:
    pickle.dump(net.state_dict(), stateFile)
```

Appendix B: Python MLP data Extraction

```
import pickle
import torch
from fxpmath import Fxp
import torch
from torchvision import transforms, datasets
import pickle


def tohex(val, nbits):
    return hex((val + (1 << nbits)) % (1 << nbits))

state_dict = None

bits = 16
frac_bits = 11

with open("MLP.state_dict","rb") as stateFile:
    state_dict = pickle.load(stateFile)
    last = index = 0
    for param_tensor in state_dict:
        print(param_tensor, "\t", state_dict[param_tensor].size())
        fileName = param_tensor.replace(".","_") + "_" + str(bits) + ".mem"
        with open(fileName, "w") as out:
            for param in state_dict[param_tensor]:
                if (param.size() != torch.Size([])):
                    for subParam in param:
                        val = tohex(Fxp(subParam.item(), n_int=bits-frac_bits
,n_frac=frac_bits, signed=True).raw(), bits)
                        out.write(str(val).replace("0x","")+" ")
                else:
                    val = tohex(Fxp(param.item(), n_int=bits-frac_bits
,n_frac=frac_bits, signed=True).raw(), bits)
                    out.write(str(val).replace("0x","")+"\n")

test = datasets.MNIST("", train=True, download=False, transform =
transforms.Compose([transforms.ToTensor()]))
testset = torch.utils.data.DataLoader(test, batch_size=10, shuffle=False)
count = 1
amount = 1
index = 0
photoSet = []
with torch.no_grad():
    for data in testset:
```

```
        X, y = data
        pic = X.view(-1, 784)
        array = []
        for param in pic:
            for subParam in param:
                if not index<784:
                    photoSet.append(array)
                    array = []
                    index = 0
                val = tohex(Fxp(subParam.item(), n_int=bits-frac_bits
,n_frac=frac_bits, signed=True).raw(), bits)
                array.append(str(val).replace("0x",""))
                index += 1
        photoSet.append(array)
        count += 1
        if (count > amount): break

    index = 0
    with open("answers.txt", "w") as answers:
        for photo in photoSet:
            index += 1
            fileName = "picture_" + str(index) + "_" + str(bits) + ".mem"
            answers.write(str(y[index-1].item())+" ")
            print(str(y[index-1].item())+" " + fileName)
            with open(fileName, "w") as out:
                for pixel in photo:
                    out.write(pixel + " ")
```

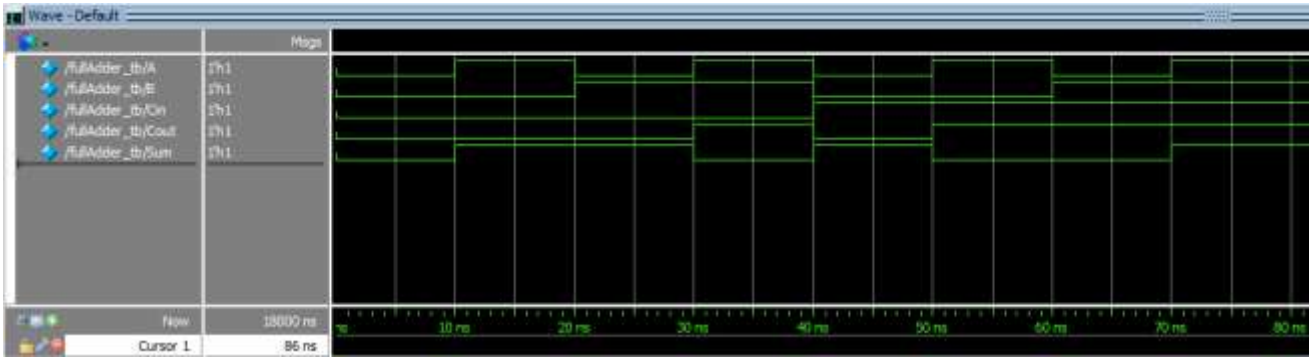
Appendix C: Full Adder SystemVerilog Code

```
module fullAdder(input A, input B, input Cin, output Cout, output Sum);
    assign Sum = Cin^(A^B);
    assign Cout = (A&B)|((A^B)&Cin);
endmodule
```

Appendix D: Full Adder Test Bench Code

