ECE 441 Final Project

Abby Rothschild, Stephen Lamczyk, Jacob Strother, Lamar Simmons-Porter

ECE 441

Spring 2022

I pledge to support the Honor System of Old Dominion University. I will refrain from any form of academic dishonesty or deception, such as cheating or plagiarism. I am aware that as a member of the academic community it is my responsibility to turn in all suspected violations of the Honor Code. I will report to a hearing if summoned.

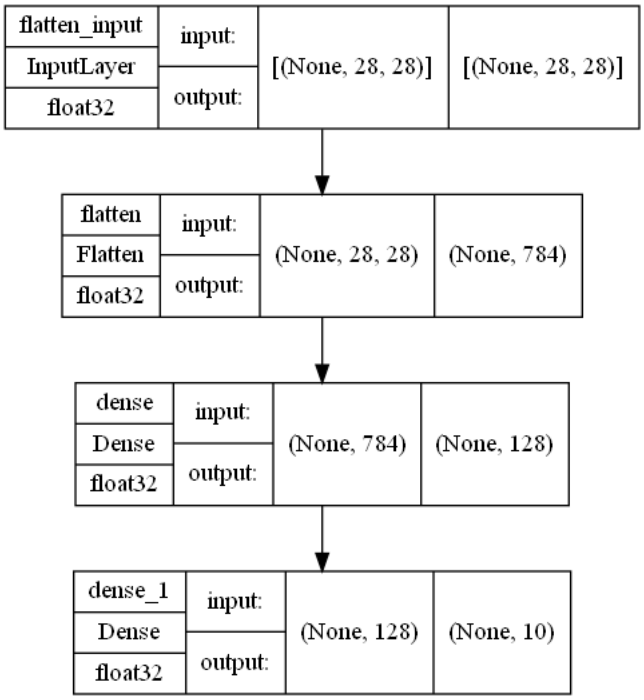
Introduction

In our project, we will be exploring deep neural networks by implementing a neural network in VHDL to use our DE10-Standard FPGA to classify pictures of numbers. Neural networks are sets of algorithms that are designed after the human brain, in which we can input data through machine learning algorithms and develop patterns. We can then translate this data, and for our experiment, we will translate it into pictures of numbers. For this project we utilize the MNIST dataset. The MNIST dataset, created by Yann LeCunn, is a dataset of digits 0-9 and is composed of about 60k images.

Design Work

Before implementing this into VHDL, we decided to start by coding in Python to obtain a better understanding of our problem. We will implement our neural network into Python using TensorFlow, and train it on the MNIST dataset so it can learn to classify each number. Once we trained it with this data, we are able to obtain the weights of each image. We also obtained a sample set from our data so we can compare it with our results from VHDL.

We felt that training the neural network in Python was appropriate since we did not want to implement backpropagation in VHDL. Following the Tensorflow tutorial for creating a neural network to classify the images of the MNIST dataset, we implemented the following neural network. This neural network takes as input an 28x28 grayscale image from the dataset and flattens it to get a 784 long vector. Consequently, it performs a multiplication given the 1x784 vector with the 784x128 weight matrix of the first layer. After that, it adds the 1x128 bias vector. After that addition, we perform the relu activation. After the hidden layer is done computing, it multiples the resulting 1x128 vector by a 128x10 weight matrix and adds the 1x10 bias vector to get our final result. While ideally this neural network would have softmax activation so we could express our final results as a probability, we felt that it was not appropriate since we would have to eventually implement this on the fpga. Consequently, we left off the softmax activation function and just decided that we would perform the argmax on the resulting output to get our class.



Neural Network Python Shape

We then exported these weights and biases over to VHDL in a package as Aldec has trouble with opening large files in the editor. Before doing so, we realized one important thing. Most of our weights and biases were floating point numbers roughly in the e-1 to e-2 range. We did not want to utilize floating point arithmetic in our VHDL implementation so we firstly multiplied the weights and biases by 256 before exporting them to VHDL. We also decided that we would again multiply the normalized image by 256 to avoid floating point arithmetic. Consequently, we decided to spread out 3 divides by 256 into our VHDL implementation to get our final result. We chose 256 over a more round number like 200 so we could use bit shifting instead of more expensive multiplication.

Specifically, we exported the 784x128 and 128x10 weight matrix. We also exported the 1x128 and 1x10 bias vectors. We also exported 10 randomly sampled images from the MNIST dataset to represent each number. Of course, these are all flatten, normalized, and then multiplied by 256. These were all exported as integer arrays.

Before implementing this in VHDL, we implemented this modified version with only integers and multiplies and divides by 256 to ensure it would act reliably.

