ECE532: Group Report

Image Processing Pipeline with Handwritten Digit Recognition

Group #6
Richard Barrie
Savo Bajic
Xiaonong Sun (Frank)
Chun Yin Au Yeung (Cyrus)

Professor: Jason Anderson
TA: Camilo Vega

University of Toronto
April 14th, 2023

Contents

List of Figures

List of Tables

1 Project Overview

As we continue into the digital age and develop faster cameras with better resolution, the need for efficient video processing rises. Because of the huge data throughput used in high definition video, modern CPUs struggle to keep up with the demand for personal, commercial, and scientific applications.

A key operation in modern image processing techniques such as convolutional neural networks is the 2D convolution operation. This relies heavily on multiply-accumulate (MAC) operations where a "kernel" of weights is multiplied by the value of the pixels underneath and then these products are summed. With a simple 3-by-3 kernel, each pixel in the output image is the sum of nine multiplications.

As processor clock speeds plateau, accelerators are imperative to keep up with faster and higher definition video. Fortunately, this task is well-suited for such acceleration due to its repetitive nature.

The goal of our project was to create a reliable, hardware-accelerated image processing pipeline. We selected the Nexys 4 DDR board for its VGA output capability, and the OV7670 VGA camera module from Omnivision. We had the goal of implementing edge detection with an algorithm that uses 2D convolution. Customizable image processing thresholds were desired, so that the user can fine tune the image processing depending on the lighting conditions. Our initial goals of the were to achieve at least 10 frame per second video throughput, with a latency under 200ms. The system should also be reliable with no connection drop or pixel glitches. Figures 1 and 2 show block diagrams of the final system.

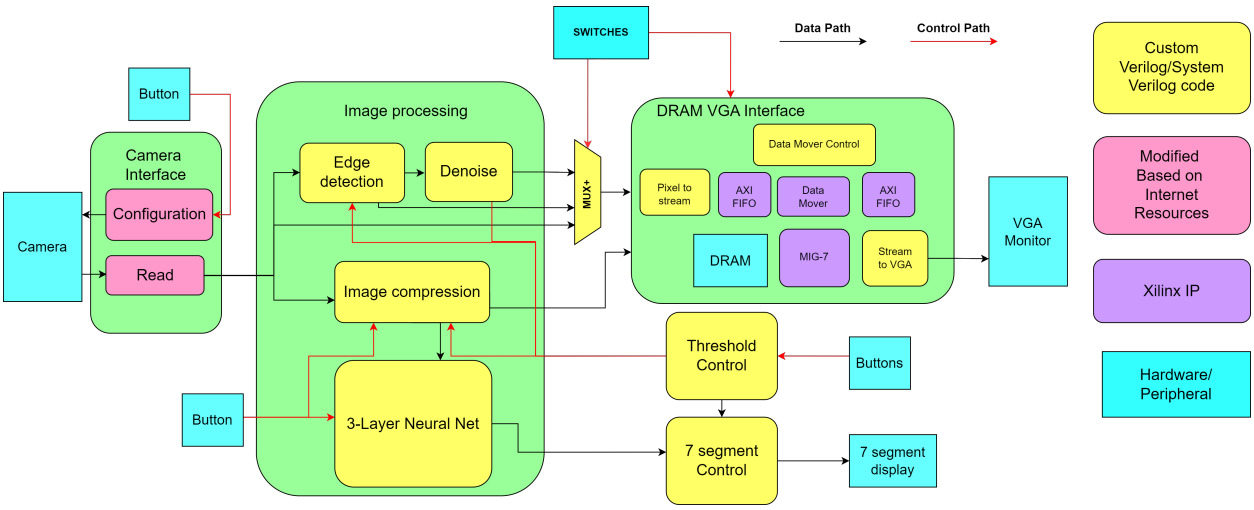

Figure 1: The block diagram of the entire system

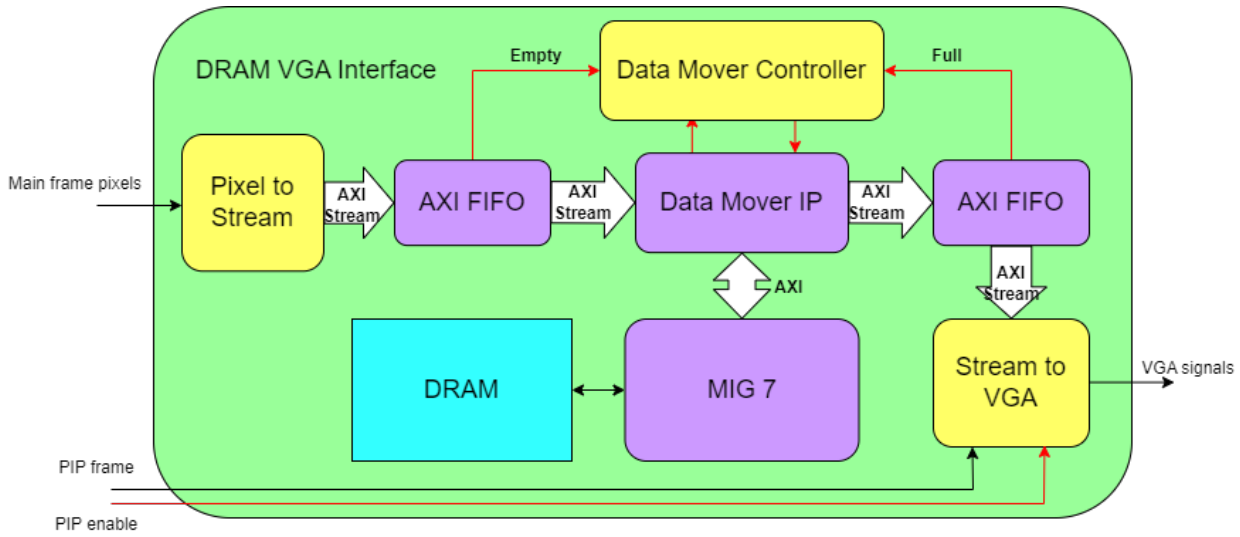

Figure 2: Focused block diagram of the DRAM VGA interface

From the block diagrams above and the key on the right, it can be seen that the whole project is purely hardware without using any soft processor. The project was mainly completed using custom Verilog modules. The Verilog code for the camera interface was based on an online resource [1].

The SCCB block was modified by flipping the polarity of the communication signals and the camera configuration settings we used were based on our group's testing, with help from another group using the same camera.

The VGA portion made extensive use of Xilinx IP to help interface with the dynamic memory off-chip (DRAM). At the heart of the VGA portion was a DataMover IP which was responsible for storing an incoming stream of pixels to a region in the DRAM, while also drawing pixels from another portion of memory into a stream to be displayed. The custom Verilog was used to prepare the streams from pixel data, converting stream data to a VGA signal, and most importantly maintaining a "ping-pong" buffer for frame data in the DRAM so that only completely rendered frames were displayed. There was also a provision for a Picture-In-Picture (PIP) feature to be used for displaying an image over the frame in the top left of the screen, this used on-chip memory for its data.

2 Outcome

We achieved our original goals, with some additional improvements. The final project could perform video streaming from a PMOD camera to the Nexys-4 DDR board through VGA, delivering a consistent output of 60 frames per second without any drops in frames. We achieved real-time edge detection using 2D convolutions with Sobel filters, as well as a custom de-noising technique. These have customizable thresholds so that the user can manually adjust the sensitivity of the edge detection and the denoising, controlled using switches and buttons for inputs, and incorporating discrete LEDs and seven-segment

(HEX) displays for outputs. The controls for these thresholds are shown in Figure 3. A comparison between the performance of final system and our goal is shown in Table 1.

