Linear Spatial Filtering Implemented in RTL for Binary Convolutional Neural Network

Name: **Arpad Attila Voros**
Unity ID: **aavoros**
Student ID: **200215778**

Delay: **1725 ns**
Clock period: **3 ns**
cycles: **575**

Logic Area: (um^2)

2398.522

Memory: **N/A**

1/(delay.area) ($ns^{-1}um^{-2}$)

2.41695e-7

Delay (TA provided example. TA to complete)

1/(delay.area) (TA)

Abstract

For the past couple of decades, applications for convolutional neural networks (CNN) have increased in popularity due to increased performance of computing and research in deep learning. CNNs are nowadays considered a foundational type of layer in deep learning architecture for image processing, image generation, object detection, and even speech detection & synthesis. However, computation time is still a big issue when training a network that uses CNNs, where some training can take hours or even days. Binary CNNs have replaced the weights of a CNN from a decimal point value to a unit vector in both positive and negative directions, which surprisingly does not take a lot of information away from the network and severely decreases training time and evaluation. To further speed up CNNs, the concept of Binary CNNs be extended from the software to a register transfer language (RTL) to be synthesized into hardware. The example being synthesized has the smallest square kernel size of three, commonly used in linear spatial filtering, with operators such as Sobel, Roberts, Prewitt, etc. The inputs to the network, rather than being images are simply small binary arrays. This report serves as a rudimentary example of how machine learning hardware comes to fruition.

Linear Spatial Filtering Implemented in RTL for Binary Convolutional Neural Network

Arpad Attila Voros

Abstract

For the past couple of decades, applications for convolutional neural networks (CNN) have increased in popularity due to increased performance of computing and research in deep learning. CNNs are nowadays considered a foundational type of layer in deep learning architecture for image processing, image generation, object detection, and even speech detection & synthesis. However, computation time is still a big issue when training a network that uses CNNs, where some training can take hours or even days. Binary CNNs have replaced the weights of a CNN from a decimal point value to a unit vector in both positive and negative directions, which surprisingly does not take a lot of information away from the network and severely decreases training time and evaluation. To further speed up CNNs, the concept of Binary CNNs be extended from the software to a register transfer language (RTL) to be synthesized into hardware. The example being synthesized has the smallest square kernel size of three, commonly used in linear spatial filtering, with operators such as Sobel, Roberts, Prewitt, etc. The inputs to the network, rather than being images are simply small binary arrays. This report serves as a rudimentary example of how machine learning hardware comes to fruition.

1. Introduction

Ideally, if the synthesized result of this module were to be produced, it could potentially serve an actual purpose other than for demonstration purposes. Even if the input sizes are limited to word sizes, the module can run in parallel for segments of a binary image to improve processing time. However, there are still severe limitations to this design and was mainly realised and designed for educational purposes.

The design in this report was able to achieve a synthesized area of ~2400um^2 with a minimum clock period of 3 ns. Verilog was used as the RTL and Synopsys was used to synthesize the module.

2. Micro-Architecture

The size limitations of the inputs are 16 bits in each dimension, while the size limitations of the kernel is rigidly restricted to a 3 bit by 3 bit array. Because of this, the non-flexible kernel size can be exploited when designing the hardware. It was decided that the entirety of the kernel can be represented in the hardware, and the input data can be pipelined through to represent the kernel “sweeping” across the image.

To do this, 9 ‘convolution modules’ are produced which simply load in the weights, pipeline the incoming data to the outgoing data, all the while calculating the “product” of the bits. Since we represent a -1 with a binary 0, and a 1 with a binary 1, we can XNOR the two bits to find its binary equivalent “product”. However, in this design XORs are used to calculate the inverse, which will be explained in a second.

To compute the convolution, the result of the element-wise matrix multiplication must be summed together. If all the weights and inputs are binary, the highest possible result would be a sum of all positive bits (9) and the lowest possible result would be a sum of all negative bits (-9). There will never be a sum of 0, since the kernel size is odd. The result simply has to be signed to store the output into a single bit. Because of this, we can simply calculate whether the there are greater number of 0's or 1's in the 9 output bits. XORs were used instead of XNORs because calculating the number of 0's or 1's is commutatively equivalent. As far as I am aware, XNOR gates on a transistor level are simply XORs with a negation at the end, so adding 9 inverters would be superfluous when only the signed output is needed.

