Final Report

Timothy Baalman

ECEN 4303

Contents

Introduction

This was a very interesting project to do, but due to the complexity of the project I did not get as much done as I initially intended. I have the Network fully built and tested verifying that correct digits were detected for the loaded in pictures. So, I did complete the project, but I did not get around to implementing the dadda-multiplier, the CLA, or any extra layers. I do have a tested CLA module though. I also implemented a softmax I have the network's output without softmax saved, but I am only including the softmax network results. There was some optimization complexity of connecting everything together that I will discuss later, but the required optimization took a long amount of time to implement. Which is why I did not get to the dadda-multiplier, or the CLA. I made my own custom float to fixed-point converter as well as a fixed-point to float converter. The later one was used to verify results of the 32-bit adder and 32-bit multiplier. I wrote my code in such a way that increasing the bit address to 64-bits or adding more layers would require changing a couple parameters. Also, if more layers were to be added I would have needed to retrain the network and convert the weight and bias values. My weights and bias were based off my trained PyTorch network that I got to be 100% accurate.

Training the Network

My network consists of one linear layer and an output layer. The linear layer has 784 inputs to accommodate the 784 pixels values from the loaded in 28x28 pictures. The linear layer then outputs 64 values. Since, that seemed like a nice computer number amount. The output layer takes in the 64 outputs from the linear layer and will have 10 outputs where each output corresponds to the digit guess value from the network. Meaning that, the highest value on index 0 means the network guessed the input digit to be a 0.

Multi-Parameter Training

To allow me to quickly get to the best accuracy with my PyTorch Network I utilized a python dictionary. It contained arrays for learning rate, batch size, shuffling, and epochs. The arrays had varying data types and could vary in the amount of data in the array. This method allowed me to test multiple types and get to the 100% accuracy. To get the values that would be best it is mainly trial and error. So, doing multiparameter just speeded up the trial and error, but understanding each parameter also help.

Where the learning rate determines how far the optimization will step. Having it too big could result in the optimization jumping back and forth from two points as it gets closer to the min value, but too small and it might not jump far enough within the given number of epochs being ran. When initially setting my values for the learning rate I only strayed three decimal places either way from 0.01. Ultimately, the learning rate that had the best result was 0.01. Also, the batch size determines how many pictures are going to be loaded in, to train on. Generally, the more the better. Next is shuffling, where shuffling just determines if the pictures will be shuffled each time. This does means that shuffle is either true or false. Finally, the number of epochs indicates the number of times that PyTorch will correct the Network.

Utilizing the GPU

My laptop has a Nvidia GPU which is cuda capable, and PyTorch makes using cuda cores ridiculously easy. By simply setting the torch.device to cuda:0 indicating the first GPU on the system, and then using .to(the set device) everything was loaded to the GPU. Using the GPU allows for larger batch sizes to be trained on, and does it way faster than what can be done on the CPU. Which allowed me to reach the 100% accuracy faster by using big batch sizes. If I was not going to use large batch sizes using the GPU could be slower since there is some delay from passing the data from the CPU to the GPU.

Outputting to JSON File

Inside my training code I kept track of the accuracy and loss of each epoch. Whenever a better accuracy and loss appears I would save the weights and biases to the corresponding json element. After all epochs are ran the code will write the json data to best_results.json and add a new entry into history_best_results.json. The history was used as a reference for which parameters have gotten me the best results. The weights and biases are saved as floating point in those files to be parsed later.

Converting to Signed Fixed Point

Parsing Best Results

For converting the floating point to fixed point, I load the data from best_resuls.json file looping through the weight and bias arrays passing the value through my converter to a new json element. Then outputting each weight array to its corresponding dat file. Since the weights are 2D arrays where the first index corresponds to the node the weights belong to making one dat file for each node. I set the code to put the biases for one layer in one file passing the correct bias to the node in the layer.

My converter does signed fixed point conversion with Q17.14 in Qm.n notation where m is the number of integer bits and n is the number fractional bits, and I am using a 32-bit address. Initially I was thinking that there did not need to be many bits for m, but then I started thinking about how the final values are calculated. Where all the input pixels are times by a weight value and then added together. Doing a worst-case calculation to determine m does not really make sense to do since an entire white picture is not going to happen, and only a few weights ever reached the highest value of 10 that I saw. But for doing the worst-case I made that assumption that all the pixels have a value of 1 and the weights all have a value of 10. Leaving 784 times 10 + 1 on the first layer adding one for bias. Which is then passed to 64 other nodes where each weight is also 10 and we add 1 for the bias resulting in a value of 5,018,241 for the worst-case integer value. Now the smallest observed fractional value was .0001010…, and meeting the smallest fractional was the standard I used since the worst-case for the integer values did not seem like it would happen and 2^{-14} yielded 0.00006103515625 as my smallest fractional value. I then gave the rest of the bits minus one for the sign bit to the integer part. Resulting in a max in value of 131,071. The converter takes in a number multiplies the number by 2^{14} then converts to binary if negative number does 2C’s conversion.

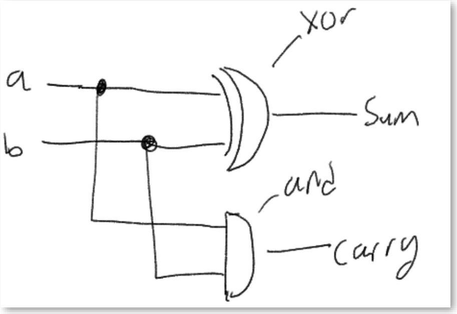
Parsing Pixel values

For getting image files I utilized the data loader similarly to training the network. I passed in a large batch size and then loop through until a 0 target label was found, then for a 1, and so on ending at 10. Each time the target label matches the current wanted image the code would loop through the image passing every pixel value through my converter then to the dat file for that image target. I only output one image file for each digit, but the image used is randomized. I can also rerun the parse_image_data.py file to get new image file values without needing to change anything in the Verilog.

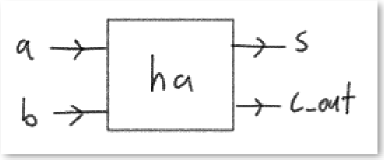
Verilog Modules

Half Adder

1) Design

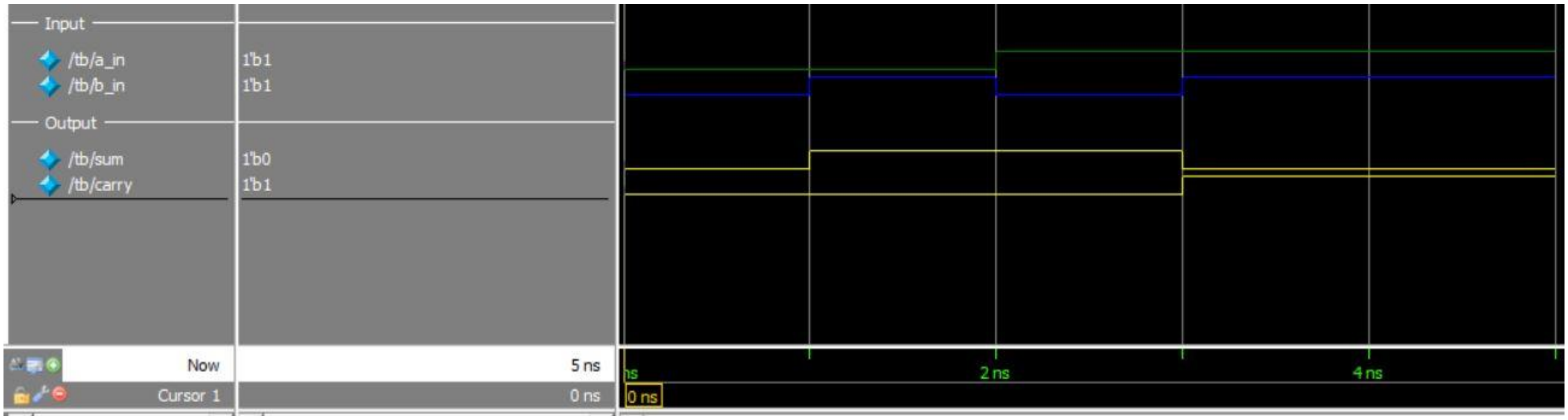


Half Adder Gate Level



Half Adder Module

2) Verification

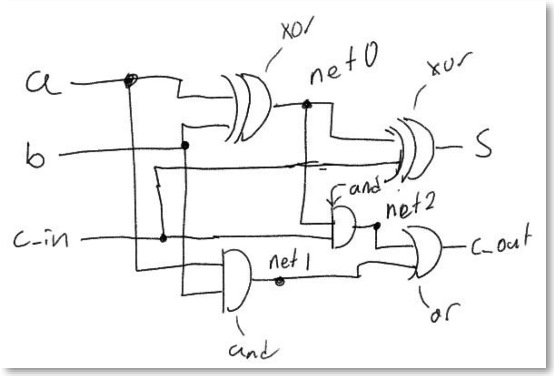


```
Input
/tb/a_in     1'b1
/tb/b_in     1'b1
Output
/tb/sum      1'b0
/tb/carry    1'b1
Now          5 ns
Cursor 1     0 ns
```

