**Final Year Project (2023 – 2024)**

**ELEC/CPEG Progress Report**

**Hardware Acceleration of Data Processing**

**Project ID: ZW01a-23**

**Supervisor: Professor Wei Zhang**

**Author (Student ID): CHIO Yat Hei (20765204), LUK Pak Him (20766571), SHUM Kwan Ho (20767898)**

**Date: 5 January 2023**

**Main Objective**

This project aims to design and implement an image processing workflow with color to grayscale, edge detection, noise reduction and black-and-white masking algorithms, which is usually applied for pre-processing of image processing algorithms on an FPGA board to enable hardware acceleration of the workflow.

**Objective Statements**

1. To design and implement an optimized data flow and pipeline stages for median filter, verify the correctness of it.

2. To implement a data cache system and consolidate it with the implemented sobel filter system.

3. To design and implement an optimized sobel filter for the edge detection part for the whole system, and verify the correctness of it, and integrate into the system.

4. To design and implement Otsu’s method that can be updated concurrently and compute final weighting by combining concurrent parts and integrate into the system.

5. To implement color to grayscale and integrate into the system.

6. To design and implement interface for image input and output and integrate into the system.

# **Table of contents**

[**Table of contents** 2](#_Toc155731942)

[**SECTION 1**—**INTRODUCTION** 4](#_Toc155731943)

[**1.1** **Background and Engineering Problem** 4](#_Toc155731944)

[**1.2** **Objectives** 5](#_Toc155731945)

[**1.3** **Literature Review of Existing Solutions** 6](#_Toc155731946)

[**SECTION 2**—**METHODOLOGY** 8](#_Toc155731947)

[**2.1 Overview** 8](#_Toc155731948)

[**2.1.1 System Description** 8](#_Toc155731949)

[**2.1.2 System Block Diagram** 11](#_Toc155731950)

[**2.1.3 Components List** 11](#_Toc155731951)

[**2.1.4 ECE Knowledge** 11](#_Toc155731952)

[**2.2**. **Objective Statement Execution** 12](#_Toc155731953)

[**2.2.1 Median filter for noise reduction** 12](#_Toc155731954)

[**2.2.2 Data cache for fetching and storing** 25](#_Toc155731955)

[**2.2.3 Sobel filter for edge detection** 26](#_Toc155731956)

[**2.2.4 Otsu’s method for grayscale to black-and-white** 36](#_Toc155731957)

[**2.2.5 Color to grayscale** 36](#_Toc155731958)

[**2.2.6 Input and Output** 37](#_Toc155731959)

[**SECTION 3— Project Planning** 38](#_Toc155731960)

[**3.1 Project Schedule** 38](#_Toc155731961)

[**3.2 Budget** 43](#_Toc155731962)

[**References** 44](#_Toc155731963)

[**APPENDICES** 46](#_Toc155731964)

[**Appendix A — Meeting Minutes** 46](#_Toc155731965)

[**Appendix B — Group Members’ Contributions** 52](#_Toc155731966)

[**Appendix C – Deviation(s) from the proposal and supporting reason(s)** 53](#_Toc155731967)

[**Appendix D – Source code** 54](#_Toc155731968)

**SECTION 1**—**INTRODUCTION**

## **1.1 Background and Engineering Problem**

Nowadays, High performance computing has become an important fast-growing branch in the computation industries, as more and more demand of computation came from the widespread growth of Artificial Intelligence (AI), data analytics and mining, computational simulations such as computational fluid dynamics (CFD), etc. While Central Processing Units (CPUs) have been the backbone of computation over the past decades, the upsurge of demand in computational power meant that these general-purpose processors could not simply keep up with the required computational power. Numerous processor architectures are being developed, where various ASICs are being developed targeting specific applications, such as GPUs used to accelerate 3D-graphics rendering, as well as dedicated proprietary circuits for modems. However, ASICs are expensive to develop, which could only be justified by ordering and producing a large amount of chips [1].

In order to provide a middle point between proprietary ASIC and general-purpose CPUs, Field Programmable Gate Array (FPGA) is being developed by Xilinx to provide programmable IC and circuits to suit the need for low-volume customized ICs. FPGA chips are based on matrices of programmable computational-logic-blocks (CLBs) placed across the chip, which are being connected by customizable interconnects. Currently, FPGA is being used in various applications, such as automobile ECUs, Data Centre switches, and even plane and rocket guidance and control systems. With the increasing demand in high-performance computing and time-critical high-speed computation, FPGA has seen a rise in popularity, being deployed in numerous applications, including cloud datacenters and AI training [2].

In this project, we are going to use FPGA to its advantage, and apply it for the use of acceleration in image processing algorithms. FPGA provides a platform for dedicated circuit designs and allows concurrent computation and operation with its various CLBs, which also enables a flexible placement of logic blocks and circuits, making it suitable for a wide variety of applications. Image processing methods such as edge detection and noise reduction are highly parallelizable, and being able to be split into pipelining stages, which is why we chose to implement these methods with optimized and efficient algorithms on FPGA boards. Our project is aiming to implement an image processing workflow from a colored image to a noise-reduced black-and-white image on an FPGA board, which is quite a common pre-processing technique used for machine learning and image recognition applications.

We believe that using FPGA for image processing algorithms hints new possibilities for future computational applications, on server-grade processing, personal computation and embedded systems. Currently, programmable logic device manufacturers such as Altera and Xilinx, are producing SOCs that combine CPU and other functional blocks with programmable circuits, which could possibly enhance performance of heterogeneous computing by configuring the logic and circuit according to the computational need. It also provides a low-power solution for parallel processing in embedded applications, such as IOT devices that require a high degree of image processing.

## **1.2 Objectives**

This project aims to design and implement an image processing workflow with color to grayscale, edge detection, noise reduction and black-and-white masking algorithms, which is usually applied for pre-processing of image processing algorithms on an FPGA board to enable hardware acceleration of the workflow.

**1.2.1 Objective Statements**

Objective 1: To design and implement an optimized data flow and pipeline stages for median filter, verify the correctness of it.

Objective 2: To implement a data cache system and consolidate it with the implemented Sobel filter system.

Objective 3: To design and implement an optimized Sobel filter for the edge detection part for the whole system, and verify the correctness of it, and integrate into the system.