The design of the hardware can be seen in the drawing in Figure 1 below. It was also realised that an efficient way to sum the outputs would be to use 5 full-adders. Each of the 9 output bits can first be fed into the three inputs (two sum bits and one carry-in bit) of three full-adders. The respective outputs would be the output bit and a carry-out bit, which realistically represents the values 1 and 2 when the output wires are high. Those 6 outputs are then pipelined into another set of two full-adders, where the "1" bits go to one and the "2" bits go to another. The former will have the same outputs as the adders from the previous stage, but the latter will represent the values of 2 and 4 when the output wires are high. Using these four wires of {1, 2, 2, 4}, if all the wires are high then we get a sum of 9 (as expected), and if all of them are low then we get -9. To determine the sign of the output, we simply have to use some combinational logic to see if the sum is 4 or more. In Verilog, this would be written as:

assign SIGN = (four & (one | two1 | two2)) | (one & two1 & two2);

Where **four** represents the four wire, **two1** and **two2** represent the 2 two wires, and **one** represents the one wire. This simply means that if the four wire is high, we need at least one other wire to be high to get a sum greater than 4 (since 4 + anything > 4). Otherwise, if all of the one and two wires are high, we get a sum greater than 4 (since 1 + 2 + 2 > 4). This is represented in drawing using a 2 bit MUX selector on the right of Figure 1. Since the original outputs calculated the negative flag, the summed result is negated before stored to the output SRAM.

All of the convolution modules control the pipelining of the data by a convolutional "go" flag, which, when high, acts as the kernel sweeps the data. Since the kernel size is limited to 3 bits by 3 bits, the first 3 lines of input can be pre-loaded before performing convolution. The first index of each input is loaded into a 3 bit register to the input of the convolution module array, ready to be pipelined down. As the convolution "go" flag goes high, the data enters the convolution modules, and the next index of data is read in. Since the kernel size is restricted to 3, it is simply timed to know when to start calculating the sum (when the first index of each row of data has been pipelined down to the end of the convolution modules). It is only then when the pipelined output is read and considered for storage.

To store the output, we can simply take the convolution module which holds the starting index in both dimensions and get the data index at that module for all convolutions. In this case, that module would be the upper-right module seen in Figure 1. To get the index of the data, it would make the most sense to simply pipeline the data index alongside the data, so that's what it does.

