ECE 385 Final Project

Neural Network for Handwritten Digit Recognition

Saaya Nath, Min Jin Chong

Introduction

For our final project, we chose to implement a neural network for handwritten digit recognition. The user can take a photo of a written-down number using a digital camera module connected to the DE2 board, and then our system will process and classify the value, outputting the number on the hex displays.

In our project, we implemented both the training (back propagation) and testing (forward propagation) portions of the neural network. The implementation and functionality will be further described below.

Instructions:

Testing

1) Ensure SW 17 is LOW
2) Press KEY 0 to reset the FPGA
3) Press KEY 3 to start the camera and display is on the monitor
4) SW [12:0] controls the contrast. Pressing KEY 1 updates the contrast
5) KEY 3 also toggles between RGB/Greyscale/B/W mode on screen
6) Pressing KEY 2 freezes the screen
7) Prediction is displayed on the hex display

Training

1. Set SW [16:13] to the correct label in binary
2. Set SW 17 to HIGH for as long as you want to train

Forward Propagation

Weights

Our network consist of only two layers, an input layer of size 784, and an output layer of 10, which represents the probability of the ten digits. We first train the network using TensorFlow with an input of size [1,784] and a weight matrix of [784, 10]. Since we are only using two layers, no sigmoid function is used. The prediction is done by choosing the digit with the largest output. The dataset we trained the weights on are MNIST digits which are flattened to a dimension of [1,784]. We also converted all the pixels from 8 bits to 1 bit using a threshold. This essentially converts it to a black and white image.

To force the weight values to have similar magnitudes, we added an L2 regularizer. The weights we get are in the order of 1×10^{-3} and since we decide to do integer calculations on our FPGA, we multiply the weights by 1000 to make them integers. This will increase the output by magnitude of 1000 but will not impact the prediction. The accuracy of the network is around 92%.

We then save the weights to our FPGA's on-chip memory.

Camera

The DE camera we used have outdated software that is compatible with DE1 instead of DE2. To get it working, we have to remap all the pin assignments and debug certain software bugs. For example, the VGA signals are 10 bits instead of 8 bits and the SRAM address is 18 bits instead of 20 bits. Lots of trial and error, documentation reading are done in order to have it working as we intend to.

Image Processing

I first implemented bit-wise RGB to Greyscale and Greyscale to black/white module. This is important because our training set are black/white images. Since our weights accept inputs of size [1,784] we have to downsample the image obtained from the camera [480, 640] to [28,28] so that we have an input of length 784. The image we get from the camera are fed to us from the SDRAM a pixel at a time with H_Cont and V_Cont keeping track of which pixel it is supposed to be. Thus we are not able to downsample and do matrix multiplication directly because we do not have all the data at once. We can only get a pixel in one by one.

Thus to do downsample the image to our desired dimension, we first have to keep track of which pixel is being fed to us and ignoring certain pixels. We first truncate the image from [480, 640] to [448, 448]. We use H_Cont and V_Cont to keep track of the pixel and by ignoring certain pixels, we can obtain a [448, 448] pixel. In order to debug this, I setup the VGA controller to color the [448,448] RGB and everything else B/W so it shows up on the monitor. From Figure 1, we can see that our truncation work well.

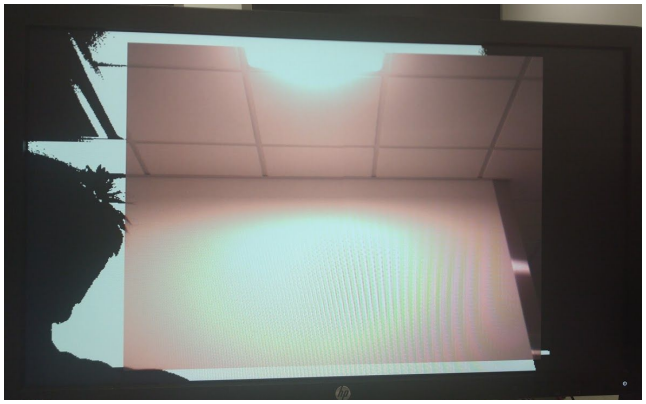


Figure 1

Once we have the [448,448] image, we now downsample it to [28,28] by selecting every 16 pixels horizontally and vertically. In other words, in every 16 x 16 block, we sample one pixel. Now that we have our [28,28] image, this serves as input. However, this image are still being fed to us pixel by pixel, we are only “obtaining” this [28,28] image by masking and unmasking the pixel stream given to us by the SDRAM.

In order to ensure that the [28,28] image we obtain are correct, we need a way to view this image. To do this, we chose to send the image to SRAM so we can export it to the computer. The skeleton code we started from have an SRAM interface that we decided to use. However, the SRAM address was 18 bits, which is compatible with the DE1 instead of the 20 bits for DE2. Thus we’ve spent hours figuring the bug out. Once we got the SRAM working, we are able to debug our code and ensure that the [28,28] image we as shown in Figure 2 obtained are expected.

Figure 2

Matrix Multiplication and Classification

After the image has been processed, we need to perform pixel-wise matrix multiplication in order to mimic the neural network and classify the input image as a number 0-9. The matmux10 module takes as input a pixel, start_stream signal, and an unpacked array of 10 shortints which represent the weights. The input weights have been selected as input corresponding to the input pixel, based on its location.

The multiplication occurring is between a 1x784 image vector, passed in pixel-by-pixel, and a 784x10 array of weights, accessing one row of 10 weights at a time. In order to perform this operation, we use the mult_accum IP which accumulates the sum of the multiplied values when our accum_start signal is high. In this module, we create a state machine includings states: WAIT, WAIT2, MULTIPLY, MULTIPLY_DELAY, DONE, and COMPARE.

