BMP Image Edge Detector

Phase 3 Proposal

Submission Date: 03/08/2018
Thursday 14:30 Lab (Lab Section 11)
TA: Reena Elangovan

Prepared by:
Ni Kang
Haoming Zhang
Haobo Chen
Graham Sninsky

1. Executive Summary

Edge detection is a basic problem for image processing and computer visualization, whose aim is to denote the points which change obviously in digital image. The obvious changes in image properties usually reflect the property's important events and changes, which includes: discontinuity in depth, discontinuity on surface, the changes in matter properties and the brightness change in a specific scenario. Image edge detection drastically decreases the number of data and eliminates the irrelevant information which keeps the important properties of the image structure. There are a lot of ways to be used in edge detection, and they can be divided into two parts. The first part is to find the maximum and minimum value in the first derivation of the image. The second part is to find the edge through find the second derivation of the image. This result in a large amount of arithmetic and repeated processing, which, makes an ASIC design a perfect fit for the purpose. In addition, having a SoC design produced specifically for them function described above will drastically diminish the workload for processing images.

The design will utilize AHB Lite Interface, which is capable for both reading and writing. The AHB Lite is required for communication between systems in the design. Since edge detection requires large scale computation, AHB Lite itself won't be able to store all necessary data. All data will be stored in registers in order for edge detection. The edge detector will also have two dedicated core, one for image filtering, the other is for edge detection.

The successful design of the proposed edge detector will require the following resources:

- AHB Lite Datasheet
- Reference Standard Cell Simulation Library for Mapped Design Verification
- Reference Standard Cell Technology Library for Final Design Layout Verification
- Verilog HDL Simulation and Design Synthesis Tool Chain

The following documents will describe:

- Super high level block diagram of edge detection
- Edge detector address mapping
- Design pinout
- Design architecture

2. Design Specifications

2.1. System Usage

2.1.1. System Usage Diagram

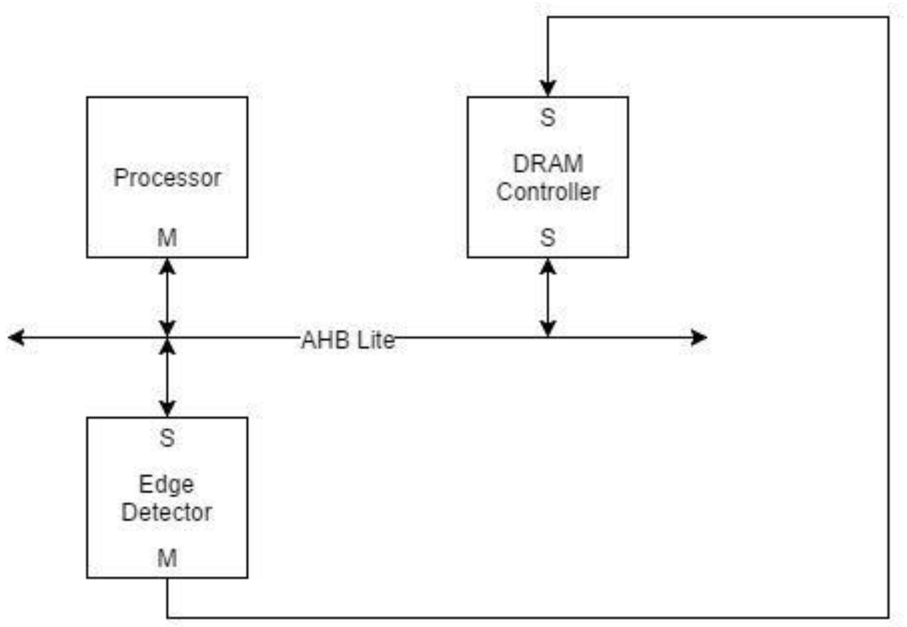

Figure 1: Example System Usage Diagram for Edge Detector

An example system illustrating the intended use of this edge detector is depicted above in Figure 1. In this system there is a main processor where any relevant software would be executed, DRAM where any intermediate values are stored in and edge detector for handling large scale calculation. The key operational ideas are that the software running on the processor would perform the following steps:

1. Configure the operational settings directly on the edge detector
3. Directly send the image data to the edge detector
4. Wait until edge detector's completion status bit is asserted
5. Output the calculated result to the data bus