Here is a test run with an image of a zero.

```
print(model.predict(np.array([image/255])))
```
[12] ✓ 0.1s

```
[[ 14.84053   -11.898269    0.6348648  -2.536228   -3.0690982  -5.322214
  -1.9863118  -4.632834   -5.037287    5.048931 ]]
```

```
image = (image / 255 ) * 256
image = image.flatten().astype(int)
```
[13] ✓ 0.3s

```
with open("image.txt", "w") as f:
    f.write("(")
    for i in range(len(image)):
        f.write(str(image[i]) + ",")
    f.write(")")
```
[14] ✓ 0.5s

```
intermediate_weights = (model.layers[1].get_weights()[0] * 256).astype(int)
intermediate_output = (model.layers[1].get_weights()[1] * 256).astype(int)

final_weights = (model.layers[2].get_weights()[0] * 256).astype(int)
final_output = (model.layers[2].get_weights()[1] * 256).astype(int)
```
[15] ✓ 0.3s

```
intermediate_output = ((np.matmul(image, intermediate_weights) + intermediate_output)/256).astype(int)
intermediate_output[intermediate_output < 0] = 0
```
[16] ✓ 0.2s

```
final_output = ((np.matmul(intermediate_output, final_weights) + final_output)/256).astype(int)
```
[17] ✓ 0.4s

```
(final_output/256).astype(int)
```
[18] ✓ 0.5s

```
array([ 14, -11,   0,  -2,  -3,  -5,  -1,  -4,  -4,   5])
```

It is evident that our integer implementation performs closely to the actual floating point implementation.

Afterwards, we felt confident in our ability to implement it in VHDL. We implemented 6 states in our VHDL model.

Our zeroeth state just sets the hidden output and final output equal to the bias. This saves us the extra state of adding the bias later on.

```
when 0 ⇒
    report("State 0");
    output_1 := bias_1;

    output_2 := bias_2;
    state ⩽ 1;
```

We originally implemented state 1 with plain matrix multiplication but found that Aldec was trying to use around 1.2 million logic elements. After making the original matrix multiplication sequential, we settled on the following implementation that instead of executing

the 784x128 loops all at once, it only executed one per clock cycle. While this did result in a large time needed for execution, we did not enough have enough time needed to explore how far we could go with parallelizing this.

```
when 1 =>
report("State 1");
    if(i1 < 128 and j1 < 784) then
        output_1(i1) := output_1(i1) + img(j1) * weights_1(j1, i1);
    end if;
        --report(integer'image(j));
    if i1 = 128 then
                    i1 := 0;
                    j1 := 0;
                    state <= 2;
    else
                    if j1 = 784 then
                      j1 := 0;
                      i1 := i1 + 1;
                    else
                      j1 := j1 + 1;
                    end if;
                    state <= 1;
    end if;
```

After calculating the 1x128 hidden output vector, we performed relu activation in a similar manner to how we made the loops in state one sequential. This means that instead of utilizing a regular for loop and having quartus unroll it, we just had each clock cycle implement one index of the loop.

```
when 2 =>
    report("State 2");
    if(i1 < 128) then
        if(output_1(i1) < 0) the
            output_1(i1) := 0;
        end if;
    end if;

    if i1 = 128 then
        i1 := 0;
        state <= 3;
    else
        i1 := i1 + 1;
        state <= 2;
    end if;
```

After this, we divided our 1x128 vector by 256 and performed our 128x10 multiplication in a similar manner to the 784x128 matrix multiplication. Afterwards, we followed these two states with two extra states dividing by 256 to get our final result. The result is also implemented as a 1x10 output integer array.

The results will be output to the six seven-segment displays [HEX0...HEX5] using a multiplexer, whose inputs are SW[4] and the integer value output by the neural network. For the integer value to be used a conversion to signed 32-bit is necessary. Since the multiplexer will be selecting the resulting upper and lower 24-bits to the displays and the select is 1-bit, this allows for a simple implementation of a case or an if-else statement. For this project a case was chosen, whose conditions are when SW[4] is ‘0’ or off, the resulting lower 24-bits are output to the displays, whereas the upper 24 bits are displayed when SW[4] is ‘1’ or on.

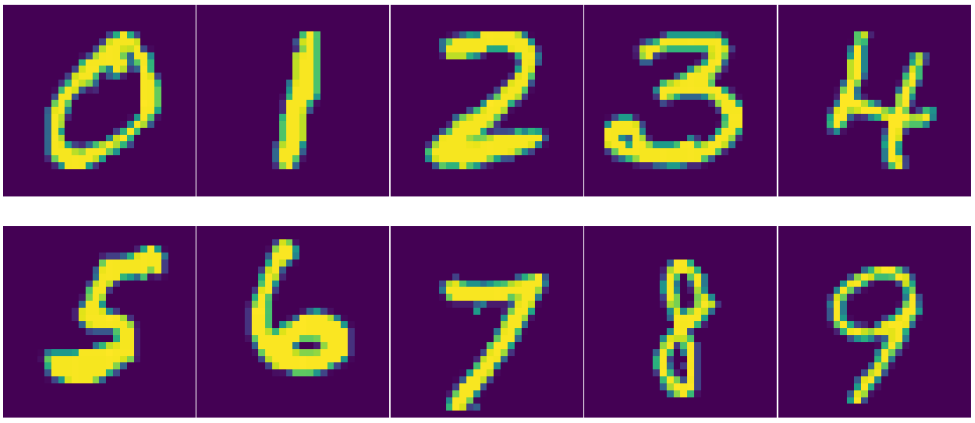
```
process(all)
begin
   val <= to_signed(value,32);
   case S is
         when '1' => sendOut <= val(31 downto 8);
         when others => sendOut <= val(23 downto 0);
   end case;
```