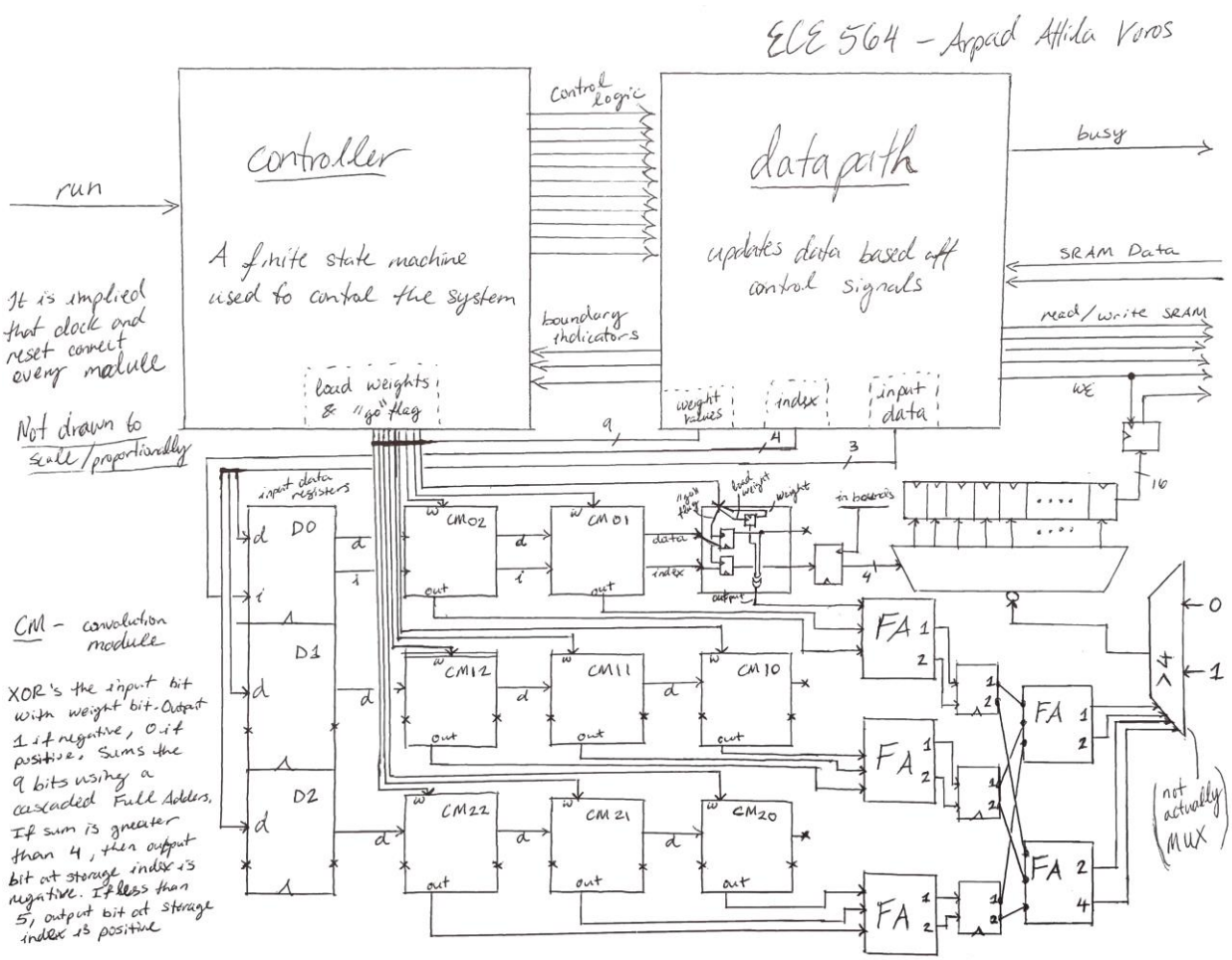


Figure 1: Hand drawn schematic of the module

When the index is received on the end of the output, it is checked to see if it is a proper storage index that does not exceed boundary conditions. If so, the signed result is stored to an output data register given the provided index.

A controller and datapath module are used in conjunction of the rest of the design. As the name suggests, the controller controls the design via control flags while the datapath responds to the control logic and manipulates data and other registers.

One of the jobs of the datapath register is when the input data is read in, it will determine the boundary conditions for when the controller should move to the next column, move to the next row, move to the next input, or completely stop. The datapath module is also what interfaces with top (without memory), and as a result memory (top with memory) in order to read and write data to/from simulated SRAMs.

The controller is a synchronous finite state machine (FSM) with 12 states. In short, the thought process behind the function of the FSM is the following (this is not reflective of the exact function and number of states used in the FSM, it is merely used to put the function of the FSM into words):

1. Wait for top to tell you when to run
2. Start reading in weight and input dimensions
3. Start reading in weigh values and first row of input. Calculate boundary conditions given input dimensions
4. Load in weights to convolution modules. Read in second row of input
5. Read in third row of input and start pipelining data through convolution modules
6. Once the first input data has reached the final stage of the convolution modules, calculate the sum. Store in the output register given the output index
7. Continue pipelining data until a column-wise boundary condition is reached. This means, last output has to be stored and new row has to be read in.
8. Store last output, write to SRAM output
9. Repeat steps 5-8 until row-wise boundary condition is reached
10. This means, we are ready for new inputs all together. Start reading from step 2 HOWEVER if instead of input dimensions are read in, the module reads in an EOF condition (in this case, the EOF was represented by 0x00FF), then go back to step 1 because the module has run its course