2.1.2. Implemented Standard(s) and Algorithm(s)

Edge Detector

Slave whose input is controlled by the processor

Sublaminar master that controls read and writing data to the DRAM

AHB Lite Standard Slave

32-bit data bus

Read and Write Transfers

Burst Transfers Supported

Pipelined Transfers

3-bit, 8 word-address namespace (0x000 → 0x007)

2.1.3. Edge Detector Namespace Address Mapping

Table 1: Namespace Address Mapping Table

Slave Word	Address Read / Write	Data Size (Byte)	Description
0x000	R/W	4	Image width
0x001	R/W	4	Image height
0x002	R/W	4	Image header size
0x003	R/W	1	Config bit for completion of reading image width 1 -> complete 0 -> not complete
0x004	R/W	1	Config bit for completion of reading image height 1 -> complete 0 -> not complete
0x005	R/W	1	Config bit for completion of reading image header size 1 -> complete 0 -> not complete
0x006	R/W	1	Config bit indicating that the process is ready to start 1 -> ready 0 -> not ready
0x007	R	12	Input Image Data Starting Location

2.2. Design Pinout

Table 2: Miscellaneous Pinout Table

Signal Name	Direction	Number of bits	Description
vcc	PWR	/	Power Pin
gnd	GND	/	Ground Pin
clk	IN	1	System clock(100MHz)
n_rst	IN	1	Asynchronous Reset. (Active Low)

Table 3: AHB Lite Interface Pins

Signal Name	Direction	Number of bits	Description
image_data	IN	24	Single pixel image input
filter	IN	18	Sobel filter value
process_done	OUT	1	Set to high when process finished
image_out	OUT	24	Single pixel image output
data_ready	IN	1	Set to high when all configuration information is ready so that the data is ready to be processed

2.3. Operational Characteristics

2.3.1. Image Data Input

Before data processing, all the image data will be stored in a buffer in the data bus as a two-dimensional array. The configuration data of the image will be read into the module. The configuration flags won't be set until the preparation procedure is complete. After all the configuration flags are asserted, the data_ready flag is asserted. Only after then, the actual data processing can start. The procedure will be done in pipelined transfers such that the clock cycles needed for waiting is minimized.

2.3.2. Window Buffer Initialization

Each pixel is represented by 3 bytes. Since by default AHB Lite bus process a word (4 bytes) at a time, the efficient way to read data from the data bus is to read a chunk of 12 bytes (4 pixels) at a time so that there is no redundant byte. In order to minimum the number of data reading, it is necessary to construct a 8x8 window buffer so that after processing each pixel, only the replacement of single row or column is needed for further processing.

During the initialization, the window buffer will load the first 8x8 pixel data at the upper right corner of the image. The reason for loading in this amount of data is articulated below.

- For any given pixel in the image, it requires the information from 8 pixels around it for applying the kernels that will be explained in the next couple sections. Thus, to process a 3x3 pixel group, it requires data from a 5x5 set including their surrounding pixels. However, as mentioned above, the easiest way to read data from AHD Lite is to read 4 pixels at a time. Taking efficiency into account, it is reasonable to take 2 sets of 4 pixels for the initialization.

2.3.3. Window Buffer Iteration

For any further iterations, if the pixel of interest moves to the left or the right and doesn't exceed the current buffer window, the data don't need any update. When the pixel of interest and its surrounding pixels exceeds the side bounds of the buffer window. It is only necessary to shift the existing columns of values to the left by 4 bytes and load in a 4x8 set of data. If the pixel of interest moves downward, it behaves in a similar way as explained above.

Instead of starting at the beginning of each row, in order to minimize the amount of data updating, the pixel of interest will move in a zigzagging pattern such that only 32 pixels need to be updated each time except the initialization.

In the corner cases where the remaining of a row is not enough for 4 pixels or the remaining rows is less than 4, The data shift process will react accordingly.

2.3.4. Gaussian Blur

For each pixel of interest, Gaussian blur will apply the following kernel to it.

1	**1**	**1**
1	**1**	**1**
1	**1**	**1**

Figure 2. Gaussian Blur Kernel

In such a way, the image gets blurred and only the major edges can be detected by the Sobel filter. Thus, the quality of the edge detection can be improved.

The blurred data will be sent to another window buffer for further processing.

2.3.5. Window Buffer for Blurred Data

The Gaussian Blur module will produce a 6x6 matrix to the second window buffer, this window buffer operate in the same way as the previous one.