The module waits for start stream to be high, which indicates that the input pixel is part of the down-sampled image and needs to be utilized in order to classify the value. We also have a counter which counts until 784, to keep track of the incoming pixels and indicating when the full 28x28 image has been processed. For this reason, we have two WAIT states. In WAIT, our counter is reset to 1, but not in WAIT2. Therefore, after each pixel multiplication, the state machine goes to the WAIT2 state. It does not re-enter the WAIT state until all 784 pixels have been multiplied by their corresponding weights and the classification is complete. The multiplication happens in the MULTIPLY state, in which we assign our pixel as datab[0] which is multiplied by dataa, each distinct weight in the 10-value row. The multiplied output is stored in an unpacked array of shortints, output_reg. After this, we go to the MULTIPLY_DELAY state. The mult_accum IP has a latency of N+2 where N is the number of multiplication operations. Therefore, after each multiply, our machine stays in the MULTIPLY_DELAY state for two clock cycles, while it counts a value ‘delay’. Once delay = 2, the machine allows movement to the DONE state. In this state, we check for start_stream. If it is still high, we stay in the DONE state. Once it goes low, if the counter is not yet 784, we go to the WAIT2 state until the start_stream becomes high again. If the counter has reached 784, we have processed all pixels of the down-sample image and can now go to COMPARE in order to perform the classification.

In the COMPARE state, the classification is set based on the output of our comparator. We use the lpm comparator IP in order to determine the correct value 0-9 of the input image, based on the 1x10 pixel*weights array. The comparator unit takes in the output_reg from the mult_accum IP and compares all 10 values with each other. We use the alb signal, which is high if dataa < datab. The comparator module, largest_reg, then determines the index of the largest value in the array. Thus, our output of the comparator unit is 9 minus this index, which gives us the actual number value 0-9 which has been determined to match the handwritten value in input image. This classification is then outputted from the matmux10 module, and the multiply module. In the COMPARE state, a variable output_test is also assigned the value of output_reg in order to save the output for the purpose of backpropagation during training. Below is an

image of what output_reg might look like like after the multiplication and accumulation, and how it would be classified through the comparator unit.

output shortint output_reg[9:0]

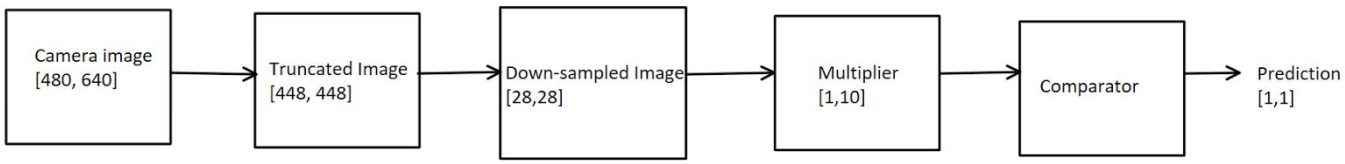
Class.	0	1	2	3	4	5	6	7	8	9
Score	-40	-20	8	12	-14	21	-24	-28	-10	16

In this case, the handwritten value in the input image would be classified as the number 5.

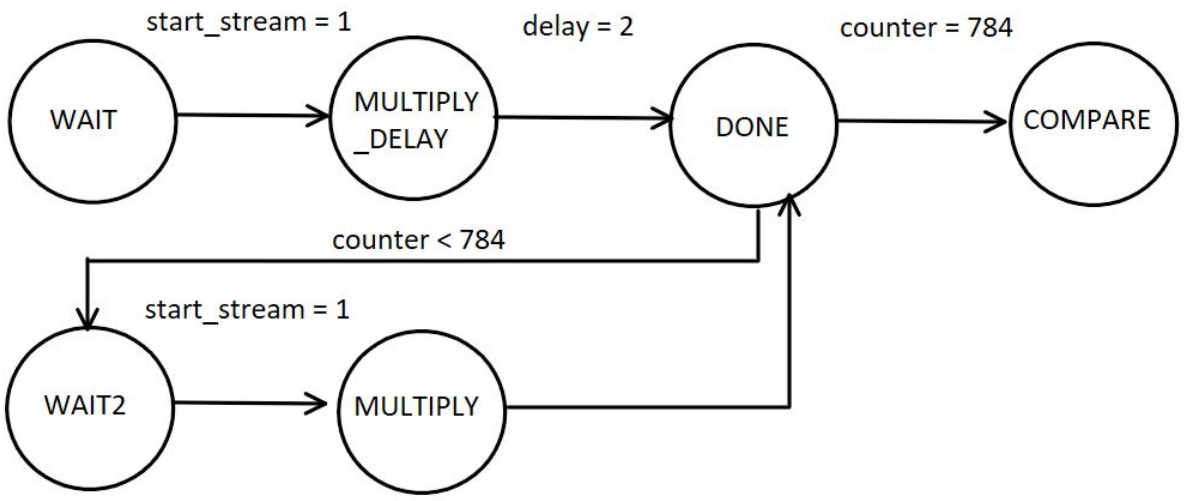
In our state machine, we also have control signals enable_clk and accum_start which control when the multiplier should be accumulating and when not. Because we don't want it to constantly be accumulating, we only enable the clock and set accum_start high during the MULTIPLY and MULTIPLY_DELAY states. In the WAIT state, we set control signal aclkr3 high which clears the accumulator for the next pixel.

Diagram of data transformation (Testing)

The blocks represent transformation of the data at each step and the dimension inside the block represent the dimension of the data at that point.



State Machine (Multiplication)



WAIT:
At this state, we are waiting for the start_stream to tell us when the correct pixel has arrived. Counter for pixel count and our multiplier is continuously cleared here. We are only in this state at before the first pixel arrive. Once start_stream arrives, go to MULTIPLY_DELAY state.