The six seven-segment displays are common anode displays, which are active low (requiring a ‘0’), to activate its individual segments. To accomplish this, a function containing each possible hexadecimal value matched its corresponding 7-bit value. The 7-bit value will be sent to the displays to either turn the segments on or off.

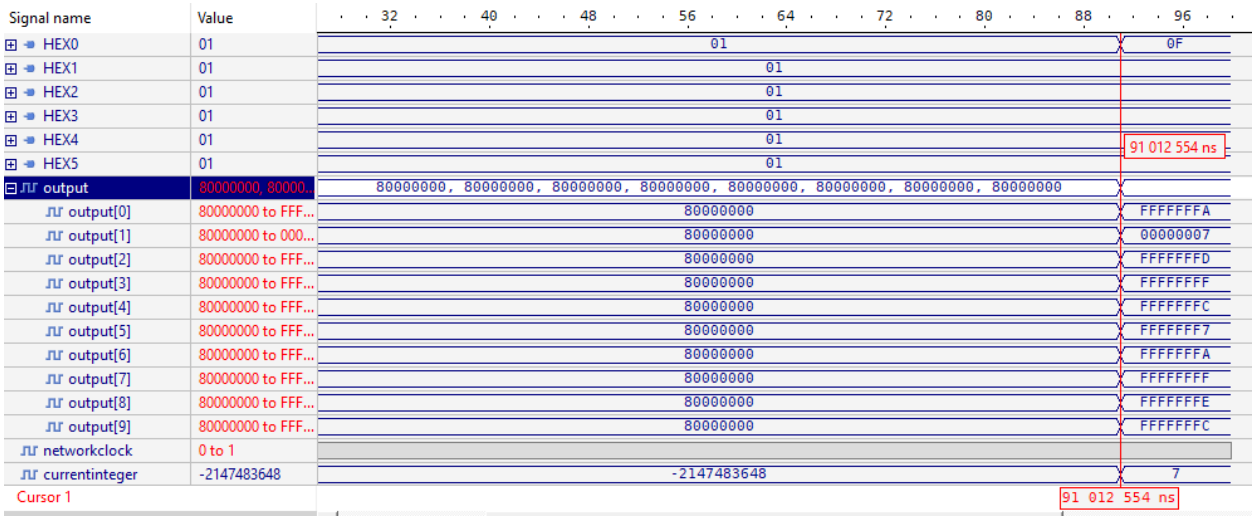
```
when x"7" => ret := not "1110000";
```

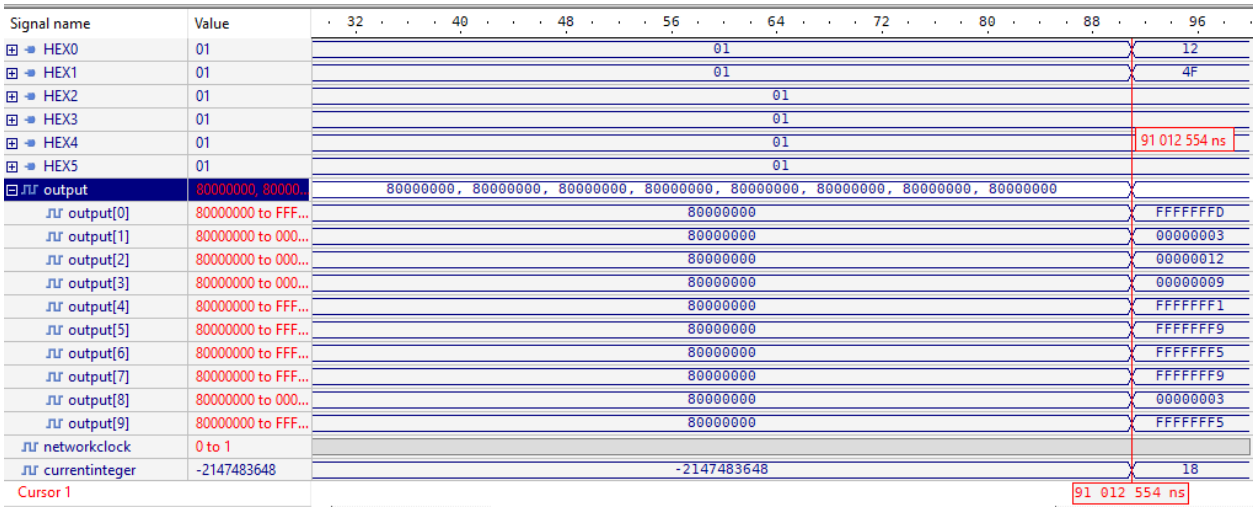
Simulation Results

After training our MNIST data, we can observe that our results in python will be images of each number 0 to 9.