Criterion	**Original Goal**	**Achieved**
Throughput	10+ FPS	30 FPS
Latency	Less than 200 ms	33 ms
Reliability	No dropped connections and pixel glitches	No dropped connections and pixel glitches

Table 1: Desired and achieved values for the system

In addition to the project's core functionalities, we incorporated a handwritten digit classifier, using a neural network to further demonstrate the capability of fully hardware-accelerated image processing. The system was able to accurately detect digits captured by the camera under a controlled environment. Images from the camera were first compressed to a 28-by-28 pixel image before classification. The compressed image was displayed using the PIP feature, and the classified digits were displayed on the HEX display shown in Figure 3

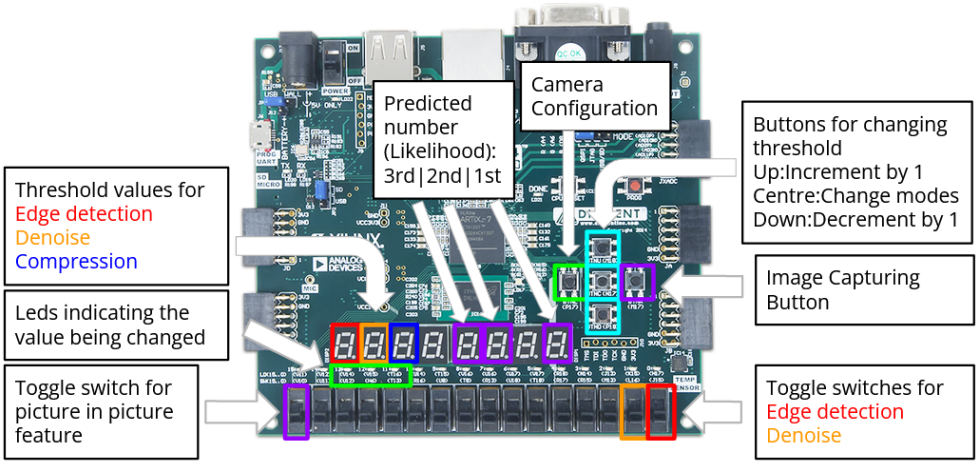

Figure 3: Board controls and output layout

In addition to this, we implemented a ping-pong DRAM frame buffer, which played a pivotal role in reducing the amount of BRAM needed, allowing for additional project features. The resource usage for our final project is shown in Table 2.

Resource	Quantity Utilized	Percent Utilized	Comments
LUTs	33,041	52.12%	
LUTRAM	1645	8.66%	
Flip Flops	22,422	17.64%	
DSP	72	30.00%	Used for matrix algebra
BRAM	2.5	1.85%	Used for buffers around DRAM

Table 2: Final FPGA resource utilization

In hindsight, we realized that incorporating documentation from the beginning would have been beneficial. This includes better commenting in the Verilog code, as well as a step to step guide on how to set up the system. Although our team worked closely together and documentation was not strictly necessary for our project, it would have saved time and facilitated knowledge transfer between team members by clearly explaining the project concept and the work done by each member. Additionally, this best practice would have prepared us for the possibility of other people picking up the work in the future and would have been highly useful in an industry setting where collaboration and handovers are essential.

In the event that someone takes over the project, there are several enhancements that we could implement to improve its functionality and accuracy. The first area of improvement would be the image recognition system. Our current system has limitations, specifically with regard to the need for well-written and centered digits. To address this limitation, one could explore implementing a larger neural network, subject to the available memory on the FPGA. Additionally, one could consider a convolutional neural network instead of a simple fully-connected network like we implemented. Furthermore, one could add additional features to enhance the project's capabilities. This would include recognizing operators such as '+', '-', 'x', and '/', enabling the system to perform expression detection and give the answer to the detected expression. To accomplish this, image segmentation would be needed, either through the naive approach of segmenting the field into three equal-sized parts or with more advanced machine learning algorithms. These enhancements would make the project an even more versatile and powerful tool, capable of handling more complex image recognition tasks with greater accuracy and efficiency. Another approach to increasing the project's versatility is incorporating additional input/output capabilities. For instance, adding a keyboard interface that enables users to manually adjust the convolution kernels. This would provide greater flexibility and control over the image processing operations, allowing users to customize and fine-tune the system's performance to their needs.

3 Schedule

Milestone 1 - Feb. 1

- Research on camera and VGA interface
- Planning on system architecture

Milestone 2 - Feb. 8

- Camera interface block completed and tested in test bench
- VGA block completed and test pattern can be shown on screen
- Convolution core completed and max pooling was tested using test image
- System skeleton based on AXI streaming was set up

Milestone 3 - Feb. 15

- BRAM frame buffer completed
- Integrating camera with frame buffer and VGA interface
- Rework on convolution core to gain 10X LUT utilization improvement

Milestone 4 - Mar. 1

- Edge detection core completed and was tested using test image
- Debugged VGA timing handling issue
  - Solved the fame shifting issue at the start of each frame
- Replaced AXI stream with direct parallel connection instead

Major achievement - Black and white image from the camera can be shown on the VGA monitor

Mid-project Demo - Mar. 8

- Debugged camera configuration issue
  - Communication signal bits were flipped, leading to initialization failure
- Integration of the edge detection core with the system

Major achievement - Edge detection could be done in real time on camera feed. Camera could be configured using user defined parameters.

Milestone 5 - Mar. 15

- Denoise algorithm was completed and implemented
  - Image after edge detection was more clean by filtering out unwanted noise
- Image compression core was completed and implemented

  - The VGA frame is cropped and compressed into a 28x28 image
- Auto white balance and gain control was realized by finding a proper camera configuration though testing

Major achievement - Better edge detection algorithm, image compression, better image quality from the camera

Milestone 6 - Mar. 22

- Trained a single layer linear regression model in Python
  - 32-bit integer weights and 4-bit black and white 28x28 Mnist data sets were used
- The digit recognition model was implemented
  - Accuracy was up to 86%
- 7 segment display block was implemented

Major achievement - Preliminary digit recognition

Final Week - Mar. 30

- Trained a three layer neural net in Python
  - Accuracy was up to 91.4%
- The 3-layer neural net was implemented
  - The top 3 most probable recognized could be shown
- Threshold control block was completed and implemented
  - Threshold of edge detection, denoising and image compression can be adjusted
- DRAM buffer was completed and implemented
  - Frame buffer was moved from BRAM to DRAM to free up more resources
  - Enabled proper ping-pong buffering of the frames (only completed frames were displayed)
- Picture-in-picture feature was completed and implemented
  - The compressed image snapshot can be shown on the top left of the monitor
  - The snapshot can be scaled into a larger form factor

Major achievement - Digit recognition with much higher accuracy and the ability to adjust the thresholds for image processing modules in real time

Final Demo - Fully functional system

4 Description of Blocks

4.1 Camera interface

The camera interface contains the following three sub-blocks which are Camera Configuration, Camera Configuration ROM and SCCB. The detailed block diagram of the camera interface is shown in Figure 4.

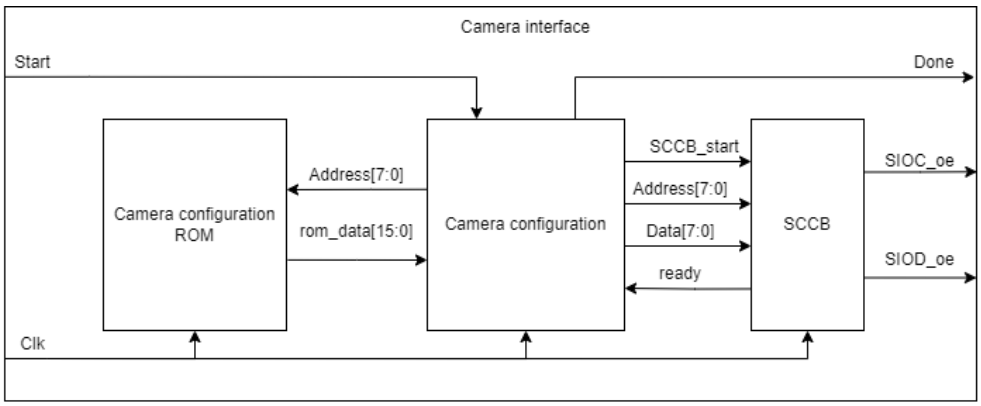

Figure 4: Camera interface sub-block hierarchy

4.1.1 Camera Configuration

The camera configuration block is a finite state machine block to control the camera configuration process. When a Start signal is received, it loops through the camera configuration address and fetches the respective configuration settings from the camera configuration ROM block. Note that the configuration settings are represented as a 2-byte words while the SCCB block can only send out 1 byte at a time. Thus, the camera configuration block also splits the fetched settings into two packets to be sent over in two cycles. The parsed data and address are then send over to the SCCB block and communicate with the camera is initialized using the SCCB_start signal. A Done signal will be flagged when all the addresses are looped through to indicate the end of configuration.

4.1.2 Camera Configuration ROM

This block acts as a bank to store the user defined camera configuration settings. The user can modify the data within this block to test and change for optimal camera settings.

4.1.3 SCCB

This block handles the communication with the camera module. It adopts the SCCB protocol which is a modified I2C protocol used by Omnivision. As mentioned before, configuration data at each address needs to be parsed into two packets and then be sent out a

byte at a time to the camera, thus a ready signal is needed to feed back into the camera configuration block to indicate the receiving of the next data after the first byte is sent out.