WAIT2:
Subsequent pixels after the first comes here after their respective multiplication instead of going to WAIT state so we can keep track of how many multiplications are done instead of continuously resetting our counter in WAIT. Similar to WAIT state, we wait for start_stream to tell signify the pixel is ready to multiply.

MULTIPLY:
At this state, we begin our multiplication by enabling the clock to our multiplier ip and feeding it with the pixel obtained from VGA controller and the weights from the on-chip memory. We also increment the pixel count so we know which pixel we are multiplying.

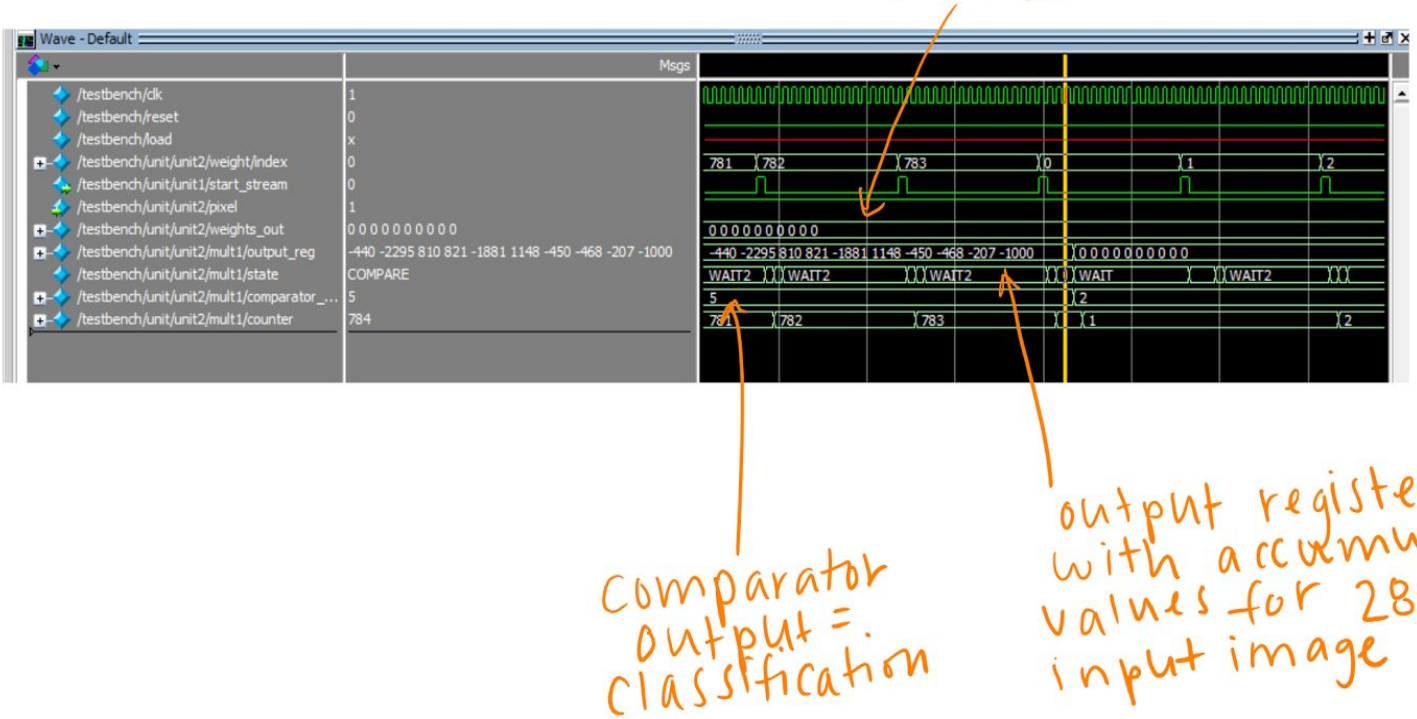
MULTIPLY_DELAY:
First multiplication takes 3 clock cycle by our multiplier ip and thus our first multiplication comes to this state where we give it 3 clock cycles to perform. After the first multiplication, each subsequent multiplication only requires 1 clock cycle and thus will be routed to MULTIPLY state which takes one clock cycle instead. Moves to done state after 3 clock cycles when delay = 2.