3. Module Specifications

Register/Wire Name	Width	Description
dut_run	**1**	Run flag – set high externally to run
dut_busy	**1**	Busy flag – is set high when module is running
reset_b	**1**	Reset flag
clk	**1**	Clock
dut_sram_write_address	**12**	Write address for outgoing data
dut_sram_write_data	**16**	Outgoing data to be written
dut_sram_write_enable	**1**	Write enable flag for storing output
dut_sram_read_address	**12**	Outgoing input read address
sram_dut_read_data	**16**	Incoming input data
dut_wmem_read_address	**12**	Outgoing weight read address
wmem_dut_read_data	**16**	Incoming weight data
weights_data	**16**	Holds weights information. Used to load convolution modules
d_in	**3**	Three data inputs. Input to convolution modules, pipelined down
coli_in	**4**	Storage index of output, pipelined
s1_ones	**3**	Full-adder stage 1 “ones” output
s1_twos	**3**	Full-adder stage 1 “twos” output
s2_ones	**3**	Full-adder stage 2 “ones” input
s2_twos	**3**	Full-adder stage 2 “twos” input
initialization_flag	**1**	High when new input is read in. Used to indicate when to store result
last_col_next	**1**	Flag to indicate next clock cycle is the last column. Prepare next row
last_row_flag	**1**	Flag to indicate last row. Prepare for next input
dut_busy_toggle	**1**	Set high to toggle **dut_busy**
set_initialization_flag	**1**	Set high to set **initialization_flag**
rst_initialization_flag	**1**	Set high to reset **initialization_flag**
incr_col_enable	**1**	Set high to increment column index counter
incr_row_enable	**1**	Set high to increment row index counter
rst_col_counter	**1**	Set high to reset column index counter
rst_row_counter	**1**	Set high to reset row index counter
incr_raddr_enable	**1**	Set high to increment input read address
rst_dut_sram_write_address	**1**	Set high to reset input write address
rst_dut_sram_read_address	**1**	Set high to reset input read address

rst_dut_wmem_read_address	**1**	Set high to set weight read address to weight data (0x0001)
str_weights_dims	**1**	Set high to store **wmem_dut_read_data** (weight dimensions)
str_weights_data	**1**	Set high to store **wmem_dut_read_data** (weight data)
str_input_nrows	**1**	Set high to store **sram_dut_read_data** (dim 1, number of rows)
str_input_ncols	**1**	Set high to store **sram_dut_read_data** (dim 2, number of cols)
pln_input_row_enable	**1**	Set high to read in new input row, and pipeline inputs 'upward'
str_temp_to_write	**1**	Set high to write output register into **dut_wmem_write_data**
update_d_in	**1**	Set high to pipeline new data into convolution modules
load_weights_to_modules	**1**	Set high to load weights into convolution modules
toggle_conv_go_flag	**1**	Set high to toggle convolution module "go" flag
rst_output_row_temp	**1**	Set high to reset output register
negative_flag	**1**	Negative flag. 1 if sum is negative, 0 if positive
conv_go_flag	**1**	Set high to pipeline data through convolution modules
end_condition_met	**1**	High if end condition (EOF) is reached

The green cells in the table above represent the interface of the top module. The rest are descriptions of the registers and wires used in the top module of the system to get an idea of how the system functions. The following figures are timing diagrams from the Verilog simulation to demonstrate functionality.

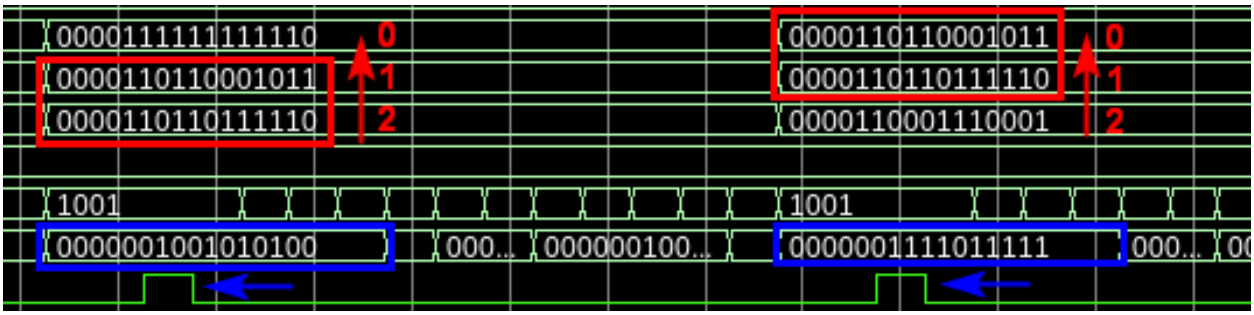


*Figure 2: Red shows the input addresses being pipelined during a ROW transition. Blue shows the output register result as well as the **str_temp_to_write** flag*

