ECE 385

Spring 2023
Final Project Report

I.M.E.R.S.I.V

Intelligent Mathematical Equation Recognition System In Verilog

Max Ma & Dhruv Kulgod
NL / Friday 9:15 AM - 9:30 AM
Nick Lu

Purpose

As the AI surge continues, artificial intelligence applications become more and more abundant, driving the purpose of dedicated hardware accelerators using FPGAs and ASICs. This project aims to demonstrate the utilization of an FPGA hardware accelerator for a handwritten numerical character neural network trained on a binary MNIST dataset. Multiple stages of verification were performed to get the final product: a highly-parameterized neural network IP peripheral and a VGA user interface to demonstrate its performance. Due to the implementation of parameters in both neural network training and the hardware implementation, this project demonstrates that a universal platform for integration of neural networks into FPGAs, and ultimately ASICs, is achievable.

Introduction

This report is broken down into three sections from neural network design, hardware development, and software development. It also includes various appendices with more detailed information and figures that will be referenced throughout the report.

Before diving into each section, an overview of the project will be presented. The first step of the project was to design the neural network. A simple multilayer perceptron model was used in order to reduce FPGA resources, and keep the project manageable within the timeline. The model was programmed and trained in Python with the weight and biases extracted from training. After completion of the neural network, the hardware development began with setting up the main components of the system: generating the SoC with both VGA and neural network IPs, connecting all board I/O, and conducting checksum tests on all components. After the checksum tests passed, work began on the neural network IP itself. This was followed by rigorous verification using a system verilog testbench in ModelSim, which passed with 90% accuracy. Upon successful verification, the NIOS-II software was developed and test cases were employed to validate the neural network IP once programmed in the FPGA fabric. Finally, the main source code was implemented to reliably interface with the VGA monitor and neural network IP peripheral to recognize characters drawn with a USB mouse.

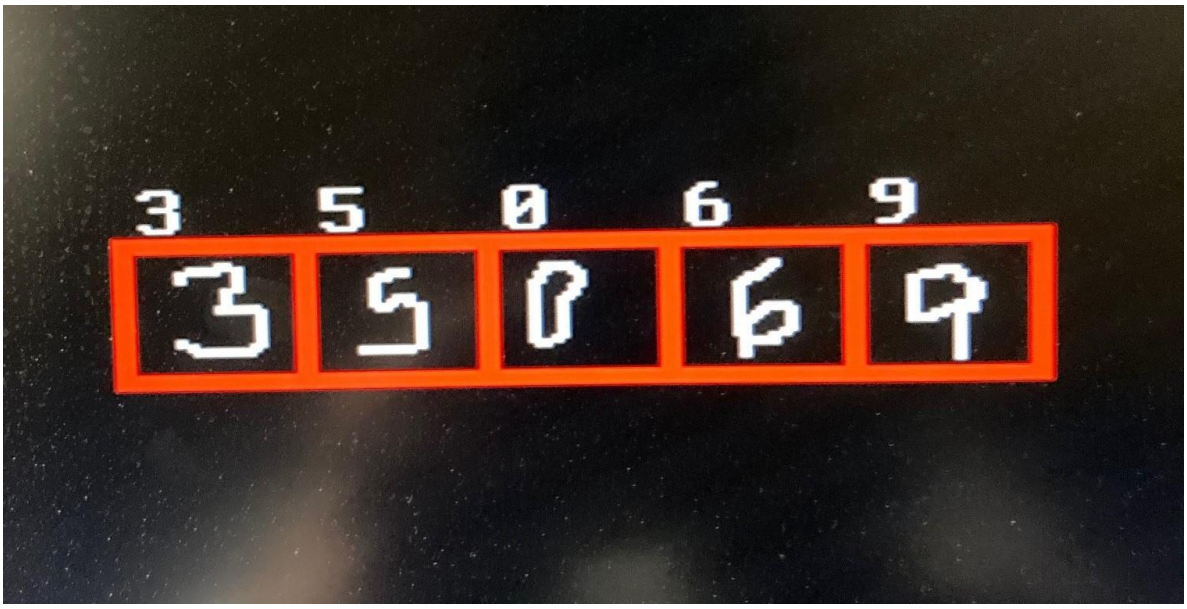

Figure 1: Character Recognition on VGA Monitor GUI

SECTION I: Neural Network Design:

Implementation

Training a neural network starts with having a good dataset. We knew about the MNIST dataset of handwritten digits, but to achieve our full functionality, we required mathematical operators as well. Through research, we found a dataset on Kaggle that contained all digits as well as common operators such as addition, subtraction etc (APPENDIX D Source #4).

The next step is model type. We had to achieve a balance between a model complexity (which allows higher accuracy scores), model size, and ease of implementation on an FPGA. The state of the art for MNIST-style character recognition is currently a Convolutional Neural Network (CNN), which applies principles from DSP in 2 dimensions. We felt this would be too large a task to successfully implement in our 4-week time frame. Hence, we adapted the model from Nielsen's github (APPENDIX D Source #2) to our new dataset. This model was a simpler Multi-Layer Perceptron (MLP), consisting of one hidden layer (APPENDIX A). Through some trial and error, we achieved an 80% validation score with this model.

However, there was a discrepancy between the dataset and our expected inputs. Because the dataset was just handwritten characters from one person, transformed in different ways to create a larger dataset, there was not enough variety in the data to create a properly generalisable model. By drawing our own test dataset (100 variants) using a mouse, we found the model performed quite poorly (only around 30%) unless we explicitly copied the drawing style of the dataset exactly. Between the limitations of the data and the simplicity of our model, we did not feel this approach would work for our final project demonstration. Thus, we switched to using the MNIST dataset (a version of the MNIST which we created with only binary data, because our drawing did not support greyscale).

After this switch, we were able to achieve a 94% accuracy on the validation dataset in Python, which translated to a 90% accuracy in hardware, discussed in later sections. It also performed far better on our own dataset, close to 60%. We were no longer able to reach our stretch goals, but this compromise allowed us to create a more consistent baseline product.

SECTION II: Hardware Development:

Implementation

Following the purpose of this project as a hardware accelerator, the hardware implementation in the FPGA is crucial. The hardware implementation is centered around the NIOS-II and supporting peripherals composed into a single SoC using Platform Designer. These peripherals and components are detailed in the system level block diagram shown in Appendix B. However, the two custom IPs are worth mentioning in detail. First, the VGA Interface IP is responsible for controlling the VGA display. This includes displaying the mouse cursor, drawing to the screen, illustrating the predicted characters, and handling the read and writes to the frame buffer. Covering each of these in more detail, the mouse cursor was composed of a sprite and x,y location. The sprite was stored in ROM as indices within the color palette, and they were displayed at the appropriate x,y location given from PIOs driven by the NIOS-II. Drawing pixels was also handled by the NIOS-II by writing to the frame buffer. The predicated characters were illustrated from a font-rom module that was used to hold the bit map of each number 0-9. Finally, the VGA display controller used a single buffered frame buffer to hold the black or white information of each pixel with 480x640 resolution.