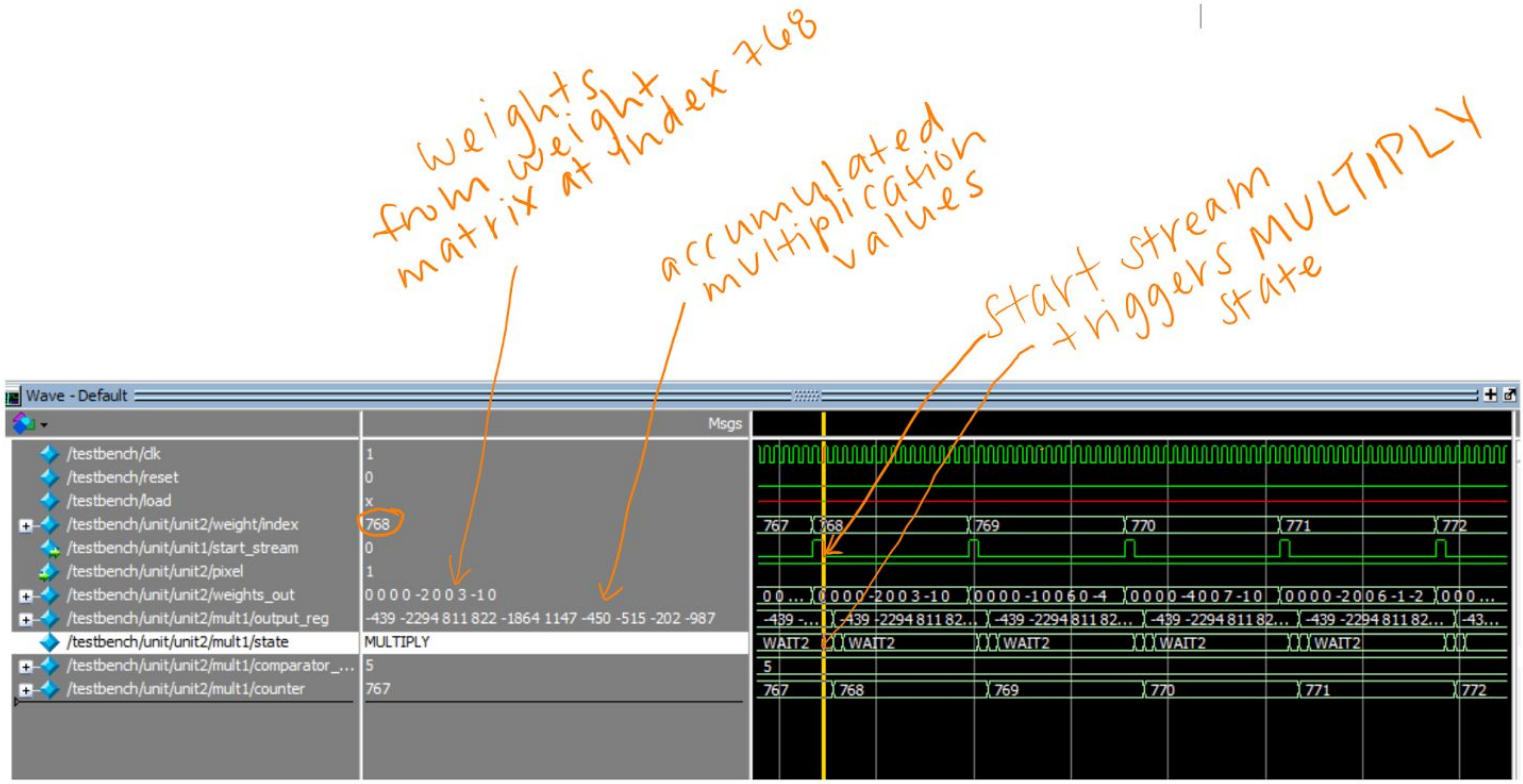
DONE:
Done with a single multiplication, goes to WAIT2 to prepare for next multiplication if we have not done 784 (28 x 28) multiplications or COMPARE state if we are done multiplying every pixel.

COMPARE:
Saves the comparator output into a register

Waveform (Multiplication)

Below are simulations of our matrix multiplication, showing the weights, pixel, and output of the mult_accum IP after the multiplication and accumulation. The simulations also show the comparator output, which is the determined classification.




weights from weight matrix at index 768
accumulated multiplication values
start stream triggers MULTIPLY state
Wave - Default
Msgs
/testbench/clk
/testbench/reset
/testbench/load
/testbench/unit/unit2/weight/index
/testbench/unit/unit1/start_stream
/testbench/unit/unit2/pixel
/testbench/unit/unit2/weights_out
/testbench/unit/unit2/mult1/output_reg
/testbench/unit/unit2/mult1/state
/testbench/unit/unit2/mult1/comparator_...
/testbench/unit/unit2/mult1/counter
1
0
x
768
0
1
0 0 0 0 -2 0 0 3 -1 0
-439 -2294 811 822 -1864 1147 -450 -515 -202 -987
MULTIPLY
5
767
767
768
769
770
771
772
WAIT2

Back Propagation

After we were able to successfully implement our forward propagation with the MNIST data set, we created a backpropagation unit. This training algorithm adjusts the original weights through two steps: 1. Feeding forward the values and 2. Calculating the error and propagating it back to our earlier layers.

The goal of backpropagation is to compute the partial derivative of the cost (pixel*weight) function with respect to the weights and bias in the network. This calculation can be described as the gradient of the error:

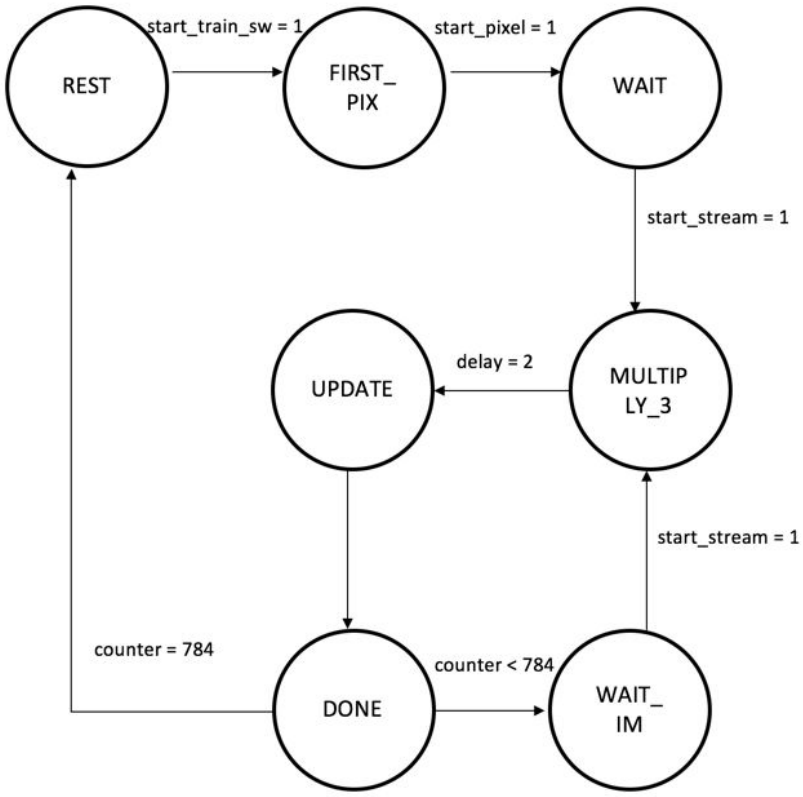
$$\nabla E = (x^T w - y)x^T$$

We calculate this error by using the original output matrix of our feedforward unit $(x^T w)$ as well as a 'correct' vector (y). This correct vector is determined by the DE2 on-board switches. We use 4 switches to indicate a binary value 0-9, which is then translated to a one-hot encoded vector, scaled by 1000, based on the input number. For instance, if the switches determine the number we are training with is 4, then the one-hot vector will be: {0, 0, 0, 1000, 0, 0, 0, 0, 0}. The difference between these two values is the first step in calculating our error.