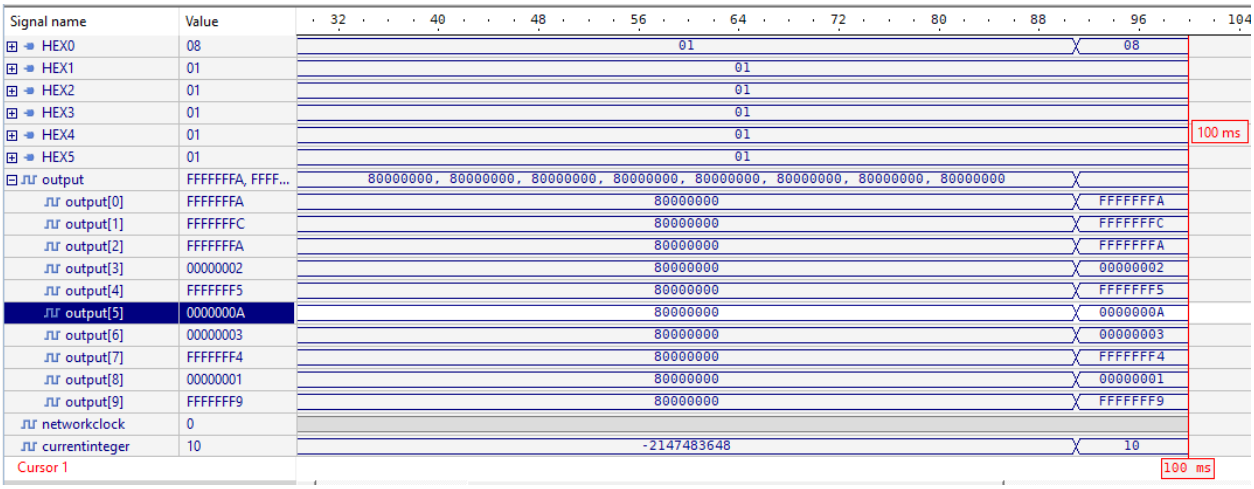
In Aldec, we can run our test bench and obtain a waveform diagram showing which numbers are sent to the output. The waveforms below simulate the images from 0 to 9.

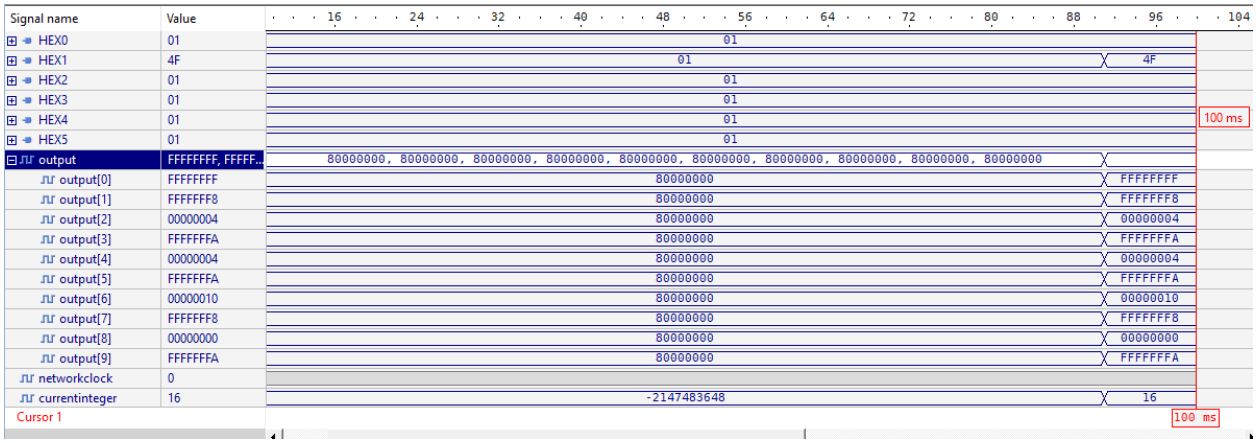

Signal name
Value
32 40 48 56 64 72 80 88 96
HEX0 01 01 0F
HEX1 01 01
HEX2 01 01
HEX3 01 01
HEX4 01 01 91 012 554 ns
HEX5 01 01
output 80000000, 80000... 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000
output[0] 80000000 to FFF... 80000000 FFFFFFFA
output[1] 80000000 to 000... 80000000 00000007
output[2] 80000000 to FFF... 80000000 FFFFFFFD
output[3] 80000000 to FFF... 80000000 FFFFFFFF
output[4] 80000000 to FFF... 80000000 FFFFFFFC
output[5] 80000000 to FFF... 80000000 FFFFFFF7
output[6] 80000000 to FFF... 80000000 FFFFFFFA
output[7] 80000000 to FFF... 80000000 FFFFFFFF
output[8] 80000000 to FFF... 80000000 FFFFFFFE
output[9] 80000000 to FFF... 80000000 FFFFFFFC
networkclock 0 to 1
currentinteger -2147483648 -2147483648 7
Cursor 1
91 012 554 ns