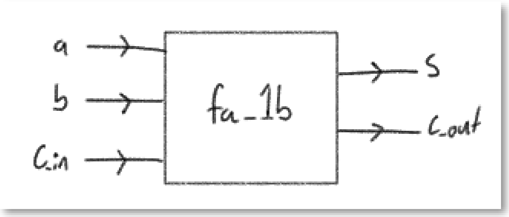
Waveform Half Adder

Full Adder and Carry Ripple Adder

1) 1-Bit FA Design

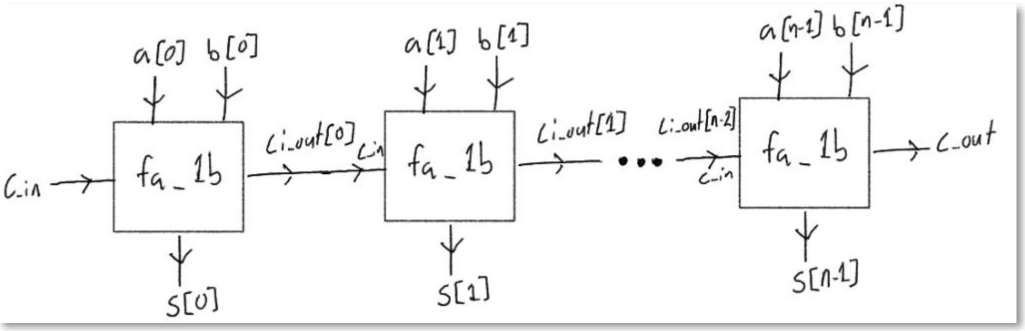


1-Bit Full Adder Gate Level

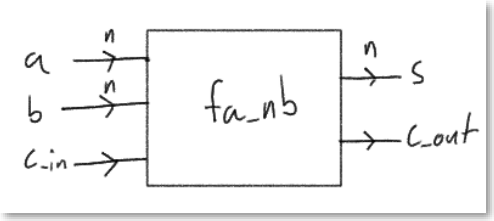


1-Bit Full Adder Module

2) N-Bit CRA (N-Bit FA) Design

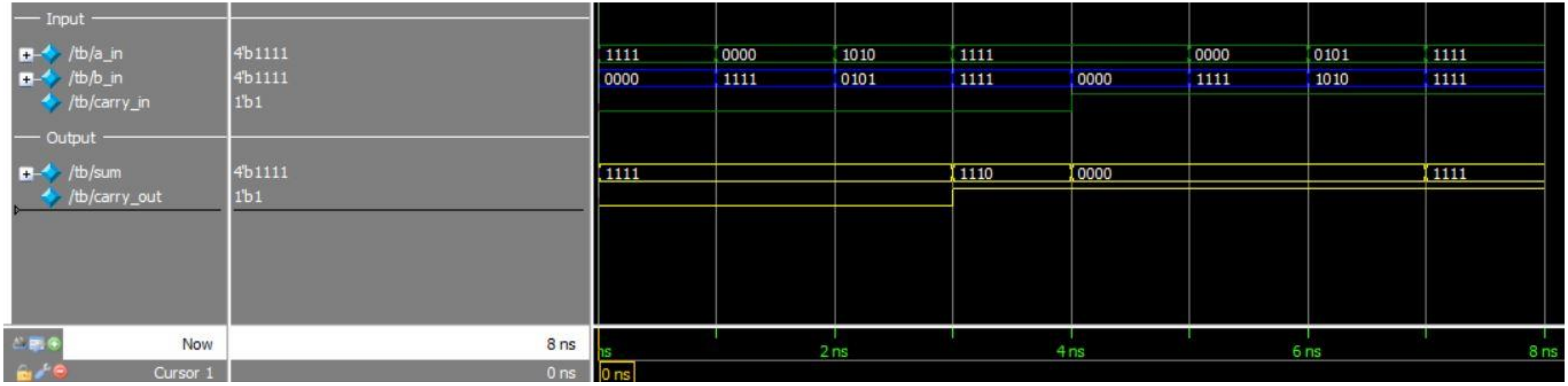


N-bit Full Adder Submodule Level



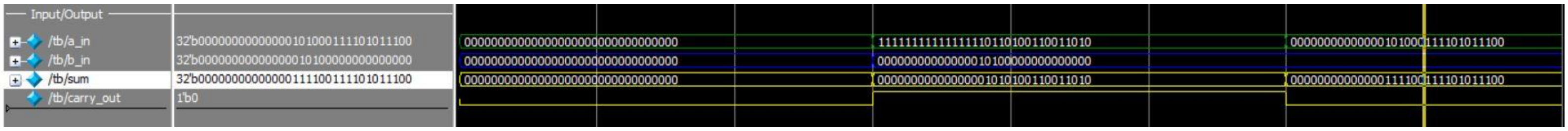
N-Bit Full Adder Module

3) 4-Bit Verification

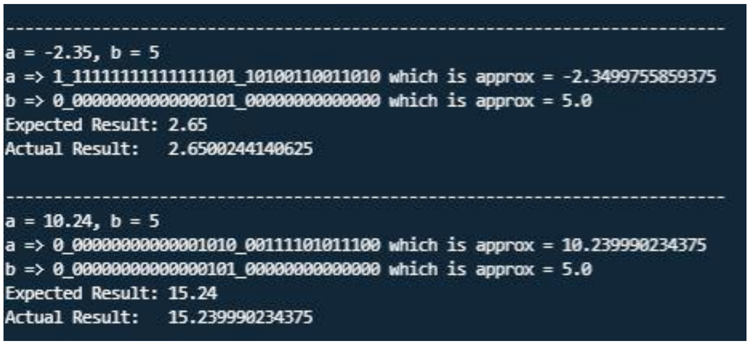


Waveform 4-Bit Full Adder

4) 32-Bit Verification



Waveform 32-Bit Full Adder

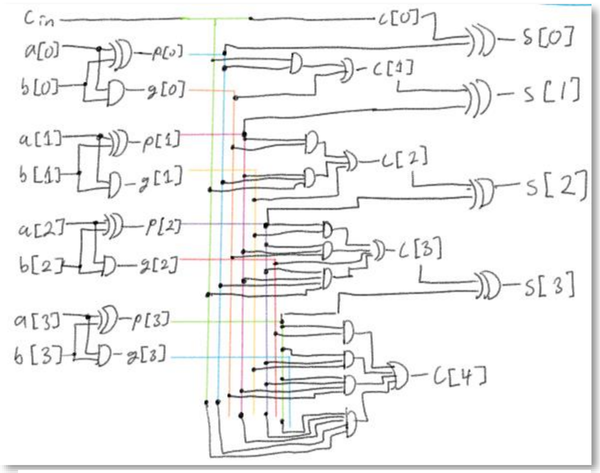


```
---------------------------------------------------------------------------
a = -2.35, b = 5
a => 1_1111111111111101_10100110011010 which is approx = -2.3499755859375
b => 0_0000000000000101_00000000000000 which is approx = 5.0
Expected Result: 2.65
Actual Result:   2.6500244140625

---------------------------------------------------------------------------
a = 10.24, b = 5
a => 0_0000000000001010_00111101011100 which is approx = 10.239990234375
b => 0_0000000000000101_00000000000000 which is approx = 5.0
Expected Result: 15.24
Actual Result:   15.239990234375
```

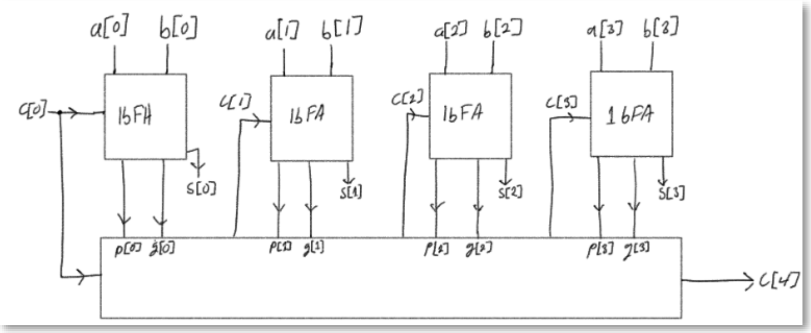
Waveform 32-Bit FA Values from Fixed Point to Floating Point