After we compute this difference, we then use the mult_accum IP to perform matrix multiplication pixel-by-pixel of a 1x784 vector of an input image (x^T) and the calculated result described before. This multiplication results in a 784x10 matrix which is the same size as the original weights matrix.Therefore, ∇E, can also be thought of as ∇ w (weights), which represents the change in value from the original weights. Therefore, the new weights are overwritten by performing the computation $weights - \nabla w$, where 'weights' are the original weight values we had written to memory during our forward propagation.

This new weights matrix is used to train our network, making it more accurate proportional to the number of input images which are used (larger data set = more accuracy).

State Machine (Training)



REST
In this state, we wait for start_train_sw to be high to indicate to begin training. Once we receive this signal, we go to the state FIRST_PIX.

FIRST_PIX
In this state, we wait for the signal start_pix to indicate that the start of the picture. This ensures that we start training on the first pixel of the image and not halfway through. Once that signal is received, we go to the WAIT state to wait for start_stream.

WAIT
In this state, we are waiting for the start_stream to tell us when the correct pixel has arrived. The counter for pixel count is reset to 1 here and our delay as well as the write enable signal are both set to 0. Our multiplier is also cleared here. We are only in this state before the first pixel arrives. Once start_stream arrives, we go to MULTILPY_3 state.

WAIT_IM
Subsequent pixels after the first one come to this state after their respective multiplications instead of going to the WAIT state. This is to keep track of how many multiplications are done

instead of continuously resetting our counter in WAIT. Similar to WAIT state, we wait for start_stream to signify the pixel is ready to multiply. Once start_stream arrives, we go to MULTIPLY_3 state.

MULTIPLY_3

In this state, we begin our multiplication by enabling the clock to our multiplier IP and feeding it the pixel obtained from our VGA controller and the weights from OCM. We also increment the delay count here. The mult_accum IP has a latency of 3, so we wait 3 clock cycles. Once the delay = 2, we route to the UPDATE state.