The second part of the hardware development stage includes the neural network IP. This IP was designed to act as a peripheral with only the input image buffer and output character register accessible from the Avalon bus interface. The neural network IP was also designed to trigger an interrupt upon completion of computing the character from the given 28x28 image. The neural network IP was designed with parameterized layer and neuron counts as well as fixed point location. Because we implemented a fully-connected neural network, each layer receives all the inputs from the previous layer's outputs. This was completed using data pipelining between the image buffer and between layer 1 and layer 2. Layers are composed of neurons, which each have hardware DSP multipliers to compute the multiply and accumulate operations on the input values and weights. Combinational logic was incorporated to handle proper data valid and output valid signals. In each neuron, there contains a weight, bias, and sigmoid ROM, which all acted as lookup tables. Due to the use of lookup tables, accuracy was lost because values were quantized and not continuous. Additionally, overflow checks were performed on the final output to ensure no glitches. After both data pipelining stages, the values of the output layer were used to find the maximum value, which corresponds to the final guess. This guess was stored in the character register to be read by the NIOS-II.

Neural Network Verification

A large part of the work dedicated to this project was done on the verification and validation side of developing the neural network IP peripheral. Verification was done in ModelSim using a custom testbench that emulated the NIOS-II using the Avalon bus interface. The testbench flattened test images, wrote them to the neural network's image buffer, and compared the expected and computed results. This testbench was parameterized to handle all 10,000 test images, which would take multiple minutes to finish. Verification was critical to this project because the performance of the neural network is impossible to visualize. Upon completion of the neural network IP, tests were conducted on the programmed FPGA model using the NIOS-II to validate the simulations. This is discussed in Section III.

Simulation Waveform

The simulation waveform from running 100 images from the testing dataset in ModelSim is illustrated below. Each computation takes roughly 33us for the native 50 MHz clock frequency. The simulation illustrates that the neural network has 90% accuracy on the given test images.

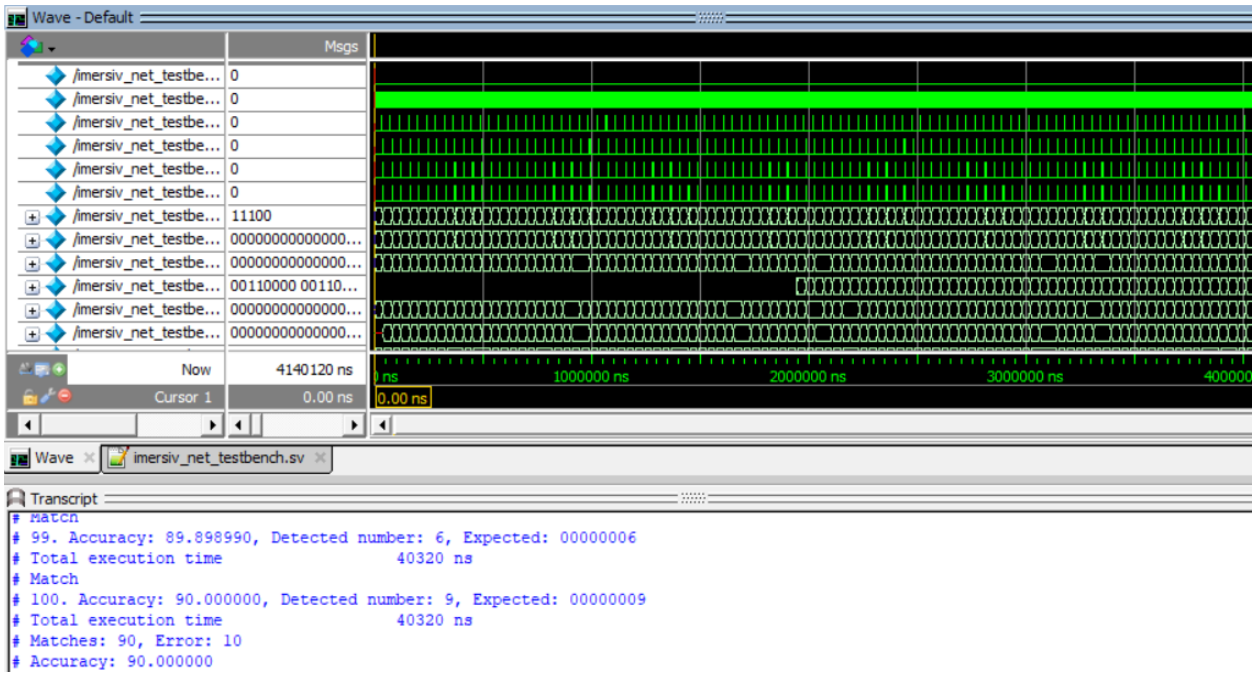

Figure 2: Simulation Waveform from Running Test Dataset

Challenges

There were some hardware challenges faced during the hardware development of this project. First, inferring BRAM for the neurons within the neural network and not inferring BRAM for the image buffer proved to be slightly troublesome. We learned that there is a fine line between what the compiler can detect as inferable memory. Additionally, working with the frame buffer and managing proper reads and writes so as to not induce any graphical glitches required careful planning. Finally, debugging had to be done on the neural network IP itself due to some timing issues. This proved to be challenging as even the complexity of the simple model introduced a lot of signals that needed to be checked.

SECTION III: Software Development

Implementation

To control all the peripherals implemented in the SoC design, the NIOS-II was used for its flexibility. The code for the main loop is, as shown in Appendix C, handles each of the peripherals using a task-based structure. The three main tasks are the neuralNetTask, VGATask, and mouseTask. We will cover the neuralNetTask as the others are self-explanatory. This task utilizes an interrupt handler to tell it whether to load the next image or read and transfer the current output character. The interrupt handler can be referenced in Appendix C; however, the basic idea is that it will signal when the neural network IP is done computations. Once done, the character is transferred to be displayed by the VGA IP and the next image is loaded until the fifth image is reached. This task is triggered once an on-board button is pressed.

Neural Network Validation

A stage of validation tests were conducted using the NIOS-II, which was programmed into FPGA fabric. The tests were composed of pre-selected images that were fed into the neural network IP peripheral. The outputs of the neural network were matched with the same outputs from the ModelSim testbench. Indeed, the outputs matched, which validated that the neural network was implemented correctly in the FPGA hardware. Because of the flexibility and ease-of-use, validation using the NIOS-II was chosen over approaches like SignalTAP.