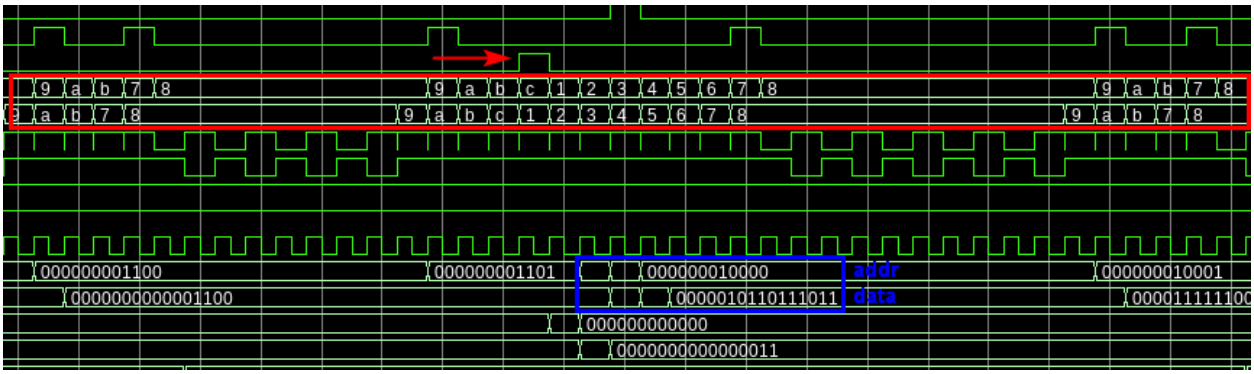


*Figure 3: Red shows next and current state as well as **rst_output_row_temp** changing during an INPUT transition. Blue shows the input SRAM reading the new input dimensions one after another then the input data, starting convolution immediately*

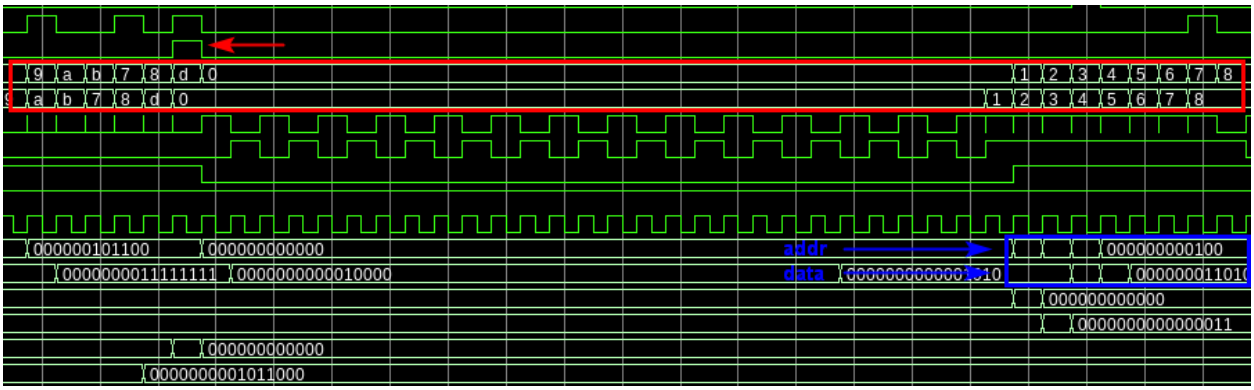


*Figure 4: Red shows next and current state as well as **rst_output_row_temp** changing during a FILE transition. Blue shows the input SRAM address being reset and starting to read again when new inputs are read in*

4. Verification

Using the testbench provided and the verified examples, the simulation was able to be verified. Other input, weight, and expected output files were tested to ensure compatibility with **any** number and combination of rows and columns (greater than or equal to kernel dimensions, less than or equal to word size) worked as well. ***This means the design works past the expected 10x10, 12x12, and 16x16 requirements for any input rectangle of any size less than or equal to 16 or greater than or equal to 3.*** This minor amount of flexibility causes the design to be slower than if the result was hard-coded for the required square sizes, where everything is done in parallel instead of reading and pipelining data.

In addition, running multiple trials consecutively helped verify how to properly reset the DUT.

5. Results Achieved

Statistics:

- Number of compute cycles achieved:
  - 575
  - See Figure 5
- Minimum clock period achieved:
  - 3 ns
  - See Figure 7
- Minimum area achieved:
  - 2398.522 um^2
  - See Figure 8
- Slack requirements met:
  - See Figures 6, 9, and 10