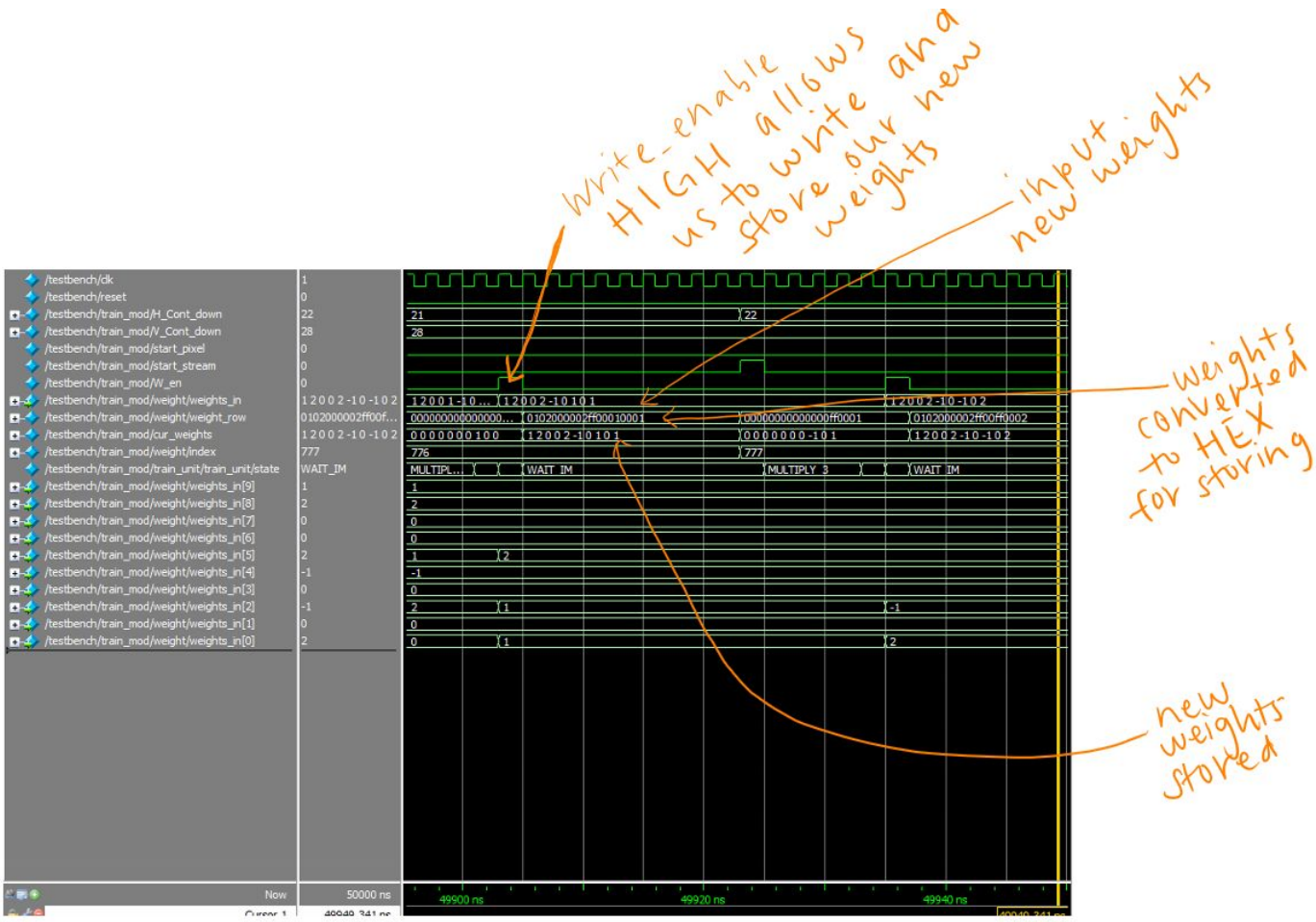
UPDATE

This is the first state where the multiplier outputs for each pixel row are available. Additionally, we turn the write enable signal (W_en) high here so that by the next clock cycle the signal is in sync with writing the new weights, which are also calculated in this state. For all 10 values in the array, the new weights are updated by subtracting the partial derivative of the error w.r.t to weights from our weights, with a learning rate of 1×10^{-3} .

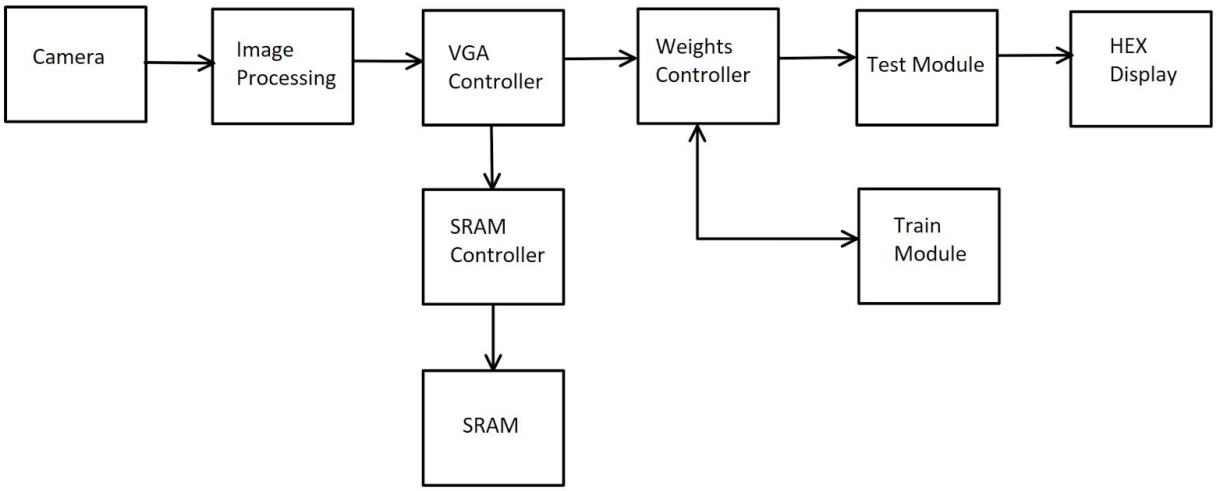
DONE

Done with a single multiplication, goes to WAIT_IM state to prepare for next multiplication if we have not done 784 (28 x 28) multiplications (counter < 784). If all multiplications have been performed, it goes to the REST state.

Waveform (Training)



Block Diagram



One thing to note is that image downsampling happens in the VGA Controller. Thus, some training and testing modules depend of VGA Controller for signals such as which pixel to multiply and its index.

Module Descriptions

Module: weights_controller
Inputs: [4:0] H_count, V_count; W_en; shortint weights_in[9:0]
Outputs: shortint weights_out[9:0]
Description: This module takes in a horizontal and vertical count value based on the incoming pixel's position, and based on that outputs the corresponding row of weights. The output is in the form shortint weights_out[9:0] which is an array of 10 shortint values, which are 16-bits. The module uses an index based on H_count and V_count to determine the correct corresponding weights. This index is (V_count*28) + H_count - 29. There are 784 rows of vectors total, each row having 10 distinct weights. In addition, this module controls whether the weights are being read (testing mode) or written (training mode).
Purpose: To determine the correct weight vector corresponding to the current pixel.

Module: mult_accum
Inputs: enable_clk, accum_start, aclr3, clock0; [15:0] dataa, datab
Outputs: [31:0] result
Description: This is an automatically generated file for our multiply accumulate IP. It takes inputs dataa and datab, and multiplies them. It then accumulates these products while accum_start is high, and clears the accumulator whenever aclr3 is set high. We also use the enable_clk signal to control when accumulation occurs.
Purpose: To perform multiplication and accumulation for our pixel*weight vectors.

Module: multiply
Inputs: clk, reset, pixel, start_stream; [4:0] H_count, V_count
Outputs: [3:0] classification; shortint output_test[9:0]
Description: This module instantiates the weights_controller and matmux10 modules in order to perform matrix multiplication of the weights and pixels. It then outputs a classification based on the weights and what the program determines the input image digit to be. It also outputs a vector, output_test which is used for backpropagation (training) purposes.
Purpose: To instantiate the weights_controller and matmux10 modules in order to perform matrix multiplication for testing and determining the classification of the input image.