Objective 4: To design and implement Otsu’s method that can be updated concurrently and compute final weighting by combining concurrent parts and integrate into the system.

Objective 5: To implement color to grayscale and integrate into the system.

Objective 6: To design and implement interface for image input and output and integrate into the system.

## **1.3 Literature Review of Existing Solutions**

Because of the increase in demand in High Performance Computing, design approaches such as heterogeneous systems and specialized architectures are being explored with the aim of providing high-performance, efficient and flexible processors to handle tasks in the current ever-changing landscape of computing. The techniques of Graphics Processor Unit (GPU), Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) would be reviewed below.

**Graphics Processor Unit (GPU)**

GPUs are processors with a vector architecture that could handle and process multiple data at once, which are being developed originally for 3D graphics processing [3]. While traditional CPU cores can only process one data at once, GPUs utilize a Single Instruction Multiple Data (SIMD) architecture to perform operations in a vector containing multiple data at once, thus suitable for data-heavy operations. In order to utilize GPUs for more generic computing purposes other than 3D graphics, GPU manufacturer Nvidia has developed a dedicated Application Programming Interface (API) named CUDA, where programmers could develop GPU programs for HPC tasks [4].

The GPU is a versatile platform for high-performance computing applications, allowing programmers to easily implement their algorithms using CUDA and vector processors to accelerate data-heavy high-performance algorithms. However, GPUs have lower performance efficiency compared to specifically-designed accelerators. GPUs were originally designed to run graphics applications and are optimized for high-precision calculations, so there are portions of redundant hardware unused for some low-precision operations, reducing the efficiency of the GPU as an accelerator. Performance of GPU in HPC applications is also being affected due to the memory hierarchy of GPU chips, where data is being stored and moved frequently between the computation core and memory, leading to a prolonged memory access overhang that further affects the efficiency and performance of GPUs in HPC applications.

**Application Specific Integrated Circuit (ASIC)**

ASICs are chips that are specifically designed and produced to perform a dedicated task, where data is being processed in a custom hardware [5]. The production of ASIC contains multiple steps, where the proof of correctness of an algorithm is required before proceeding to the design and placement stage. After the design is finalized, a mold would be produced to fit in a special tooling called a lithographic machine to produce ASIC chips.

ASICs provide the highest efficiency and performance due to its nature of on-demand design, as dedicated hardware is being produced just to perform a specified task. However, in terms of cost and flexibility, ASICs have a disadvantage compared to other kinds of devices. In this fast-changing world, computational algorithms change from time to time when novel technologies or theories are being proposed. If the algorithm is being deployed on an ASIC chip, it is very hard to adapt the existing ASIC design to the new algorithm, where a new design from the ground up is required. Because of the lengthy and expensive process of redesigning the chip and molding tools, ASICs are generally less flexible compared to other kinds of chips for HPC applications and are also only cost effective if produced in large numbers because of the complicated one-time cost of development and production.

**Field Programmable Gate Array (FPGA)**

FPGA is a special kind of chip invented by Xilinx with the intention of being highly flexible, able to be programmed by users to adapt to various tasks. Inside the chip, there are a lot of Computational Logic Blocks (CLBs) that could be configured to perform various digital functions, as well as Programmable Interconnects which can be programmed to connect different CLBs together and form complicated logic designs [6].

FPGAs are designed specifically to suit the current computing environment, where new algorithms and technological trends appear from time to time, requiring modifications on existing hardware to cope with these new requirements. FPGAs provide an excellent platform for computer engineers to design and test their ideas quickly by simply reconfiguring the circuit design with a computer, even if the chip is being deployed in data centers or user devices. While FPGA and GPU both provide the same level of flexibility, FPGA have an advantage over the latter in terms of efficiency and performance. While they might have similar performance characteristics in some optimal applications, FPGA tend to have a lower power consumption compared to GPU, as they are being configured specifically for the desired HPC application, utilizing all of its CLB’s to implement the computation logic. On the contrary, it might not be possible to run programs optimally on GPUs, which would result in additional power consumption by idle parts of the chips.

In the age where HPC has been more and more important in the field of computing, a lot of technologies have been developed to accelerate and better suit these computationally heavy tasks, including GPU, ASIC and FPGA. While all of them can accelerate data processing algorithms, we believe that FPGA suits our project the most because of its advantages of high flexibility, performance and efficiency against its counterparts. Thus, we have chosen to implement our data processing algorithm on FPGA chips.

# **SECTION 2**—**METHODOLOGY**

## **2.1 Overview**

### **2.1.1 System Description**

The proposed system aims to provide an accelerated method on doing image processing by using FPGA designs. The image processing methods that are going to be implemented include color to grayscale, edge detection, noise reduction and grayscale thresholding. These image processing methods are quite important in current real-life examples, including the preprocessing of images which would be used for image recognition and machine learning.

In a basic image preprocessing workflow, a colored image is first being turned into grayscale, as in many cases the color of the object being recognized does not matter. The resulting image would be transformed from a 3D matrix to a 2D matrix, allowing easier operations afterwards. The color to grayscale is usually done by taking the average of the three colors (RGB) with different weights. Then the image is being passed to an edge detection filter, by detecting the discontinuities of an image and identifying the edges, which help extract the overall shape of an object. The processing is usually done by convoluting a filter onto the image, where there are various filters, including Canny filter, Sobel filter, Prewitt filter etc. As the image after edge detection might contain unwanted noise signals due to the nature of the filters, a noise-reduction step is to be carried out, by methods of spatial filtering such as a median filter. A grayscale threshold is applied so that the image contains only the black and white outlines of the image. The threshold can be determined by a fixed threshold, or by global methods such as the Otsu’s method. Additional processing between different processes can be added if needed, for instance, if the input image is too noisy, an additional noise-reduction filter can be applied before edge detection to give a better result.

The whole system is being divided into two parts, the input and output connections, as well as the processing part. In order to provide high-performance and efficient hardware, the system is being designed as a spatial dataflow architecture, where data is being processed in one module and then being passed to other modules before saving into the main memory [7]. Traditional processor systems use Von Neumann architecture, where operations are being performed by fetching values from main memory, and being written back after the operation is completed [8]. While this architecture provides higher flexibility for users to program, the frequent access of memory meant that a lot of energy and time is being wasted on transmission of data. As FPGA allows us to design and implement circuits with high flexibility, spatial dataflow architecture is being chosen to be implemented for our system.