The source of the camera configuration blocks came from a github repository [1] but modifications were done on the SCCB block. Also, the camera configurations in the Camera Configuration ROM block were based on our group's testing as well as the help from another group using the same module.

4.2 Edge Detection

Overview

Edge detection was done using a custom packaged IP block written in System Verilog. Figure 5 shows an overview of how the block works. The image is processed with a 2D convolution with a three-by three kernel, called a "Sobel filter " [2]. It is processed with two different kernels, that calculate the vertical and horizontal gradients of the image. If the image gradient positive and above a threshold, this corresponds to a light-to-dark edge, and if it is less than zero and and below a threshold, this corresponds to a dark-to-light edge. To capture both light-to-dark and dark-to-light edges in the vertical and horizontal directions, we take the absolute value of the image gradients in both directions, and sum them together. We then apply a threshold to pick out the pixels that are edges. We found that different thresholds were needed to achieve clean edges in different lighting conditions, and so we added a manual threshold setting to our IP block.

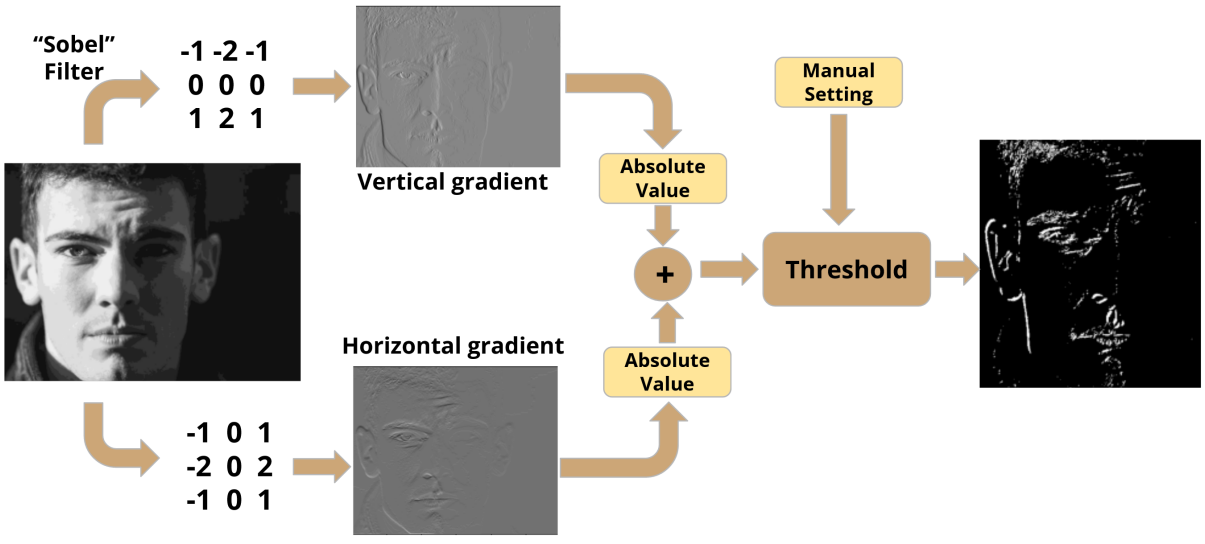

Figure 5: Edge detection flow diagram

Implementation

4-bit black-and-white pixel intensity values are input to the block serially. An internal buffer stores 4 full rows of pixel intensity values. Only three consecutive pixel rows are needed to calculate one row of the convolution output. Once three consecutive rows are loaded, each 3-by-3 square is passed to a custom multiply-accumulate module that performs the Sobel filter calculation, while the fourth row is being filled up with new pixel values. Once the fourth row is filled, the rows are switched in a rolling fashion, so that there are always three consecutive full rows in the buffer, and the fourth row is being filled with incoming pixel intensity values. Zero-padding is needed along the edges of the image to keep the input image and output image dimensions the same, to comply with VGA image standards. An example of the internal buffering for a 6-by-6 pixel image is shown in Figure 6. Each step corresponds to the time to buffer in a new row of pixels. The calculation of output pixel intensity for the first column of each row is highlighted in pink.

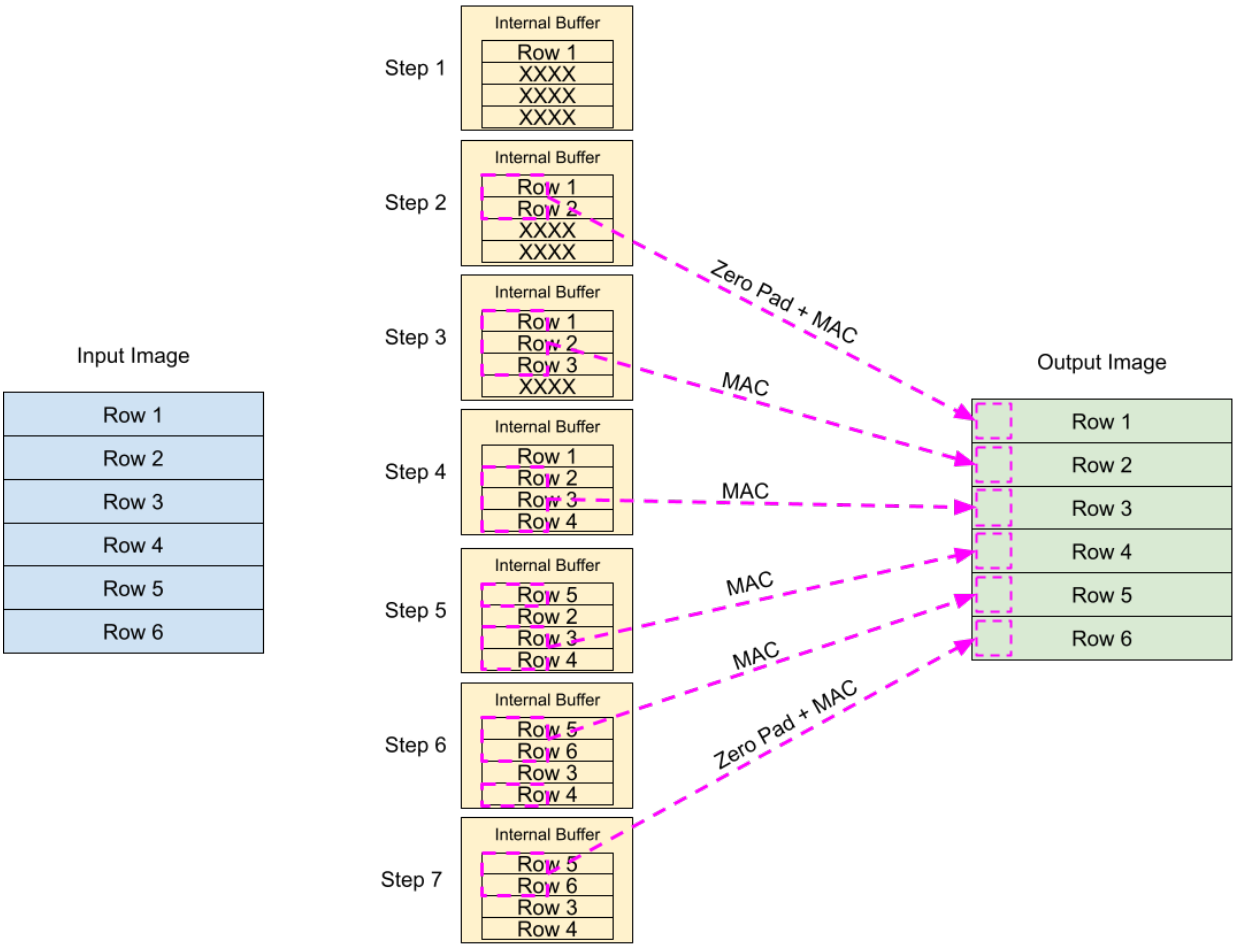

Figure 6: Edge detection internal buffering

4.3 DeNoise

After edge detection, we found that there was noise in our image, and so we implemented a simple algorithm to remove this noise:

Algorithm 1 DeNoise Algorithm

```
1: for Each pixel p in image do
2:     if p is an edge then
3:         n ← # adjacent pixels to p that are also an edge
4:         if n ≥ THRESHOLD then
5:             Output pixel is an edge
6:         else
7:             Input pixel is noise, output pixel is not an edge
8:         end if
9:     else
10:        Output pixel is not an edge
11:    end if
12: end for
```

This algorithm is efficiently implemented using the same method as with the 2D convolution described in section 4.2. The difference is that instead of using a multiply-accumulate module on each 3-by-3 window of pixels, we use a new module that counts the number of pixels adjacent to a pixel that are an edge, and returns 1 if the amount is greater than or equal to the threshold.