Carry Lookahead Adder

1) 4-Bit Design

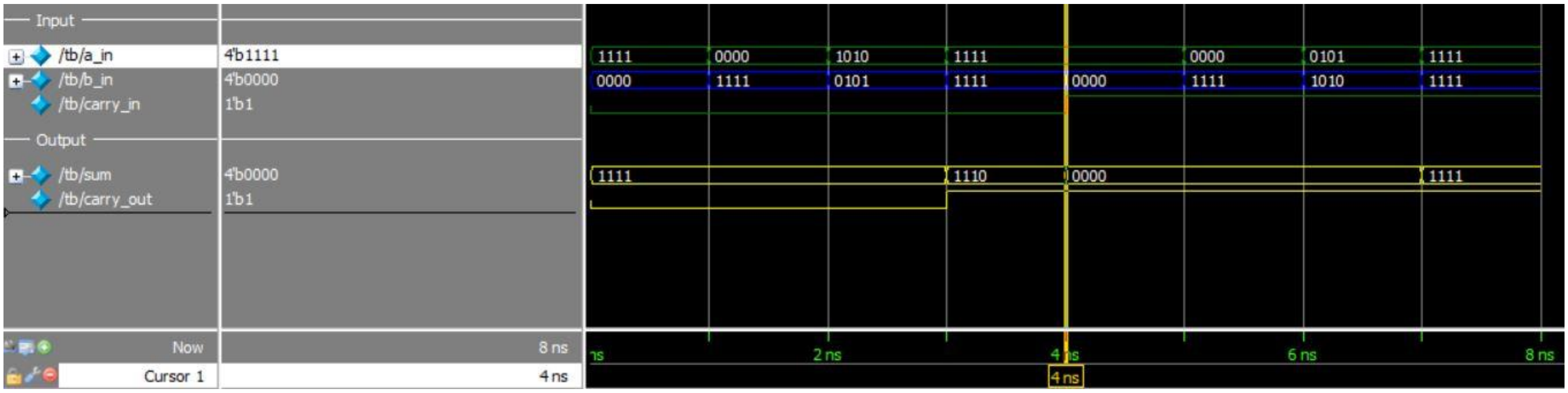


4-bit CLA Gate Level



4-bit CLA Module

2) 4-Bit Verification

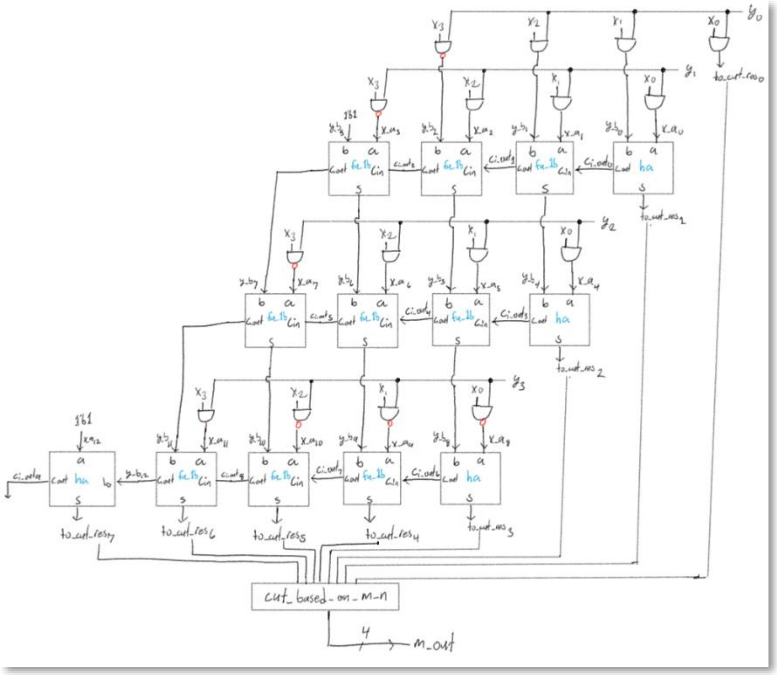


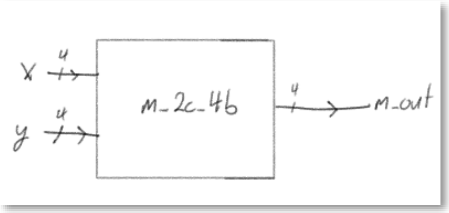
```
Input
/tb/a_in        4'b1111   1111 | 0000 | 1010 | 1111 |      | 0000 | 0101 | 1111
/tb/b_in        4'b0000   0000 | 1111 | 0101 | 1111 | 0000 | 1111 | 1010 | 1111
/tb/carry_in    1'b1
Output
/tb/sum         4'b0000   1111 | 1110 | 0000 | 1111
/tb/carry_out   1'b1
Now             8 ns      ns   2 ns   4 ns   6 ns   8 ns
Cursor 1        4 ns      4 ns
```

4-bit CLA Waveform

Two’s Compliment Multiplier

1) 4-Bit Design

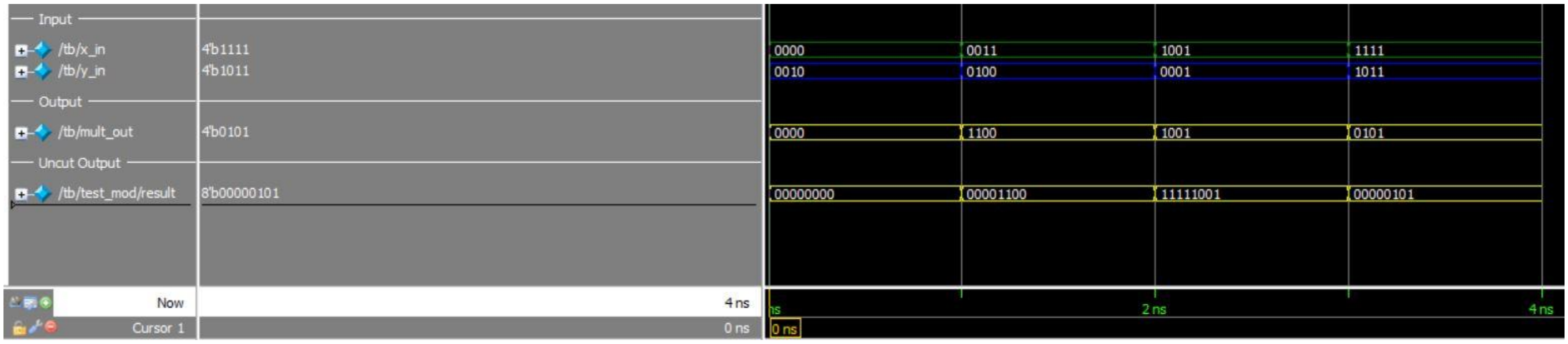




4-bit 2C Multiplier Module

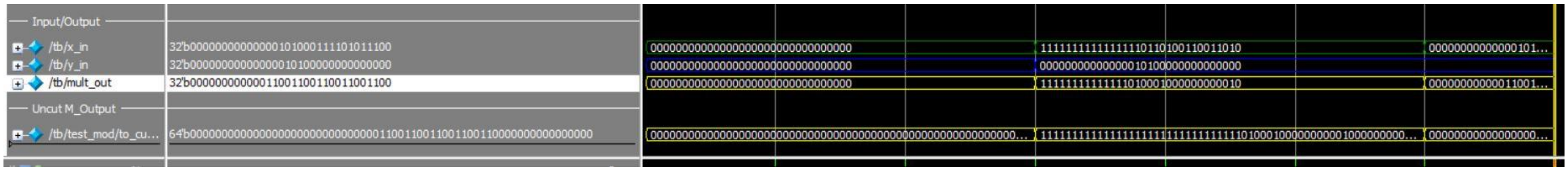
4-bit 2C Multiplier Submodule Level

2) 4-Bit Verification

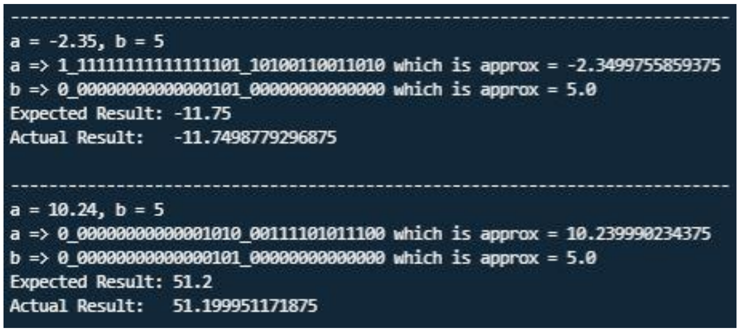

```
Input
/tb/x_in           4'b1111     0000      0011      1001      1111
/tb/y_in           4'b1011     0010      0100      0001      1011
Output
/tb/mult_out       4'b0101     0000      1100      1001      0101
Uncut Output
/tb/test_mod/result 8'b00000101 00000000 00001100 11111001 00000101

Now       4 ns          ns        2 ns        4 ns
Cursor 1  0 ns      0 ns
```

4-bit 2C Multiplier Waveform

3) 32-Bit Verification



32-bit 2C Multiplier Waveform

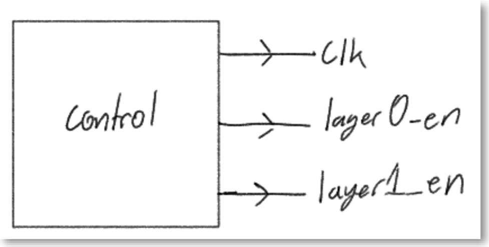


```
---------------------------------------------------------------------------
a = -2.35, b = 5
a => 1_11111111111111101_10100110011010 which is approx = -2.3499755859375
b => 0_00000000000000101_00000000000000 which is approx = 5.0
Expected Result: -11.75
Actual Result:   -11.7498779296875

---------------------------------------------------------------------------
a = 10.24, b = 5
a => 0_00000000000001010_00111101011100 which is approx = 10.239990234375
b => 0_00000000000000101_00000000000000 which is approx = 5.0
Expected Result: 51.2
Actual Result:   51.199951171875
```