A black background with white squares

Description automatically generated

Figure 1. Illustration of spatial dataflow architecture

In our system, the image is being processed row by row, where in each clock cycle, one row of pixel is being fetched into the system for processing, while one row of processed image is being output to the memory. The whole row of pixels would be processed simultaneously in multiple computational units. Each computational unit contains arithmetic operators, as well as some registers for pipelined operations as well as temporary memory for local operations. Implementing the accelerator on FPGA chips means that we can easily enlarge or shrink the scale of the processor, where the number of computational units are flexible. Figure 1 also shows how the number of computational units can scaled accordingly and connect together, where middle modules can be increased and increased accordingly.

The whole system is being divided into a few functional blocks: the input memory block, the logic blocks for processing, as well as the output memory block. For some of the operations, there is also intermediate memory for temporary storing. The Input memory block stores the original image for processing. The processing procedure would possibly be in a pipelined style, where the image is being processed from one step to another. Figure 2 shows the data flow diagram of the envisioned completed system.

For color to grayscale functional block, as it is a point operation, each pixel is calculated separately and stored in the intermediate cache memory. To speed up the processing time, the processing is also being split up into multiple stages, where the weighted values of the RGB colors are calculated separately, then being added up. If possible, multiple pixels can be processed concurrently, subjected to the number of functional blocks available.

After passing the grayscale functional block, the processed pixel is stored to an intermediate memory block, waiting to be fetched to the edge detection block. For the edge detection block, we opt for using the Sobel filter, as it is easier to be implemented on an FPGA board. There are some optimization methods, such as saving intermediate calculation results back to the memory and reducing the number of arithmetic, suitable only for boards with fast memory access speed. Pipelining could also be implemented, as the final pixel of a convolution is calculated by summing up all the weighted neighboring pixels, benefiting from the characteristic of FPGA.

For the noise reduction block, we opt to use the median filter, where we take in nine adjacent pixels and compute the median of those nine pixels and update the value of the middle pixel. Median is quite a good representation of the average of nine pixels, as it won’t be affected by outliers, while providing a good estimation of the average on values with a wide range. There are multiple different ways to compute the median of nine values, by further breaking down the values into smaller groups and sort them first. The method that is used will be further explained below. Pipelining could also be implemented to increase the throughput of the accelerator, as it breaks down lengthy processes into smaller stages that could handle more pixels.

After the whole image is being noise reduced and stored in the memory block, a global operation of Otsu’s method is being implemented to determine the threshold of the image, in order to turn it from a grayscale image to black and white only, thus revealing the edges. The resultant image is stored and outputted via an I/O module provided on the FPGA board.

The project will be designed through Vivado, an IDE specifically designed for writing Verilog code for Xilinx FPGA product, where software simulation is also done through the IDE. The verification of functionality is being done with the Simulation Behavioral Model, where test data could be coded to run through the simulated circuit and see if the results correspond to the target output. The Post-implementation Simulation tool also allows resource usage and timing measurements being made, thus allowing the evaluation of different design approaches.

During the project, different design approaches such as those with different optimization and pipelining methods, as well as different resource allocations for memory block or processing unit, are being evaluated. They will be implemented on the same FPGA board to provide a level playing ground for resource limitation, while the timing simulation measured would reflect the performance of each design, with the least time spent the better.

Real-life tests of the performance and efficiency will also be made after all the design is being completed and verified, which focuses on measuring both the time of computation of a batch of test data to determine the performance of the implemented design, as well as measuring the power drawn to determine the efficiency of the design. If possible, the performance is being compared against normal CPU operations with both single-thread/multi-thread performance, as well as against other vector-machine architectures.

### **2.1.2 System Block Diagram**

Figure 2. System Diagram of the completed image processing system

### **2.1.3 Components List**

Table 1. List of Specification

| **Item\*** | Specifications/Model |
| --- | --- |
| **Artrex ultrascale** |  |

### **2.1.4 ECE Knowledge**

Knowledge from various ECE courses could be applied to the project, especially those about circuit and logic designs.

**ELEC2350**, as an introductory course for computer organization and architecture, outlines how modern processors are being designed, in order to maximize performance and efficiency [9]. Nowadays, most processors are being designed with pipelined stages, where a process is being broken down into smaller stages and functional blocks so that the hardware can be used more efficiently with less idle time, and thus improving performance and efficiency. In this project, each functional block or process is also being broken up into pipeline stages, which improves processing performance for the high-performance computing applications.

In courses **ELEC3310** and **ELEC4320**, the knowledge of FPGA and Verilog is being introduced in the course [10] [11], which are the tools being used to implement the design of our project on. ELEC3310 also focuses on digital-circuit and logics, including combinational and sequential logic gates, as well as finite state machines, counters and registers. While the course ELEC4320 focuses on the design principles and considerations of an FPGA system, including hardware placement, resources available and memory management, which aids on the design and consideration of the implementation of our system.

The course **ELEC4130** introduces the methods of digital image processing, how digital images are being formed by pixels, and the view of them in both spatial and frequency domain [12]. The course introduces how and why filters are being designed in the way they are, and how to perform convolution on images with kernels. The sobel and median filter introduced in the course serve as the algorithm applied in the project for the edge detection and noise reduction parts of the whole system.

## **2.2**. **Objective Statement Execution**

**2.2.1 Median filter for noise reduction**

To design and implement an optimized data flow and pipeline stages for median filter, verify the correctness of it.

In this objective, we designed and implemented a median filter circuit design that takes in nine adjacent pixels and outputs their median. The aim is to provide a design that can be run highly concurrently, utilizing the characteristics of FPGA chips, while being scalable and able to adapt to images of various sizes.

Sort and take the median of 9 numbers

Figure 3 shows how a median filter works. Pixel values of adjacent nine pixels are being drawn, the median is being computed and applied as the value of the pixel of the centermost position.

Figure 3. An illustration of median filter

In the project, we have undergone two different approaches for the median filter design, one taking in all values in one clock cycle, while the other takes three values in one clock cycle only, requiring a total of three clock cycles to gather all necessary pixel values. A staged and non-staged design for sorting of three pixels is also being considered.