Challenges

There were some software challenges faced during the hardware development of this project. First, flattening the 2D image to send to the neural network IP peripheral was difficult to program due to the formatting of the 28x28 image vector. This could be improved for future implementations. Second, handling the interrupt logic from the neural network IP was crucial to not bottleneck the NIOS-II with polling requests. This proved to be difficult since the SoC had to be changed in Platform Designer to handle hardware interrupts. Finally, system performance had to be troubleshooted because reading and writing to the neural network IP caused mouse drawing to become delayed. Thus, the neural network IP only computed images upon pressing an onboard button.

Conclusions

In conclusion, a character recognition neural network was successfully implemented with 90% accuracy. In hardware, our model took an average of 33us to evaluate each image, while our Python script averaged around 28us. However, if we theoretically run at Fmax (APPENDIX B), we would expect our hardware average to come down to 23us, beating Python. There are many caveats here - Python was not utilizing all the capabilities of the hardware available, actual stability at Fmax has not been tested. Despite these, our hope here was that we could demonstrate that fixed-function neural network accelerators do have a place in the world, and can show performance increases in the right applications. We believe our project showcases enough evidence for this to be true.

Thus, this project demonstrates that successful neural network hardware accelerators will become the status quo in products from IoT devices to mobile phones. Furthermore, work to develop different image and speech recognition models can be implemented using the framework of our project in the future.

APPENDIX A: Neural Network Design & Training

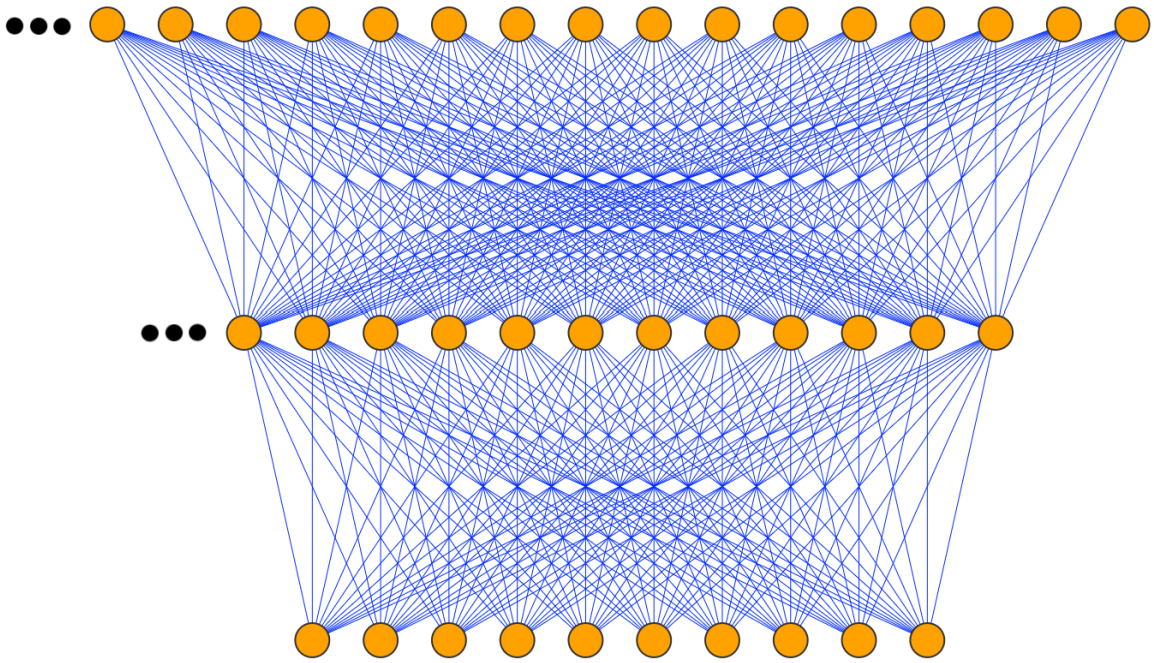

Figure 3: Neural Network structure with 784 input nodes, 1 hidden layer with 30 nodes, and one output layer with 10 nodes.

```
[[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]]
Prediction:  6
```

```
Out[211]: array([[1.57425821e-04],
                 [1.71786918e-06],
                 [3.48755325e-03],
                 [4.66588009e-09],
                 [1.15525485e-02],
                 [1.30681215e-06],
                 [1.11868762e-01],
                 [1.40339870e-04],
                 [7.80485131e-04],
                 [4.30666042e-04]])
```

Figure 4: Example input and output of the model. The 2D array shows a binary digit being fed into the model, while the 1D output array shows us what the output nodes hold for a given input.

APPENDIX B: FPGA Design and Statistics

Appendix B illustrates the system level block diagram implemented in FPGA fabric. It also details how the system verilog source code on the FPGA is organized and the resulting resource utilization.

FPGA System Level Block Diagram

From the system level, the main bulk of the project was done in the neural network IP. This IP interfaces with the Avalon Bus like any other peripheral. The master should write the intended image to the 28x28 image buffer. Upon receiving an interrupt from the peripheral when computation completes, the appropriate character can be read from the memory mapped character register located within the IP.

