CSM152a Lab 4 Report:

Iris Flower Classification with Neural Networks

Team:

Gajan Nagaraj

Frank Ren

Everett Sheu

Professor: Miodrag Potkonjak

TA: HongXiang Gu

Lab 1

Introduction:

The goal of lab 4 was to utilize our knowledge of the FPGA, Xilinx ISE, and building Verilog systems from the past labs to create a project of our choice from scratch. We proposed to create a neural network that would be trained to classify the UCI iris flower data. However, due to the FPGAs limited computational power, we determined that it would logical to create the neural network in Python and train it on the computer. We would capture the model weights and biases, and transfer them to the Verilog neural network to run predictions. The inputs to the Verilog neural network would be four integers inputted by the user on the PmodKYPD numpad. These four integers would then be converted into 32 bit floating point numbers and fed into the trained neural network. The neural network will then output three floating point numbers that correspond to the classification of the inputs for each label (flower species). We return the greatest classification and output this to UART (original plan, but incomplete) and to the seven segment display.

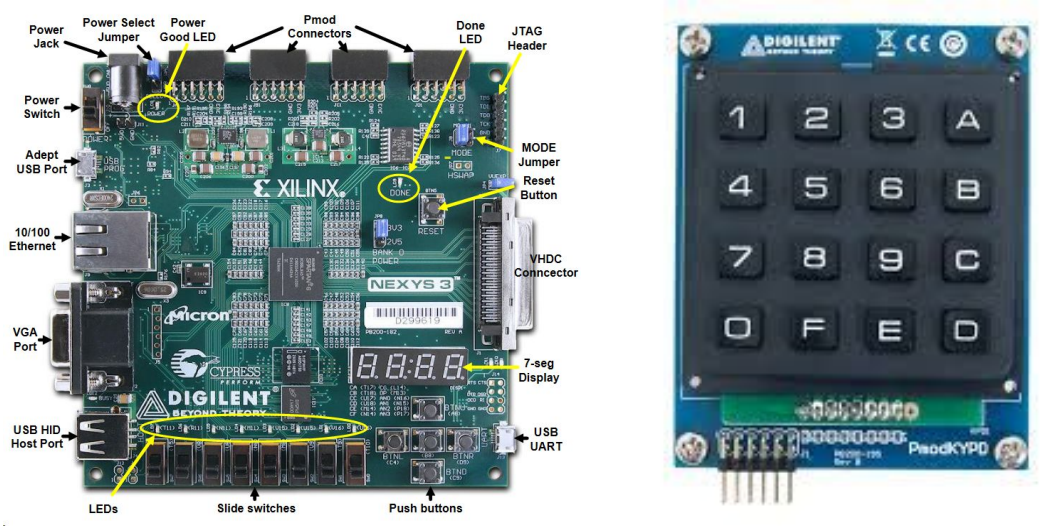

Figure 1: FPGA board highlighting input slider switches, input push buttons, and output onboard seven segment display. The numpad was connected to the JA port on the FPGA.

Design Description:

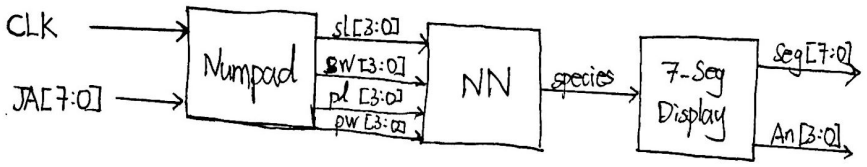

Figure 2: Top Schematic For Iris Flower Classification with Neural Networks

For the design of the stopwatch, we implemented three main modules: numpad input, neural network prediction, and UART/seven segment display output. These three modules were combined together in the top module to produce the final iris flower classifier. **Figure 2** shows the overall design diagram for the FPGA iris flower classifier.