2.3.5. Sobel Filter Operation

With the window buffer produced from Gaussian blue. The Sobel filter module apply the following filter kernels on each pixel of the image.

-1	**0**	**1**

-1
0
1

Figure 3. Sobel Filter Kernel

By applying the filter kernel in both horizontally and vertically, we will be able to find the gradient relation between the pixel of interest and its surrounding pixels in both directions.

2.3.6. Gradient Computing Operation

After a group of gradients is found, the Pythagorean theorem is applied to the two gradient vectors. Since it is possible to have a negative gradient, it is necessary to take the absolute values of them.

$$Edge = \sqrt{\Delta x^2 + \Delta y^2}$$

If the edge value is larger than a threshold of our choice, it is recognized as a valid edge and will output FF (i.e. white pixel). Otherwise it will output 00 (i.e. black pixel).
By performing the operation above, an image with only major edges will be generated.

2.4. Supported AHB-Lite SoC Bus Transactions

2.5. Requirements for Design

The current cell phone camera provides an image quality of about 8 megapixels. With given clock speed restriction, it is expected that the edge detection would finish below one second. With a clock speed of at least 100 MHz, it is a reasonable aim to complete the image processing in less than 1 second. This requires pipeline transfer of data. Meaning that all tasks are done parallelly so that ideally no clock cycles are wasted for waiting for new data coming in. The pipelining will be handled by state machines to the greatest extend. The most method to buffer 8x8 data matrix is selected so that the less data reading is performed such that no spaces are wasted. Moreover, the zigzagging pattern that the pixel of interest moves forward will also help to reduce the number of data reading.

In terms of area, since the image processing requires a large amount of registers to store intermediate values. Given the specification above it is estimated that we will use at least 2000 registers.

When implementing the edge detector, the choice of threshold number is critical. The actual choice will be tested during the experiment. Applying the different thresholds of the design will affect the final output rate. If a large threshold is chosen, the output will be vague (the edges of the picture will not be shown clearly, there might be missing parts if the color gradient is not large enough). Otherwise, if the threshold is low, the output will be messy (the edge will include redundant edges, which will not perfectly outline the image). Thus, we have to find the perfect threshold of the bmp edge detector through different tests, which is also the first requirement of our design. Secondly, knowing the size of the output image is also critical. We have a one thousand samples flag to measure how many bytes were passed into and processed by the system. This flag will not only be used in the purpose of indicating how many bytes are passed in but will also be implemented for debugging purpose. This is the second main requirement for our design.