4.4 Image Compression

Compressing an image frame from full resolution VGA (480 by 640 pixels) to a 28 by 28 pixel image before attempting digit recognition allows us to use a much smaller model without loss of accuracy. 28 by 28 pixels was chosen as our resolution to match with the MNIST database for handwritten digits [3]. Algorithm 2 shows our algorithm for compressing the image from full resolution 4-bit black-and-white to 28-by-28 pixel 4-bit black-and-white images using the full dynamic range from 0 to 15 pixel intensity in the output image. In order to match the MNIST dataset using a black marker on white paper, the image pixel intensity is inverted (i.e. white number on a black background). Also, we implemented a threshold that sets pixels below a certain intensity to pure black to combat camera vignetting (the edges of the frame being darker than centre).

The following algorithm was implemented in a custom IP block, which allowed us to achieve image compression that closely matched with the MNIST database.

Algorithm 2 Compression Algorithm

for each of the 784 17-by-17 pixel squares S_i shown in Figure 7 **do**
    $n_i \leftarrow$ # sum of all pixel intensities in the S_i
end for
$min \leftarrow min(n_i)$ ▷ Find minimum of all 784 sums
$max \leftarrow max(n_i)$ ▷ Find maximum of all 784 sums
for each n_i **do**
    $n_i \leftarrow (n_i - min) \times 15)/(max - min)$ ▷ Scale sums so that they range from 0 to 15
    $n_i \leftarrow 15 - n_i$ ▷ Invert pixel intensity
    if $n_i \leq$ THRESHOLD **then**
        $n_i \leftarrow 0$ ▷ Set pixel intensities below threshold to 0 (pure black)
    end if
end for

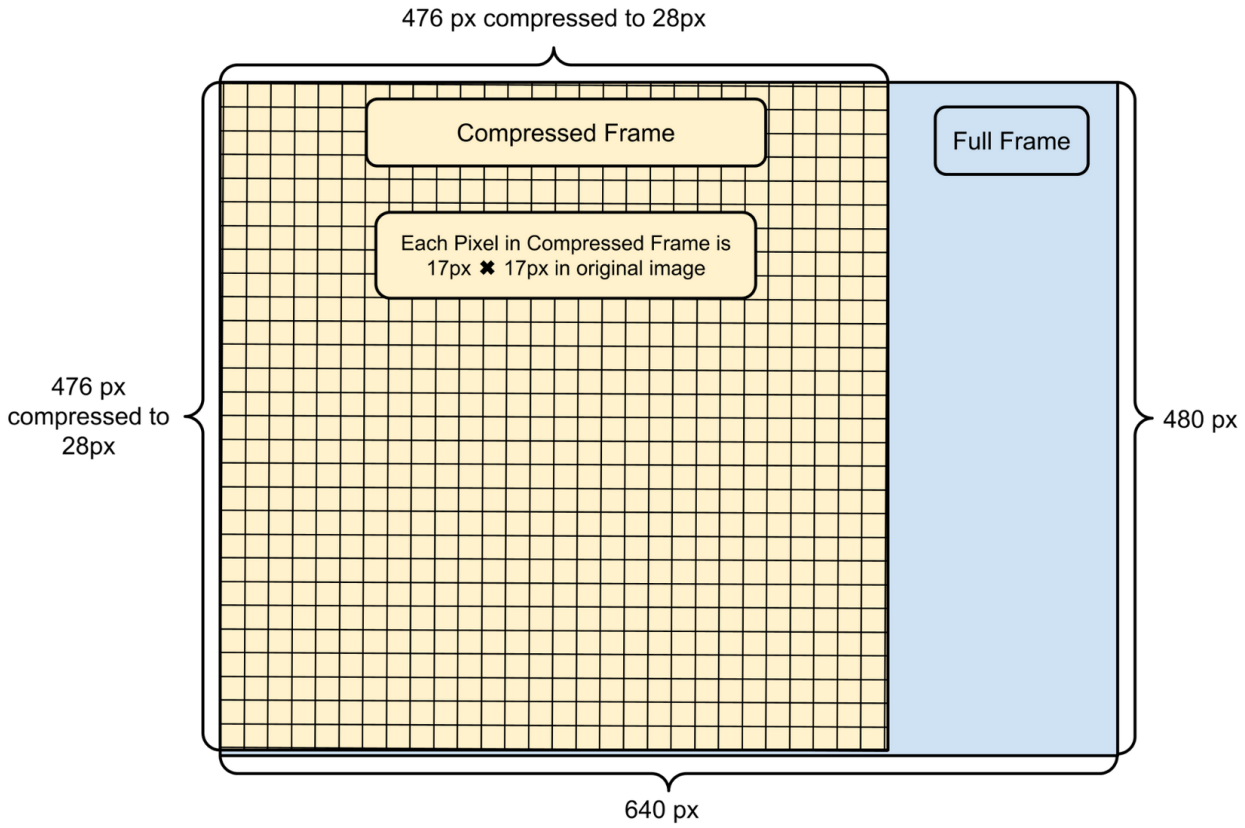

Figure 7: Diagram of VGA frame compression

4.5 Neural Network

For digit recognition, we developed a three-layer neural network shown in figure 8, note that the first layer is only represented as a size 28 array when it should be size 784 for ease

of visualization. The input layer is a 1 by 784 matrix, corresponding to the compressed 28 by 28 image from section 4.4. The first hidden layer has 10 neurons, and so the first weight matrix 784 by 10. The second hidden layer has 10 neurons so the second weight matrix is 10 by 10, and finally one more 10-by-10 weight matrix is used to produce the output layer. To accommodate the paramaters for this architecture were stored in distributed ram on the FPGA. The hidden layers are activated by ReLU activation, while the output layer utilizes linear activation. Furthermore, we incorporated biasing terms for each layer to optimize the network's performance.

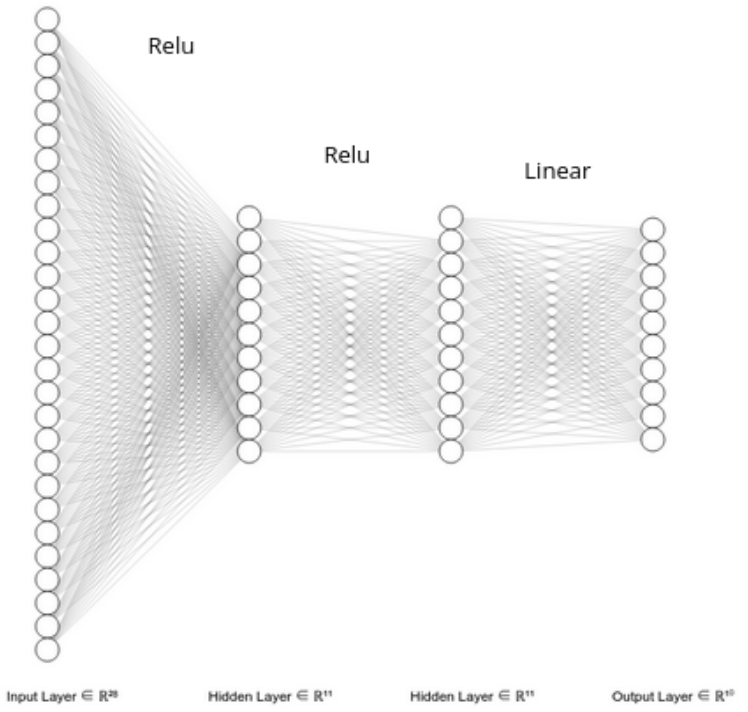

Figure 8: Three layer digit recognition neural network model

To train our neural network, we used the MNIST dataset, which contains a collection of handwritten digits represented as 28-by-28 pixel images. We used gradient descent to train the network paramaters. Figure 9 shows an example of each digit from the MNIST dataset, and Figure 10 shows the decreasing loss over 3000 epochs of training. THis allowed us to achieve an accuracy of around 90% correct classifications on our test dataset. We chose hidden layers of size 10 to balance the network accuracy with the number of paramaters. Increasing the size to 20 by 20 would increase the memory requirements by more than a factor of 2, but only improve the accuracy by approximately 3%.