Waveform 32-Bit 2C Mult. Values from Fixed Point to Floating Point

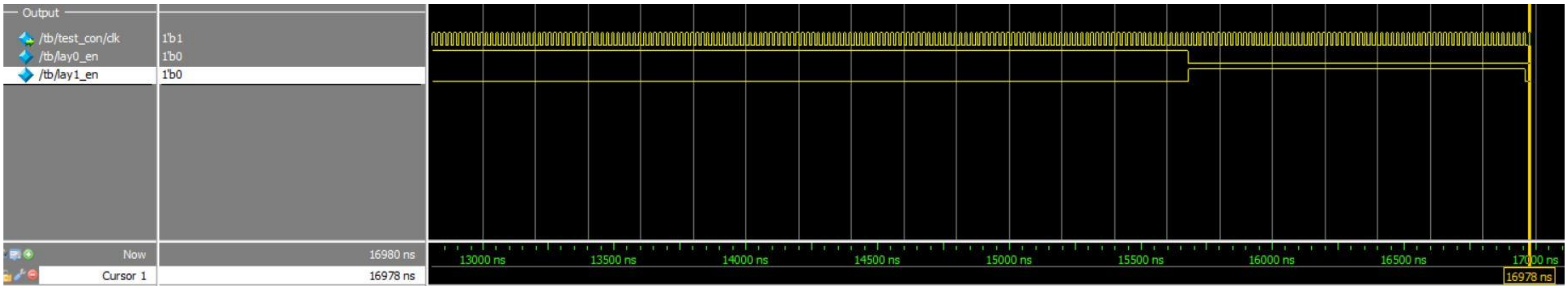
Control

1) Design



Control Module

2) Verification

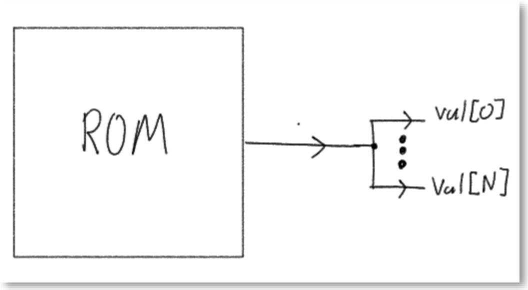


Control Waveform

ROMs

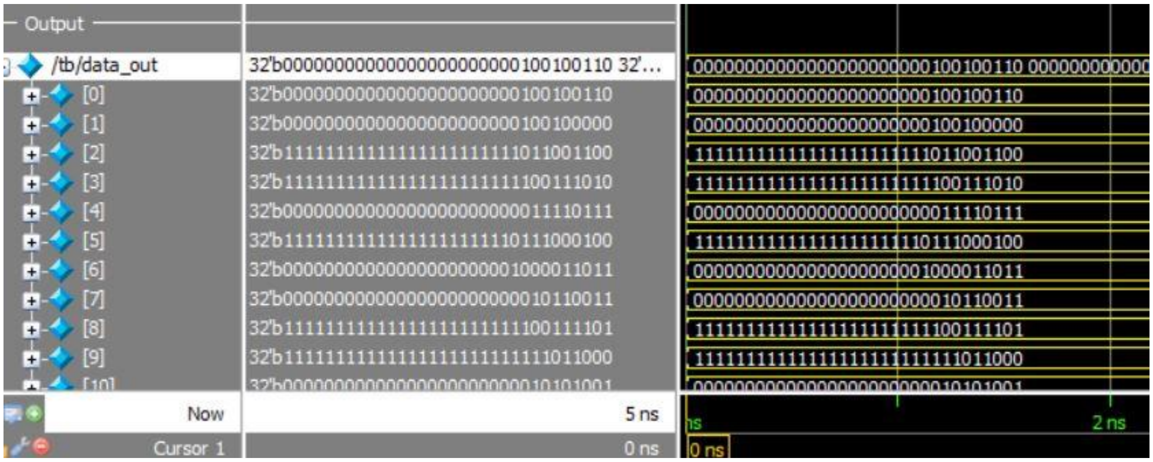
Each ROM reads from their corresponding dat file and passes the read in values to a wire array.

1) ROM Design



ROM Module

2) ROM Verification

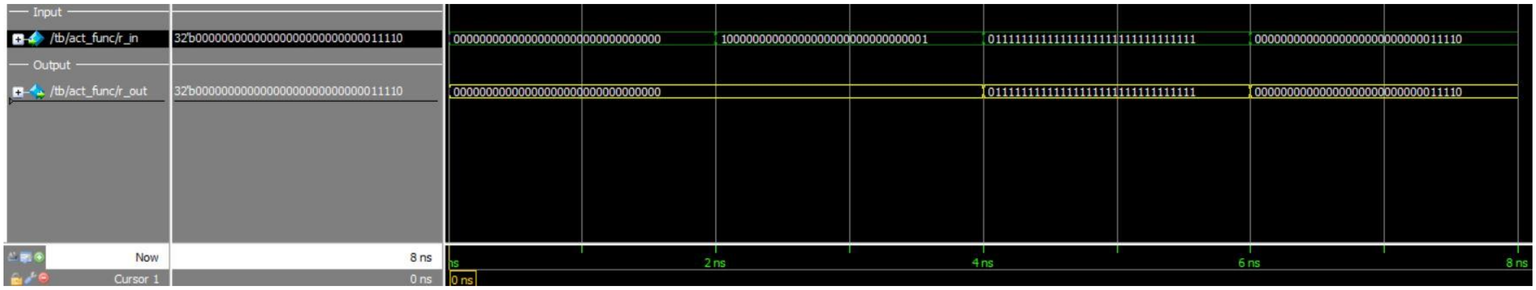


```
Output
/tb/data_out   32'b00000000000000000000000100100110 32'...   00000000000000000000000100100110 0000000000000
  [0]          32'b00000000000000000000000100100110          00000000000000000000000100100110
  [1]          32'b00000000000000000000000100100000          00000000000000000000000100100000
  [2]          32'b11111111111111111111111011001100          11111111111111111111111011001100
  [3]          32'b11111111111111111111111100111010          11111111111111111111111100111010
  [4]          32'b00000000000000000000000011110111          00000000000000000000000011110111
  [5]          32'b11111111111111111111110111000100          11111111111111111111110111000100
  [6]          32'b00000000000000000000001000011011          00000000000000000000001000011011
  [7]          32'b00000000000000000000000010110011          00000000000000000000000010110011
  [8]          32'b11111111111111111111111100111101          11111111111111111111111100111101
  [9]          32'b11111111111111111111111111011000          11111111111111111111111111011000
  [10]         32'b00000000000000000000000010101001          00000000000000000000000010101001
Now            5 ns                                          ns   2 ns
Cursor 1       0 ns                                          0 ns
```

ROM Waveform for Layer 0 Weight 0

Relu

1) Verification



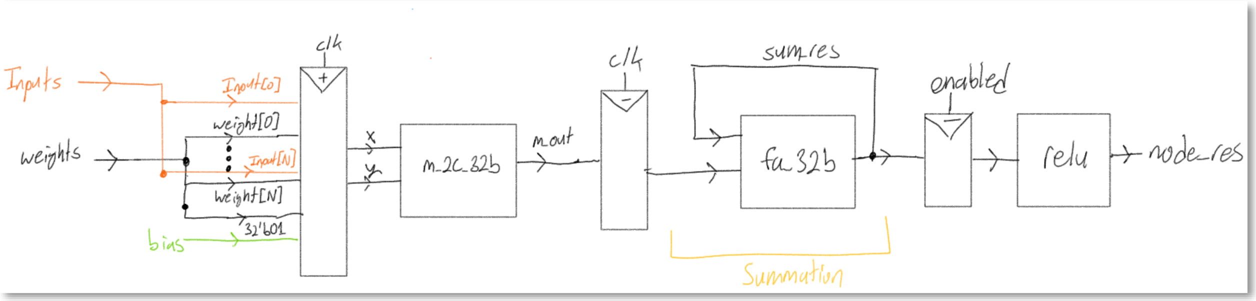
```
Input
/tb/act_func/r_in    32'b00000000000000000000000000011110   00000000000000000000000000000000   10000000000000000000000000000001   01111111111111111111111111111111   00000000000000000000000000011110
Output
/tb/act_func/r_out   32'b00000000000000000000000000011110   00000000000000000000000000000000                                      01111111111111111111111111111111   00000000000000000000000000011110
Now                  8 ns                                   ns   2 ns   4 ns   6 ns   8 ns
Cursor 1             0 ns                                   0 ns
```

Relu Waveform

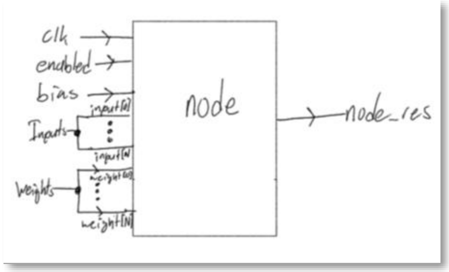
I did not individually test the node, layer, softmax, or network modules. I did not test them because with my python generated Verilog code it was easier to verify the fully connected Network gave the correct output. Which eliminated the need to do individual testing. Also, individual testing of these modules would have resulted in more work needing to be done to spilt them into individual parts and determine what the expected result of the individual part should be. I did have some issues, but once I found the issue, I just needed to make the change in my python code, and any issue would be fixed everywhere.