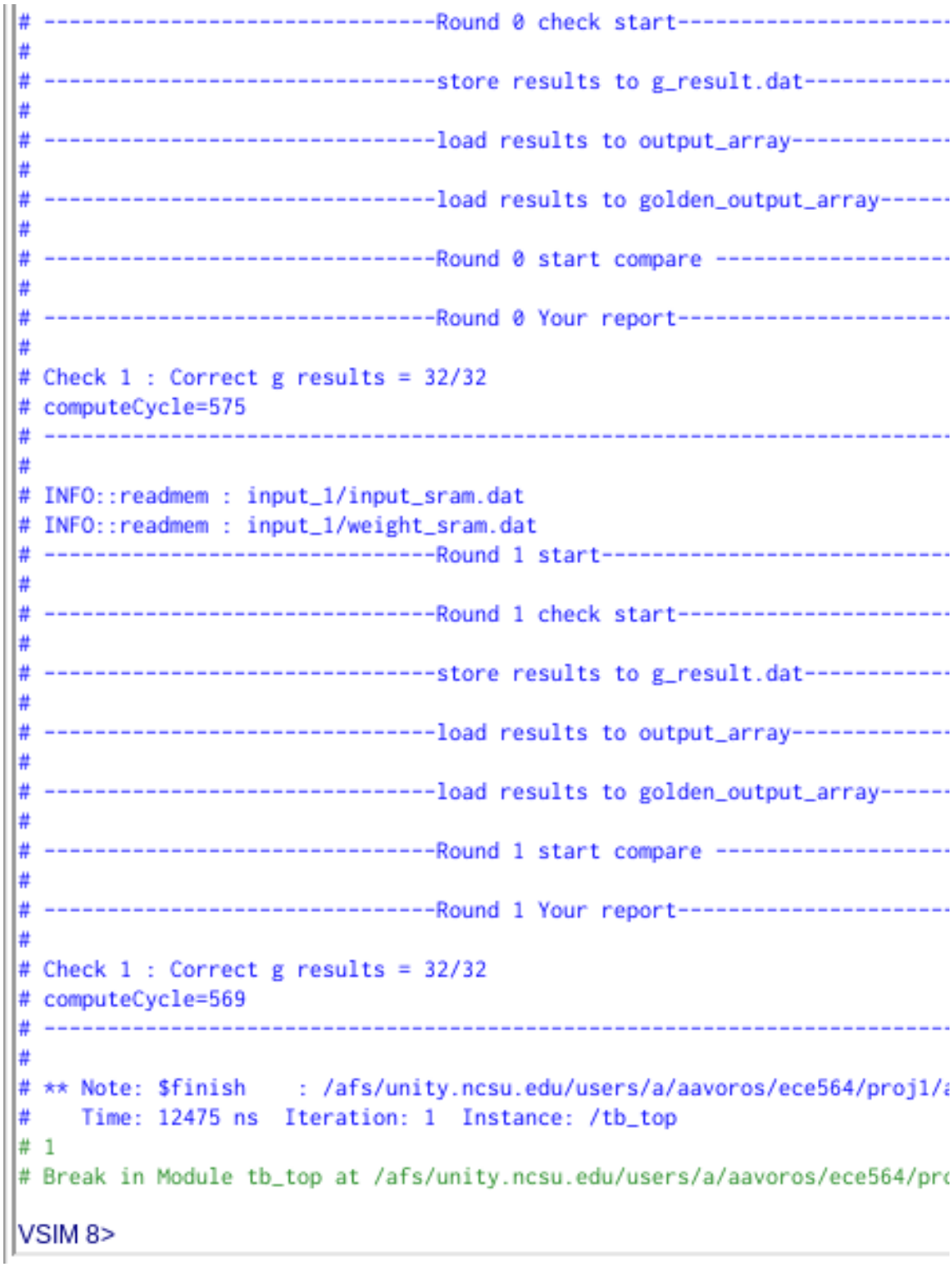
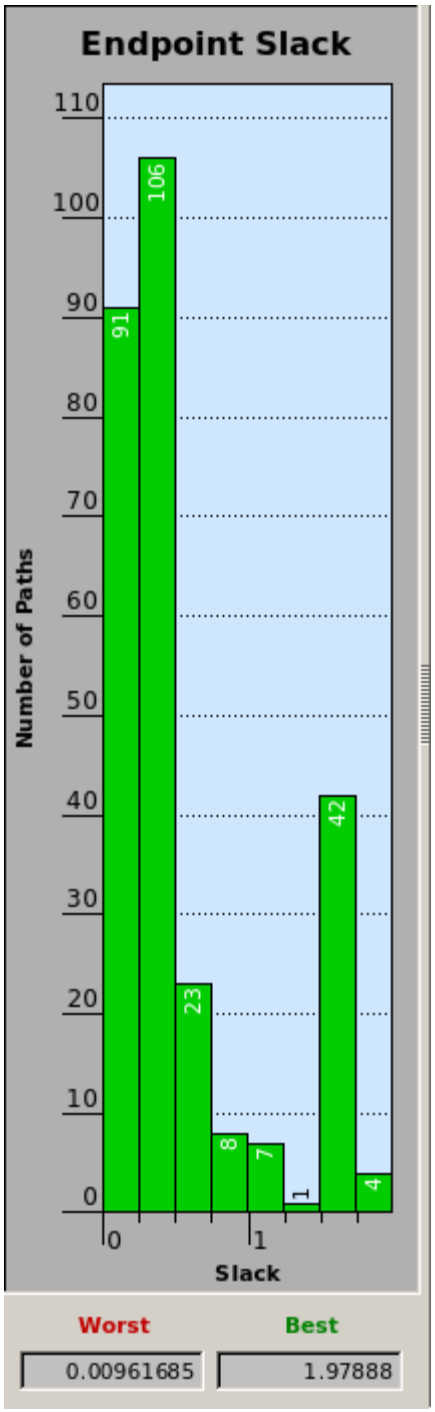