```
module fullAdder_tb;
    logic A;
    logic B;
    logic Cin;
    logic Cout;
    logic Sum;

    fullAdder fullAdder(.A(A), .B(B), .Cin(Cin), .Cout(Cout), .Sum(Sum));

    initial begin
        assign A = 0;
        assign B = 0;
        assign Cin = 0;
        #10
        assign A = 1;
        assign B = 0;
        assign Cin = 0;
        #10
        assign A = 0;
        assign B = 1;
        assign Cin = 0;
        #10
        assign A = 1;
        assign B = 1;
        assign Cin = 0;
        #10
        assign A = 0;
        assign B = 0;
        assign Cin = 1;
        #10
        assign A = 1;
        assign B = 0;
        assign Cin = 1;
        #10
        assign A = 0;
        assign B = 1;
        assign Cin = 1;
        #10
        assign A = 1;
        assign B = 1;
        assign Cin = 1;
        #10
  end endmodule
```

Appendix E: Full Adder Test Bench Results



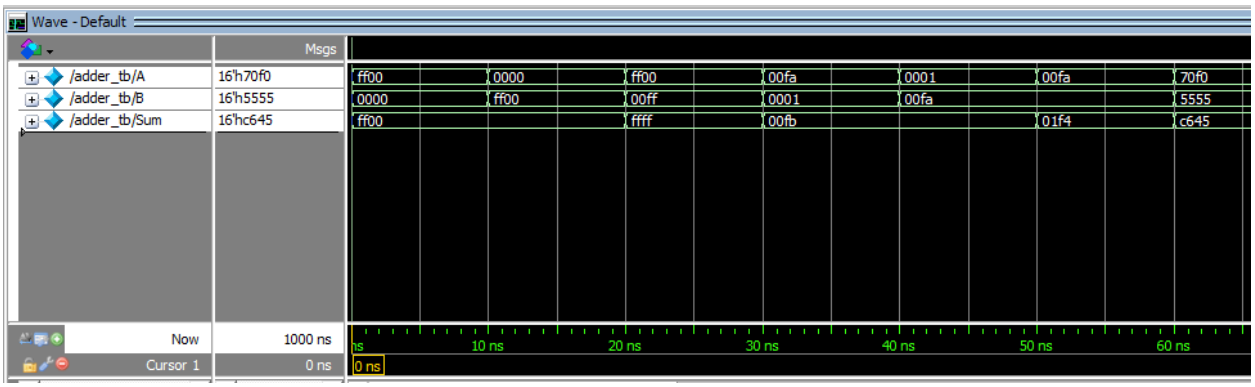
Appendix F: Adder SystemVerilog Code

```
module adder #(parameter bits)(
    input signed [bits-1:0] A,
    input signed [bits-1:0] B,
    input Cin=0,
    output signed [bits-1:0] Sum,
    output Cout
    );

    reg signed [bits:0] C_;
    reg signed [bits-1:0] buffered_Sum;
    assign C_[0] = Cin;

    genvar i;
    generate
        for(i = 0; i < bits; i=i+1) begin : adders
            fullAdder fullAdder(.A(A[i]), .B(B[i]), .Cin(C_[i]), .Cout(C_[i+1]),
.Sum(buffered_Sum[i]));
        end
    endgenerate

    assign Cout = C_[bits];
    assign Sum = buffered_Sum;
endmodule
```

Appendix G: Adder Test Bench Code

```
module adder_tb;
    logic [15:0] A;
    logic [15:0] B;
    logic [15:0] Sum;

    adder #(.bits(16)) adder(.A(A), .B(B), .Sum(Sum));

    initial
        begin
        assign A = 'hFF00;
        assign B = 0;
        #10
        assign A = 0;
        assign B = 'hFF00;
        #10
        assign A = 'hFF00;
        assign B = 'h00FF;
        #10
        assign A = 250;
        assign B = 1;
        #10
        assign A = 1;
        assign B = 250;
        #10
        assign A = 250;
        assign B = 250;
        #10
        assign A = 'h70F0;
        assign B = 'h5555;
        end

endmodule
```

Appendix H: Adder Test Bench Results



Appendix I: Multiplier SystemVerilog Code

```
module multiplier #(parameter bits, fractional_bits=0)(
    input signed [bits-1:0] A,
    input signed [bits-1:0] B,
    output signed [bits-1:0] Product
    );

    logic signed [bits-1:0] A_logic;
    logic signed [bits-1:0] B_logic;
    logic signed [bits*2-1:0] temp;
    logic signed [bits*2-1:0] shift;
    logic signed [bits-1:0] previous [0:bits-1];
    logic signed [bits-1:0] setup;

    assign sign = (A<0 || B<0) && ~(A<0 && B<0);
    assign A_logic = (A>0) ? A : -A;
    assign B_logic = (B>0) ? B : -B;

    bitAnd #(.bits(bits)) bitAnd(.X(B_logic),.Y(A_logic[0]), .Out(setup));
    assign temp[0] = setup[0];
    assign previous[0] = setup>>1 ;