Nodes

1) Design



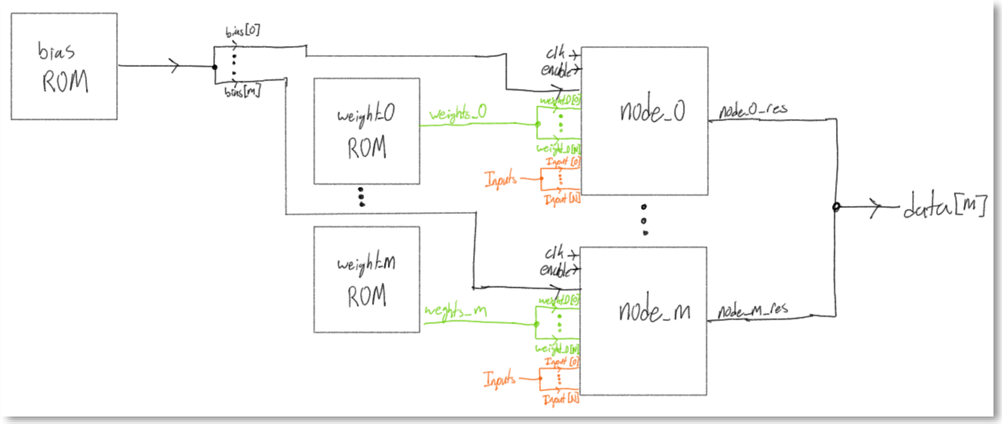
Node Submodule Level



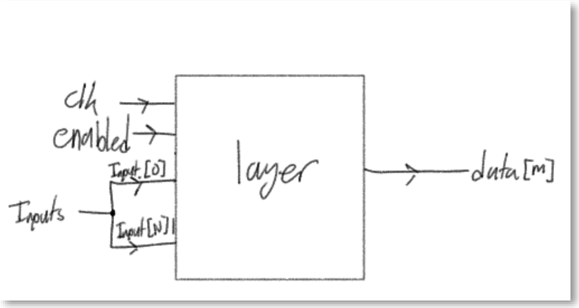
Node Module

Layers

1) Design



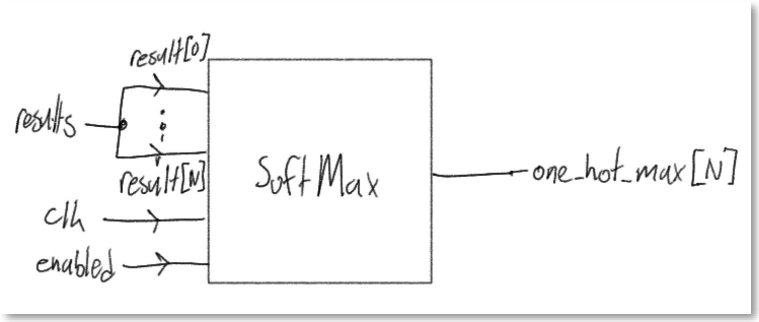
Layer Submodule Level



Layer Module

SoftMax

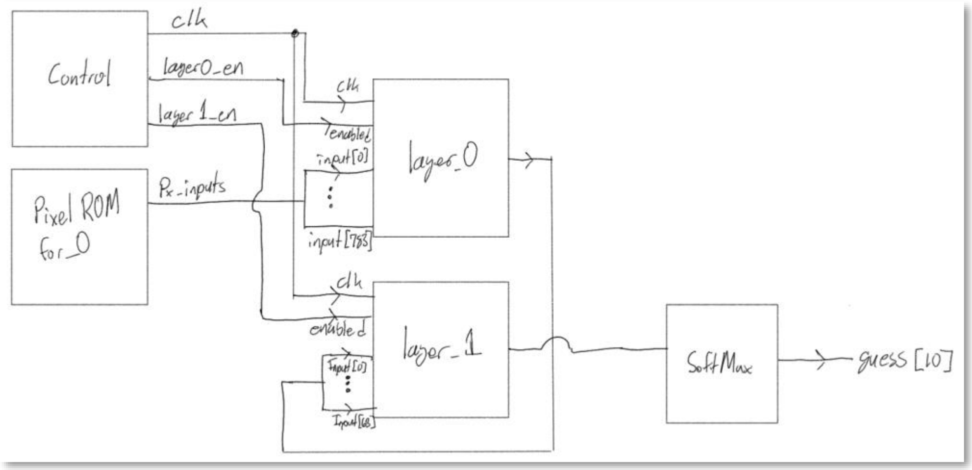
1) Design



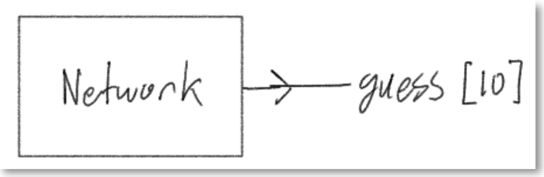
SoftMax Module

Network

1) Design



Network Submodule Level



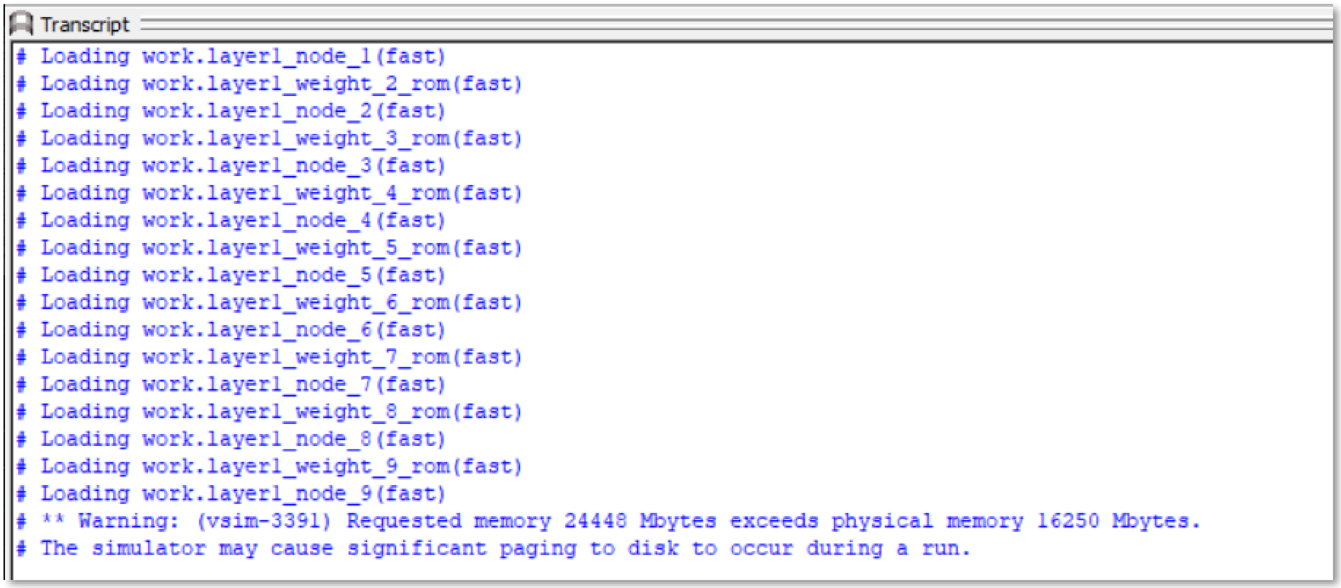
Network Module

Connecting the Network

All the modules and connections are generated through python. I created multiple class for unique modules that a separate python file can access to implement each module. Then it writes all the modules to a single System Verilog file. The files are in the Python-Verilog-Export folder named Network.sv, Network.do, and Network_tb.sv.

Initial Attempt

I originally was not using a clk or layer enable signals. Not using a clk meant that I was creating over 784 adders and multipliers for each node in first layer, and over 64 in the second. The output System Verilog file was over 300,000 lines and ModelSim gave a warning saying that I needed 24448 Mbytes of RAM to run my design. ModelSim did crash after the warning giving memory allocation errors.



```
# Loading work.layer1_node_1(fast)
# Loading work.layer1_weight_2_rom(fast)
# Loading work.layer1_node_2(fast)
# Loading work.layer1_weight_3_rom(fast)
# Loading work.layer1_node_3(fast)
# Loading work.layer1_weight_4_rom(fast)
# Loading work.layer1_node_4(fast)
# Loading work.layer1_weight_5_rom(fast)
# Loading work.layer1_node_5(fast)
# Loading work.layer1_weight_6_rom(fast)
# Loading work.layer1_node_6(fast)
# Loading work.layer1_weight_7_rom(fast)
# Loading work.layer1_node_7(fast)
# Loading work.layer1_weight_8_rom(fast)
# Loading work.layer1_node_8(fast)
# Loading work.layer1_weight_9_rom(fast)
# Loading work.layer1_node_9(fast)
# ** Warning: (vsim-3391) Requested memory 24448 Mbytes exceeds physical memory 16250 Mbytes.
# The simulator may cause significant paging to disk to occur during a run.
```