3. Design Implementation

3.1. Design Architecture

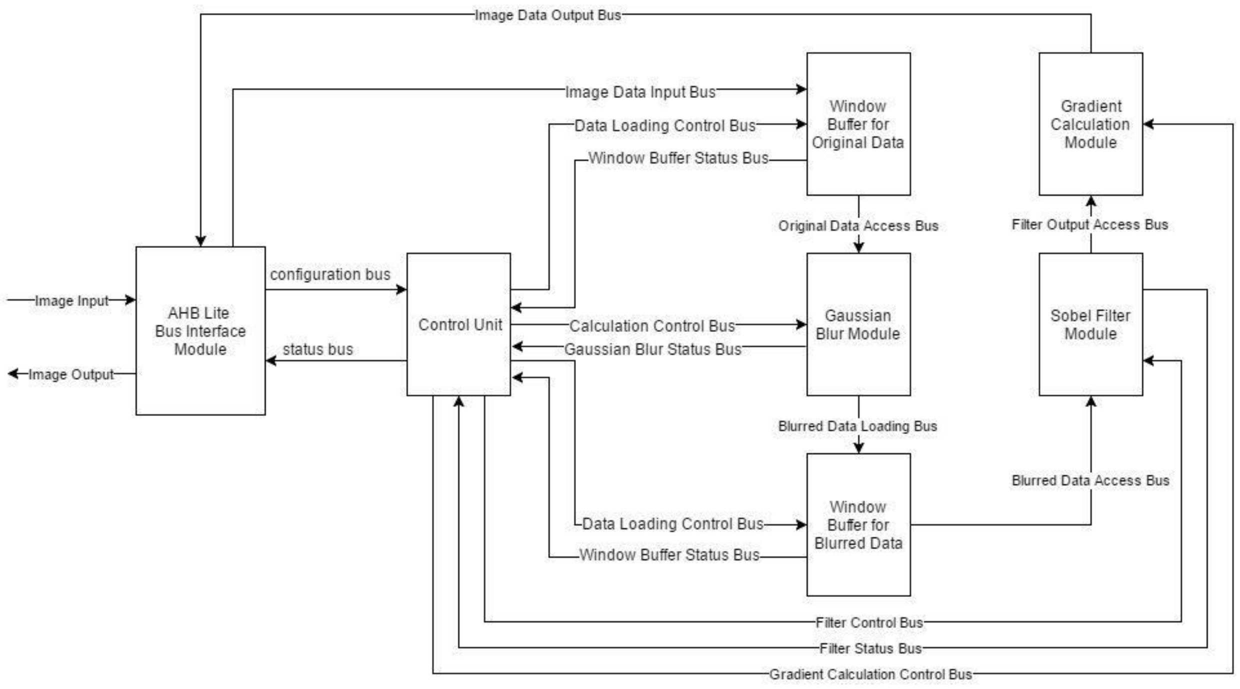

Figure 4: Edge Detection Architecture Diagram

The intended implementation architecture is depicted above in Figure 2. A module handling the implementation of the AHB Lite interface is used to bridge between the external SoC environment and the internal data and status locations. The control unit will be used to control the pipeline transfer of the design. There will be two window buffers for storing the intermediate processing data, one is for storing the original data, the other is for storing the preprocessed data coming from the Gaussian Blur module. The Sobel Filter and Gradient module will serve all the core functions for the edge detector. There are no other storage devices except for the intermediate processed data. The algorithm will be controlled by a Finite-State-Machine with state counters for tracking the progression through the needed stages and handle the data transfer between modules with pipeline transfer feature.

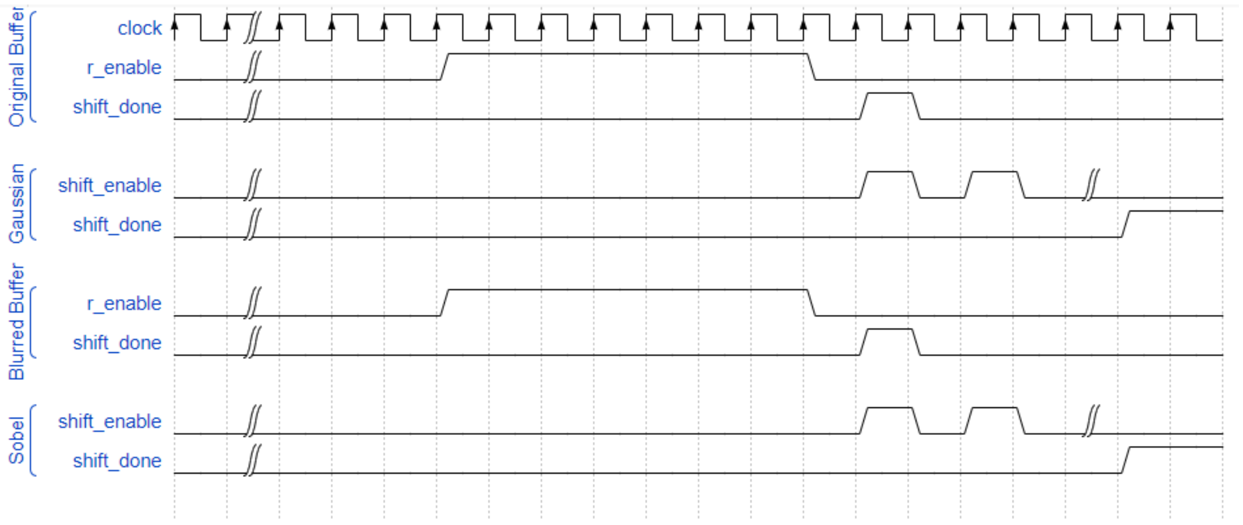

Figure 5: Edge Detection Architecture Timing Diagram

The timing diagram shows the internal relationship between each blocks. The read enable signal on the original buffer going high indicates that the buffer is loading. While the read enable goes to 0, it means that shifting is done and the shift_done flag gives a strobe to the next block, which is Gaussian block, to make it start doing the corresponding work. While the Gaussian block finishes the shifting and calculation, it sets the shift_done signal to high and stores the value into the blurred buffer. When all data from Gaussian block was transmitted, the shift_done in the blurred buffer sets to high to make the sobel block start working. Finally, when the sobel block finishes working, it shifts out the final output to the Gradient block, which is a purely combinational block.