    genvar i;
    generate
        for(i = 1; i<bits; i=i+1) begin : multiplier
            multiplier_layer #(.bits(bits)) multiplier_layer(.B(B_logic),
.A_bit(A_logic[i]), .last(previous[i-1]),.result(previous[i]),
.out_bit(temp[i]));
        end
    endgenerate

    assign temp[bits*2-1:bits] = previous[bits-1];
    assign shift = (sign == 0) ? temp : -temp;
    assign Product = shift>>>fractional_bits;

endmodule


module multiplier_layer #(parameter bits) (
    input signed [bits-1:0] last,
    input signed [bits-1:0] B,
    input A_bit,
    output out_bit,
    output signed [bits-1:0] result
```

```
    );

    logic C;
    logic signed [bits-1:0] sum;
    logic signed [bits-1:0] transfer;

    bitAnd #(.bits(bits)) bitAnd(.X(B), .Y(A_bit), .Out(transfer));
    adder #(.bits(bits)) adder(.A(last),.B(transfer),.Sum(sum),.Cout(C));

    assign result[bits-2:0] = sum[bits-1:1];
    assign result[bits-1] = C;
    assign out_bit = sum[0];
endmodule
```

Appendix J: Multiplier Test Bench Code

```
module multiplier_tb;
    logic signed [15:0] A;
    logic signed [15:0] B;
    logic signed [15:0] Product;
    logic signed [31:0] shift;
    logic signed [15:0] Actual;

    multiplier #(.bits(16), .fractional_bits(11)) multiplier(.A(A), .B(B),
.Product(Product));

    initial
        begin

        assign A = 65279; //254
        assign B = 128;   //.2
        assign shift = A*B;
        assign Actual = shift>>>11;
        #10
        assign A = 128;   //.2
        assign B = 65280; //255
        assign shift = A*B;
        assign Actual = shift>>>11;
        #10
        assign A = 5000;
        assign B = -4;
        assign shift = A*B;
        assign Actual = shift>>>11;
        #10
        assign A = -4;
        assign B = 5000;
        assign shift = A*B;
        assign Actual = shift>>>11;
        #10
        assign A = 16;
        assign B = 0;
        assign shift = A*B;
        assign Actual = shift>>>11;
        #10
        assign A = 0;
        assign B = 16;
        assign shift = A*B;
        assign Actual = shift>>>11;
        #10
```

```
        assign A = 65280;
        assign B = -256;
        assign shift = A*B;
        assign Actual = shift>>>11;
        #10
        assign A = 250;
        assign B = 250;
        assign shift = A*B;
        assign Actual = shift>>>11;
    end

endmodule
```

Appendix K: Multiplier Test Bench Results

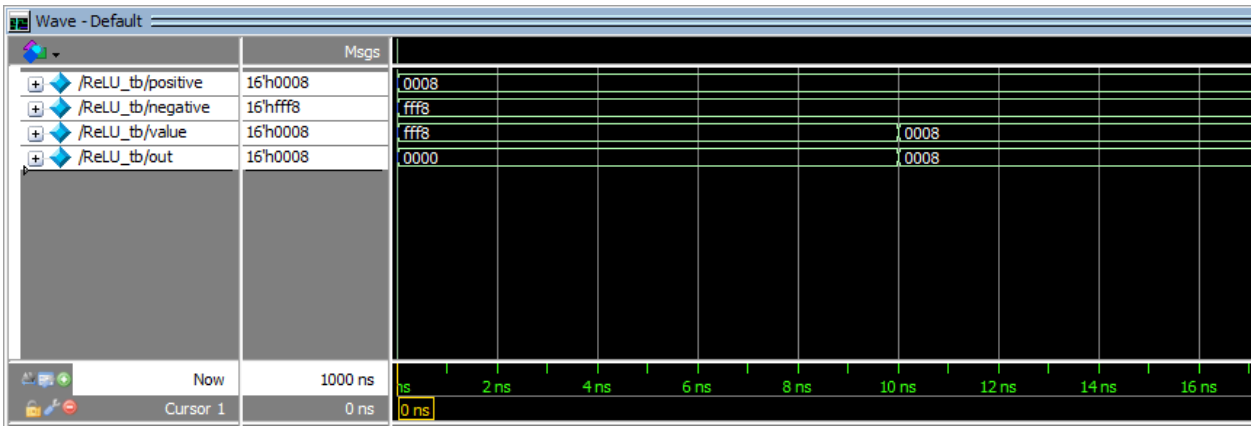
Appendix L: ReLU SystemVerilog Code