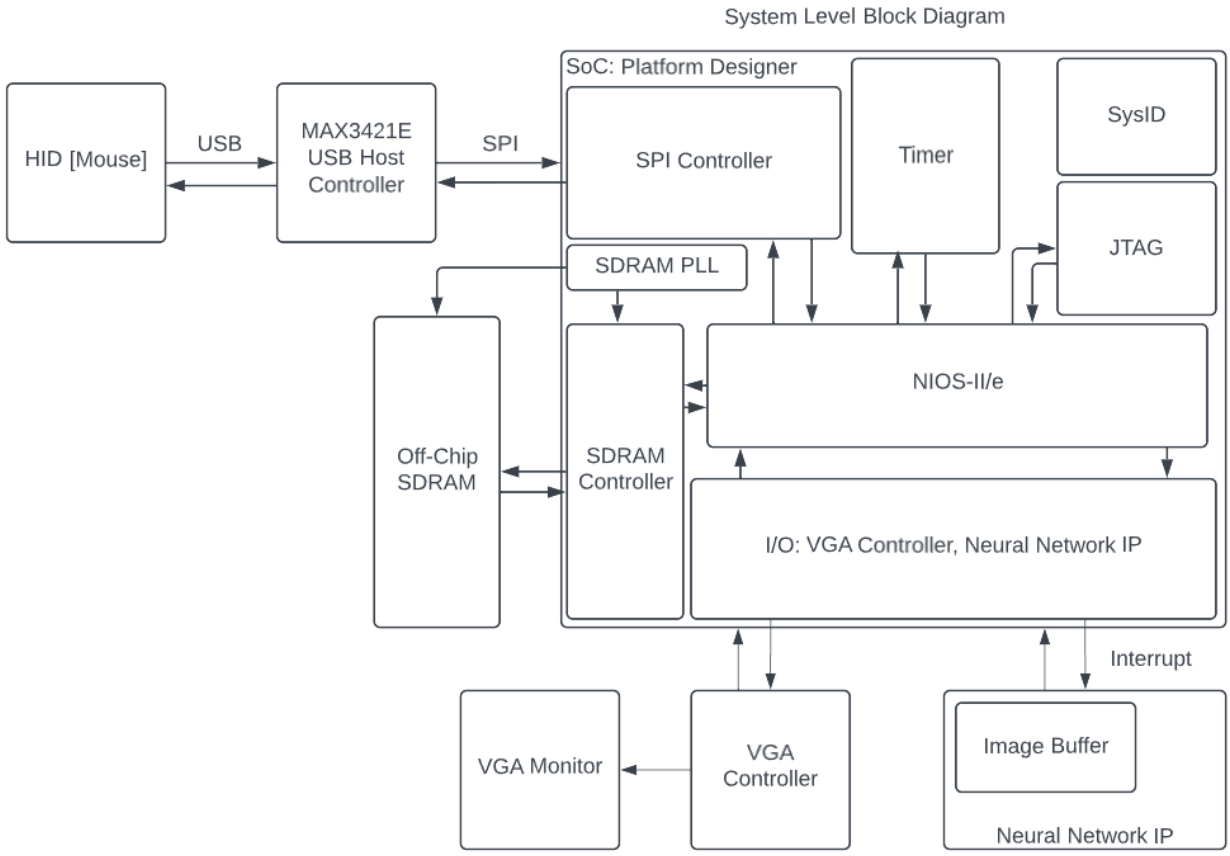

Figure 5: System Level Block Diagram

FPGA Design Module Information

Many different SystemVerilog and Verilog files went into building this project; thus, only files pertaining to the neural network IP and its verification will be discussed. Interest in the system design should be directed to Appendix E.

imersiv_net_avl_interface.sv: Top level module designed to interface with the Avalon bus and generate the IP in Platform Designer. Contains the image buffer, character register, and handles memory mapped reads and writes from the Avalon bus.

imersiv_net.sv: Module containing the neural network model within the “imersiv_net_avl_interface.sv” module. This module mainly handles pipelining the input and output to and from each layer of the neural

network. Additionally, it finds the maximum value of the ten output neurons, which is the final prediction of the neural network.

imersiv_net_testbench.sv: Testbench used to verify the neural network IP's performance against the original pythonic implementation. This testbench is designed to feed the test dataset to the neural network by simulating reads and writes using the Avalon bus interface. Each 2D image is flattened and fed into the neural network IP by writing to the image buffer through the Avalon bus. This functionality simulates the job of the NIOS-II processor during the validation and final product stages.

weight_memory.sv: A ROM module used to store the weights of a particular neuron within the neural network model. Address width is parameterized for quality-of-life and reusability in each layer.

sigmoid_rom.sv: A ROM module used as a lookup table for the sigmoid activation function of each neuron. Address width is parameterized so an appropriate resolution can be used.

neuron.v: Main module of the neural network. Computes the multiplication and accumulation operation on the given input and weight. The bias value is added after all the computations are complete, and the activation function is evaluated to determine the final output of the neuron. 16-bit fixed point arithmetic was implemented to increase the precision of the neural network.

maxFinder.v: A module used to find the maximum value within the final neuron output layer. It implements a basic larger-than-current comparison algorithm and maps the output to the corresponding numerical character.

Layer_1.v: A module to hold the neurons composing the first layer of the neural network. Each neuron was assigned a weight '.mif" file and a bias '.mif" file. The outputs from all neurons were combined into a single vectored output from the layer module.

Layer_2.v: A module to hold the neurons composing the second layer of the neural network. Each neuron was assigned a weight .mif file and a bias .mif file. The outputs from all neurons were combined into a single vectored output from the layer module.

include.v: Header file used to store parameters, such as fixed point integer width, number of layers, and size of each layer. These parameters were used throughout the neural network IP.

FPGA Design Statistics

After reading the compilation report, the FPGA resource utilization for this project was very large compared to previous experiments. However, this resource utilization is justified. For the large amount of LUTs, contributing factors were the logic needed for the pipelining and image buffering utilized in the neural network IP along with the actual implementation of the neurons in the model itself. Next, 30 DSP slices were used for the multiply and accumulate operations in each neuron. There were 30 neurons total, each taking a single DSP slice. A large amount of the VRAM was consumed in the 480x640 VGA frame buffer; additionally, each neuron was composed of weight, bias, and sigmoid activation function ROM.

Design Resources and Statistics Table

LUT	34,103
DSP	30
Memory (BRAM)	414,720 bits
Flip-Flop	5,548
Frequency	69.52 MHz
Static Power	97.06 mW
Dynamic Power	175.26 mW
Total Power	293.73 mW

Figure 6: FPGA Design Resources and Statistics

FPGA RTL Diagrams:

Because of the depth and complexity of the hardware used in this project. Only the RTL diagrams pertaining to the neural network IP are included.