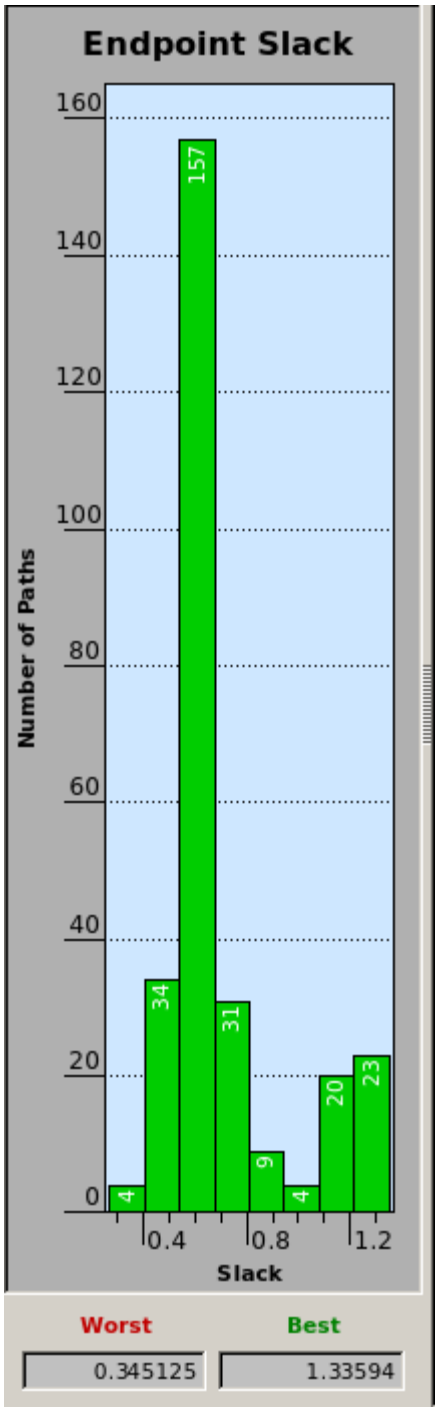