```
module ReLU #(parameter bits)(input signed [bits-1:0] in,
    output signed [bits-1:0] out);

    assign out = (in>0) ? in : 0;
endmodule
```

Appendix M: ReLU Test Bench Code

```
module ReLU_tb;
    reg signed [15:0] positive;
    reg signed [15:0] negative;

    reg signed [15:0] value;
    reg signed [15:0] out;

    ReLU #(.bits(16)) ReLU(.in(value), .out(out));

    assign positive = 8;
    assign negative = -8;

    initial begin
        assign value = negative;
        #10
        assign value = positive;

    end
endmodule
```

Appendix N: ReLU Test Bench Results



Appendix O: Node SystemVerilog Code

```
module node #(parameter bits, fractional_bits, in_size)(
    input signed [bits-1:0] in [0:in_size-1],
    input signed [bits-1:0] bias,
    input signed [bits-1:0] weights [0:in_size-1],
    input clock,
    input start,
    output reg ready,
    output reg signed [bits-1:0] out
    );

    reg signed [bits-1:0] sum;
    reg signed [bits-1:0] sumIn;
    reg signed [bits-1:0] sumOut;
    reg [bits-1:0] count;
    reg [bits-1:0] countOut;
    reg signed [bits-1:0] weight_var;
    reg signed [bits-1:0] weighted_var;
    reg signed [bits-1:0] input_var;
    reg triggered;
    reg signed [bits-1:0] one;

    multiplier #(.bits(bits), .fractional_bits(fractional_bits))
multiplier_instance (
            .A(input_var),
            .B(weight_var),
            .Product(weighted_var)
            );
    adder #(.bits(bits)) adder(.A(sumIn), .B(weighted_var), .Sum(sum));
    adder #(.bits(bits)) counter(.A(count), .B(one), .Sum(countOut));
    adder #(.bits(bits)) addBias(.A(sumOut), .B(bias), .Sum(out));
    assign one = 1;

    initial begin
        count = in_size;
        sumIn = 0;
        triggered = 1;
    end

    always @(posedge clock or posedge start or negedge start) begin
        if (count < in_size) begin
            ready <= 0;
            input_var <= in[count];
            weight_var <= weights[count];
```

```
            sumIn <= sum;
            count <= countOut;
            triggered <= 0;
        end
        else if (~triggered) begin
            sumOut <= sum;
            ready <= 1;
            triggered <= 1;
        end
        else if ((count >= in_size) && (start)) begin
            count <= 0;
            sum <= 0;
            sumIn <= 0;
        end
        else begin
            ready <= 0;
        end
    end
endmodule
```

Appendix P: Hidden Node Test Bench Code

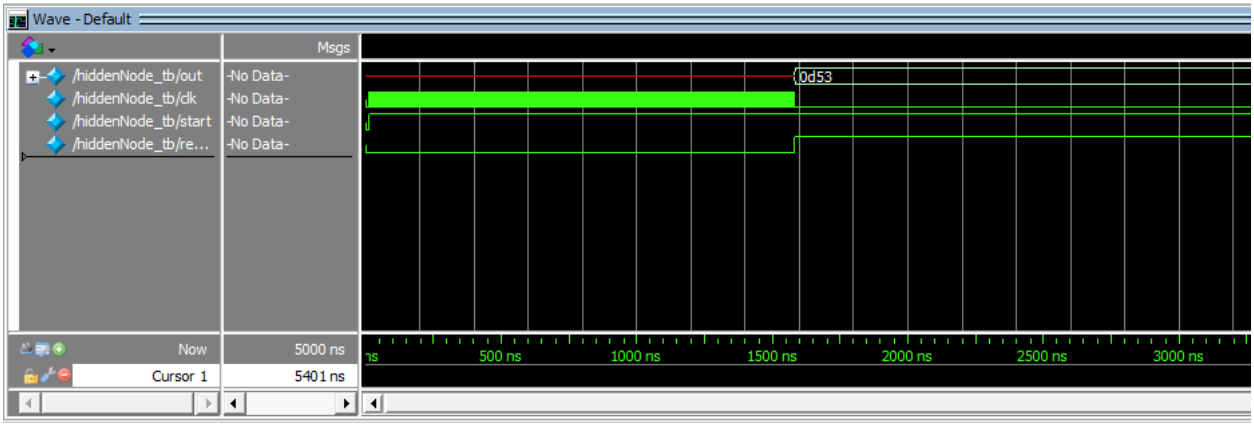
```
module hiddenNode_tb;

    reg signed [15:0] test_memory [0:783];
    reg signed [15:0] in [0:783];
    reg signed [15:0] out;

    reg signed [15:0] weights [0:39199];
    reg signed [15:0] bias [0:49];
    reg signed [15:0] weight [0:783];
    reg signed [15:0] bia;