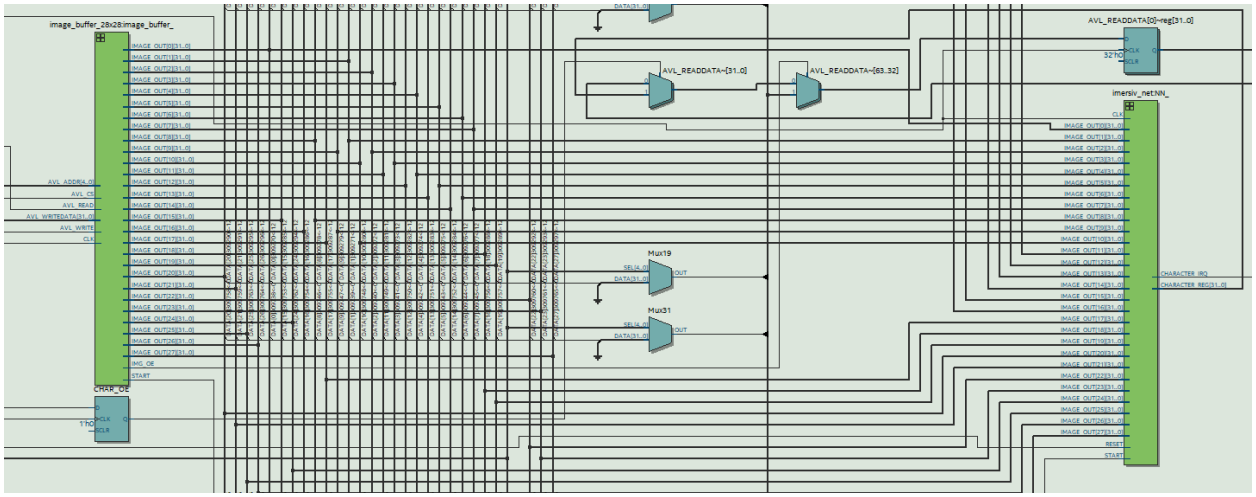

Figure 7: Top Level Neural Network IP Interface using Avalon Bus

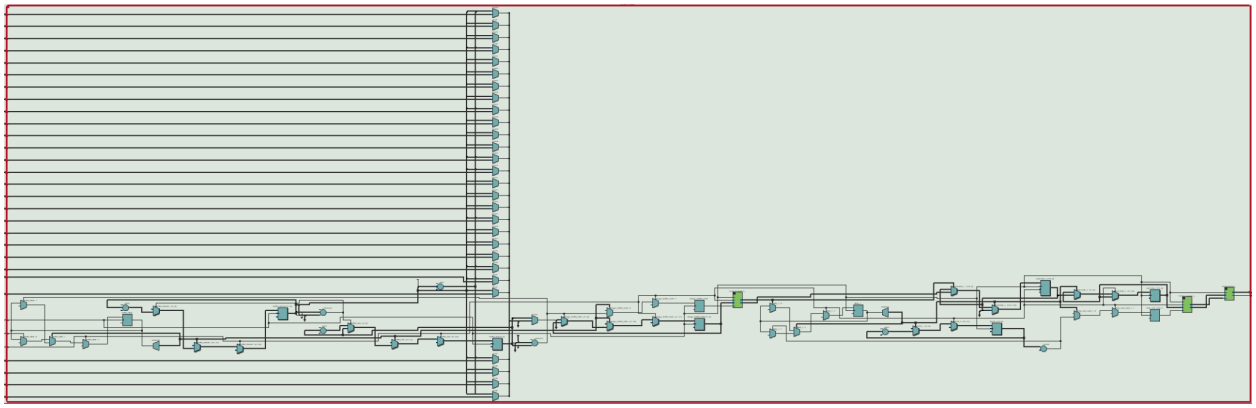

Figure 8: Neural Network Module to Illustrate General Design

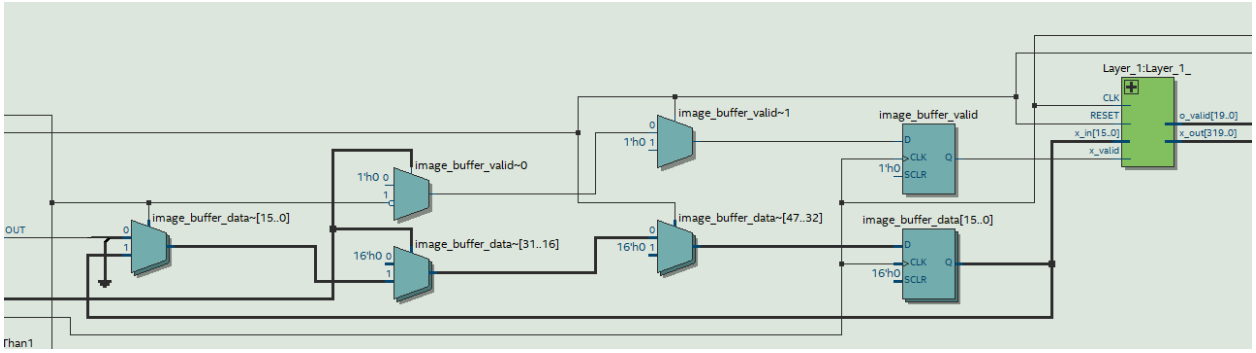

Figure 9: Neural Network Data Pipelining Input to Layer 1

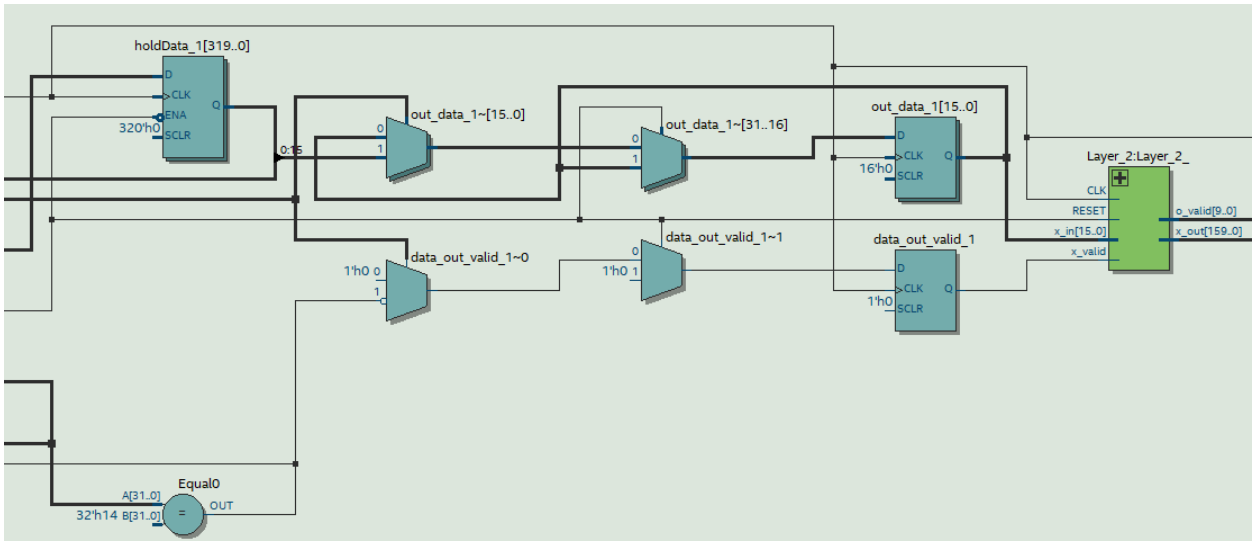

Figure 10: Neural Network Data Pipelining Layer 1 to Layer 2

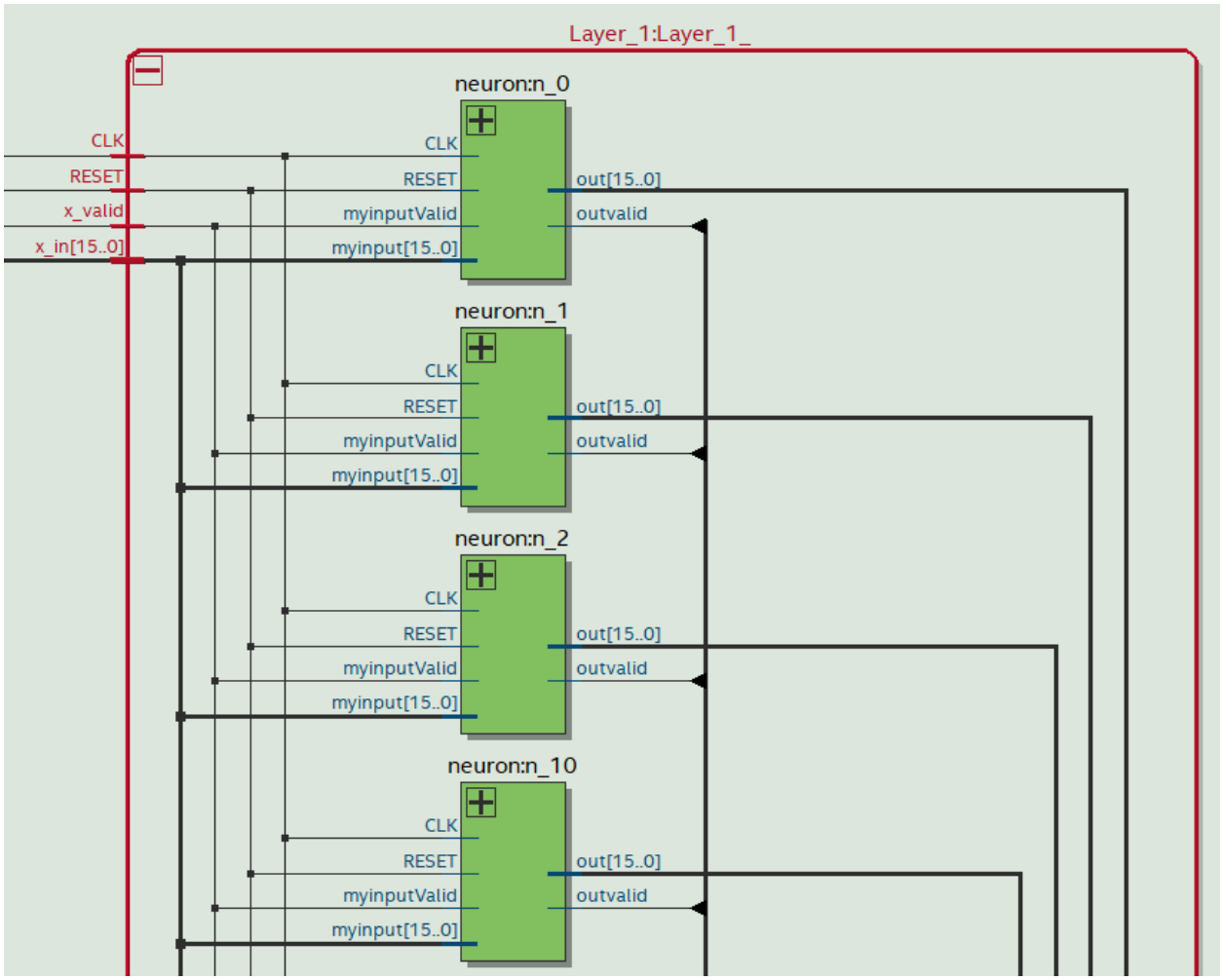

Figure 11: Layer 1 Module

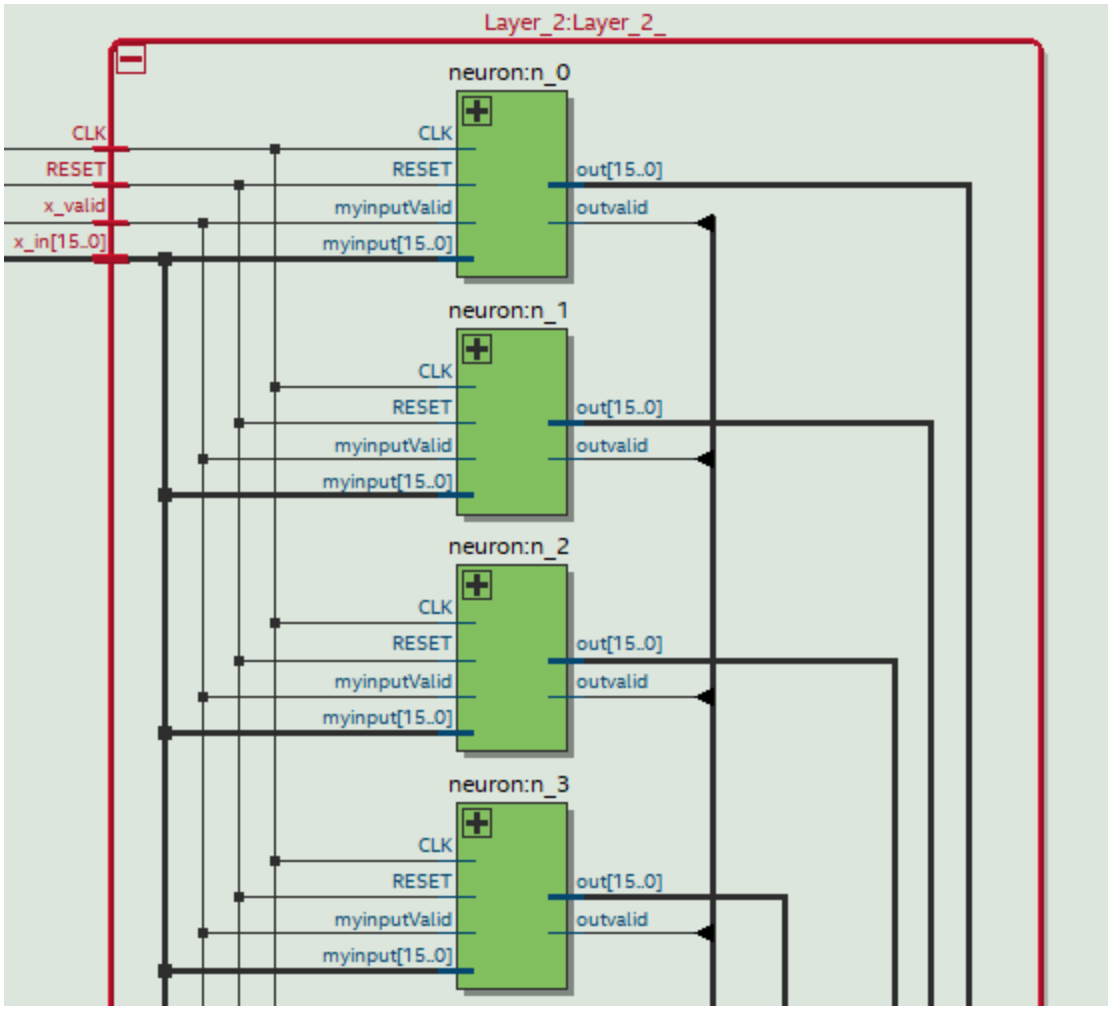

Figure 12: Layer 2 Module

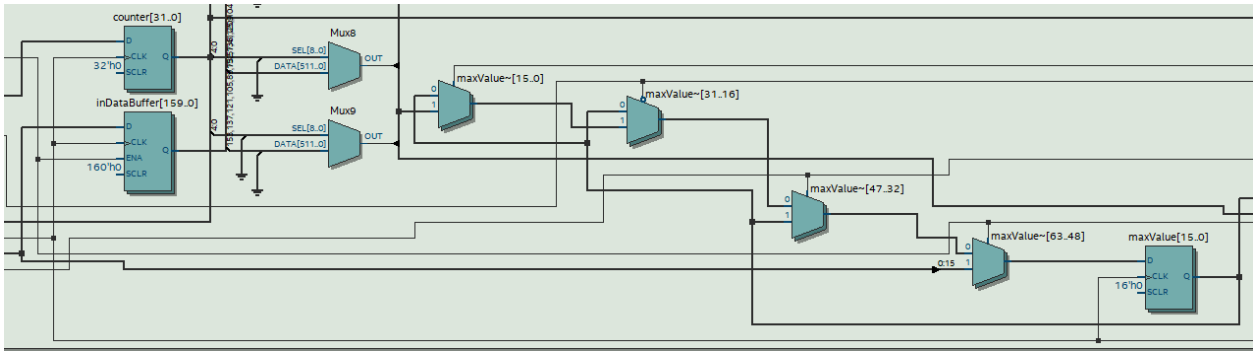

Figure 13: Maximum Finder Module

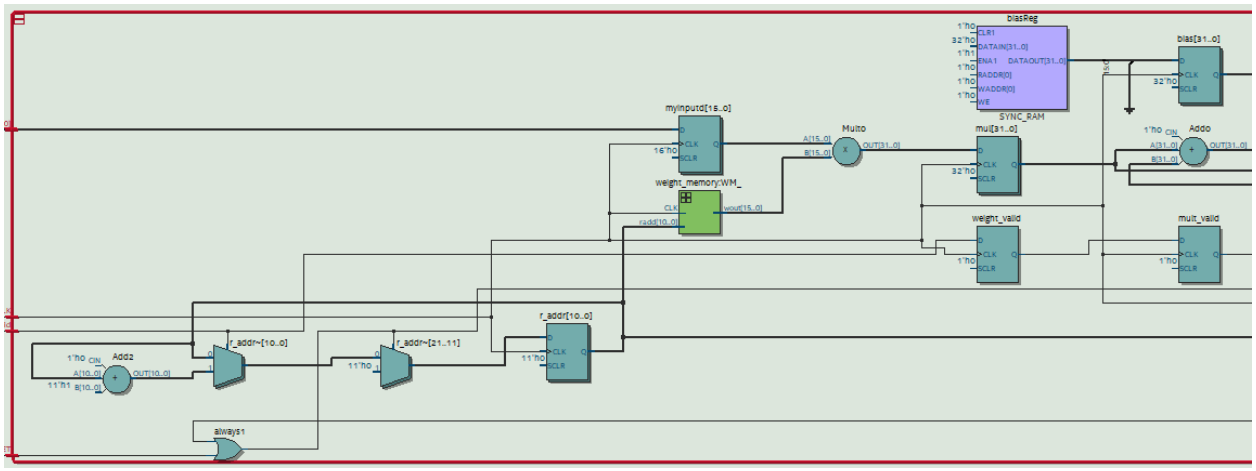

Figure 14: Neuron Layout and Design Part 1

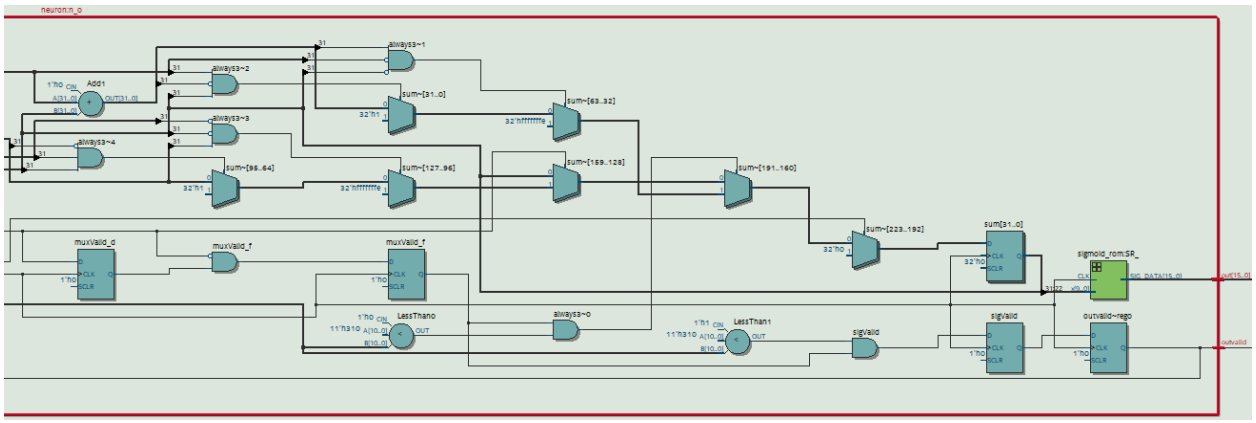

Figure 15: Neuron Layout and Design Part 2

APPENDIX C: NIOS-II Source Code Analysis