3.2 Functional Block Diagram

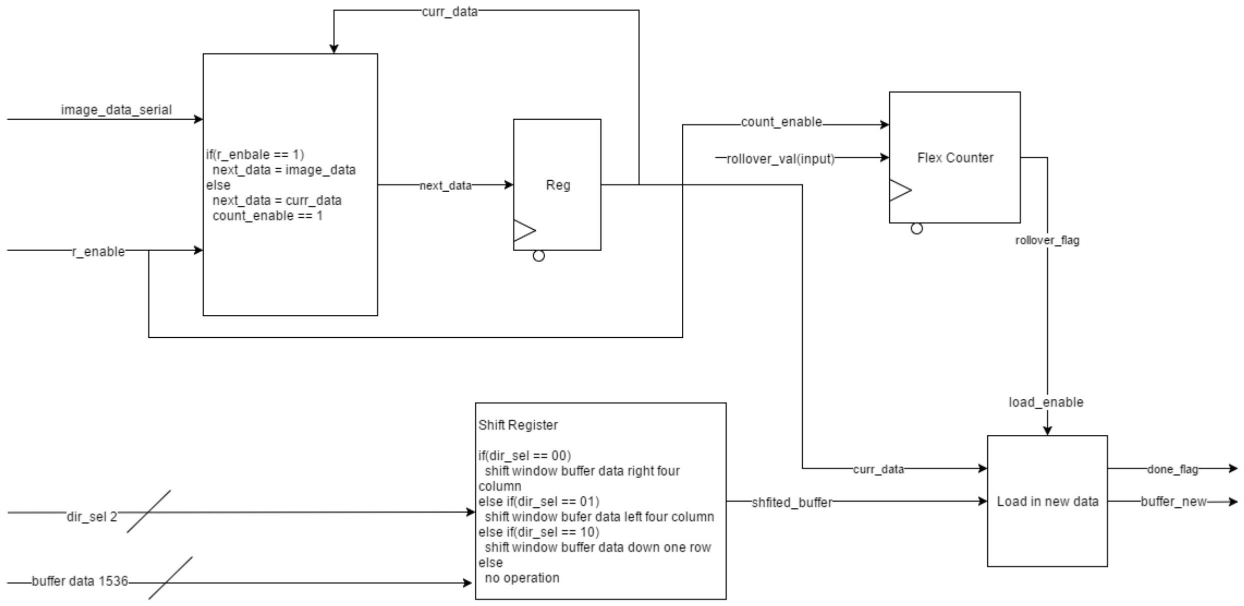

Figure 6: Window Buffer Module Block Diagram

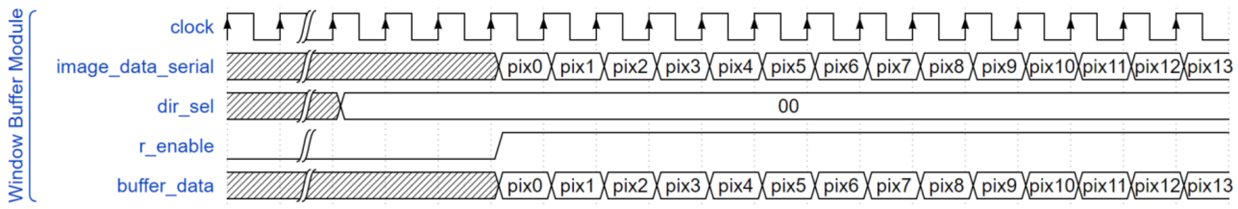

Figure 7: Window Buffer Module Timing Diagram

The window buffer will be utilized to store the original and blurred data. But the implementation is identical. First the dir_sel flag will be used to decide the behavior of the shifting operation. If dir_sel is 11, it means that this is the first time the module gets executed. Otherwise, it indicate the direction of shifting operation. If r_enable is asserted, the image data will be read in serially and be stored in the empty parts of the shifted buffer. A combination of new data and old data will form the next buffer_data to be calculated. The done_flag is asserted only when the internal counter reaches it’s rollover flag.