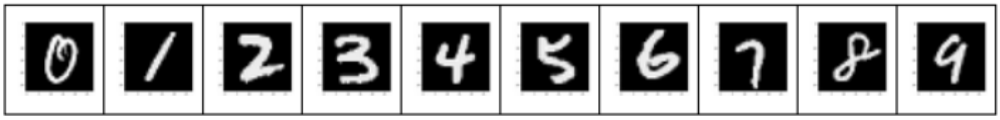

Figure 9: Example of the digits in the MNIST dataset used to train our model

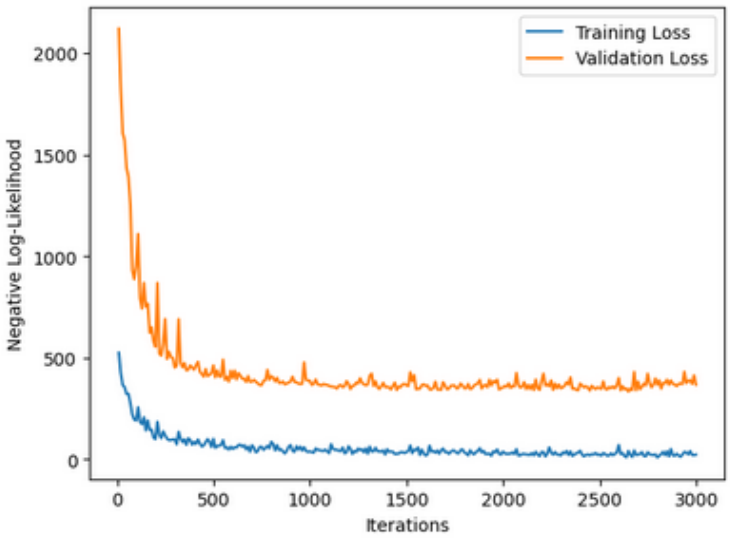

Figure 10: Training and validation loss over number of iterations

After training, the weights were processed from floating point numbers to 16-bit integers for easier multiplication and accumulation in the FPGA. More information on this is availalbe in section **??**. Afterwards, the weights were read into the FPGA's RAM. To implement the desired multiply and accumulate, we used a finite state machine with three states for each of the three layers.

For each layer, one state performs the multiplication, one state adds the biasing term and one state performs the ReLU activation. The multiplication state is implemented by iterating over the 28 by 28 compressed pixel values in c_pixels and multiplying them with the corresponding weights in W1, and then adding the results to the corresponding element in the L1_multiout array. The multiplication is done by first sign-extending the 4-bit pixel value to 5-bit by concatenating a 0-bit to the most significant bit to allow for signed multiplication. The resulting 32-bit value is then multiplied by the corresponding weight in W1, which is also a 32-bit value. Finally, the result is added to the corresponding element in the L1_multi_out array. Once all the pixels have been multiplied with their corresponding weights, the state variable is updated to the next state, and the i variable, which keeps track of the pixel index, is reset to 0. The matrix multiply stage is shown in algorithm 3, note the second loop is done in parallel (10 multiplications done per cycle), and the master loop is done sequentially (784 clock cycles to do all multiplication). For subsequent layers, the weights are changed from $W1\_n$ to Wx_n.

Algorithm 3 Matrix Multiply Algorithm

for each of the 784 pixels in the 28 by 28 compressed image **do**
    if $i \geq 784$ **then**
        $state \leftarrow$ adding in bias state
        $i \leftarrow 0$
    else
        $i \leftarrow$ i + 1
        for each n in 10 **do**
            $L1\_multi\_out[n] \leftarrow L1\_multi\_out[n] + \$signed(1'b0, c\_pixels[i])) \times W1\_n[i]$
        end for
    end if
end for

In the bias stage, the bias term is added to the output of the previous matrix multiplication step, which was stored in the L1_multi_out array. The "if" statement checks if the index "i" is greater than or equal to 10, and if so, the state variable is updated to the next state and the index is reset to 0. If the index "i" is less than 10, then it is incremented by 1 and the bias term (stored in the B1_0 array) is added to the corresponding element in the L1_multi_out array. This process is repeated for all 10 elements in the L1_multi_out array. The bias adds stage is shown in Algorithm 4. This stage is done in 10 clock cycles. For subsequent layers, the bias is changed from B1_0 to Bx_0.

Algorithm 4 Bias Add Algorithm

for each of the 10 bias weights **do**
    if $i \geq 10$ **then**
        $state \leftarrow$ ReLU activation stage
        $i \leftarrow 0$
    else
        $i \leftarrow$ i + 1
        $L1\_multi\_out[i] \leftarrow L1\_multi\_out[n] + B1\_0[i]$
    end if
end for

The ReLU stage introduces non-linearity. This stage checks if each of the 10 outputs of the previous matrix multiplication stage is less than zero. If it is, it sets that output to zero, effectively "turning off" that neuron. If the output is greater than or equal to zero, it remains unchanged. This ensures that the output of the neuron is always non-negative, which can help prevent the "dying ReLU" problem that can occur when the output of a neuron consistently remains negative and prevents the neuron from learning. Once all 10 outputs have been processed, the state variable is updated to move on to the next stage of the program. The ReLU stage is shown in Algorithm 5.

Algorithm 5 ReLU activation Algorithm

```
1: for each of the 10 outputs do
2:     if i ≥ 10 then
3:         state ← next layer multiply
4:         i ← 0
5:     else
6:         i ← i + 1
7:         if L1_multi_out[i] ≤ 0 then
8:             L1_multi_out[i] ← 0
9:         end if
10:    end if
11: end for
```

4.6 Threshold Control

The threshold control block stores three user defined values and delivers them to the three image processing modules. When the centre button is being pushed, it cycles through the three variables to be changed, which are the edge detection threshold, denoising threshold and the image compression threshold. The user can increase or decrease the selected threshold by pressing the up or down buttons. The threshold vales are shown on their respective seven segment display with led indicator underneath it to show the currently selected variable to be changed. The threshold values are then passed to the three image processing blocks for the effect to take place.

4.7 VGA Overview

The video output port available on our development board was VGA, which we operated at the 640 by 480 pixel resolution offered by our camera. A major problem that we identified at the beginning of the project was that this system had to output at 60 frames per second to be compliant with the VGA specifications (and thus be accepted and displayed properly on a monitor), however the camera only output at 30 frames per second. Thus some form of frame buffer was needed to show each frame twice, to achieve 60 frames on the output without blanking or tearing the image on screen.

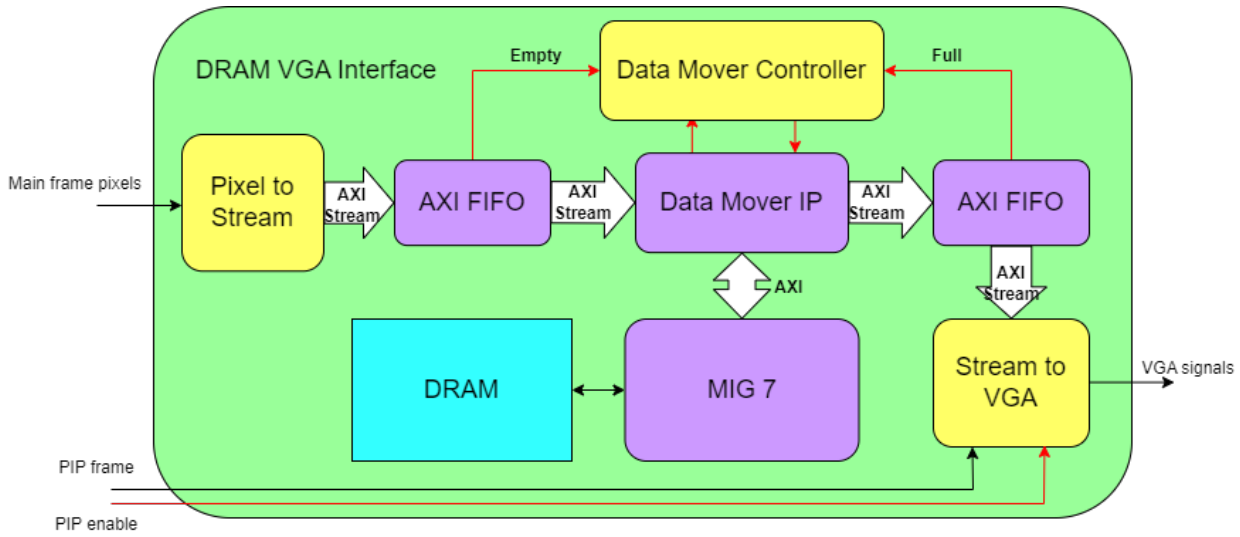

Figure 11: Block diagram of VGA portion (Repeat of Fig. 2)

As shown in Figure 11, the VGA system was roughly half custom Verilog blocks (yellow) and half Xilinx IP blocks (purple). The Xilinx IP was used to move data from a stream to a memory mapped location in the DRAM and back into a stream, with our custom code operating on each end of these streams and controlling the DataMover. All modules in this section were designed to be AXI-Stream compliant since that was the method used to pass data into and out of the Xilinx IPs, the final data width used was 64 bits.