Appendix C contains images of important snippets of the NIOS-II source code needed in understanding the overall implementation of the software used in the project. Interest in more detailed implementation should be directed to Appendix E.

```
initTest();
VGAClear();
printf("Initializing MAX3421E...\n");
MAX3421E_init();
printf("Initializing USB...\n");
USB_init();
init_neural_network_interrupt();
while (1) {
    /// Neural Net Tasks ///
    if (*keys == 0x02) {
        neuralNetTask();
    }

    if (*keys == 0x01) {
        VGAClear();
    }

    /// USB Tasks ///
    VGATask(buf, mouseX, mouseY);
    MAX3421E_Task();
    // usleep(1000);
    USB_Task();
    if (GetUsbTaskState() == USB_STATE_RUNNING) {
        if (!runningdebugflag) {
            runningdebugflag = 1;
            device = GetDriverandReport();
        } else if (device == 2) {
            // Check more mouse data
            rcode = mousePoll(&buf);
            if (rcode == hrNAK) {
                //NAK means no new data
                continue;
            } else if (rcode) {
                printf("Rcode: ");
                printf("%x \n", rcode);
                continue;
            }
            // Process Mouse Data to VGA Interface
            mouseTask(&buf, &mouseX, &mouseY, &sensitvity);
        }
    } else if (GetUsbTaskState() == USB_STATE_ERROR) {
        if (!errorflag) {
            errorflag = 1;
            printf("USB Error State\n");
        }
    } else {
        printf("USB task state: ");
        printf("%x\n", GetUsbTaskState());
        if (runningdebugflag) { //previously running, reset USB hardware just to clear out any funky state, HS/FS etc
            runningdebugflag = 0;
            MAX3421E_init();
            USB_init();
        }
        errorflag = 0;
    }
}

return 0;
```