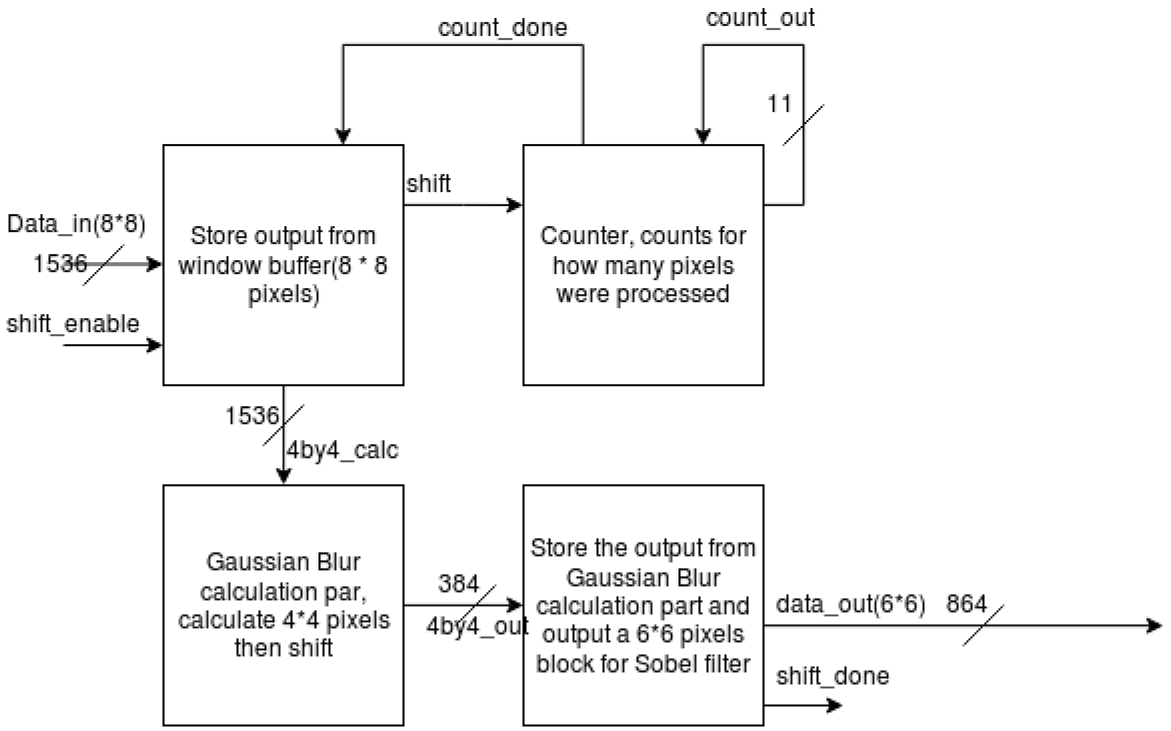
count_done
count_out
11
shift
Data_in(8*8)
1536
shift_enable
Store output from window buffer(8 * 8 pixels)
Counter, counts for how many pixels were processed
1536
4by4_calc
Gaussian Blur calculation par, calculate 4*4 pixels then shift
384
4by4_out
Store the output from Gaussian Blur calculation part and output a 6*6 pixels block for Sobel filter
data_out(6*6)
864
shift_done

Figure 8: Gaussian Filter Module Block Diagram

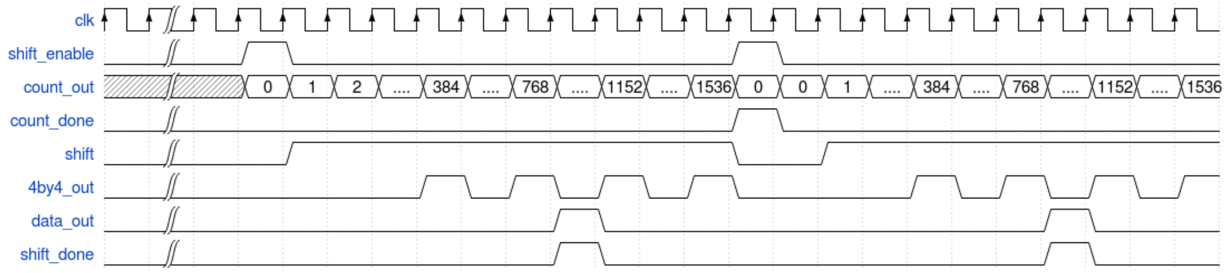

Figure 9: Gaussian Filter Module Timing Diagram

The Gaussian filter module will take in the data(8*8*3) output from window buffer module and execute the value. It will first go into a counter that counts how many data were shifted in, and when there are 1538 bit data were successfully shift in, the count_done flag will set. In the meanwhile, the data are shifted into a block that takes in 4 * 4 data (384 bit) to process, when the 4 * 4 data finish transition, the 4by4_out flag will set. Then shift to registers that save the final data. When the 6 * 6 final data are finally processed, the data_out flag will set and the shift_done will also set to tell the next block the data is ready.