    reg clk;
    reg start;
    reg ready;

    node #(.bits(16), .fractional_bits(11), .in_size(784)) hiddenNode (
        .start(start),
        .clock(clk),
        .in(in),
        .ready(ready),
        .out(out),
        .bias(bia),
        .weights(weight)
        );

    initial begin
        clk = 0;
        start = 0;
        ready = 0;
        $readmemh("picture_1_16.mem", test_memory);
        $readmemh("hidden_weight_16.mem", weights);
        $readmemh("hidden_bias_16.mem", bias);
        assign in = test_memory;
        assign weight = weights[0:783];
        assign bia = bias[0];
        #10
        start = 1;
        while (~ready) begin
            #1 clk = ~clk;
        end
    end
endmodule
```

Appendix Q: Output Node Test Bench Code

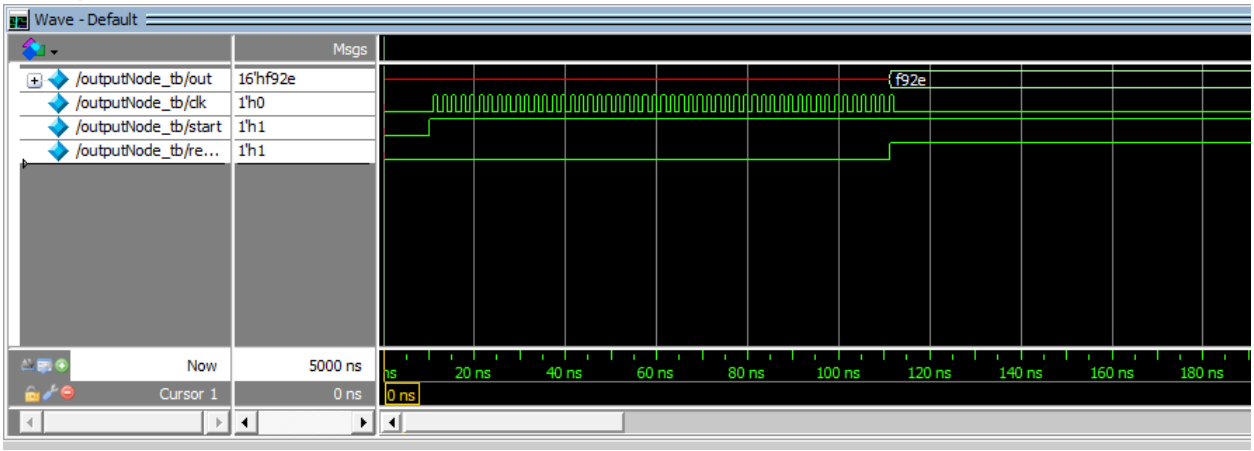
```
module outputNode_tb;


    reg signed [15:0] test_memory [0:49];
    for (genvar i = 0; i<50; i=i+1) assign test_memory[i] = 'h100;


    reg signed [15:0] in [0:49];
    reg signed [15:0] out;
    reg signed [15:0] weights [0:499];
    reg signed [15:0] bias [0:9];
    reg signed [15:0] weight [0:49];
    reg signed [15:0] bia;
    reg clk;
    reg start;
    reg ready;

    node #(.bits(16), .fractional_bits(11), .in_size(50)) outputNode (
        .start(start),
        .clock(clk),
        .in(in),
        .ready(ready),
        .out(out),
        .bias(bia),
        .weights(weight)
        );

    initial begin
        clk = 0;
        start = 0;
        ready = 0;
        $readmemh("output_weight_16.mem", weights);
        $readmemh("output_bias_16.mem", bias);
        assign in = test_memory;
        assign weight = weights[0:49];
        assign bia = bias[0];
        #10
        start = 1;
        while (~ready) begin
            #1 clk = ~clk;
        End end
endmodule
```

Appendix R: Hidden and Output Node Test Bench Results

Hidden



Output



Appendix S: Layer SystemVerilog Code