In the end, we chose to implement a pipelined staged sequential median filter, which takes in three new pixels and outputs one median of the values of the previous three clock cycles every clock cycle.

#### **Task 1:**

**Aim:** To design a circuit to compute the median of nine input values each with 8-bit depth

**Expected outcome:** Finish a schematic of an efficient and flexible median filter module

**Member in charge:** All group members

**Work Description:**

*Concurrent and pipelined approach of median filter for nine values*

Traditional single-threaded processor is most efficient by doing a merge sort, where the median is taken easily from the sorted sequence. However, with the case of implementing on an FPGA board, a better approach can be taken, by utilizing the parallelizable characteristics of FPGA. The paper by Bevara and Sanki [13] suggests an algorithm of concurrent median filter processing, by splitting the 9-pixel local area into 3 groups, computing their own medians respectively and taking the median of those three local medians to find the desired value. This approach not only speeds up the process by utilizing the concurrent processing ability of FPGA circuits, but also by possibly reducing the number of operations if the previous result is being stored and reused for the next pixel. The algorithm is illustrated in Figure 4.

A screenshot of a computer

Description automatically generated

Figure 4. Median Filter algorithm by Bevara and Sanki [13]

As seen in the figure, the median of nine values can be split into three sorting stages. The first stage sorts each row of the nine pixels, with three different sorters sorting concurrently. Next, the sorted values are grouped into three different groups, each representing the largest, middle and smallest value of the previous sort. These groups are also known as column sort, as they are grouped in each column of the result of previous sort, totaling in three sorts for this stage. The last stage, also known as diagonal sort, contains one sort for three values on the diagonal of the result of the previous sort. The three values are the smallest member of the largest group, the middle member of the middle group, as well as the largest member of the smallest group respectively.

In the first two stages, as three sorts are being performed in parallel, which decreases the time complexity of computation compared to traditional programming algorithms of recursive merge sort. As the whole module can be split into three pipeline stages with two sets of registers in the middle, this design also allows a greater throughput with multiple pipeline stages, providing an efficient and high-performance median filter design for our project. Figure 5 shows a design of the median filter module.

A screenshot of a computer

Description automatically generated

Figure 5. Design of pipelined median filter

*Sequential input of median filter*

As the input of adjacent median filters are partially overlapped, multiple median filters could share same inputs and reduce the number of total sorting algorithms performed. In our project, each row of pixels is being processed concurrently, where a new row of output would be generated in each clock cycle. According to the algorithm by Bevara and Sanki [13] mentioned above, each row is being sorted before the output would be sorted by column. This means that we can sort each input row and store the sorted values for two more clock cycles, which are being used to compute the sorting of subsequent columns. Figure X shows how sorted row values are being used in the next clock cycles.

Because of this characteristic, we developed the median filter with sequential input and output. For each clock cycle, the three inputs in the same row are being sorted and stored in one of three groups of first in first out registers, which are groups of largest, medium and smallest values from the sort. These registers would store the sorted values for three clock cycles, where they will be sorted again by column. The column sort module is essentially a mealy machine with one input, two registers and an output. The resulting value of the three groups would be sorted again and output as the median of the nine values. Figure 6 shows the full design of module incorporating the column sort module as a sub-module.

A screenshot of a computer

Description automatically generated

Figure 6. Design of sequential median filter module

*Staged and non-staged sort module*

As mentioned in the above schematic, a total of five three-sort modules are required for the sequential input median filter, forming an essential part of the algorithm. An easy way to sort three values in ascending order in hardware is to first do comparisons for each pair of values, then determine the output sort using the combinations of comparisons. Table 2, Table 3 and Table 4 shows how the combination of three comparisons are being used to determine the order of the values.

Table 2. Decision of large output in sort module

|  |  |  |  |
| --- | --- | --- | --- |
| **Large out** | **A>B** | **B>C** | **C>A** |
| **A** | 1 |  | 0 |
| **B** | 0 | 1 |  |
| **C** | 1 |  | 1 |
| 0 | 0 |  |

Table 3. Decision of medium output in sort module

|  |  |  |  |
| --- | --- | --- | --- |
| **Medium out** | A>B | **B>C** | C>A |
| **A** | 0 | 1 | 0 |
| 1 | 0 | 1 |
| **B** | 1 | 1 |  |
| 0 | 0 |  |
| **C** | 1 | 0 | 0 |
| 0 | 1 | 1 |

Table 4. Decision of small output in sort module

|  |  |  |  |
| --- | --- | --- | --- |
| **Small out** | A>B | **B>C** | C>A |
| **A** | 0 |  | 1 |
| **B** | 1 | 0 |  |
| **C** | 1 | 1 | 0 |
| 0 |  | 0 |

With these tables, we can easily formulate an efficient three-input sorter, where three comparisons are first made, then fed into a set of multiplexers (MUXs) that choose the correct outputs. Because the sorter could be divided into two parts, the comparison part and the multiplexer part, a dedicated pipeline stage can be applied to the sorter to further increase the performance of the filter. The schematic of staged and non-staged filters is shown in task 2.

#### **Task 2:**

**Aim:** Implement the computational block of the median filter algorithm

**Member in charge:** Issac and Gordon

**Work Description:**

To implement the median filter in a clear structure, we implemented the sequential median filter module with multiple sub-modules, where unit testing of each sub-module can also be done and minimize mistakes. Figure 7 shows the overall structure and submodules of the top module *sequential\_nine\_median*. The filter contains one *three\_sort* module at the beginning, acting as the row sort, and another *three\_sort* module at the end as the diagonal sort. As the column sorting requires dedicated FIFO registers for storage, a dedicated *column\_sort* module is being implemented with two FIFO registers and a three\_sort module inside. Details of each module will be described below.

A computer screen shot of a diagram

Description automatically generated

Figure 7. Overall structure of module sequential\_nine\_median, and its sub-modules

*Module three\_sort*

Module *three\_sort* takes in three 8-bit values at once and sorts the input values, and give the sorting result (largest, middle, smallest) via three wires. Figure 8 below shows the RTL schematic of the implemented module, while the code snipplet of the module is also shown below. The module is split into two parts, the comparison part and the multiplexer part. In the comparison part, the input values are compared to one another, which yields three Boolean. The three comparisons are whether input A is greater than input B, whether input B is greater than input C and whether input C is greater than input A. The Booleans are then stored into registers for the multiplexer part. In the multiplexer part, the Booleans from above are used to determine which is the largest, middle and smallest among those three input values and send them out.