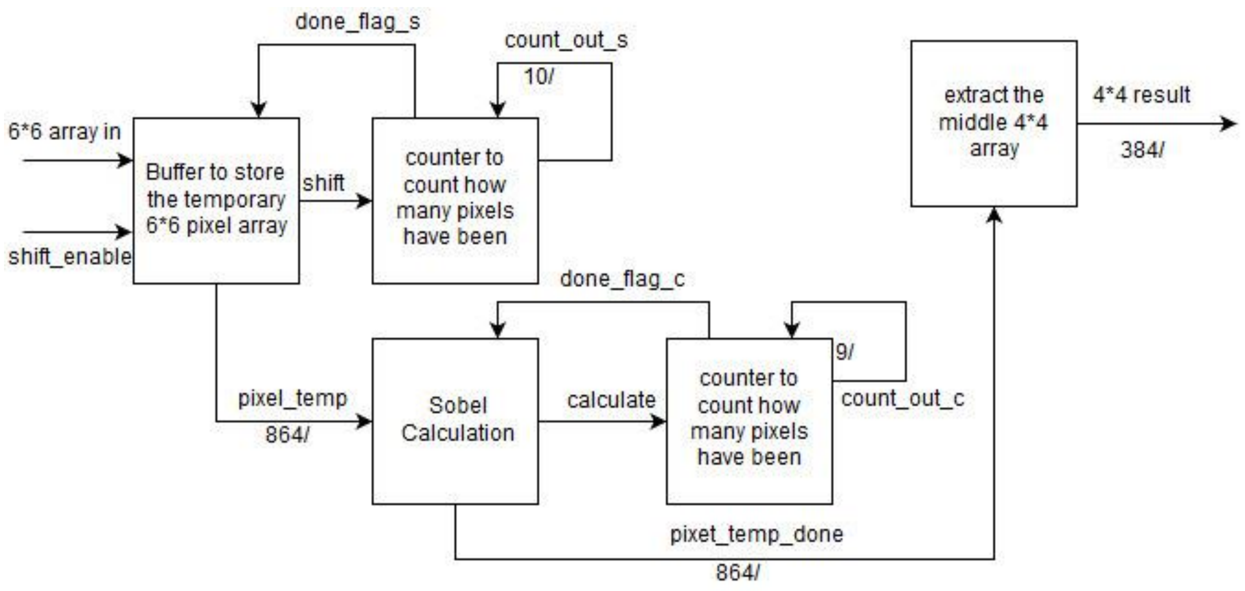
done_flag_s
count_out_s
10/
6*6 array in
shift_enable
Buffer to store the temporary 6*6 pixel array
shift
counter to count how many pixels have been
extract the middle 4*4 array
4*4 result
384/
done_flag_c
9/
count_out_c
pixel_temp
864/
Sobel Calculation
calculate
counter to count how many pixels have been
pixet_temp_done
864/