Numpad Input:

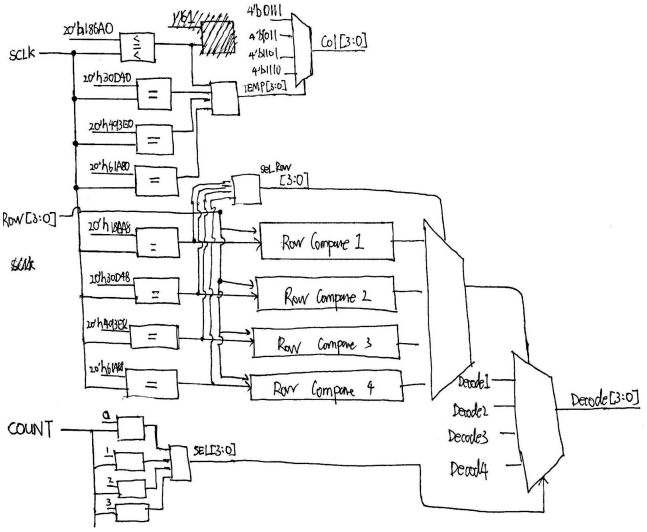

Figure 3: Schematic for Number Pad

In order to input the 4 integer value for sepal length, sepal width, petal length, and petal width, we used number pad to record our input. We first found the sample code online. By analyzing the sample code, we get the general idea of using the number pad. The fpga takes in a 8 bit input from the number pad, the first 4 bits indicating the row of the number pad, where 0111 indicating the first row is selected. The last 4 bits indicating the columns for the number pad.

In order to poll from the keypad, we have a value sclk that is incremented according to the onboard clock. Whenever sclk reaches certain values, it polls from a different row. Since We need to take four separated value from number pad, we also utilized another counter. We converted the input value from number pad to integer and store it to the register with

corresponding counter, and we then increment the counter so the next time we receive an input from number pad, it will be stored in the next register until we stored all four inputs from the number pad.

Neural Network:

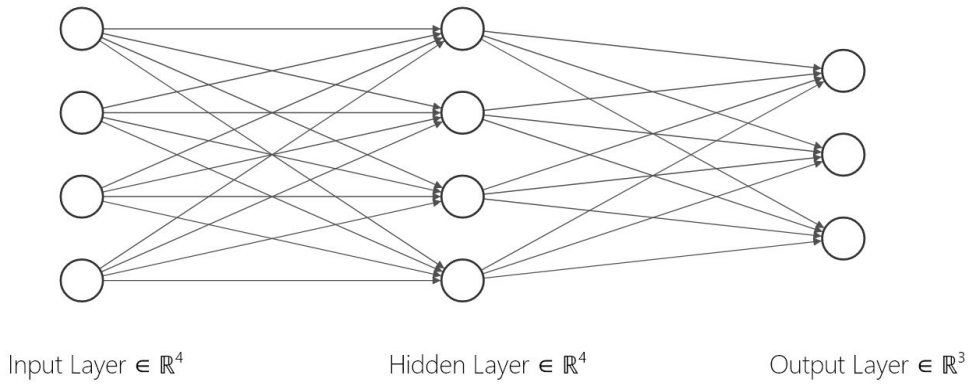

Figure 4: Diagram of Neural Network used for Iris Flower Classification. Four input nodes, four hidden nodes (1 layer), three output nodes.

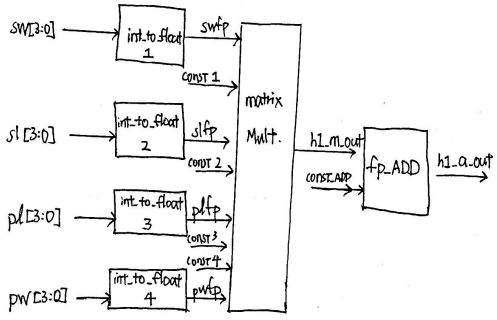

Figure 5: Schematic for Activation of one input node in hidden layer. Process repeated four times