Figure 5: Verification of simulation using provided testbench



Max path slack histogram – fast and slow



Min path slack histogram – fast and slow

Figure 6: Slack histograms

```
# setup name of the clock in your design.
set clkname clk

# set variable "modname" to the name of topmost m
set modname MyDesign

# set variable "RTL_DIR" to the HDL directory w.r
set RTL_DIR    ../v

# set variable "type" to a name that distinguishe
set type projl

#set the number of digits to be used for delay re
set report_default_significant_digits 4

set CLK_PER 3
#-----------------------------------------------
```

Figure 7: Clock period of 3ns achieved

```
m21/weight_reg          DFF_X2          NangateOpenCellLibrary_PDK
                                                          4.7880
m22/data_out_reg        DFF_X1          NangateOpenCellLibrary_PDK
                                                          4.7880
m22/weight_reg          DFF_X2          NangateOpenCellLibrary_PDK
                                                          4.7880
--------------------------------------------------------------------
Total 1245 cells                                       2398.5220
1
```

Figure 8: ~2400um area achieved

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : MyDesign
Version: P-2019.03-SP1
Date   : Wed Nov 17 13:24:43 2021
****************************************

Operating Conditions: slow   Library: NangateOpenCellLibrary_PDKv1_2_v2008_10_
Wire Load Model Mode: top

  Startpoint: ctrl/current_state_reg[1]
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint: dp/input_r0_reg[0]
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Point                                                   Incr       Path
  -----------------------------------------------------------------------
  clock clk (rise edge)                                 0.0000     0.0000
  clock network delay (ideal)                           0.0000     0.0000
  ctrl/current_state_reg[1]/CK (DFFR_X1)                0.0000     0.0000 r
  ctrl/current_state_reg[1]/Q (DFFR_X1)                 0.6968     0.6968 f
  U792/ZN (NOR2_X1)                                     0.2285     0.9253 r
  U1341/ZN (NAND2_X1)                                   0.1089     1.0341 f
  U615/ZN (INV_X1)                                      0.0928     1.1269 r
  U1345/ZN (NAND2_X1)                                   0.0571     1.1841 f
  U625/ZN (AND2_X2)                                     0.2770     1.4611 f
  U662/ZN (INV_X16)                                     0.2558     1.7169 r
  U1390/Z (MUX2_X1)                                     0.7846     2.5014 f
  dp/input_r0_reg[0]/D (DFFR_X1)                        0.0000     2.5014 f
  data arrival time                                                2.5014

  clock clk (rise edge)                                 3.0000     3.0000
  clock network delay (ideal)                           0.0000     3.0000
  clock uncertainty                                    -0.0500     2.9500
  dp/input_r0_reg[0]/CK (DFFR_X1)                       0.0000     2.9500 r
  library setup time                                   -0.4426     2.5074
  data required time                                               2.5074
  -----------------------------------------------------------------------
  data required time                                               2.5074
  data arrival time                                               -2.5014
  -----------------------------------------------------------------------
  slack (MET)                                                      0.0059
```

Figure 9: Timing on max path on slowest condition. Slack met

```
Information: Updating design information... (UID-85)

****************************************
Report : timing
        -path full
        -delay min
        -max_paths 1
Design : MyDesign
Version: P-2019.03-SP1
Date   : Wed Nov 17 13:24:42 2021
****************************************

Operating Conditions: fast   Library: NangateOpenCellLibrary_PDK
Wire Load Model Mode: top

  Startpoint: m00/idx_out_reg[0]
              (rising edge-triggered flip-flop clocked by clk)
  Endpoint: dp/writ_idx_reg[0]
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: min

  Point                                    Incr       Path
  -----------------------------------------------------------
  clock clk (rise edge)                  0.0000     0.0000
  clock network delay (ideal)            0.0000     0.0000
  m00/idx_out_reg[0]/CK (DFF_X2)         0.0000     0.0000 r
  m00/idx_out_reg[0]/Q (DFF_X2)          0.0533     0.0533 r
  dp/writ_idx_reg[0]/D (DFFR_X1)         0.0000     0.0533 r
  data arrival time                                 0.0533

  clock clk (rise edge)                  0.0000     0.0000
  clock network delay (ideal)            0.0000     0.0000
  clock uncertainty                      0.0500     0.0500
  dp/writ_idx_reg[0]/CK (DFFR_X1)        0.0000     0.0500 r
  library hold time                     -0.0086     0.0414
  data required time                                0.0414
  -----------------------------------------------------------
  data required time                                0.0414
  data arrival time                                -0.0533
  -----------------------------------------------------------
  slack (MET)                                       0.0119
```

Figure 10: Timing on min path on fastest condition. Slack met

6. Conclusions

In conclusion, in can be seen that hardware for machine learning is extremely viable and easy to design. It can also be shown that creating a more flexible design with variable input support can slow down the design, whereas creating a rigid/limited design can significantly speed things up, which is the trade-off. The application will determine which type of design is preferred. As a personal statement, I would say the idea behind the project is incredibly simple and so is writing it in Verilog RTL. However, being completely unfamiliar with synthesis and Synopsys, it took a significant amount of time to properly write the module after redesign after redesign (where each design perfectly simulated in Verilog but had difficulty synthesizing). The items that helped the most in proper synthesis were completely separating the controller and datapath, as well as using one of the “CPU” notes as a reference to gauge how the datapath and controller module were formatted. Once I based my design off of this format (though the logic and the idea behind the design did not change), synthesis suddenly started working. It was definitely an interesting experience.