Figure 10: Sobel Filter Module Block Diagram

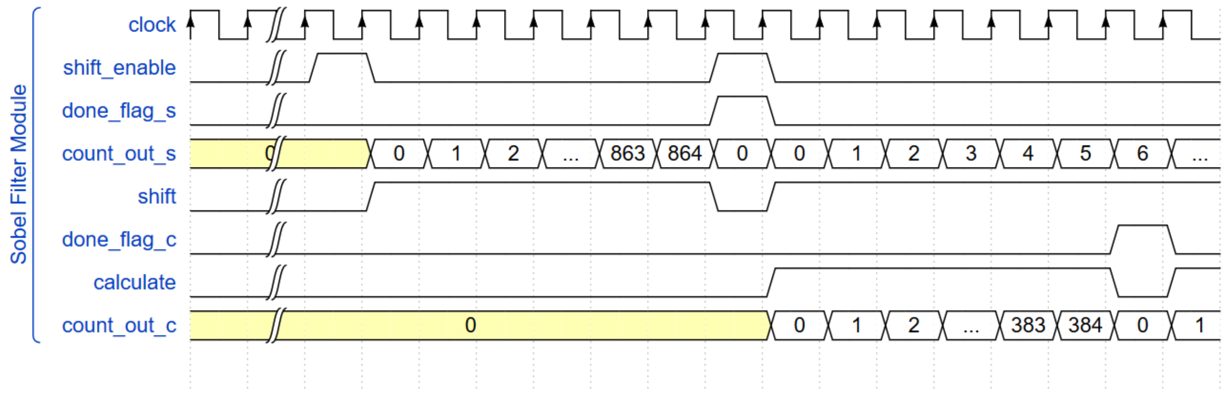

Figure 11: Sobel Filter Module Timing Diagram

The shift enable is an input signal from the other module. Once the shift_enable signal is strobed, the shifting-in counter starts to count. There are two counters used in the Sobel Filter Module. The first one is for the pixel array shifting-in, and the second one is for counting how many pixel has been calculated by the Sobel algorithm. For the shifting-in one, only one data bus is used, so that we need a counter to follow if the whole 6*6 pixel array has been shifted in. Once the all 864 bit data has been shifted in, the done_flag_s would change to 0 to indicate that the data has been shifted in in a whole. For the second counter, which is for the calculation part. Only 4*4 pixel in the middle of the 6*6 one needs to be calculated, and there’s a counter used to follow the number of bits being calculated. Once all of the bits have been calculated, the done_flag_c flag will be changed to 1 and the final result will be shifted to the next block to perform output.

4.1. Project Timeline and Division of Tasks

- Week of Mar. 19: Write Verilog code for the window buffer parts and write test bench
  a. Haoming Zhang & Haobo Chen: Write verilog code for the window buffer
  b. Ni Kang & Graham Sninsky: Write test bench for the window buffer
- Week of Mar. 26: Write Verilog code for the Gaussian Blur part and write test bench
  a. Ni Kang & Haobo Chen: Write verilog code for the Gaussian Blur part
  b. Haoming Zhang & Graham Sninsky: Write test bench for the Gaussian Blur part
- Week of Apr. 2: Write Verilog code for Sobel Filter and Gradient parts and write test bench
  a. Ni Kang: Write verilog code for the Sobel Filter
  b. Haoming Zhang: Write verilog code for the Gradient part
  c. Haobo Chen: Write test bench for the Sobel Filter
  d. Graham Sninsky: Write test bench for the Gradient part
- Week of Apr. 9: Put parts of code together and write test bench for the edge detector
  a. Ni Kang: Put parts of code together
  b. Haobo Chen & Haoming Zhang & Graham Sninsky: Write test bench for the Gaussian Blur part
- Week of Apr. 16: Debug the code and write the report
  a. Haoming Zhang & Haobo Chen: Debug the code
  b. Ni Kang & Graham Sninsky: Write the report
- Week of Apr. 23: Prepare for the presentation

4.2. Success Criteria

4.2.1 Fixed Criteria: (Total of 12 SC Points)

1. (2 points) Test benches exist for all top-level components and the entire design. The test benches for the entire design can be demonstrated or documented to cover all of the functional requirements given in the design specific success criteria.

2. (4 points) Entire design synthesizes completely, without any inferred latches, timing arcs, and, sensitivity list warnings.

3. (2 points) Source and mapped version of the complete design behave the same for all test cases. The mapped version simulates without timing errors except at time zero.

4. (2 points) A complete IC layout is produced that passes all geometry and connectivity checks.

5. (2 points) The entire design complies with targets for area, pin count, throughput (if applicable), and clock rate. The final targets for these parameters will be determined by course staff based on your design review. Failure to reach any of the targets will result a score of 1 out of 2 provided that you are within 50% on area, 10% on pin count, and 25% on throughput. Doing worse in any category will result in a score of 0 out of 2.

4.2.2 Design Specific Success Criteria: (Total of 8 SC Points)

1.(2 points) Demonstrate by simulation of a Verilog test bench that window buffer block works.

2.(1 points) Demonstrate by simulation of a Verilog test bench that gaussian block works.

3. (2 points) Demonstrate by simulation of a Verilog test bench that sobel filter works.

4. (2 points) Demonstrate by simulation of a Verilog test bench that gradient calculator works.

4. (1 points) Demonstrate by simulation of a Verilog test bench that the overall function works.

Reference Cited:

"AMBA® 3 AHB-Lite Protocol," 2006. [Online]. Available: http://mazsola.iit.uni-miskolc.hu/~drdani/docs_arm/IHI0033A_AMBA3_AHB_Lite.pdf. [Accessed: 24-Feb-2018].