In our original design and the one we used for our mid-project demo, we used a different arrangement for frame memory. We simply wrote to and read from mapped memory inside the FPGA as our buffer. This was acceptable for a baseline system but due to the limited memory resources in the FPGA were not able to store two separate full colour frames since each frame needed 3,686,400 bits (640 by 480 pixels, 12 bits of colour) and the FPGA we used had about 6 Mb of memory available. Thus to implement proper ping-pong buffering of frames we needed to utilize off-chip memory. This brought the benefit of clearer video once implemented, in addition to the swathes of memory that were freed up.

The majority of our effort in the VGA portion went into tuning the frame buffer system, namely ironing out bugs with DataMover.

4.7.1 VGA Driver (Stream to VGA)

The endpoint of pixel data in the system is the VGA driver which accepted pixel data using the AXI-Stream interface and put it on screen sequentially. Although the resolution of the system was constrained by the resolution of the camera, the colour depth was limited to 4 bits per colour channel due to the driving hardware on our development board. This meant that each pixel needed only 12 bits of colour information, which to make mapping pixels to memory easier - each pixel was padded to make each pixel 16 bits wide, thus each 64-bit stream packet would be buffered and sliced internally to provide data for four pixels.

The VGA driver ran off a 25.2 MHz clock to meet the specifications for the VGA output. It uses counters to generate the supporting horizontal and vertical synchronization signals (HSYNC and VSYNC) in accordance to the specification, as well as to know when to display the pixels or not due to the blanking region around each video frame. These counters were also used to verify if the screen output position was synchronized to the incoming pixel stream. The reason being that the VGA driver intentionally didn't wait for data to arrive before it began outputting to the screen, it would merely repeat the contents of the four pixel buffer until new data arrived. However to make sure that the pixel stream from the driver did align with the data stream (e.g. output the pixel meant for the top left in the top left position) the TLAST signal was used for the last packet in an image. If this arrived at any point other than when expected (displaying the last four pixels) based on the internal counters, the driver would reset and the next set of pixel data would be drawn in the first position (top left) of the screen.

As part of the digit recognition, we found it helpful to have a preview of the digit image on screen. This way we could see what the compressed image was on screen that the FPGA was processing, since it was meant to be drawn over the live video like on a television, it was dubbed "Picture-in-Picture" (PIP). Given the small size of the image (28 by 28 pixels) it was also helpful to also have this image scaled up to be easily seen.

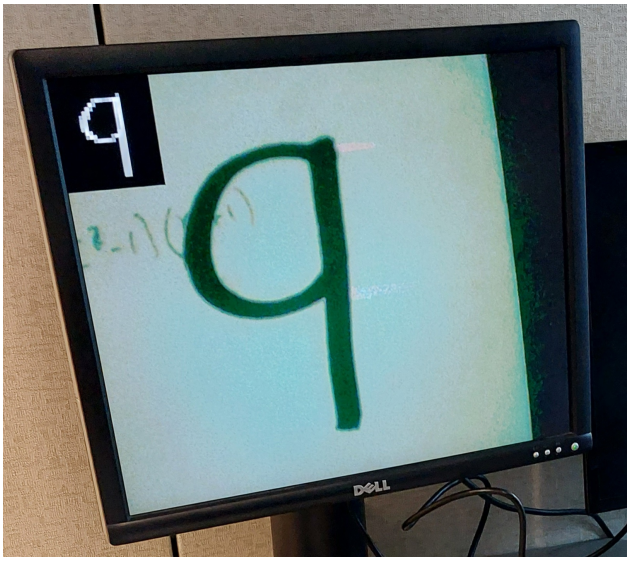

Figure 12: PIP scaling demonstrated over source video

This feature was implemented using a small portion of memory in the the FPGA to store the image which was then read when the driver was in the the region of the screen and the PIP enable signal was set. To scale the image the region was increased by some integer factor (four in our final submission), so each pixel was now repeated. This required minimal memory on the chip thanks to the small size of the image and that it was in gray scale - needing only 4 bits per pixel, for a total of 3136 bits of memory. When rendering the PIP pixels, the pixel data for the live video feed is still sent by the memory management system, so the driver discards the pixels it doesn't use so it can access the ones it

does need in the stream later in each row. The final working system is shown in Figure 12.

4.7.2 Pixel to Stream

This block performed the relatively simple task of reading in the pixel data from the processing portion and collected four pixels worth of data (12 bits each), and padding them with zeros into one 64-bit packet to be forwarded along the AXI-Stream to memory.

In addition to this, since the intended "address" of the pixel was also provided by the previous stage (a holdover from when we used mapped memory), this was used to raise the TLAST signal on the final pixel's packet to synchronize the memory mapped writing between frames so that the next packet would be written to the start of a frame buffer.

4.7.3 Memory IP Blocks

The direct memory management was handled using Xilinx IP blocks, namely the Memory Interface Generator 7 (MIG) to interface to the DRAM chip on the board, and the DataMover to write a stream to a memory location and then read that memory into another stream. There are small FIFO buffers on both sides of the DataMover to help smooth the flow of data along the streams.

All these IP blocks are used with minimal adjustments to their default parameters, one change they did share was having the data bus widths all unified to 64 bits. The specifics of each IP block are broken down per block below.

The buffers are set to different sizes, the buffer feeding the DataMover pixels has a depth of 512 packets due to it occasionally receiving the pixels for the last few rows in quick succession from the vision processing, while the buffer to the VGA driver has a depth of 64 since this was enough for steady video rendering. Other differences include the buffer feeding the DataMover has an empty signal, that will be set if the level drops below 8 packets, the buffer downstream of the DataMover has a full signal that will be set if it has space for less than 8 packets. These signals are used by controller for the DataMover to know when there is enough data/space available for a transfer without the DataMover needing to potentially hold mid-transaction for the stream to move.

The MIG 7 is used exactly as it was generated, connected via an AXI4 interconnect to the DataMover to store the frame buffers on the DRAM chip this freeing up the memory on the FPGA and allowing us to store two complete, full-colour frames. However since it was DRAM and off-chip, the latency was notable which is why the wider data buses and data bursts were used for transactions to/from the memory to offset these effects.

The DataMover IP was also largely used as provided, to move data from stream to memory and back again. The main changes applied to it from its defaults was the widening of the data buses to 64 bits, removing the internal buffering, and having the Stream to Memory Mapped (S2MM) and Memory Mapped to Stream (MM2S) share the same AXI port for the memory access to the MIG.

4.7.4 DataMover Controller

This custom IP was in charge of issuing instructions to the DataMover to stream data to and from locations in memory. The purpose of it was to have the DataMover stream the data from the video processing portion to one region of the memory while reading from another to provide the VGA output pixels. Once the input frame is completely loaded to memory, the regions are swapped for the next cycle of reading and writing.

The controller had additional features to improve the performance of the system. One such feature was monitoring of the levels of the buffers into and out of the DataMover to ensure that the DataMover wouldn't start a transaction without the appropriate data/space available. Another more important feature was that it was responsible for aligning and synchronizing frame data so that pixels that were displayed where they were needed on the screen, with no drift. This was important to do since the pixels were delivered as a stream so there was no location data accompanying them, especially to the VGA driver.

Internally it operates in two, almost entirely independent halves: one for the stream to memory mapped (S2MM) operation of the DataMover, and the other for the memory mapped to stream (MM2S). Other than for sharing a reset signal, the only reason for their connection is to communicate the state of each half of the ping-pong buffer used for the frame output so they know when to swap memory regions to operate in.

Each half has a counter to keep track of the amount of transfers it had completed and what the current target address was for the transaction. After each transaction completed by one half, it tick to up to the next value, and the next transaction instruction to the DataMover would be generated. Once the required number of transactions for a frame are completed for one half, they would decide how to proceed. The S2MM half would begin to write to the next frame region, the MM2S would check if the other frame was complete and only switch to reading that frame if it is complete.