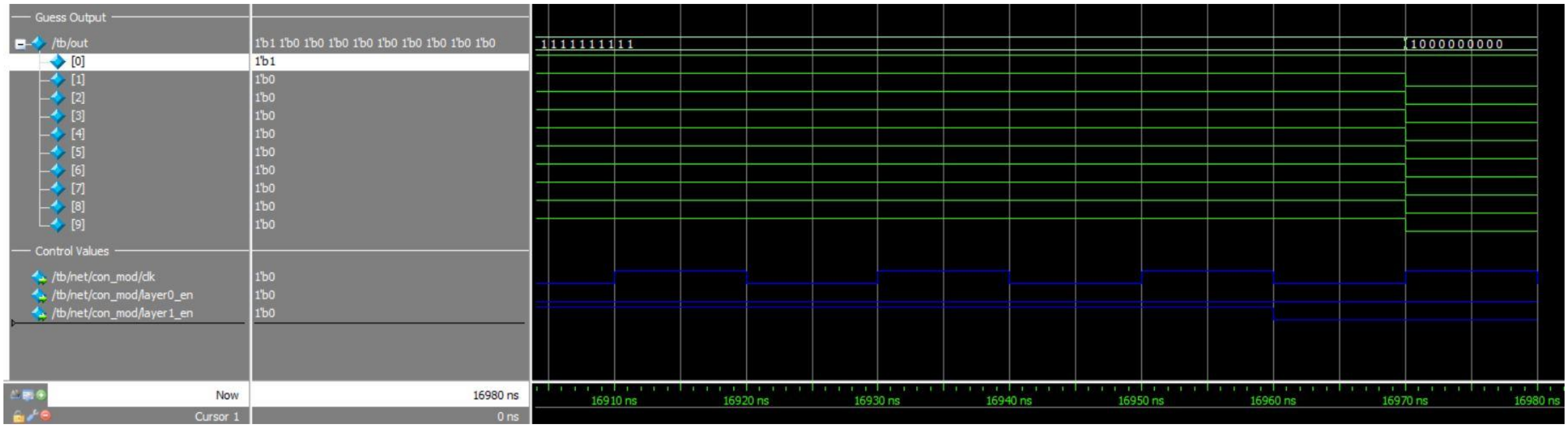
ModelSim Memory Warning Message

Optimization

After the memory issue happened, I went about figuring out how to optimize the code. First thing I needed to do was to just use one adder and multiplier for each node. To do that I utilized a control module which handled the clk and used always blocks inside the nodes to give input values and output values at different points in the clk cycle to the adder and multiplier. The next issue was having both layers running at the same time. The first layer needs to be complete before the next layer starts. So, I added in layer enable signals to the control module which prevented layers from running when not enabled. The enable signals are done based on keeping track of the clock cycles where a raise and fall is one cycle. The first layer requires 784 cycles to finish, and the second requires 784 + 64 cycles. The multiplying is done on the posedge of the clk and the adding is done on the negedge. The relu operation is perform on the negedge of the enable signal. The adding also feeds into itself for getting the summation and takes the result from multiplying the input by the weight. I also appended a 1 to the weight and the bias to the input allowing the me to not need to add another step in for adding the bias. Another optimization I would want to do if I had more time would be to have a reset for the whole system.

Completed Network Results

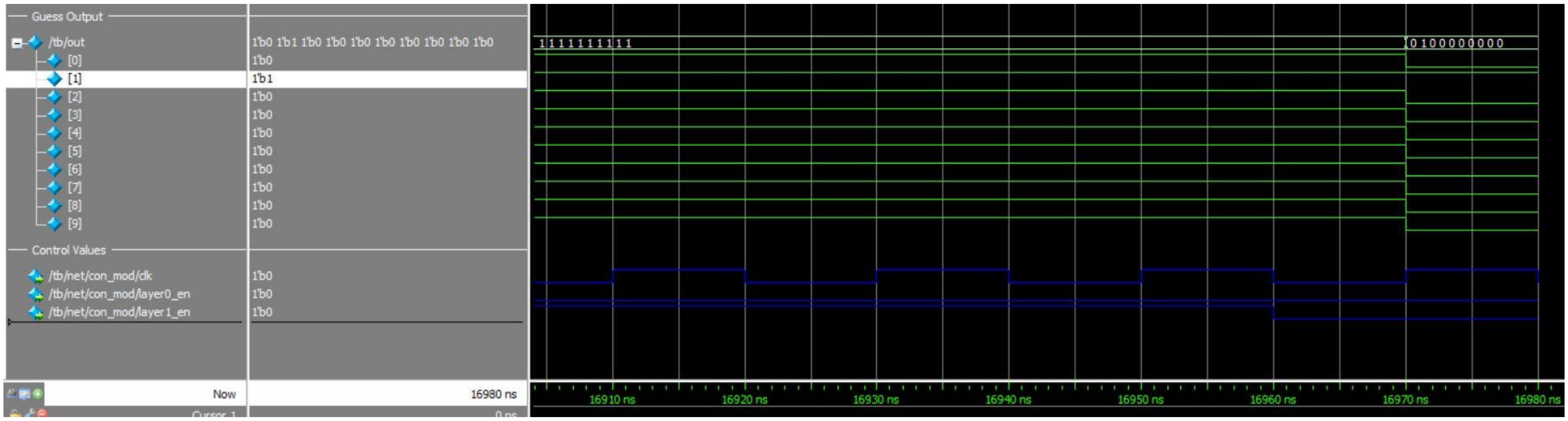
Results From Loading Digit 0



```
Guess Output
/tb/out                      1'b1 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0
  [0]                        1'b1
  [1]                        1'b0
  [2]                        1'b0
  [3]                        1'b0
  [4]                        1'b0
  [5]                        1'b0
  [6]                        1'b0
  [7]                        1'b0
  [8]                        1'b0
  [9]                        1'b0
Control Values
/tb/net/con_mod/clk          1'b0
/tb/net/con_mod/layer0_en    1'b0
/tb/net/con_mod/layer1_en    1'b0
Now                          16980 ns
Cursor 1                     0 ns
```

Waveform with SoftMax Detecting Digit 0 Correctly

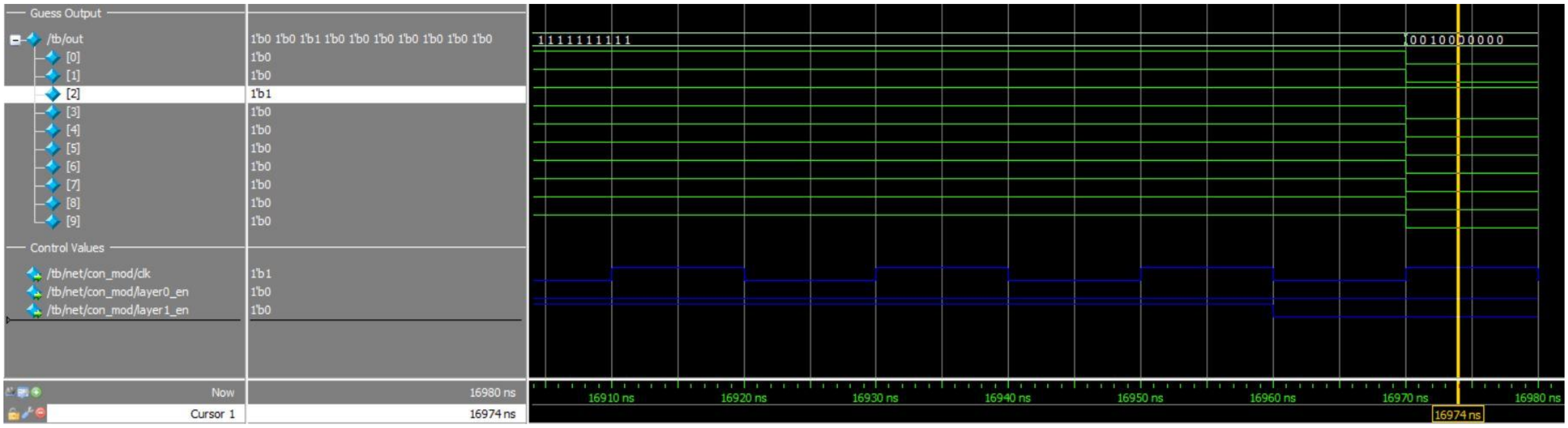
Results From Loading Digit 1



```
Guess Output
/tb/out                      1'b0 1'b1 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0
  [0]                        1'b0
  [1]                        1'b1
  [2]                        1'b0
  [3]                        1'b0
  [4]                        1'b0
  [5]                        1'b0
  [6]                        1'b0
  [7]                        1'b0
  [8]                        1'b0
  [9]                        1'b0
Control Values
/tb/net/con_mod/clk          1'b0
/tb/net/con_mod/layer0_en    1'b0
/tb/net/con_mod/layer1_en    1'b0
Now                          16980 ns
Cursor 1                     0 ns
```

Waveform with SoftMax Detecting Digit 1 Correctly

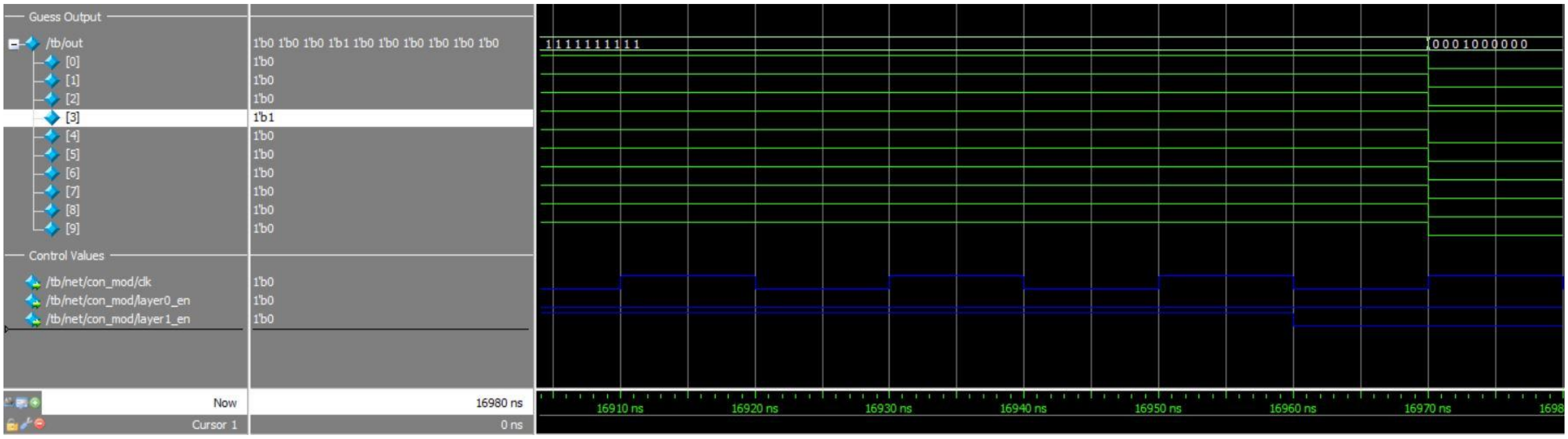
Results From Loading Digit 2



```
Guess Output
/tb/out                    1'b0 1'b0 1'b1 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0
  [0]                      1'b0
  [1]                      1'b0
  [2]                      1'b1
  [3]                      1'b0
  [4]                      1'b0
  [5]                      1'b0
  [6]                      1'b0
  [7]                      1'b0
  [8]                      1'b0
  [9]                      1'b0
Control Values
/tb/net/con_mod/clk        1'b1
/tb/net/con_mod/layer0_en  1'b0
/tb/net/con_mod/layer1_en  1'b0
Now                        16980 ns
Cursor 1                   16974 ns
```