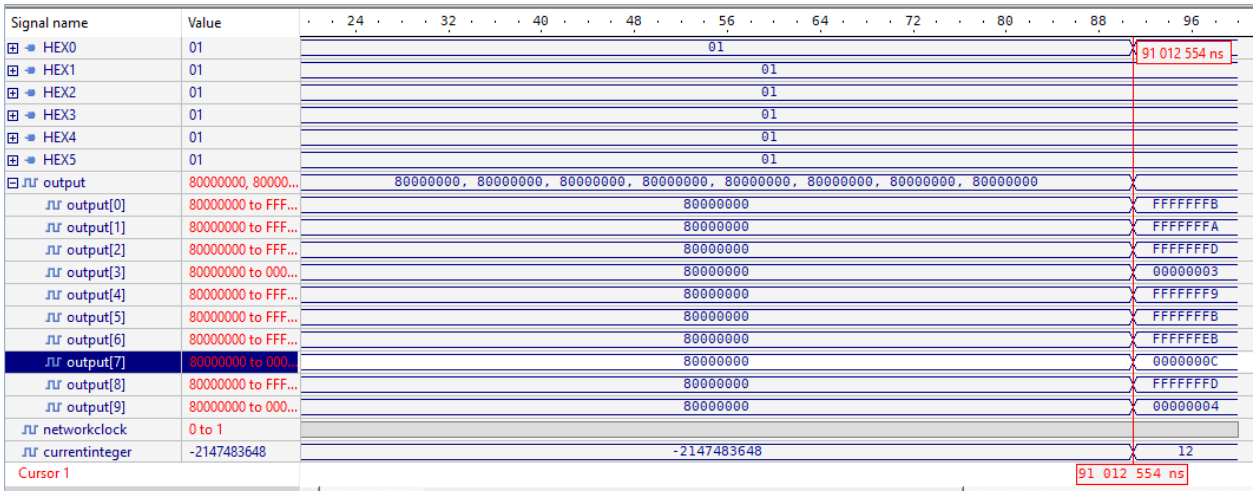

Signal name
Value
32 40 48 56 64 72 80 88 96
HEX0 01 01 12
HEX1 01 01 4F
HEX2 01 01
HEX3 01 01
HEX4 01 01 91 012 554 ns
HEX5 01 01
output 80000000, 80000... 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000
output[0] 80000000 to FFF... 80000000 FFFFFFFD
output[1] 80000000 to 000... 80000000 00000003
output[2] 80000000 to 000... 80000000 00000012
output[3] 80000000 to 000... 80000000 00000009
output[4] 80000000 to FFF... 80000000 FFFFFFF1
output[5] 80000000 to FFF... 80000000 FFFFFFF9
output[6] 80000000 to FFF... 80000000 FFFFFFF5
output[7] 80000000 to FFF... 80000000 FFFFFFF9
output[8] 80000000 to 000... 80000000 00000003
output[9] 80000000 to FFF... 80000000 FFFFFFF5
networkclock 0 to 1
currentinteger -2147483648 -2147483648 18
Cursor 1
91 012 554 ns