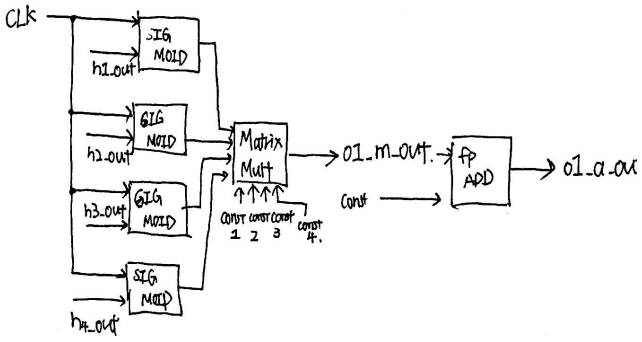

Figure 6: Schematic for Activation of one output. Process repeated three times

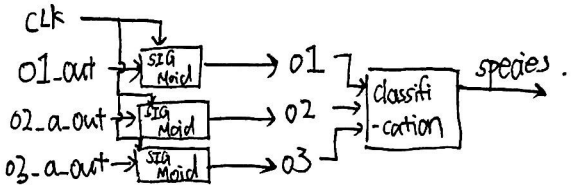

Figure 7: Schematic for calculate sigmoid of each node in output layer and output the classification.

The neural network prediction module took in the four inputs and computed a value between 0 and 1 for each label by feed forward propagation. To be exact, and use the weights from the Python model, we had to transform our inputs to floats and use 32 bit floating point arithmetic. The greatest value among all the labels is the output classification. Each hidden node and output node have an associated set of weights and biases. The inputs and weights are represented by 1x4 matrices. Thus, we must compute (1x4)*(4x1) matrix multiplication and take the sigmoid of the result to get the activation of each node. This is done for the all the hidden nodes and output nodes.

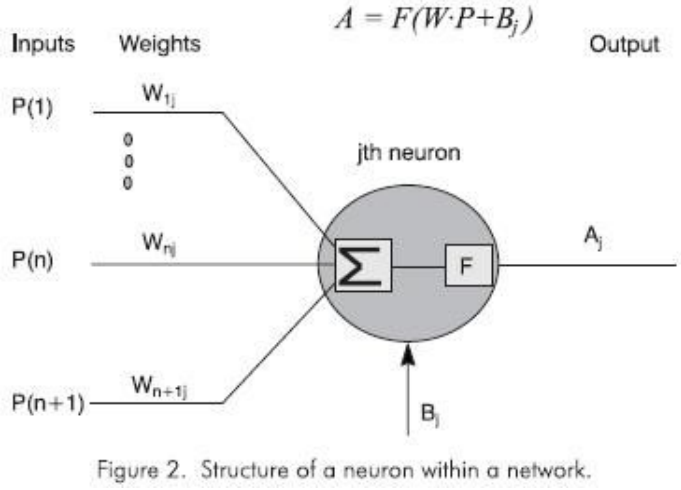

Figure 2. Structure of a neuron within a network.

Figure 8: Diagram showing how the activation for each neuron is calculated. F is the sigmoid function.

In order to verify that the neural network worked as expected, we created a testbench to test its functionality. To test the neural network effectively, we fed in inputs that we tested on the Python model and verified that the classification from the simulation matched that of the Python module.

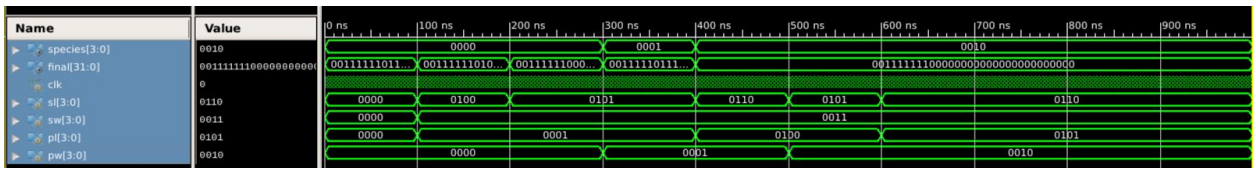

Figure 9: Waveform output from neural network testbench.

```
C:\Gajan\School\UCLA\Sophmore\Winter\CS152a\pythoNN>python nn_iris.py
0
0
0
1
2
2
2
```

Figure 10: Output from Python model that matches testbench waveform.