```
module layer #(parameter bits, fractional_bits, in_size, out_size)(
    input signed [bits-1:0] in [0:in_size-1],
    input clock,
    input start,
    input signed [bits-1:0] weights [0:in_size*out_size-1],
    input signed [bits-1:0] biases [0:out_size-1],
    output reg signed [bits-1:0] out [0:out_size-1],
    output ready
    );

    reg signed [bits-1:0] to_activation [0:out_size-1];
    reg [out_size-1:0] allReady;

    genvar i;
    generate
        for (i = 0; i<out_size; i=i+1) begin : output_layer
            node #(.bits(bits), .fractional_bits(fractional_bits),
.in_size(in_size)) node(
                    .start(start),
                    .clock(clock),
                    .in(in),
                    .ready(allReady[i]),
                    .out(to_activation[i]),
                    .bias(biases[i]),
                    .weights(weights[in_size*i:in_size-1+in_size*i]));
            ReLU #(.bits(bits)) ReLU(.in(to_activation[i]), .out(out[i]));
        end
    endgenerate
    assign ready = &allReady;
endmodule
```

Appendix T: Hidden Layer Test Bench Code

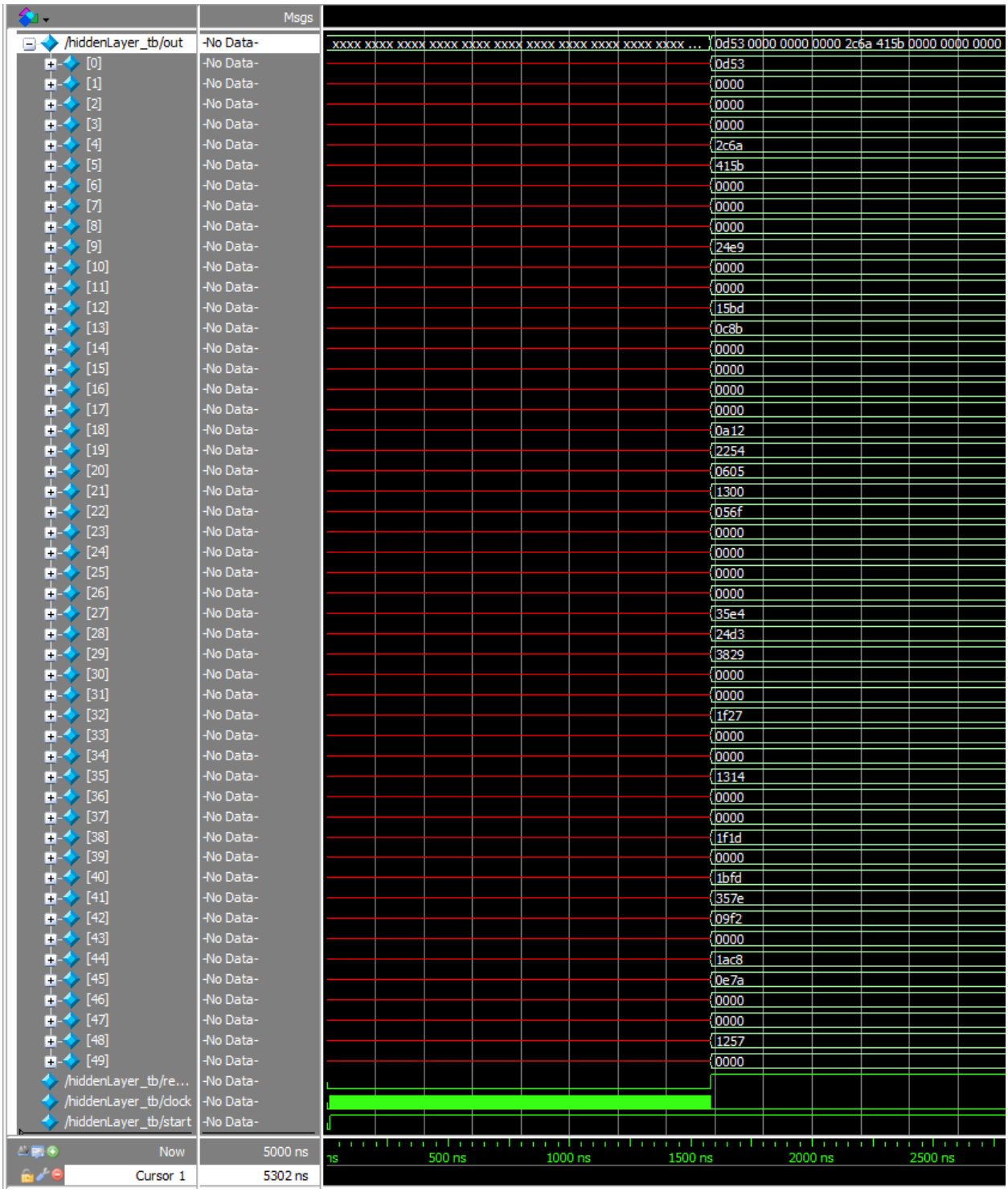
```
module hiddenLayer_tb;
    reg signed [15:0] picture1 [0:783];

    reg signed [15:0] out [0:49];
    reg signed [15:0] picture [0:783];
    reg ready;
    reg clock;
    reg start;

    reg signed [15:0] hidden_weights [0:39199];
    reg signed [15:0] hidden_biases [0:49];