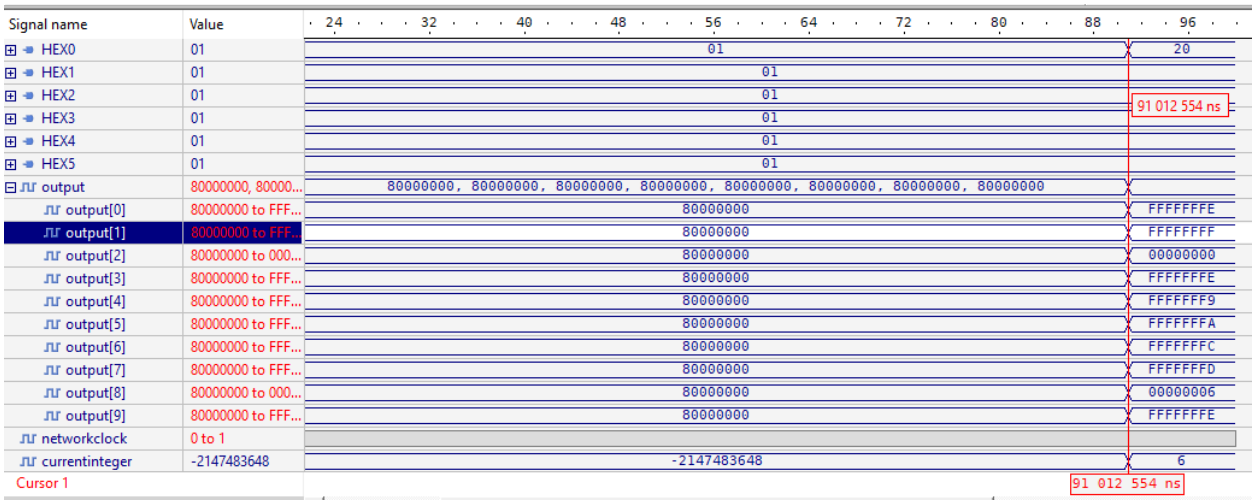

Signal name
Value
32 40 48 56 64 72 80 88 96
HEX0 01 01 42
HEX1 01 01
HEX2 01 01
HEX3 01 01
HEX4 01 01 91 012 554 ns
HEX5 01 01
output 80000000, 80000... 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000
output[0] 80000000 to FFF... 80000000 FFFFFFFB
output[1] 80000000 to FFF... 80000000 FFFFFFFE
output[2] 80000000 to 000... 80000000 00000005
output[3] 80000000 to 000... 80000000 0000000D
output[4] 80000000 to FFF... 80000000 FFFFFFF9
output[5] 80000000 to 000... 80000000 00000002
output[6] 80000000 to FFF... 80000000 FFFFFFF2
output[7] 80000000 to FFF... 80000000 FFFFFFF8
output[8] 80000000 to FFF... 80000000 FFFFFFFF
output[9] 80000000 to FFF... 80000000 FFFFFFF9
networkclock 0 to 1
currentinteger -2147483648 -2147483648 13
Cursor 1
91 012 554 ns

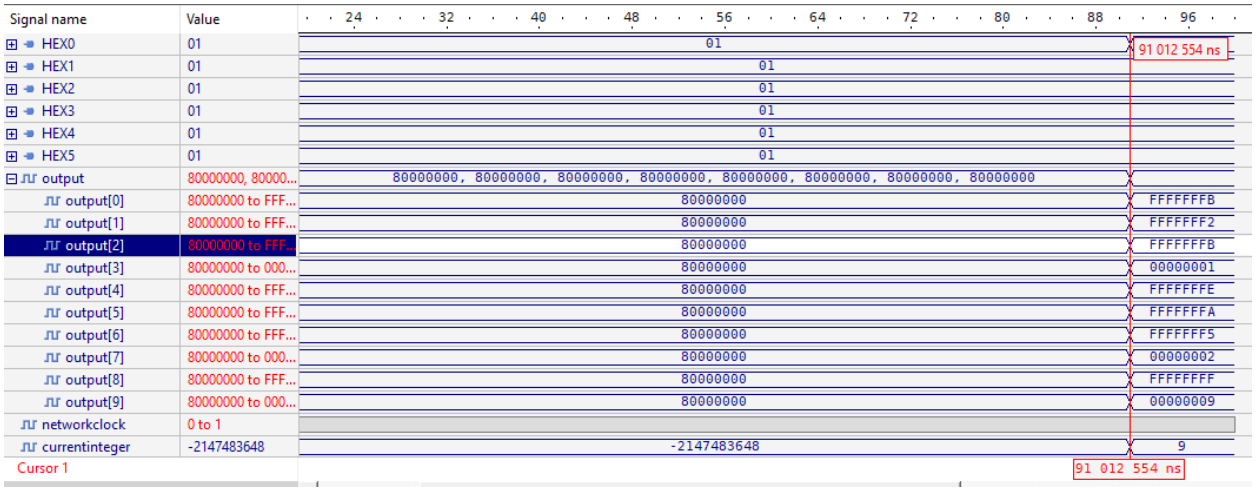

Signal name
Value
32
40
48
56
64
72
80
88
96
104
HEX0
08
01
08
HEX1
01
01
HEX2
01
01
HEX3
01
01
HEX4
01
01
100 ms
HEX5
01
01
output
FFFFFFFB, FFFF...
80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000
output[0]
FFFFFFFB
80000000
FFFFFFFB
output[1]
FFFFFFF7
80000000
FFFFFFF7
output[2]
FFFFFFFF
80000000
FFFFFFFF
output[3]
FFFFFFFB
80000000
FFFFFFFB
output[4]
0000000A
80000000
0000000A
output[5]
FFFFFFFE
80000000
FFFFFFFE
output[6]
FFFFFFFE
80000000
FFFFFFFE
output[7]
00000000
80000000
00000000
output[8]
FFFFFFFC
80000000
FFFFFFFC
output[9]
FFFFFFFF
80000000
FFFFFFFF
networkclock
0
currentinteger
10
-2147483648
10
Cursor 1
100 ms