Int to Float:

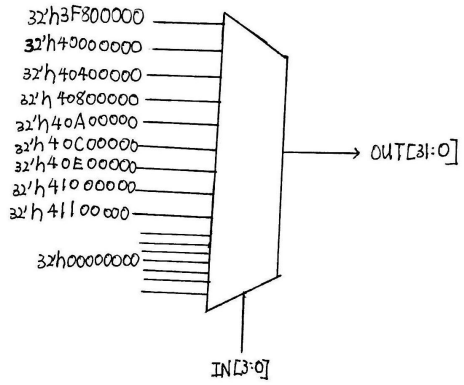

Figure 11: Schematic for int to floating point conversion

The inputs to the neural network are collected via the numpad and stored into registers for later use. However, the weights and biases in the neural network at each node are floating point numbers. In order to preserve accuracy, precision, and compute arithmetic operations we determined that we must convert the inputs into 32 bit floating point numbers. Specifically, we decided to use the IEEE 754 standard which is shown in **Figure 12.**

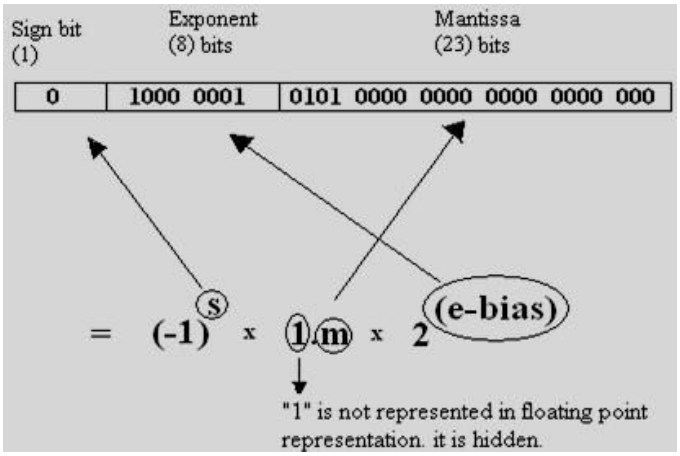

Figure 12: IEEE 754 32 Bit Floating Point Representation of Number.

The conversion from integer to float involved a simple case statement that outputted the hard coded 32 bit representation of the input number from the numpad (always between 0 and 9). Although it was hardcoded, we still wrote a testbench to make sure that our conversions were correct and the module worked as expected.

Figure 13: Waveform output for int to float conversion module.

Matrix Multiplication:

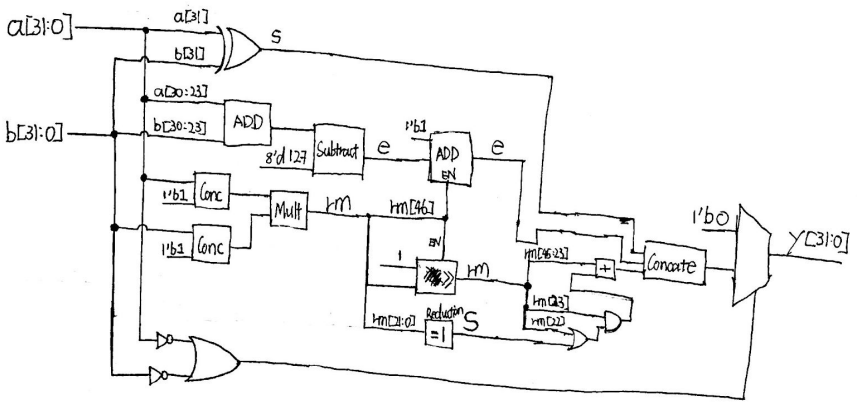

Figure 14: Schematic for matrix multiplication

In order to implement the neural network, we required a matrix multiplier to connect the inputs and outputs of each node. Due to the nature of our project, we required the use of of a 1x4 by 4x1 matrix multiplier. The mathematical details of such an operation are detailed in **Figure 15** below.

$$\begin{bmatrix} w & x & y & z \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} (w \times a) + (x \times b) + (y \times c) + (z \times d) \end{bmatrix}$$

Figure 15: Showing 1x4 time 4x1 Matrix Multiplication.