    layer #(.bits(16), .fractional_bits(11), .in_size(784), .out_size(50))
hiddenLayer(
                .start(start),
                .clock(clock),
                .in(picture),
                .biases(hidden_biases),
                .weights(hidden_weights),
                .out(out),
                .ready(ready));

    initial begin
        ready = 0;
        clock = 0;
        start = 0;
        $readmemh("picture_1_16.mem", picture1);
        $readmemh("hidden_weight_16.mem", hidden_weights);
        $readmemh("hidden_bias_16.mem", hidden_biases);
        assign picture = picture1;
        #10
        start = 1;
        while (~ready) begin
            #1 clock = ~clock;
        end
    end
endmodule
```

Appendix U: Output Layer Test Bench Code

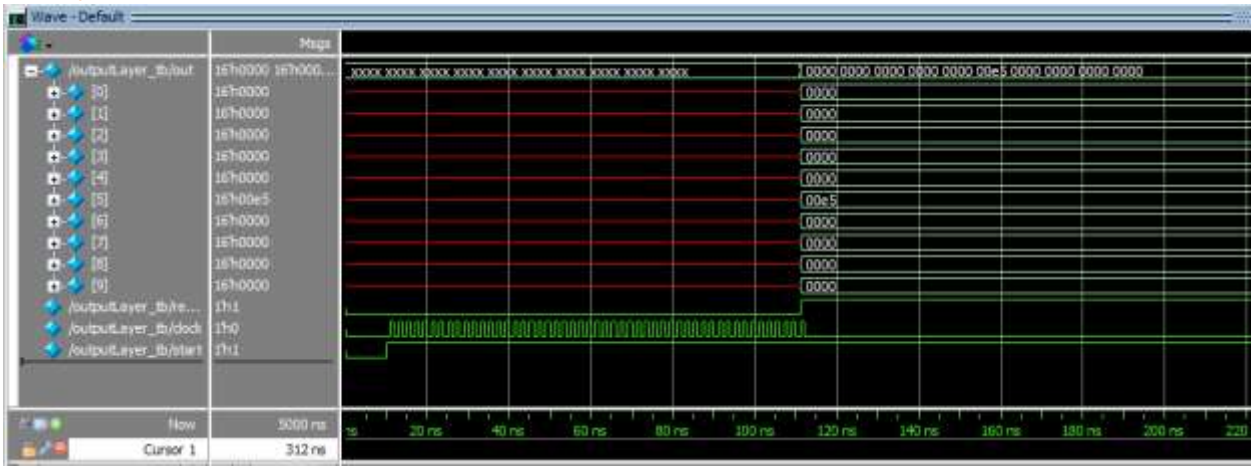
```
module outputLayer_tb;
    reg signed [15:0] test1 [0:49];
    for (genvar i = 0; i<50; i=i+1) assign test1[i] = 'h100;

    reg signed [15:0] out [0:9];
    reg signed [15:0] test [0:49];
    reg ready;
    reg clock;
    reg start;

    reg signed [15:0] output_weights [0:499];
    reg signed [15:0] output_biases [0:9];

    layer #(.bits(16), .fractional_bits(11), .in_size(50), .out_size(10))
outputLayer(
                .start(start),
                .clock(clock),
                .in(test),
                .biases(output_biases),
                .weights(output_weights),
                .out(out),
                .ready(ready));

    initial begin
        ready = 0;
        clock = 0;
        start = 0;
        $readmemh("output_weight_16.mem", output_weights);
        $readmemh("output_bias_16.mem", output_biases);
        assign test = test1;
        #10
        start = 1;
        while (~ready) begin
            #1 clock = ~clock;
        end
    end
endmodule
```

Appendix V: Hidden and Output Layer Test Bench Results

Hidden



Output


Wave - Default
Msgs
/outputLayer_tb/out
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
/outputLayer_tb/re...
/outputLayer_tb/clock
/outputLayer_tb/start
16'h0000
16'h00e5
1'h1
1'h0
Now
5000 ns
Cursor 1
312 ns
0000
00e5
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
180 ns
200 ns

Appendix W: Network SystemVerilog Code