Signal name
Value
32
40
48
56
64
72
80
88
96
104
HEX0
08
01
08
HEX1
01
01
HEX2
01
01
HEX3
01
01
HEX4
01
01
100 ms
HEX5
01
01
output
FFFFFFFA, FFFF...
80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000
output[0]
FFFFFFFA
80000000
FFFFFFFA
output[1]
FFFFFFFC
80000000
FFFFFFFC
output[2]
FFFFFFFA
80000000
FFFFFFFA
output[3]
00000002
80000000
00000002
output[4]
FFFFFFF5
80000000
FFFFFFF5
output[5]
0000000A
80000000
0000000A
output[6]
00000003
80000000
00000003
output[7]
FFFFFFF4
80000000
FFFFFFF4
output[8]
00000001
80000000
00000001
output[9]
FFFFFFF9
80000000
FFFFFFF9
networkclock
0
currentinteger
10
-2147483648
10
Cursor 1
100 ms

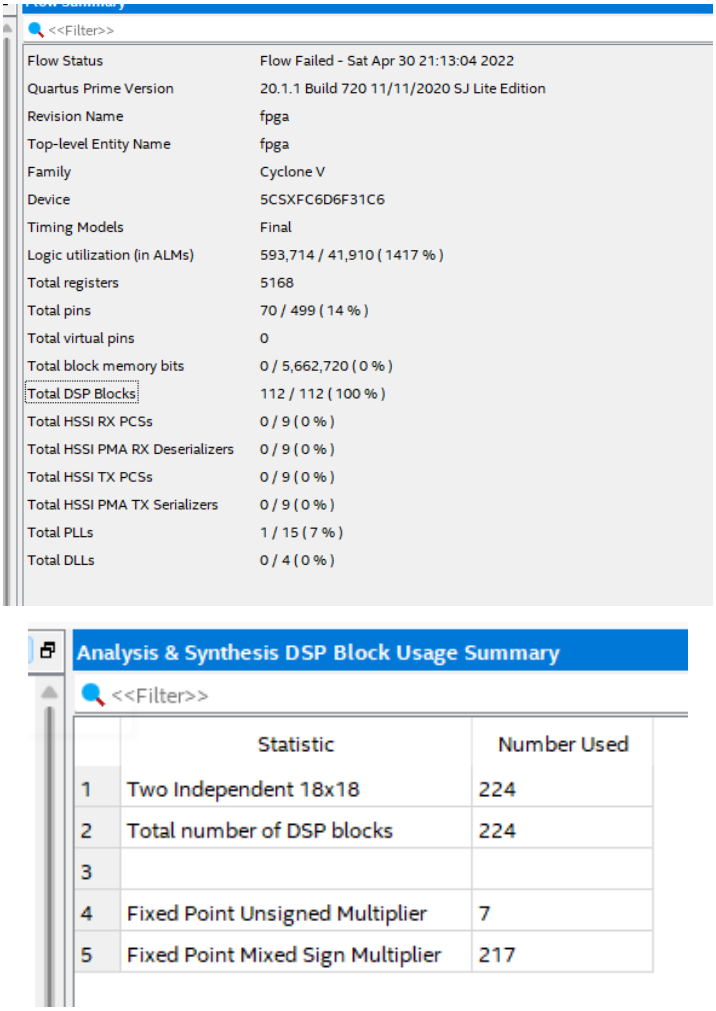

Signal name
Value
16
24
32
40
48
56
64
72
80
88
96
104
HEX0
01
HEX1
4F
HEX2
HEX3
HEX4
HEX5
output
FFFFFFFF, FFFFF...
80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000
output[0]
FFFFFFFF
output[1]
FFFFFFF8
output[2]
00000004
output[3]
FFFFFFFA
output[4]
00000004
output[5]
FFFFFFFA
output[6]
00000010
output[7]
FFFFFFF8
output[8]
00000000
output[9]
FFFFFFFA
networkclock
0
currentinteger
16
-2147483648
Cursor 1
100 ms


Signal name
Value
24
32
40
48
56
64
72
80
88
96
HEX0
01
91 012 554 ns
HEX1
HEX2
HEX3
HEX4
HEX5
output
80000000, 80000...
80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000, 80000000
output[0]
80000000 to FFF...
FFFFFFFB
output[1]
FFFFFFFA
output[2]
FFFFFFFD
output[3]
80000000 to 000...
00000003
output[4]
FFFFFFF9
output[5]
output[6]
FFFFFFEB
output[7]
80000000 to 000...
0000000C
output[8]
FFFFFFFD
output[9]
00000004
networkclock
0 to 1
currentinteger
-2147483648
12
Cursor 1
91 012 554 ns





Once we synthesize into Quartus, we can program our FPGA board and observe the results on the six hex displays. We noticed that the synthesis used a large amount of RAM on our personal computers. After letting it synthesize for an hour and a half, we noticed an error stating that we have too many logic devices, so we went back and broke up some of our for loops to reduce the memory used. The edits we made are described in the Design Work section.