As such, the matrix multiplier accepts eight 32 bit floating point numbers and outputs a single 32 bit floating point value.

Floating Point Multiplication:

The floating point multiplication module took in two 32 bit floating point numbers as factors and outputted the 32 bit floating point representation of the product. This module was built following the normal algorithm to compute floating point products.

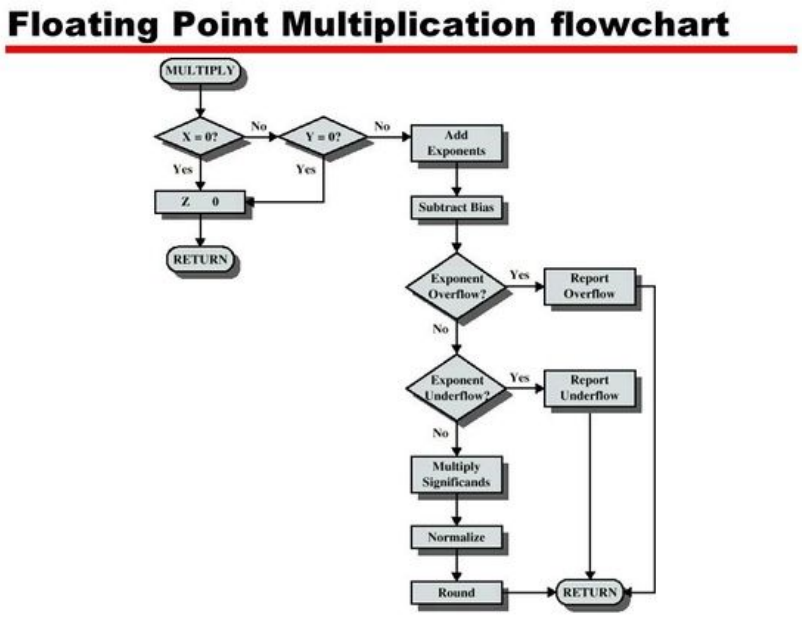

Figure 16: 32 bit Floating Point Multiplication Logic

The sign of the product was computed by XORing the signs of the factors. The exponent of the product was computed by adding the exponents of the factors and subtracting the bias value (127). We do this because each exponent of the factors already contains the bias term and we do not want this value included twice. We then calculate the "intermediate product" by multiplying the mantissa of each factor and normalize this to get the mantissa of the product. Normalizing the mantissa involves right shifting the intermediate product until we have a 1 in the 47th bit. For every right shift we increment the product exponent by 1. We then round the mantissa of the product. Concatenating all these values together gives the 32 bit floating point representation of the product.

We wrote a testbench to verify that the multiplication module worked as expected. **Figure 17** shows that the multiplication works with whole numbers, special cases, and floats.

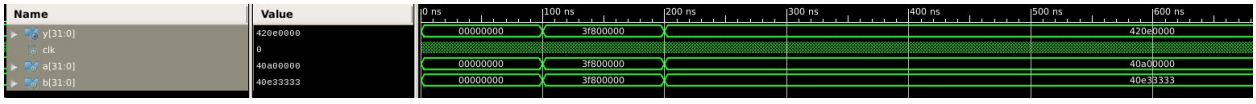

Figure 17: Waveform output for 32 bit float multiplication.