```
module MLP_Network #(parameter bits, fractional_bits) (
    input start,
    input clock,
    input signed [bits-1:0] picture [0:783],
    output ready,
    output signed [bits-1:0] results [0:9]
    );

    reg signed [bits-1:0] transfer [0:49];
    reg hiddenReady;
    reg outputReady;

    reg signed [bits-1:0] hidden_weights [0:39199];
    reg signed [bits-1:0] hidden_biases [0:49];
    reg signed [bits-1:0] out_weights [0:499];
    reg signed [bits-1:0] out_biases [0:9];

    initial begin
        $readmemh($sformatf("hidden_weight_%0d.mem",bits), hidden_weights);
        $readmemh($sformatf("hidden_bias_%0d.mem",bits), hidden_biases);
        $readmemh($sformatf("output_weight_%0d.mem",bits), out_weights);
        $readmemh($sformatf("output_bias_%0d.mem",bits), out_biases);
    end
    layer #(.bits(bits), .fractional_bits(fractional_bits), .in_size(784),
.out_size(50)) hiddenLayer(
                .start(start),
                .clock(clock),
                .in(picture),
                .biases(hidden_biases),
                .weights(hidden_weights),
                .out(transfer),
                .ready(hiddenReady));
    layer #(.bits(bits), .fractional_bits(fractional_bits), .in_size(50),
.out_size(10)) outputLayer(
                .start(hiddenReady),
                .clock(clock),
                .in(transfer),
                .biases(out_biases),
                .weights(out_weights),
                .out(results),
                .ready(outputReady));
    assign ready = outputReady;
endmodule
```

Appendix X: Network Test Bench Code

```
module MLP_Network_tb;
    reg signed [15:0] picture1 [0:783];
    reg signed [15:0] picture2 [0:783];
    reg signed [15:0] picture3 [0:783];
    reg signed [15:0] picture4 [0:783];
    reg signed [15:0] picture5 [0:783];
    reg signed [15:0] picture6 [0:783];
    reg signed [15:0] picture7 [0:783];
    reg signed [15:0] picture8 [0:783];
    reg signed [15:0] picture9 [0:783];
    reg signed [15:0] picture10 [0:783];

    reg signed [15:0] out [0:9];
    reg signed [15:0] picture [0:783];
    reg clock;
    reg start;
    reg ready;

    MLP_Network #(.bits(16), .fractional_bits(11)) Network(
        .start(start),
        .clock(clock),
        .picture(picture),
        .results(out),
        .ready(ready)
        );

    initial begin
        clock = 0;
        start = 0;
        ready = 0;
        $readmemh("picture_1_16.mem" , picture1);
        $readmemh("picture_2_16.mem" , picture2);
        $readmemh("picture_3_16.mem" , picture3);
        $readmemh("picture_4_16.mem" , picture4);
        $readmemh("picture_5_16.mem" , picture5);
        $readmemh("picture_6_16.mem" , picture6);
        $readmemh("picture_7_16.mem" , picture7);
        $readmemh("picture_8_16.mem" , picture8);
        $readmemh("picture_9_16.mem" , picture9);
        $readmemh("picture_10_16.mem" , picture10);

        assign picture = picture1;
```

```
#5
start = 1;
#5
start = 0;
#1 clock = ~clock;
#1 clock = ~clock;
while(~ready) begin
    #1 clock = ~clock;
end

assign picture = picture2;
#5
start = 1;
#5
start = 0;
#1 clock = ~clock;
#1 clock = ~clock;
while(~ready) begin
    #1 clock = ~clock;
end

assign picture = picture3;
#5
start = 1;
#5
start = 0;
#1 clock = ~clock;
#1 clock = ~clock;
while(~ready) begin
    #1 clock = ~clock;
end

assign picture = picture4;
#5
start = 1;
#5
start = 0;
#1 clock = ~clock;
#1 clock = ~clock;
while(~ready) begin
    #1 clock = ~clock;
end

assign picture = picture5;
#5
```

```
start = 1;
#5
start = 0;
#1 clock = ~clock;
#1 clock = ~clock;
while(~ready) begin
    #1 clock = ~clock;
end

assign picture = picture6;
#5
start = 1;
#5
start = 0;
#1 clock = ~clock;
#1 clock = ~clock;
while(~ready) begin
    #1 clock = ~clock;
end

assign picture = picture7;
#5
start = 1;
#5
start = 0;
#1 clock = ~clock;
#1 clock = ~clock;
while(~ready) begin
    #1 clock = ~clock;
end

assign picture = picture8;
#5
start = 1;
#5
start = 0;
#1 clock = ~clock;
#1 clock = ~clock;
while(~ready) begin
    #1 clock = ~clock;
end

assign picture = picture9;
#5
start = 1;
```

```
        #5
        start = 0;
        #1 clock = ~clock;
        #1 clock = ~clock;
        while(~ready) begin
            #1 clock = ~clock;
        end

        assign picture = picture10;
        #5
        start = 1;
        #5
        start = 0;
        #1 clock = ~clock;
        #1 clock = ~clock;
        while(~ready) begin
            #1 clock = ~clock;
        end
    end
endmodule
```