Module: matmux10
Inputs: shortint weight[9:0]; pixel, clk, reset, start_stream
Outputs: done; [3:0] classification; shortint output_test[9:0]
Description: This module performs matrix multiplication between an input pixel and its corresponding weights vector, for the entire image. It implements a state machine and utilizes the mult_accum IP in order to perform the multiplication and summation. It then uses the lpm comparator IP to determine the correct classification of the input image. This was further described above, in the multiplication section of our report.
Purpose: To perform multiplication between input pixels and their corresponding weight vectors, and then accumulate these values and use a comparator to determine the largest value of the summed up array and therefore the correct classification.
Module: VGA_controller
Inputs: [7:0] iRed, iGreen, iBlue, iGrey, iBW; iCLK, iRST_N, itoggle
Outputs: oRequest, start_stream; [7:0] oVGA_R, oVGA_G, oVGA_B; oVGA_H_SYNC, oVGA_V_SYNC; [4:0] Sample_H_Cont, Sample_V_Cont; oVGA_SYNC, oVGA_BLANK, oVGA_CLOCK
Description: This is a given module which sends control signals to the VGA based on the image taken. It sets the start stream signals and the color of the image.
Purpose: To send control signals to the VGA.

Module: Stack_RAM
Inputs: clock; [9:0] data, rdaddress, wraddress; wren
Outputs: [9:0] q
Description: (Given module) This is a ram megafunction used with the camera. It is used to hold the pixel value of an input image temporarily and then obtain the reverse order. It does this through reading and writing in the opposite direction with the help of the Mirror_Col module.
Purpose: Hold pixel value and obtain reverse order.

Module: Reset_Delay
Inputs: iCLK, iRST
Outputs: oRST_0, oRST_1, oRST_2

Description: This is a given module which generates a secondary reset signal with a delay to ensure that the data arriving is correct, after a certain latency. It is used with the camera, memory and VGA monitor.
Purpose: Generate reset signals with a delay.

Module: sram_controller
Inputs: clk, reset_n, display_complete
Outputs: sram_wren, addr_incre
Description: This module controls the read and write signals to the SRAM.
Purpose: To control signals to SRAM.

Module: Mirror_Col
Inputs: [9:0] iCCD_R, iCCD_G, iCCD_B; iCCD_DVAL, iCCD_PIXCLK, iRST_N
Outputs: [9:0] oCCD_R, oCCD_G, oCCD_B; oCCD_DVAL
Description: This is a given module which is used to reverse the order of the stored pixel data in the camera register. Initially, the data is stored from right to left. This module reverses the order in a way such that the the left column of data will be accessed first.
Purpose: Used to obtain the reverse order of input image for correct accessing.
Module: imageprocess (in gray2bw.sv)
Inputs: [9:0] iRed, iGreen, iBlue
Outputs: [7:0] grey_pixel, bw_pixel
Description: This module is used in image processing, to convert the format from grayscale to black and white, aka a binary image. The switches on our DE2 control the exposure of the image, and can be adjusted accordingly.
Purpose: To convert the input image from grayscale to binary black and white.

Module: truncate (in gray2bw.sv)
Inputs: clk, reset, start_stream
Outputs: accept
Description: This module truncates the 640x480 input image to a 448x448 image so that it can be downsampled to a 28x28 image. There is an horizontal and vertical counter which keeps track of pixel position and the borders at which to truncate.
Purpose: To truncate the input image down to 448x448, for downsampling to 28x28.

Module: down_sample2
Inputs: clk, reset, in_accept
Outputs: out_accept; [4:0] oH_cont, oV_cont; image_ready
Description: This module downsamples our truncated 448x448 image to a 28x28 image. The input in_accept chooses the first pixel of the 448x448 image, and out_accept chooses which pixel begins our 28x28 image. We also have a 16-count counter which determines when to start sampling the 448x448 image. The counter is incremented each time a pixel is accepted, therefore, every 16 counts.
Purpose: To downsample our truncated 448x448 image to a 28x28 image.

Module: shiftpixel
Inputs: [9:0] iRed, iGreen, iBlue
Outputs: [7:0] oRed, oBlue, oGreen
Description: This module shifts the RGB bits by 2. This is because the given modules contain 10 bits RGB channels, while our compatibility is with 8 bits.
Purpose: To shift the RGB values from 10 to 8 bits.

Module: tristate
Inputs: Clk, OE; [N-1:0] In
Outputs: [N-1:0] Out
Inout: [N-1:0] Data
Description: This is a tristate buffer used for I/O with memory.
Purpose: Acts as tristate buffer for SRAM.

Module: compare
Inputs: [15:0] dataa, datab
Outputs: alb
Description: This module is an automatically generated file through the creation of our lpm comparator. We use this for classification, depending upon the output signal alb. alb outputs high if dataa < datab, else outputs low. We use this to compare the 10 pixel*weight values in a given array.
Purpose: To determine the correct number classification of an input image.

Module: CCD_Capture
Inputs: [9:0] iDATA; iFVAL, iLVAL, iSTART, iEND, iCLK, iRST
Outputs: [9:0] oDATA; [10:0] oX_Cont, oY_Cont; [31:0] oFrame_Cont; oDval
Description: This is a given module used with the camera.
Purpose: This module streams pixel data from the image sensor to the FPGA.

Module: DE2_CCD
Inputs: CLOCK_27, CLOCK_50, EXT_CLOCK; [3:0] KEY; [17:0] SW; UART_RXD, IRDA_RXD
Outputs: [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, HEX6, HEX7; [8:0] LEDG; [17:0] LEDR; UART_TXD, IRDA_TXD; [11:0] DRAM_ADDR; DRAM_LDQM, DRAM_UDQM, DRAM_WE_N, DRAM_CAS_N, DRAM_RAS_N, DRAM_CS_N, DRAM_BA_0, DRAM_BA_1, DRAM_CLK, DRAM_CKE; [15:0] SRAM_DQ; [19:0] SRAM_ADDR; SRAM_UB_N, SRAM_LB_N, SRAM_WE_N, SRAM_CE_N, SRAM_OE_N, VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK, VGA_SYNC; [9:0] VGA_R, VGA_G, VGA_B; [35:0] GPIO, I2C_SDAT, I2C_SCLK.
Inout: [15:0] DRAM_DQ, SRAM_DQ
Description: This module is the top-level entity for our entire project. It instantiates all other modules including the memory, camera, VGA, and arithmetic components.
Purpose: To control the flow of data of all components and tie all modules together.