This module is used for row, column and diagonal sort at the same time, but there is some difference when it comes to column sorting, details will be discussed in the following section.

A computer screen shot of a computer

Description automatically generated

Figure 8. RTL schematic of module three\_sort

|  |
| --- |
| Code Snippet of module *three\_sort*  module three\_sort(  input [7:0] A\_in,  input [7:0] B\_in,  input [7:0] C\_in,  output [7:0] L\_out,  output [7:0] M\_out,  output [7:0] S\_out,  input clk  );  // intermediate registers  reg A\_GT\_B, B\_GT\_C, C\_GT\_A;  reg [7:0] tempA, tempB, tempC;  always @ (posedge clk) begin  A\_GT\_B <= A\_in > B\_in;  B\_GT\_C <= B\_in > C\_in;  C\_GT\_A <= C\_in > A\_in;  tempA <= A\_in; tempB <= B\_in; tempC <= C\_in;  end    assign L\_out = A\_GT\_B? (C\_GT\_A?tempC:tempA) :(B\_GT\_C?tempB:tempC);  assign M\_out = A\_GT\_B? (B\_GT\_C?tempB: (C\_GT\_A?tempA:tempC) ) :(B\_GT\_C? (C\_GT\_A?tempC:tempA) :tempB);  assign S\_out = A\_GT\_B? (B\_GT\_C?tempC:tempB) : (C\_GT\_A?tempA:tempC);  endmodule |

*Module column\_sort*

Module *column\_sort* takes in one 8-bit input value and gives three 8-bit output values. The sorting is done by one module *three\_sort* and gives the largest, middle and smallest values inside one column. However, in module *sequential\_nine\_median*, we would only use one output value from this module. Since we aim to create a pipelined dataflow, this module will be called three times at once in module *sequential\_nine\_median* each clock cycle to store the results from *three\_sort* and start giving output after three clock cycles from the beginning of our program since the results from *three\_sort* for three rows will be completed at that moment. In addition, the diagonal sort will also begin after three clock cycles since results needed for diagonal sort are from this module.

In module *sequential\_nine\_median,* three separate *column\_sort* will take the largest, middle, smallest values from each row and store them into a register inside themselves at each clock cycle. After three clock cycles, each column\_sort will have the largest, middle, smallest values of three rows and comparison will begin. The comparison will be done in columns, which will find the smallest value among the largest values, the middle value among the middle values and the largest value among the smallest values of three rows. The results will then be stored into registers and be used for diagonal sort. The schematic of the module *column\_sort* and the code snippet is shown below.

A computer screen shot of a computer

Description automatically generated

Figure 9. RTL schematic of module column\_sort

|  |
| --- |
| Code Snippet of module *column\_sort*  module column\_sort(  input [7:0] A\_in,  output [7:0] L\_out,  output [7:0] M\_out,  output [7:0] S\_out,  input clk  );  reg [7:0] B\_in, C\_in;  three\_sort sorter (.A\_in(A\_in), .B\_in(B\_in), .C\_in(C\_in), .L\_out(L\_out), .M\_out(M\_out), .S\_out(S\_out), .clk(clk));  always @ (posedge clk) begin  B\_in <= A\_in;  C\_in <= B\_in;  end  endmodule |

*Module sequential\_nine\_median*

The most important function of the module sequential\_nine\_median is to incorporate all submodules and control the dataflow inside the filter. It contains three 8-bit wide input ports for data input, a clock input for synchronous pipelined stages, as well as an 8-bit output for median. As mentioned in task 1, a median filter can be divided into three arithmetic stages, namely row sort, column sort and diagonal sort. Thus, the module contains registers that divide the stages and create a pipelined dataflow with a total of six 8-bit registers. Three of them are used to store the sorted values after the first sort, a.k.a. row sort, and the other three are being used to store the values that will be used in the last sort, a.k.a. diagonal sort. The source code and the RTL schematic of the module are shown below.

A computer screen shot of a computer screen

Description automatically generated

Figure 10. RTL schematic of module sequential\_nine\_median

|  |
| --- |
| source code of module *sequential\_nine\_median*  module sequential\_nine\_median(  input [7:0] A\_in,  input [7:0] B\_in,  input [7:0] C\_in,  output [7:0] median\_out,  input clk  );  reg [7:0] row\_L\_reg, row\_M\_reg, row\_S\_reg, column\_LS\_reg, column\_MM\_reg, column\_SL\_reg;  wire [7:0] row\_L\_out, row\_M\_out, row\_S\_out, column\_LS\_out, column\_MM\_out, column\_SL\_out;    three\_sort row\_sorter(.A\_in(A\_in), .B\_in(B\_in), .C\_in(C\_in), .L\_out(row\_L\_out), .M\_out(row\_M\_out), .S\_out(row\_S\_out), .clk(clk));  column\_sort column\_sorter\_L (.A\_in(row\_L\_reg), .S\_out(column\_LS\_out), .clk(clk));  column\_sort column\_sorter\_M (.A\_in(row\_M\_reg), .M\_out(column\_MM\_out), .clk(clk));  column\_sort column\_sorter\_S (.A\_in(row\_S\_reg), .L\_out(column\_SL\_out), .clk(clk));  three\_sort diagonal\_sorter(.A\_in(column\_LS\_reg), .B\_in(column\_MM\_reg), .C\_in(column\_SL\_reg), .M\_out(median\_out), .clk(clk));    always @(posedge clk) begin  row\_L\_reg <= row\_L\_out;  row\_M\_reg <= row\_M\_out;  row\_S\_reg <= row\_S\_out;  column\_LS\_reg <= column\_LS\_out;  column\_MM\_reg <= column\_MM\_out;  column\_SL\_reg <= column\_SL\_out;  end  endmodule |

#### **Task 3:**

**Aim:** Implement the median filter algorithm by using the computational blocks

**Member in charge:** Issac and Gordon

While the module *sequential\_nine\_median* can be used independently for processing a median for nine values with inputs across three clock cycles, our project aims to provide a scalable solution by processing one row of pixels every clock cycle, where an array of median filter modules is needed. The module *median\_filter\_scalable* provides a scalable generation module, which generates the amount of *sequential\_nine\_median* modules needed and interfaces it with an array of 8-bit values.