This behaviour was safe to use because the the MM2S would read at twice the rate data was written so even if the reader was partially done reading a frame and the S2MM starts writing to the same region, the S2MM will not catch up to or overtake the MM2S location for that frame and cause video glitches to appear. However, had the MM2S read location been put behind the S2MM one in the same region there would be a chance of the MM2S overtaking it and causing visual glitches, for example if the MM2S started in the region with the S2MM only a quarter through a frame, they would pass one another halfway through that frame so the MM2S would stream data from two different frames.

For testing the teams set the system to read the values of the switches on the board and read them in as pixel colours gradually if a switch was set. This allowed us to "draw" patterns and control the flow of pixels to ensure the system behaved as expected. E.g. if a completely white frame was fed in then the next frame was to be painted black, but not all needed pixels were supplied, then the screen would remain white until all the pixels needed for a frame came in. An example of such a test is shown in Figure 13, with PIP also enabled (showing the default PIP pattern) where the team gradually cycled through colours for a frame and then partially completed a blank frame following it.

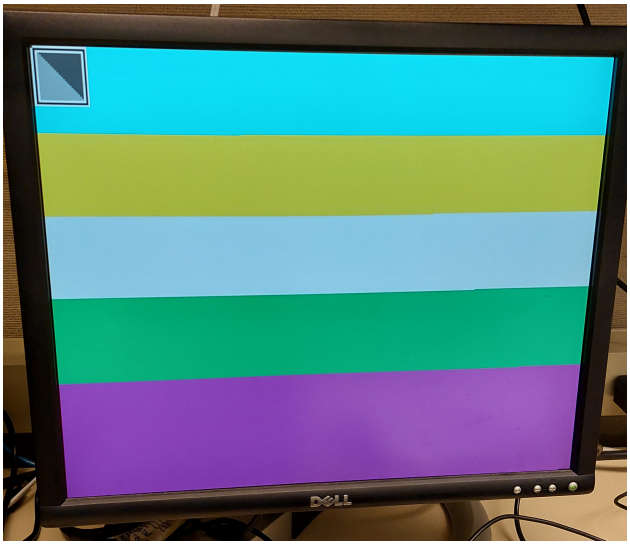

Figure 13: Successful ping-pong buffer testing image (with PIP)

To help enable synchronization through the VGA system, the controller makes use of the TLAST signals that are part of the AXI-Stream standard. When sending the last set of pixels for a frame to the VGA driver, it has the MM2S raise the TLAST flag on the last stream packet, the VGA driver then reacts to this by resetting if it is received at any point other than when expected for the last few pixels. This became vital when this was integrated with the camera.

On the other end, to ensure that the pixels from the video processing are written to the right places in the memory the controller monitors for the TLAST signal on the incoming data. The controller will only move the S2MM to the other region of memory once the internal counter indicates that the required number of transactions to memory have been performed for a frame and that the TLAST signal has been observed (indicating the video system has sent the final pixel for a frame, thus the next pixel will be the start of a new frame). If the counter reaches the expected number of transactions, the controller will keep writing the data to the last portion of the frame buffer to discard data in the buffer until the TLAST signal is received. This does cause the one frame to be incorrect, however this is only something that occurs in the transient state following system resets or camera reconfiguration and is quickly resolved and overwritten once a two more frames make it in; this has not been observed to occur in regular operation.

5 Design Tree

We have uploaded our project to a GitHub repository available `https://github.com/richard259/G6_imageprocessing`. The resources in this directory provide a complete representation of our project, adequate for a future developer to use, modify and improve on our project. In this section of the report, we discuss the structure of our GitHub repository, and provide a description of some of the important files and directories:

/doc This directory contains documentation relevant to our project

Final_Presentation.pdf The slides for our final presentation

Final_Report.pdf This report

/project This directory contains our top-level Vivado project

camera_dram_03_25.xpr - Vivado project file

/camera_dram_03_25.srcs - Design and simulation source files

/constrs_1/imports/Downloads/Nexys4_Master.xdc - Design constraints file

/sources_1/new - Custom Verilog modules

/sources_1/imports - Custom Verilog and memory files for:

- DRAM frame buffer and pixel stream
- Neural network weights
- Camera Configuration

/sources_1/bd - Top-level block design

/IP This directory contains our custom packaged Vivado IPs. These IPs are referenced in the top-level Vivado project.

/Edge_Detection_V4 - Edge detection IP block

/project_1 - Vivado project directory

component.xml - Vivado IP file

edge_detection.sv - Edge detection top-level module

mac.sv - Multiply-Accumulate sub-module, used in edge detection module

edge_detection_tb.sv - Testbench for edge-detection block

/DeNosie_V2 - De-noise IP block

/project_2 - Vivado project directory

component.xml - Vivado IP file

de_noise.sv - De-noise top-level module

mac.sv - Multiply-Accumulate sub-module

de_noise_tb.sv - Testbench for de-noise block

/Python This directory contains the Python code used to train the neural network and save the parameters to .mem files

3_layer.ipynb - Main file for training network and saving parameters

data_utils.py - Helper functions, from the course ROB313

mnist_small.npy - MNIST dataset

6 Tips and Tricks

6.1 Using the DataMover IP

Most of the work put into the VGA subsystem went into wrangling the DataMover to our will, roughly between a week or two of development. The reasons being its odd collection of peculiarities, some of which seem to be known and experienced by others from the research we did. Below are our tips when working with it to more quickly diagnose your issues and address them.

It may be worthwhile for future students to avoid this IP and evaluate if an open-source alternative such as WB2AXIP [4] is a viable replacement for it.

Selecting transaction properties and validating their behaviour in memory

Changing the properties of transactions, namely data width and allowed burst size, caused the DataMover's behaviour when writing to mapped memory to change in ways that were not expected nor explained in documentation provided for them. So after deciding on a set of properties to use, we suggest one does some tests writing controlled data to memory and seeing how it is written using a memory viewer or some alternative method such as decoding AXI exchanges using the Integrated Logic Analyzer.

The reason for this tip is that when we started development of the system we were originally using a 16-bit data bus (one pixel) with bursts of 64 transactions. As expected, the memory from our specified start location was then a continuous stream of our expected pixel data until all pixels were written. As an example, using a value of 0x0FFF for each pixel in a burst of 24 pixels, starting from an address of 0x80000000; the memory would look like in Table 3. This is what is expected, the stream being recorded in an uninterrupted portion of memory starting at the address specified.

Address	Bytes 0-3	Bytes 4-7	Bytes 8-B	Bytes C-F
0x80000000	0x0FFF0FFF	0x0FFF0FFF	0x0FFF0FFF	0x0FFF0FFF
0x80000010	0x0FFF0FFF	0x0FFF0FFF	0x0FFF0FFF	0x0FFF0FFF
0x80000020	0x0FFF0FFF	0x0FFF0FFF	0x0FFF0FFF	0x0FFF0FFF

Table 3: Original memory map

However when we decided to use a wider data bus (64-bits) with shorter bursts (8) in an effort to reduce congestion on the AXI bus, the data in the memory no longer acted as expected. We noticed that the DataMover was now skipping alternating regions of memory, it would write one packet of 64 bits (eight bytes) then skip an equivalently sized region of memory before the next packet. Looking at the memory map the uninitialized memory was visible side by side with the controlled input data. Repeating the previous example transaction would produce a result like shown in Table 4.

Address	**Bytes 0-3**	**Bytes 4-7**	**Bytes 8-B**	**Bytes C-F**
0x80000000	0x0FFF0FFF	0x0FFF0FFF	0xDEADBEEF	0x13371337
0x80000010	0x0FFF0FFF	0x0FFF0FFF	0x0420FACE	0xCAB0CAB0
0x80000020	0x0FFF0FFF	0x0FFF0FFF	0x8BADF00D	0x69696969
0x80000030	0x0FFF0FFF	0x0FFF0FFF	0x0D15EA5E	0x01234567
0x80000040	0x0FFF0FFF	0x0FFF0FFF	0xDEADFA11	0xD0D0CACA
0x80000050	0x0FFF0FFF	0x0FFF0FFF	0xBADDCAFE	0xB0000000

Table 4: Memory map after widening data bus

Detecting this behaviour took a while, since it appears the the DataMover's read operations follow a similar skipping scheme so luckily as long as the address is the same for the read and write, the the data will be passed through without issue. We had lots of space in the DRAM for our two frames so we resorted to simply doubling the address steps we made through memory to compensate, to prevent "shingling" where a later write would overwrite a portion of data present beforehand.