Figure 15: Main Loop

```c
#define WIDTH           80U
#define HEIGHT          480U

#define EMPTY_SPACE     10752U

#define NUM_CODES       5U

#define IMG_WIDTH       28U
#define IMG_HEIGHT      28U

//  Word Addresses  | Byte Addressses | Purpose
//  ------------------------------------------------------------
//  0x0000 - 0x257F | 0x0000 - 0x95FF | B&W VRAM
//  0x3000 - 0x3004 | 0xC000 - 0xC013 | x5 NN Codes
//  0x3005          | 0xC014 - 0xC017 | Status Register (Read-Only)
typedef struct vga_interface {
    alt_u8 VRAM [WIDTH * HEIGHT];
    alt_u8 empty [EMPTY_SPACE];
    alt_u32 neuralNetRegs [NUM_CODES];
    alt_u32 statusReg;  // [0] = Blanking Signal
} vga_interface;

//  Word Addresses   | Byte Addressses | Purpose
//  ------------------------------------------------------------
//  0x0000 - 0x001B  | 0x0000 - 0x006F | 28 x 28 Image File
//  0x001C           | 0x0070 - 0x0073 | Character Register (Read-Only)
typedef struct NN_interface {
    alt_u32 imgRegFile [IMG_HEIGHT];    // 28 Pixel in Upper 28 Bits
    alt_u32 charReg;
} NN_interface;
```

Figure 16: Memory Map for IP Peripherals

```c
// Interrupt Handler
#ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
static void handle_neural_network_interrupt(void* context)
#else
static void handle_neural_network_interrupt(void* context, alt_u32 id)
#endif
{
    // printf("Interrupt Handling!\n");
    // Cast context to edge_capture's type
    volatile int* edge_capture_ptr = (volatile int*) context;
    // Read the edge capture register on the button PIO.
    *edge_capture_ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(CHARARCTER_IRQ_BASE);
    // Write to the edge capture register to reset it.
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(CHARARCTER_IRQ_BASE, 0x0);
    // Read the PIO to delay ISR exit
    IORD_ALTERA_AVALON_PIO_EDGE_CAP(CHARARCTER_IRQ_BASE);

    // Neural Network Done Computation
    doneNeuralNetCompute = TRUE;
}

/* Initialize the button_pio. */
static void init_neural_network_interrupt()
{
    printf("Initializing Neural Network Interrupt!\n");
    // Recast the edge_capture pointer
    void* edge_capture_ptr = (void*) &edge_capture;
    // Enable Neural Network Interrupt
    IOWR_ALTERA_AVALON_PIO_IRQ_MASK(CHARARCTER_IRQ_BASE, 0x1);
    // Reset the edge capture register
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(CHARARCTER_IRQ_BASE, 0x0);
    // Register the IRQ
    #ifdef ALT_ENHANCED_INTERRUPT_API_PRESENT
    alt_ic_isr_register(CHARARCTER_IRQ_IRQ_INTERRUPT_CONTROLLER_ID,
    CHARARCTER_IRQ_IRQ,
    handle_neural_network_interrupt,
    edge_capture_ptr, 0x0);
    #else
    alt_irq_register( BUTTON_PIO_IRQ,
    edge_capture_ptr,
    handle_button_interrupts );
    #endif
}
```

Figure 17: Neural Network IP Interrupt Handler

```c
// Feed Data to the Neural Network Once per Frame (Adjust to per xxx amount of Frames?)
void neuralNetTask() {
#if DEBUG
    // characterTest();
#else
    static int numRun = 0;
    printf("Neural Network Task! Run: %d\n", numRun);
    int index = 0;
    while (index < NUM_CODES) {
        if (doneNeuralNetCompute) {
            // Next Image
            transferCharacter(index);
            printNeuralNetCharacterReg(index);
            doneNeuralNetCompute = FALSE;
            index++;
        }
        else {
            // Load 28x28 Pixel Image
            loadImage(index);
        }
    }
    numRun++;
#endif
}

// Set the Indexed 28x28 Pixel Character into the Image Buffer for Neural Network to Compute
void loadImage(int index) {
    // printf("Loading Image: %d\n", index);
    alt_u32 buf[IMG_HEIGHT];
    getImageVRAMBlock(index, buf);

    // Set Image Buffer
    for (int i = 0; i < IMG_HEIGHT; ++i) {
        neuralNet->imgRegFile[i] = buf[i];
    }
}

// Loads Character into VGA IP to Display
void transferCharacter(int index) {
    // printf("Transfer Character: %d\n", index);
    setNeuralNetCharacterRegs(index, neuralNet->charReg);
}
```

Figure 18: Neural Network Task

APPENDIX D: Sources

Appendix D contains the main external sources and reference materials utilized in developing the neural network for the project. A description of each source is also provided.

Source #1: http://neuralnetworksanddeeplearning.com/chap1.html
Description: This source was used as the base neural network implementation that was implemented in the FPGA fabric. The implementation for this project was a multilayer perceptron.

Source #2: https://github.com/mnielsen/neural-networks-and-deep-learning
Description: This GitHub repository was the source code that was modified for the off-the-FPGA training of the character recognition model.

Source #3: https://github.com/vipinkmenon/neuralNetwork/tree/master
Description: This GitHub repository contains an example implementation of a neural network in FPGA hardware. Source code from this repository was modified and adapted to the system described in this report.

Source #4: https://www.kaggle.com/datasets/michelheusser/handwritten-digits-and-operators
Description: This Kaggle dataset contains data of handwritten digits and operators, turned into binary 28x28 images. We based our initial model off this dataset.

APPENDIX E: GitHub Repository

Appendix E contains a link to the source code repository. This report was high-level overview of the project; however, deep interest in the source code for the neural network design, system verilog hardware implementation, NIOS-II source code, and testing and validation files should be directed to the GitHub Repository: https://github.com/Max-Y-Ma/ECE-385-FinalProject.