Waveform with SoftMax Detecting Digit 2 Correctly

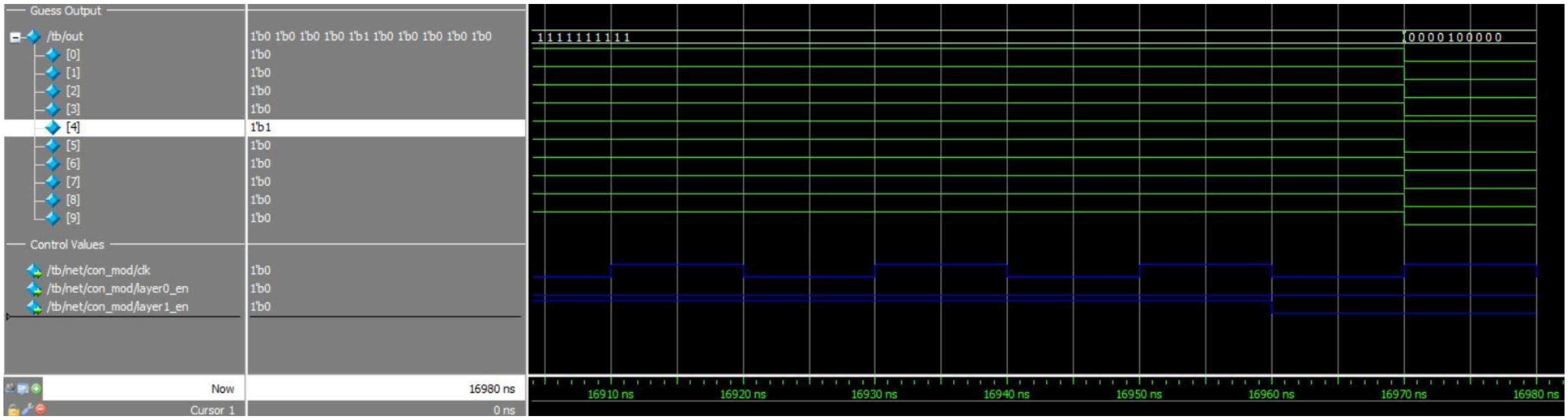
Results From Loading Digit 3



```
Guess Output
/tb/out                    1'b0 1'b0 1'b0 1'b1 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0
  [0]                      1'b0
  [1]                      1'b0
  [2]                      1'b0
  [3]                      1'b1
  [4]                      1'b0
  [5]                      1'b0
  [6]                      1'b0
  [7]                      1'b0
  [8]                      1'b0
  [9]                      1'b0
Control Values
/tb/net/con_mod/clk        1'b0
/tb/net/con_mod/layer0_en  1'b0
/tb/net/con_mod/layer1_en  1'b0
Now                        16980 ns
Cursor 1                   0 ns
```

Waveform with SoftMax Detecting Digit 3 Correctly

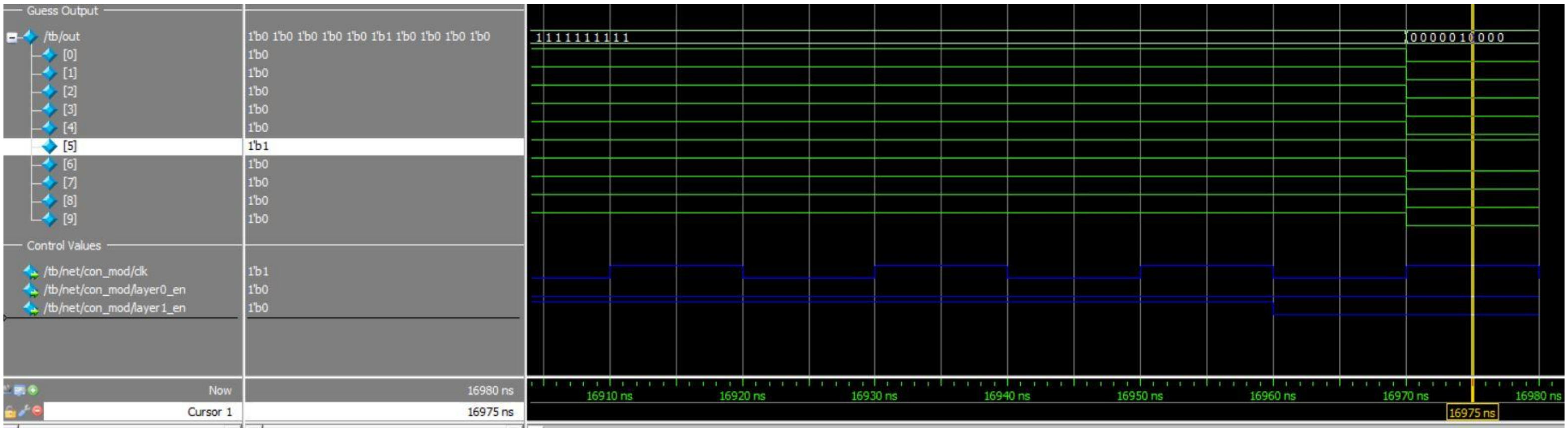
Results From Loading Digit 4



```
Guess Output
/tb/out                    1'b0 1'b0 1'b0 1'b0 1'b1 1'b0 1'b0 1'b0 1'b0 1'b0
  [0]                      1'b0
  [1]                      1'b0
  [2]                      1'b0
  [3]                      1'b0
  [4]                      1'b1
  [5]                      1'b0
  [6]                      1'b0
  [7]                      1'b0
  [8]                      1'b0
  [9]                      1'b0
Control Values
/tb/net/con_mod/clk        1'b0
/tb/net/con_mod/layer0_en  1'b0
/tb/net/con_mod/layer1_en  1'b0
Now                        16980 ns
Cursor 1                   0 ns
```

Waveform with SoftMax Detecting Digit 4 Correctly

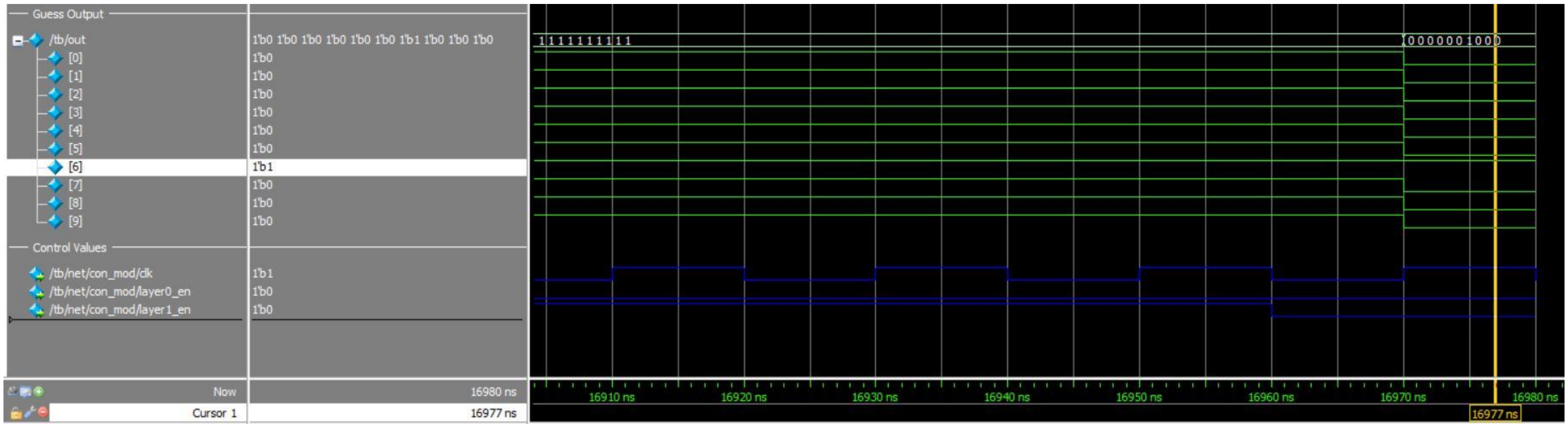
Results From Loading Digit 5



```
Guess Output
/tb/out                      1'b0 1'b0 1'b0 1'b0 1'b0 1'b1 1'b0 1'b0 1'b0 1'b0
  [0]                        1'b0
  [1]                        1'b0
  [2]                        1'b0
  [3]                        1'b0
  [4]                        1'b0
  [5]                        1'b1
  [6]                        1'b0
  [7]                        1'b0
  [8]                        1'b0
  [9]                        1'b0
Control Values
/tb/net/con_mod/clk          1'b1
/tb/net/con_mod/layer0_en    1'b0
/tb/net/con_mod/layer1_en    1'b0
Now                          16980 ns
Cursor 1                     16975 ns
```