Floating Point Addition:

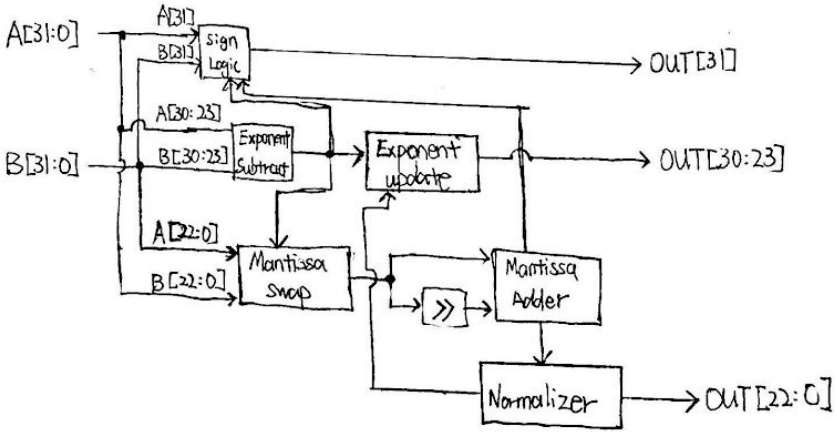

Figure 18: Schematic for floating point addition

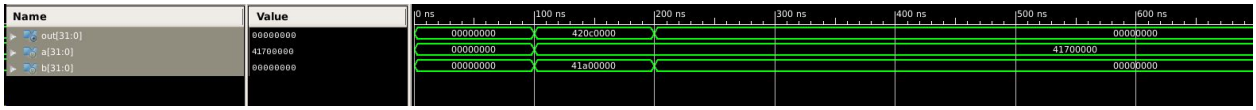

Figure 19: Waveform output for 32 bit float addition.

The logic for the 32 bit floating point adder can be seen in **Figure 18.** Due to limited knowledge on floating point arithmetic we used a module that we found here: https://github.com/danshanley/FPU/blob/master/fpu.v. We verified that the logic of this code followed that of the general floating point addition algorithm. Additionally, we wrote a testbench to verify that the cited code works as expected.

Sigmoid Approximation:

For the neural network, we needed the sigmoid function to calculate the activation of each node based upon its inputs. Since the sigmoid function is extremely computationally intensive for the FPGA, we elected to do the computation for it. Our sigmoid approximation module was accomplished by essentially using a fine granularity look-up table. The look-up table was made with a Python script to generate values of the sigmoid function between negative and positive 5. Anything greater than 5 or less than -5 were assigned values of 1 and 0 respectively.

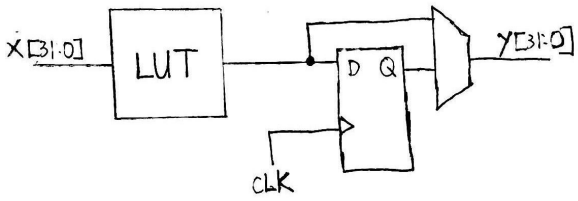

Figure 20: Schematic for the Sigmoid Approximation

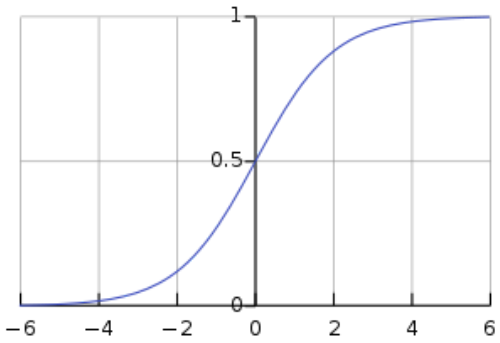

Figure 21: Sigmoid Function visualization

Since the module utilizes a look-up table, we reasoned that it would be acceptable to test the module with an assortment of values in the positive and negative spectrum. All input values returned output values that were expected of the sigmoid function. These test cases and their results are shown in **Figure 22**.

```
// Add stimulus here
x = 32'b11000000111000000000000000000000;    //-7
#10;
x = 32'b11000000101000000000000000000000;    //-5
#10;
x = 32'b11000000010000000000000000000000;    //-3
#10;
x = 32'b10111111110000000000000000000000;    //-1.5
#10;
x = 32'b10111111000000000000000000000000;    //-0.5
#10;
x = 32'b00000000000000000000000000000000;    //0
#10;
x = 32'b00111111000000000000000000000000;    //0.5
#10;
x = 32'b00111111110000000000000000000000;    //1.5
#10;
x = 32'b01000000010000000000000000000000;    //3
#10;
x = 32'b01000000101000000000000000000000;    //5
#10;
x = 32'b01000000111000000000000000000000;    //7
#10;
end
```