One of the challenges faced when developing this module is that Verilog on its own does not support multi-dimensional arrays. While we can flatten the array and pass into the module without compromise in performance, using these methods would add unnecessary complexity on coding, and thus might result in bugs and errors in development. Thus, the module is being written in SystemVerilog, which supports multi-dimensional arrays.

The module is scalable, where the change of *SIZE* at the second row of the module can easily change the size of input and output arrays, as well as the number of processing modules. It is noteworthy that the output size is two smaller than the input array size, as median filter has a locality of three, and thus the first and last modules corresponding to the input array would not have enough information to calculate the median.

A generate block with for loop is being implemented to easily scale the number of modules needed and interface with the input and output arrays. Each module takes input from current and adjacent pixels, and outputs to a smaller output array position, as the output array is smaller than the input array at both ends. The source code and RTL schematic of a demo system with input array size of 10 is being shown below as demonstration.

A screenshot of a computer

Description automatically generated

Figure 11. RTL schematic of module median\_filter\_scalable

|  |
| --- |
| Source code of module *median\_filter\_scalable*  module median\_filter\_scalable  # (parameter SIZE = 10) ( // change SIZE to vary system size  input [7:0] arr\_in [SIZE-1:0],  output [7:0] arr\_out [SIZE-3:0],  input clk  );  genvar i;  generate  for (i=1;i<SIZE-1;i=i+1) begin  sequential\_nine\_median median(.A\_in(arr\_in[i-1]), .B\_in(arr\_in[i]), .C\_in(arr\_in[i+1]), .median\_out(arr\_out[i-1]), .clk(clk));  end  endgenerate  endmodule |

#### **Task 4:**

**Aim:** Verify the correctness of the median filter

**Member in charge:** Issac

**Work Description:**

There are multiple modules in median filter where testing and thorough verification has to be performed. Currently, module three\_sort has completed thorough verification, while most other modules have only basic test cases test completed, where edge case and all possible test case verifications currently underway. The test progresses for each module are as following.

*Module three\_sort*

Testing for all possible test cases for module *three\_sort* has been completed by using for loops for inputs. It is being verified by using behavioral simulation on the output results and compare if the values are correct.

The testbench code is shown below.

|  |
| --- |
| Source code for testbench of module *three\_sort*  module three\_sort\_tb(  );  reg [7:0] tb\_A\_in, tb\_B\_in, tb\_C\_in;  reg tb\_clk;  wire [7:0] tb\_L\_out, tb\_M\_out, tb\_S\_out;  reg correct;  three\_sort uut(.A\_in(tb\_A\_in), .B\_in(tb\_B\_in), .C\_in(tb\_C\_in), .L\_out(tb\_L\_out), .M\_out(tb\_M\_out), .S\_out(tb\_S\_out), .clk(tb\_clk));    integer i, j, k;  initial begin  for(k=0;k<256;k=k+1)begin  tb\_C\_in = k;  for(j=0;j<256;j=j+1)begin  tb\_B\_in = j;  for(i=0;i<256;i=i+1)begin  tb\_A\_in = i;  tb\_clk = 1'b0;  #5;  tb\_clk = 1'b1;  #5;  end  end  end  end    always @(\*) begin  if (tb\_L\_out>=tb\_M\_out && tb\_M\_out>=tb\_S\_out)  correct = 1'b1;  else  correct = 1'b0;  end  endmodule |

*Module sequential\_nine\_median*

Currently, we are still figuring out the coding of a thorough testbench for all possible test cases for module *sequential\_nine\_median*, as the module has memory inside, which affects the testing results. However, preliminary testing has been conducted, where a few test cases with some edge cases has been tested out and the results are deemed correct. The testbench code is shown below.

|  |
| --- |
| Source code for testbench of module *sequential\_nine\_median*  module sequential\_nine\_median\_tb(  );  reg [7:0] tb\_A\_in, tb\_B\_in, tb\_C\_in;  reg tb\_clk;  wire [7:0] tb\_median\_out;  sequential\_nine\_median dut(.A\_in(tb\_A\_in), .B\_in(tb\_B\_in), .C\_in(tb\_C\_in), .median\_out(tb\_median\_out), .clk(tb\_clk));  initial begin  tb\_A\_in = 7'd0;  tb\_B\_in = 7'd1;  tb\_C\_in = 7'd2;  #2;  tb\_A\_in = 7'd3;  tb\_B\_in = 7'd4;  tb\_C\_in = 7'd5;  #2;  tb\_A\_in = 7'd6;  tb\_B\_in = 7'd7;  tb\_C\_in = 7'd8;  #2;  tb\_A\_in = 7'd0;  tb\_B\_in = 7'd1;  tb\_C\_in = 7'd2;  #2;  tb\_A\_in = 7'd3;  tb\_B\_in = 7'd4;  tb\_C\_in = 7'd5;  #2;  tb\_A\_in = 7'd6;  tb\_B\_in = 7'd7;  tb\_C\_in = 7'd8;  end  initial begin  tb\_clk = 0;  forever #1 tb\_clk = ~tb\_clk;  end  endmodule |

The testbench code of other modules are attached in the appendix.

**Task 5:**

**Aim:** Measure the performance of the implemented median filter

**Member in charge:** Haven

**Progress:** Awaiting completion of previous tasks.

### **2.2.2 Data cache for fetching and storing**

**Aim:** To implement a data cache system, and consolidate it with the implemented system.

**Tasks:**

Implement the data cache

Gordon

Verify the correctness of data cache

Gordon

Combine data cache with the Sobel filter

Issac

Progress: Awaiting completion of other objectives.

### **2.2.3 Sobel filter for edge detection**

To design and implement an optimized Sobel filter for the edge detection part for the whole system, and verify the correctness of it, and integrate into the system.

In this objective, we designed a Sobel filter system that sequentially takes in one row of pixels in and output the calculated Sobel operator at each clock cycle.

Sobel operator is one of the main operators that can be used for edge detection In digital image processing, with an advantage of a relatively small kernel, and reducing the number of arithmetic operations, while retaining a suitable degree of accuracy as a high-pass filter to filter out edges in an image. Sobel filter has two different kernels, X kernel and Y kernel for computing horizontal and vertical edges respectively. Figure 12 below shows the X kernel and Y kernel of Sobel filter.