Check if the DataMover is buffering data internally

Another one of the DataMover's oddities was that it held a small internal buffer, *and it was not large enough for an entire burst of data!* The recommendation for this is to probe the AXI bus out of the DataMover to the memory to see if this is present or not by comparing the data sent out of the DataMover to what was expected. Based on the findings adjust your system as needed.

In our project this manifested as an internal buffer of four of the 64-bit packets, while using burst sizes of eight on the S2MM, which was filled with data on reset. In our project this meant that the image had a few pixels from the right side of the image loop around to the start of the next row. Initially we thought this meant that one entire burst was off so we adjust the controller to counter this, but since it was only half a burst that was off there was now the leftmost portion of the frame rolling back to the right part of the display.

Eventually we discovered this and were unable to reliably counter this on the S2MM side with the controller, so we had the MM2S operations from the controller adjusted instead to deal with this shift in written data. The write operations start at the base address for

each frame region, but the reads for MM2S start at this offset from the base address and there is also a partial read of the other region's start to get the last few pixels that were carried across due to this buffer.

6.2 Using Standard Interfaces

In this project we made use of standard interfaces to link parts of our project together, notably the ASI-Stream protocol, and essentially the BRAM protocol for our original memory mapped operations. This meant that there were a lot of connections that ran from one IP block to another in parallel. To help tidy up the block diagram, it was a game-changer to use proper bus port definitions to flag each port that was part of a single interface, for example for our AXI stream ports we bundled them with the following structure:

```
// S2MM Ports
(*X_INTERFACE_INFO= "xilinx.com:interface:axis:1.0 S2MM_CMD_TDATA" *)
output [71:0] s2mmCommandData,
(*X_INTERFACE_INFO= "xilinx.com:interface:axis:1.0 S2MM_CMD_TREADY" *)
input s2mmCommandReady,
(*X_INTERFACE_INFO= "xilinx.com:interface:axis:1.0 S2MM_CMD_TVALID" *)
output s2mmCommandValid,
```

This meant the difference between the a clean and single connection as shown in Figure 14 to having to ensure each is connected properly as in Figure 15. This not only made for cleaner and easier to follow block diagrams, but also more reliable connections since the individual ports in the interfaces were automatically connected based on assigned purpose, so we couldn't accidentally misconnect them.

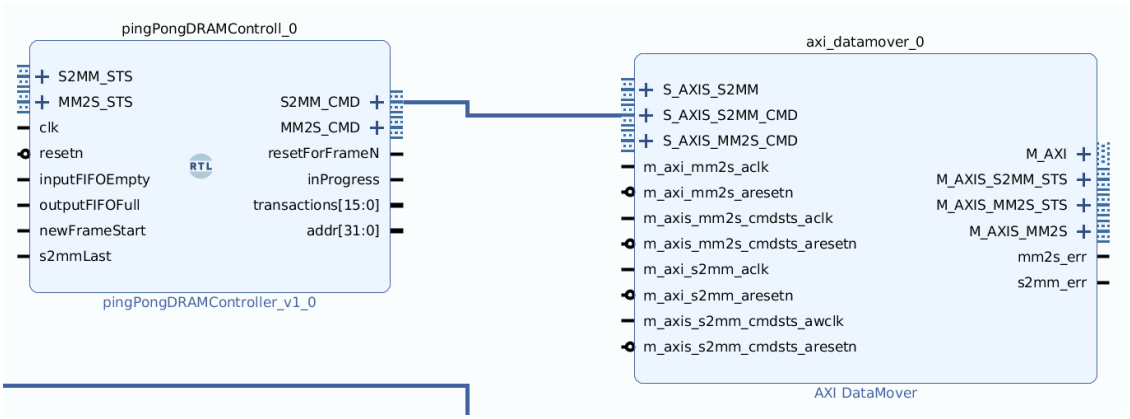

Figure 14: Bundled block interface ports

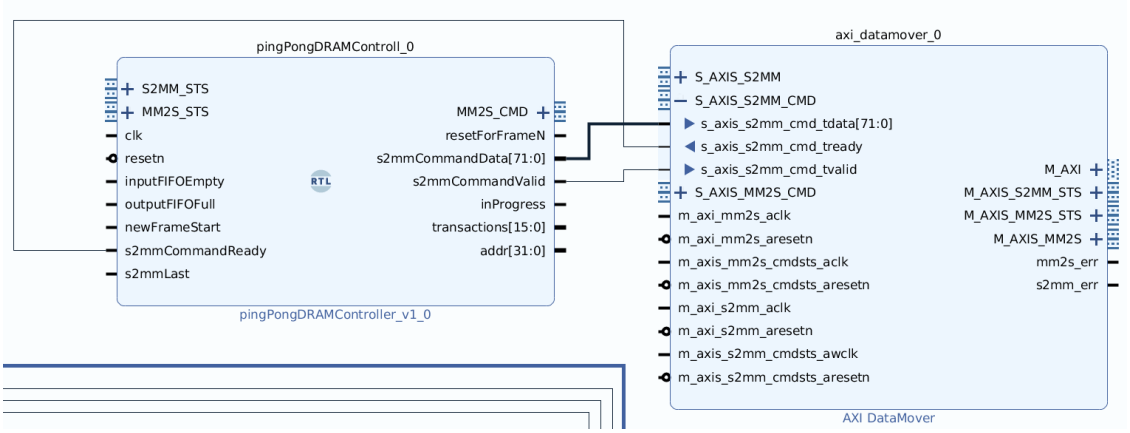

Figure 15: Unbundled block interface ports

6.3 Neural Network Weights

Keep a small parameter bit width

The bit width of the parameters greatly affects the computation and memory resources required in a neural network inference. Aside from simply the cost of storing the parameters in memory, the parameter bit-width affects the width of the elements that are used for computations at later stages in the network. For example, in our network with a 784-by-1-by-4-bit input layer, with 64-bit weights, the first layer of the network requires 4-bit times 64-bit multiplications. The output of this layer is the sum of 784 4-bit times 64-bit multiplications. Since 784 can be represented in a 10-bit value, this means that the output of the first layer must have a with of at least 64+4+10 bits to ensure no overflow. Table 5 shows the memory requirements at each stage of a forward-pass for our three-layer neural network, assuming w-width parameters.

	Input	**Layer 1**	**Layer 2**	**Output**
Size of Array	784×1	10×10	10×10	10×1
Required Bit Width	4	$w + 4 + 10$	$w + 4 + 10 + 4$	$w + 4 + 10 + 4 + 4$

Table 5: Neural net memory requirements

We found that using 64-bit or even 32-bit numbers to represent our network's parameters (weight and bias arrays) was prohibitively DSP-block intensive. The Nexys-DDR board only has 420 DSP blocks, each having a maximum input bit width of 28-bits. To achieve good parallelism, we needed to keep each multiplication in the input layer isolated to only one DSP block. To accomplish this we chose to represent parameters as 16-bit signed integers. This limits the range of allowable weights to -32,768 to +32,767.

Floats to Integers

We trained our neural network using a Python package that has functions to help with back-propagation based gradient descent. Most Python packages for training neural networks use parameters represented as floating point numbers, however multiplications and additions are much more easily done in hardware with signed integers. For this reason we cast our floating point weights to integers. Essentially, this "throws away" the information after the decimal point, so this technique only works if the weights are on a scale such that the information after the decimal point is not important.

We have now specified two constraints on our network parameters:

1. The parameters must be between -32,768 to +32,767 to fit in a 16-bit signed integer
2. The parameters must be coarsely spaced so that casting floats to integers does not result in a loss of accuracy

There are two tricks we discovered that can help with this:

Scale the training database Generally, making the training data more coarsely spaced by scaling by a factor greater than one will result in more coarsely spaced parameters. This can help with casting the parameters to integers.

Scale the parameters Since our entire network is linear, except for the RELU activation function which is close to being linear, scaling all the parameters by a constant factor does not affect the classification output. This can help bounding the weights to fit in a 16-bit signed integer.

References

[1] Westonb, "Westonb/ov7670-verilog: Verilog modules required to get the ov7670 camera working." [Online]. Available: https://github.com/westonb/OV7670-Verilog

[2] "Sobel filtering for image edge detection." [Online]. Available: https://onlinedocs.microchip.com/pr/GUID-37AD5EEE-6FAB-48FC-89F6-CAA649534B2A-en-US-1/index.html

[3] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[4] B. V. Dan Gisselquist, "Wb2axip: Wishbone to axi ip," Available at https://github.com/ZipCPU/wb2axip.