<<Filter>>

Flow Status	Flow Failed - Sat Apr 30 21:13:04 2022
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	fpga
Top-level Entity Name	fpga
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	593,714 / 41,910 (1417 %)
Total registers	5168
Total pins	70 / 499 (14 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total DSP Blocks	112 / 112 (100 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	1 / 15 (7 %)
Total DLLs	0 / 4 (0 %)

Analysis & Synthesis DSP Block Usage Summary

<<Filter>>

	Statistic	Number Used
1	Two Independent 18x18	224
2	Total number of DSP blocks	224
3		
4	Fixed Point Unsigned Multiplier	7
5	Fixed Point Mixed Sign Multiplier	217

Errors from Quartus syntheses

Once we successfully synthesize, we can program and observe our FPGA board.

DE10-Standard Setup/Observations

We will be displaying data on the 6 hex displays, HEX[0..5], and using the on board switch, SW[4] to switch between the lower and upper bits.

Switches SW[3..0] will display the current integer, switches SW[8..5] will display the image. The results from uploading the program onto the FPGA are the same as the simulation results from Aldec and Python. An important note is that the 91 ms it takes to switch images and achieve the result is significantly noticeable.

Analysis

The table below compares our integer results between Python, Aldec, and the output on our FPGA board.

Image	Python (Integer TensorFlow)	Aldec	FPGA
0	14, -11, 0, -2, -3, -5, -1, -4, -4, 5	15, -12, 0, -2, -4, -6, -2, -6, -5, 5	15, -12, 0, -2, -4, -6, -2, -6, -5, 5
1	-5, 7, -3, 0, -3, -8, -5, -1, -1, -3	-6, 7, -3, -1, -4, -9, -6, -1, -2, -4	-6, 7, -3, -1, -4, -9, -6, -1, -2, -4
2	-3, 3, 18, 9, -14, -6, -10, -6, 3, -11	-3, 3, 18, 9, -15, -7, -11, -7, 3, -11	-3, 3, 18, 9, -15, -7, -11, -7, 3, -11
3	-5, 0, 5, 13, -6, 2, -13, -7, 0, -6	-5, -2, 5, 13, -7, 2, -14, -8, -1, -7	-5, -2, 5, 13, -7, 2, -14, -8, -1, -7
4	-4, -9, 0, -4, 10, -1, -1, 1, -3, 0	-5, -9, -1, -5, 10, -2, -2, 0, -4, -1	-5, -9, -1, -5, 10, -2, -2, 0, -4, -1
5	-5, -3, -5, 2, -10, 10, 3, -11, 1, -6	-6, -4, -6, 2, -11, 10, 3, -12, 1, -7	-6, -4, -6, 2, -11, 10, 3, -12, 1, -7
6	0, -7, 4, -5, 5, -5, 16, -8, 0, -5	-1, -8, 4, -6, 4, -6, 16, -8, 0, -6	-1, -8, 4, -6, 4, -6, 16, -8, 0, -6
7	-4, -6, -2, 3, -6, -4, -19, 13, -2, 4	-5, -6, -3, 3, -7, -5, -21, 12, -3, 4	-5, -6, -3, 3, -7, -5, -21, 12, -3, 4
8	-1, 0, 0, -1, -6, -5, -3, -2, 6, -1	-2, -1, 0, -2, -7, -6, -4, -3, 6, -2	-2, -1, 0, -2, -7, -6, -4, -3, 6, -2
9	-4, -13, -3, 1, -1, -5, -9, 2, 0, 9	-5, -14, -5, 1, -2, -6, -11, 2, -1, 9	-5, -14, -5, 1, -2, -6, -11, 2, -1, 9

We can see that our results between the three methods roughly match up together, showing that we successfully implemented the neural network into VHDL. Our final model in Quartus used 20,432 (49%) ALMs and 4 (4%) DSP blocks. The setup and hold slack values were all positive with the slow model having values of 805 and 0.2. The max frequency that our model can utilize is 20.79 MHz.

Summary and Conclusion

Overall, this experiment showed us the behaviors of the MNIST data set as a neural network, and we learned how to implement this into VHDL and onto our FPGA boards. We were able to train a neural network on a data set, obtain images from the numbers, and successfully import this data into a VHDL file. We also successfully synthesized our code into Quartus and onto our DE10-Standard FPGA board. To improve our design, we can parallelize just the right amount of multiplication to use the maximum amount of DSPs, and thus make our model run even faster.

References

http://yann.lecun.com/exdb/mnist/

Github with all files: https://github.com/salamczyk/ece441final