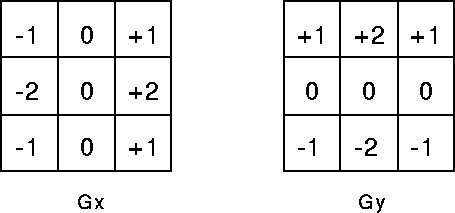


Figure 12. X kernel and Y kernel [14]

The filter is being designed to calculate one row of pixels concurrently, with neighboring Sobel operators sharing intermediate calculation results to reduce the number of arithmetic operations performed, increasing the efficiency of design. The design is also scalable and able to adapt to images of different sizes.

**Task 1:**

**Aim:** Design a scalable and optimized Sobel operator

**Member in charge:** Issac

**Work Description:**

As shown in the figure above, the Sobel Kernels have two characteristics that can be used for efficient design of the Sobel filter. One of the characteristics is the complimentary mirroring of weights. For instance, first row and last row of X kernel of Sobel filter has weights of same magnitude, differs only in the sign of the weights, with weights of the first row being negative and those on the last row being positive. The same applies to the Y kernel as well, where the weights on the left column has same magnitude with the right column, only differing on the sign. This means that we can reuse some of the calculations for previous results for other pixels.

Another characteristic is that the kernels have values of only two weight magnitude of one and two. Because of this characteristic, we can multiply the weight of pixels only once and add them according to the Sobel kernels.

The whole filter can be divided into three main parts, the X kernel part, the Y kernel part, and another part which adds up and calculates the final edge. Figure X below shows how the three parts corresponds to each other, where the intermediate values calculated by both kernels are added together and form the final result of the edge. Figure 13 shows the system view of a Sobel processor.

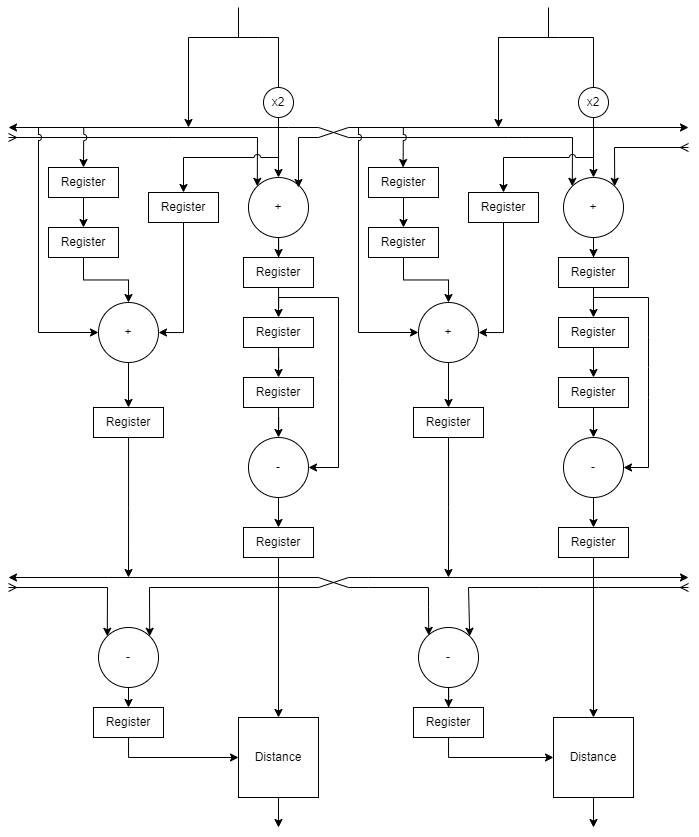


Figure 13. System view of Sobel processor

*X kernel design*

As described in figure X above, X kernel can be divided into the left column, which only differs in terms of the sign. Thus neighboring Sobel operators could also use the same intermediate calculations for their final result.

In our design, each X Sobel operator takes in one input value at once into an SIPO register with three slots. For each cycle, the data in the register shifts and each data is being multiplied with their respective weights. The newest and oldest data would have a weight of one, while the middle data has a weight of two.

After the weighted sum is being calculated, the output of the Sobel X kernel is being taken as the difference between the adjacent weighted sums (without current pixel’s weighted sum).

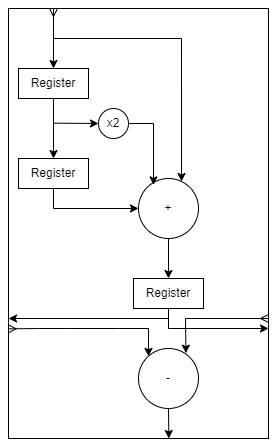


Figure 14. Design of X kernel processor

*Y kernel design*

As described in figure Y above, Y kernel can be divided into three rows, while the top row and the bottom row are complementary. Because the whole system is being designed to process the image by processing row by row in each clock cycle, in each clock cycle, the Y kernel operator only needs to take in three values adjacent pixels to calculate the result. The complementary characteristic of the Y-kernel means that the intermediate value calculated for each row can be reused by the Y kernel of the second next pixel for its output.

The design of Y kernel module emphasizes storing the weighted intermediate values of the kernel and use it for calculations of the next outputs. Figure X below shows the design of the Y kernel module. First, the three input values of adjacent pixels in the row are being input to the operator, where they are being multiplied by specific weights. The current pixel is being multiplied by weight two, while the adjacent two pixels are multiplied by weight 1 only. Then, the weighted inputs are added together and stored inside an SIPO register for further use. The complementary characteristics of X kernel means that every other output of pixels would share the same intermediate results, so we can directly calculate the difference between the current intermediate and previous intermediate for the Sobel Y edge output. Figure 13 below shows an example of how every other output requires the same intermediate value, only differing in its sign. Because of this, we can easily perform subtraction to the two saved intermediate values to result in the final Sobel X result.