Waveform with SoftMax Detecting Digit 5 Correctly

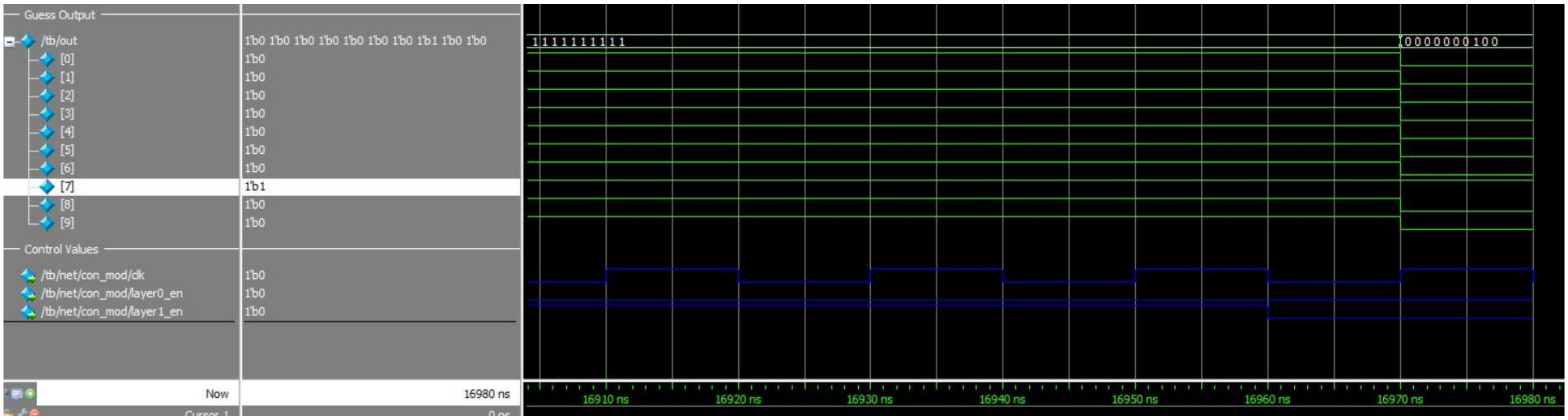
Results From Loading Digit 6



```
Guess Output
/tb/out                      1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b1 1'b0 1'b0 1'b0
  [0]                        1'b0
  [1]                        1'b0
  [2]                        1'b0
  [3]                        1'b0
  [4]                        1'b0
  [5]                        1'b0
  [6]                        1'b1
  [7]                        1'b0
  [8]                        1'b0
  [9]                        1'b0
Control Values
/tb/net/con_mod/clk          1'b1
/tb/net/con_mod/layer0_en    1'b0
/tb/net/con_mod/layer1_en    1'b0
Now                          16980 ns
Cursor 1                     16977 ns
```

Waveform with SoftMax Detecting Digit 6 Correctly

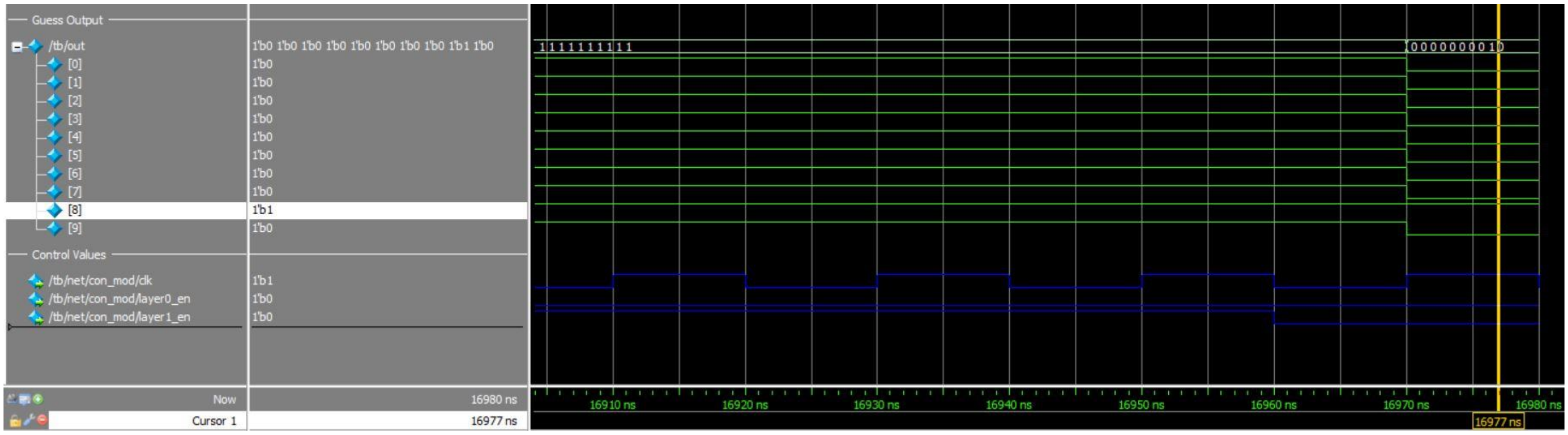
Results From Loading Digit 7



```
Guess Output
/tb/out                      1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b1 1'b0 1'b0
  [0]                        1'b0
  [1]                        1'b0
  [2]                        1'b0
  [3]                        1'b0
  [4]                        1'b0
  [5]                        1'b0
  [6]                        1'b0
  [7]                        1'b1
  [8]                        1'b0
  [9]                        1'b0
Control Values
/tb/net/con_mod/clk          1'b0
/tb/net/con_mod/layer0_en    1'b0
/tb/net/con_mod/layer1_en    1'b0
Now                          16980 ns
Cursor 1                     0 ps
```

Waveform with SoftMax Detecting Digit 7 Correctly

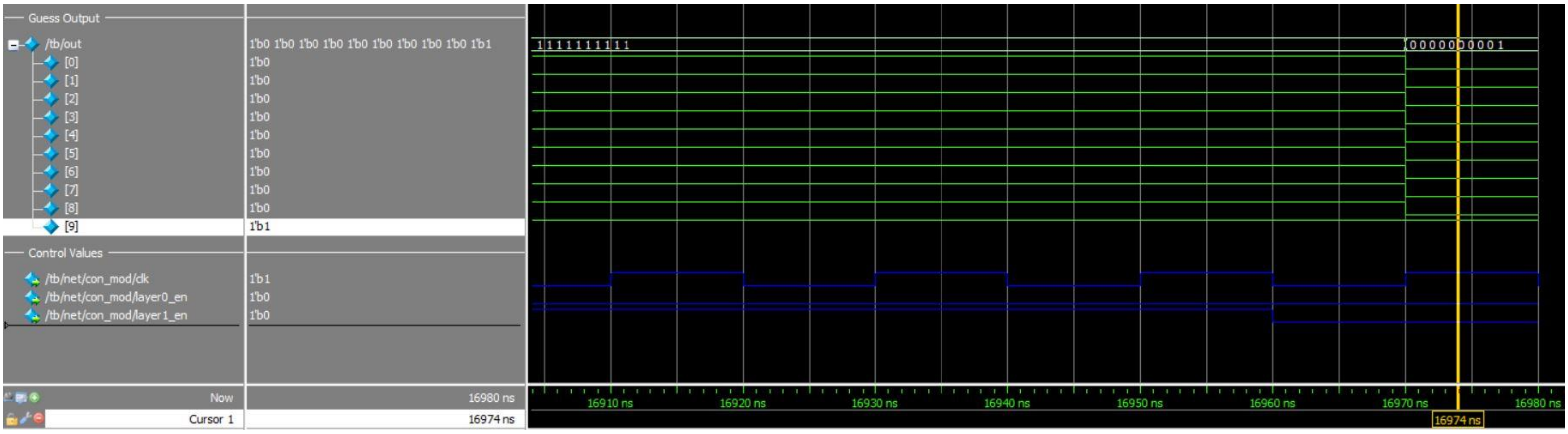
Results From Loading Digit 8



```
Guess Output
/tb/out                     1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b1 1'b0
  [0]                       1'b0
  [1]                       1'b0
  [2]                       1'b0
  [3]                       1'b0
  [4]                       1'b0
  [5]                       1'b0
  [6]                       1'b0
  [7]                       1'b0
  [8]                       1'b1
  [9]                       1'b0
Control Values
/tb/net/con_mod/clk         1'b1
/tb/net/con_mod/layer0_en   1'b0
/tb/net/con_mod/layer1_en   1'b0
Now                         16980 ns
Cursor 1                    16977 ns
```

Waveform with SoftMax Detecting Digit 8 Correctly

Results From Loading Digit 9



```
Guess Output
/tb/out                     1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b0 1'b1
  [0]                       1'b0
  [1]                       1'b0
  [2]                       1'b0
  [3]                       1'b0
  [4]                       1'b0
  [5]                       1'b0
  [6]                       1'b0
  [7]                       1'b0
  [8]                       1'b0
  [9]                       1'b1
Control Values
/tb/net/con_mod/clk         1'b1
/tb/net/con_mod/layer0_en   1'b0
/tb/net/con_mod/layer1_en   1'b0
Now                         16980 ns
Cursor 1                    16974 ns
```

Waveform with SoftMax Detecting Digit 9 Correctly