Module: I2C_CCD_Config
Inputs: iCLK, iRST_N; [15:0] iExposure
Outputs: I2C_SCLK
Inout: I2C_SDAT
Description: This is a given module used with the camera in order to adjust the exposure of the captured image. The adjustment is made through the on-board switches.
Purpose: To set exposure time to adjust lighting of image.

Module: I2C_Controller
Inputs: CLOCK, GO, RESET, W_R; [23:0] I2C_DATA
Outputs: I2C_SCLK, END, ACK, SDO; [5:0] SD_COUNTER
Inout: I2C_SDAT
Description: This is a given module that controls the I2C.
Purpose: To control the functionality of the I2C.

Module: largest_reg
Inputs: clk, reset; shortint result[9:0]
Outputs: [3:0] classification
Description: This module utilizes the LPM_Compare component to determine the classification of the handwritten digit in the input image. It reads the comparator output signal alb to determine which of 2 values in the input array of shortints is larger, and keeps track of this index. Because of the way our arrays are indexed, the final classification is 9-i, where i is the index of the largest component. It then outputs the classification.
Purpose: To use the comparator to determine the classification of the input image.

Module: Line_Buffer
Inputs: clken, clock; [9:0] shiftin
Outputs: [9:0] shiftout, taps0x, taps1x
Description: This is an automatically generated file for a ram-based shift register. It is a given module used with the camera part of our project. It gets the raw data from the image sensor, utilizing a Bayer Filter.
Purpose: This module is used to get raw data from the image sensor.

Module: RAW2RGB
Inputs: [10:0] iX_Cont, iY_Cont; [9:0] iDATA; iDVAL, iCLK, iRST
Outputs: [9:0] oRed, oGreen, oBlue; oDVAL
Description: This is a given module used with the camera and image data. It instantiates the module Line_Buffer, taking in data from the Bayer Filter. This image data is in raw format, so this module converts it to RGB.
Purpose: Convert raw image data to an RGB channel.

Module: SEG7_LUT_8

Inputs: [3:0] iDIG
Outputs: [6:0] oSEG
Description: This is a given module used for I/O. It allows us to display data in hex.
Purpose: Allows data to be displayed in hex.

Module: rdbw (in sram_wr.sv)
Inputs: clk, reset_n, ctr_en
Outputs: [9:0] bw_rdaddr
Description: This is a helper module for incrementing SRAM address.
Purpose: To generate the write address to SRAM.

Module: SW_b_to_dec
Inputs: [4:0] train_label_b
Outputs: shortint train_label_dec[9:0]
Description: This module is used in the training mode of our project. It takes Switches 13-16 from the DE2 as input and based on that, produce a one-hot encoded vector as output. This vector can be thought of as the ‘correct’ vector for training purposes. For instance, if the switches display binary number 0001, then train_label_dec[1] (because 0001 is decimal 1), will be set to 1000, while the rest of the vector will have values 0.
Purpose: To convert the binary switches value to a decimal array for training purposes.

Module: MSE_derivative
Inputs: shortint train_label_dec[9:0], shortint output_test[9:0]
Outputs: shortint result[9:0]
Description: This module takes in the one-hot encoded vector created in the SW_b_to_dec module as well as the output vector from our matrix multiplication and subtracts them to get an error value between the actual weights*pixel vector (output_test) and the correct one (train_label_dec). This is outputted in the array, result.
Purpose: To determine the derivative of error between the correct label and the predicted label.

Module: train _state_machine
Inputs: clk, reset, start_stream; shortint result_prev[9:0]; pixel, start_train_sw, start_train; shortint weights[9:0]
Outputs: shortint new_weights[9:0]
Description: This module takes in the error from the previous module as well as the current pixel and if it is in training mode, indicated by start_train_sw and start_train, it rewrites the old weights values. This trains the module to determine/output a certain digit based on an input image. This module uses the mult_accum IP to perform the multiplication between the error and the input pixel, and stores it in an array, delta_w, which is used to determine the new weights.
Purpose: To train the neural network by determining new weights and rewriting the old ones.

Module: train

Inputs: clk, reset, start_train_sw, start_train, pixel; [4:0] train_label_b; shortint output_test[9:0]; [4:0] H_count, V_count
Outputs: NONE
Description: This is a wrapper module for all our training functions, instantiating the modules weights_controller, SW_b_to_dec, MSE_derivative, and train_state_machine. This module was created for the purpose of making our top-level entity simpler and more readable.
Purpose: Wrapper module for training functions to make top-level more readable.

Design Resources and Statistics

LUT	71,670
DSP	100
Memory(BRAM)	119,808 bits
Flip-Flop	64,306
Frequency	111 MHz
Static Power	107.60 mW
Dynamic Power	0.00 mW
Total Power	295.24 mW

Discussion

Since our original weights are trained with MNIST dataset, we cannot it to perform as well using images taken by our camera. With good lighting and positioning, our classifier is able to predict correctly around 80% of the time. This accuracy goes up to around 90% if we specially trained it using the images taken by the camera and using our backpropagation module.

Conclusion

Overall, we were able to complete a very successful project which could detect the value of a handwritten digit with high accuracy. Since our original weights are trained on MNIST and because of compatibility issues with IPs, we were forced to implement our entire network using integers, rather than floating point values. Because of this, the accuracy was decreased. However, overall we were pleased with our forward and back propagation units and our ability to implement them all in SystemVerilog on the FPGA regardless of difficulty.