Figure 22: Waveform and test cases for the Sigmoid Approximation module

Seven Segment Display/UART:

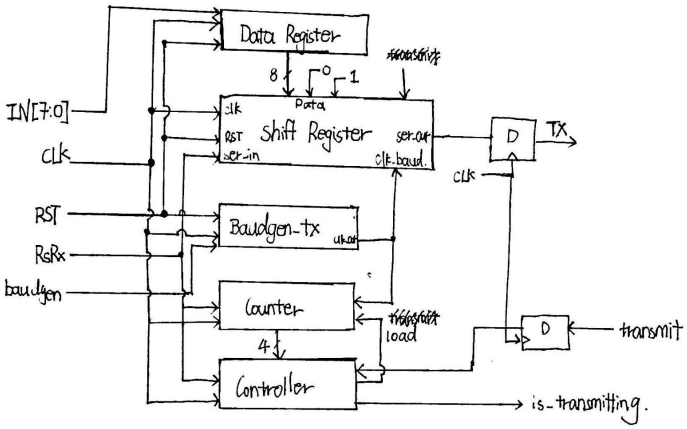

Figure 23: Schematic for Uart

We first choose to use Uart to output the result of our neural network. The uart first takes in the clk, 8 bits input as well as the RsRx, which is the incoming serial line. When the transmit signal is set high, the uart will output the input bit by bit. The shifter will be used to shift the input to output the correct bit, and the counter will be counting the bits we have already sent. The uart outputs through the outgoing serial line Tx, and will stop once the counter hits 8, indicating all 8 bits were sent. The uart were suppose to send the current mode: input, output, or standby by changing the value of IN[7:0] and value of transmit.

However, we didn't get to finish the uart part on time, and therefore we choose to use the seven segment display instead to output our result. The seven segment module takes in a 4-bit binary number (from 0 to 9) and converts it to an 8-bit binary number that will be the representation of that 4-bit number on the seven segment display. This is done via a simple case statement which sets the output of the seven segment display given the corresponding 4-bit binary number. When we finished the neural network, we input the species and output the species number on the seven seg.

Conclusion:

In this lab we used our knowledge of the FPGA, Xilinx ISE, and building Verilog systems from the past labs to create iris flower classifier from scratch. In simulation, our classifier was able to use the numpad to take in features inputs from the user, use the trained neural network to process the inputs and compute a classification, and output the classification to the seven segment display. This lab serves as proof that we can in fact design and create our own Verilog project from scratch according to our own standards.

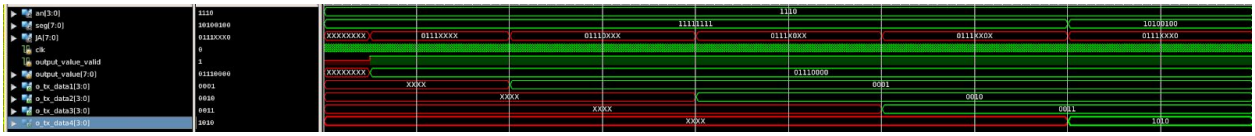

Figure 24: Waveform for top module. Running the simulation twice yields the correct output based off the collected inputs.

From this lab we learned the limitations of the Nexys3 FPGA and the Verilog project development process. Specifically, we learned that it is not enough to verify that the logic of the code works in simulation. We must also check to make sure that the code is synthesizable on the FPGA and can fit the hardware limitations of the board. When synthesizing our code, we realized that our floating point operations had consumed too many resources on the board and thus could not run on it. Nonetheless, the logic works as expected and predicts labels the same as the Python model.