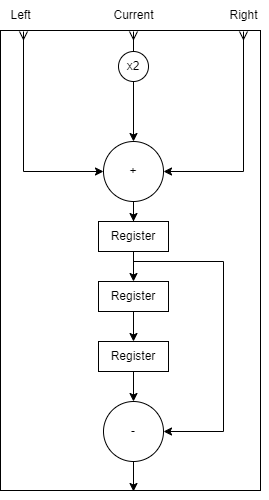


Figure 15. Design of Y kernel processor

**Task 2:**

**Aim:** Implement the designed Sobel filter

**Member in charge:** Issac and Haven

**Work Description:**

The whole Sobel filter processor is designed to take in one row of pixels and output a Sobel value every clock cycle, in a staged format. The whole system can be scaled to cope with the size of the image, controlling how many Sobel filter modules are being used. The top module, *sequential\_sobel\_scalable*, generates the number of required Sobel modules, where each Sobel module further includes sub-modules to calculate the Sobel output value. The system of *sequential\_sobel\_scalable*  is shown below, where the input output ports, sub-module placements as well as the intermediate exchange of weighted values are shown below.

A black background with white circles and red lines

Description automatically generated

Figure 16. Schematic design of scalable Sobel filter

Module *sequential\_sobel\_X*

Module *sequential\_sobel\_X* is being designed to take in one input of the current and adjacent pixels and output a Sobel X value every clock cycle. The Module is based on a pipelined design, with a total of four pipelined stages for processing. The module has one 8-bit input for pixel value input, one 10-bit output for current pixel’s weighted sum, two 10-bit inputs for adjacent pixels’ weighted sum, a 10-bit Sobel X output and a synchronous clock input.

First, the input is passed into an SIPO register, where the value is saved for the later two clock cycles. For each clock cycle, the saved values are being multiplied with their weights and added together. Since the weights only contain one and two, we can simply perform a shift left 1-bit operation for the weight of two, while directly routing them into the adder. The three values are added together to form an intermediate value of weighted sum and being stored in a temporary register for staged pipeline. As we are taking the image with 8-bit color depth, the value with weight two would have a maximum value of nine bits, and the sum of two 8-bit value and one 9-bit value would have a maximum value of 10-bits, which would be the bit depth of the weighted sum.

Next, the intermediate weighted sum is being exchanged between adjacent processing modules. The current weighted sum, named *current\_intermediate*, is being output to the neighboring modules, while adjacent weighted sums, named *left\_intermediate* and *right\_intermediate* are input to the module. The linkage of the adjacent modules is handled by upper modules, while the module *sequential\_sobel\_X* only offers input and output ports for this exchange of values.

The difference of the adjacent weighted sums being input to the module are then being output as the output of the Sobel X operator, also as the output of the module. As the intermediate weighted sum has a maximum value of 10-bits, taking the difference between the two weighted sums, ie. *right\_intermediate* minus *left\_intermediate* would yield a result of a signed 11-bit value. As we want the difference between the two weighted sums, only the magnitude of the subtraction is important. The difference is taken by inverting any negative numbers to a 10-bit unsigned number. It is noteworthy that the value is different from the true magnitude of a negative number with a difference of 1, but the small difference is negligible and saves resources for the addition, thus the choice of not taking the true two’s complement value is being made, only taking the complementary of the negative values.

The RTL schematic and the source code of module *sequential\_sobel\_X* is shown below.

A computer screen shot of a diagram

Description automatically generated

Figure 17. RTL schematic of module sequential\_sobel\_X

|  |
| --- |
| Source code of module sequential\_sobel\_X  module sequential\_sobel\_X(  input [7:0] current\_in,  input [9:0] left\_intermediate,  input [9:0] right\_intermediate,  output [9:0] current\_intermediate,  output [9:0] sobel\_X\_out,  input clk  );  reg [7:0] shift\_add\_reg [1:0];  reg [9:0] add\_reg;  reg [10:0] sobel\_X\_reg;  wire [8:0] shift\_add\_reg\_0\_shift;  always @(posedge clk) begin  shift\_add\_reg[0] <= current\_in;  shift\_add\_reg[1] <= shift\_add\_reg[0];  add\_reg <= current\_in + shift\_add\_reg\_0\_shift + shift\_add\_reg[1];  sobel\_X\_reg <= right\_intermediate - left\_intermediate;  end  assign shift\_add\_reg\_0\_shift = {shift\_add\_reg[0], 1'b0};  assign current\_intermediate = add\_reg;  assign sobel\_X\_out = (sobel\_X\_reg[10] == 1)?~sobel\_X\_reg[9:0]:sobel\_X\_reg[9:0];  endmodule |

Module *sequential\_sobel\_Y*

Module *sequential\_sobel\_Y* is being designed to take in three inputs of the current pixel and output a Sobel Y value every clock cycle. The Module is based on a pipelined design, with a total of four pipelined stages for processing. The module has three 8-bit inputs for pixel value input, one 10-bit Sobel Y value output, as well as a synchronous clock input.

For every clock cycle, the three input values are being passed into the module and the weighted sum is taken. The adjacent pixel values are given a weight of one, while the current pixel value has a weight of two. The current pixel value’s input to the adder is being shifted to the left for one bit because of the weight of two given to the value. The sum of two 8-bit and one 9-bit number (due to the shift) has a maximum value of 10-bits, so the output of the weighted sum is 10-bits in width.

After the weighted sum is being calculated, the value is stored in an SIPO register for calculations. As mentioned above in Task 1, only the first and the third row of the kernel contains non-zero weights, the design only takes in the oldest and newest data value from the SIPO register and take the difference between them. The difference is being calculated by subtracting the oldest data from the newest data of the SIPO register, and then taking a complement if the value is negative. The subtraction of two 10-bit unsigned numbers results in a 11-bit signed number, and with the complement, the output of Sobel Y is a unsigned 10-bit number.

The RTL schematic and source code of module *sequential\_sobel\_Y*  is shown below.

A computer screen shot of a diagram

Description automatically generated

Figure 18. RTL schematic of module sequential\_sobel\_Y

|  |
| --- |
| Source code of module sequential\_sobel\_Y  module sequential\_sobel\_Y(  input [7:0] current\_in,  input [7:0] left\_in,  input [7:0] right\_in,  output [9:0] sobel\_Y\_out,  input clk  );  wire [8:0] current\_in\_shifted;  reg [9:0] add\_reg [2:0];  reg[10:0] sobel\_Y\_reg;    always @ (posedge clk)begin  add\_reg[0] <= current\_in\_shifted + left\_in + right\_in;  add\_reg[1] <= add\_reg[0];  add\_reg[2] <= add\_reg[1];  sobel\_Y\_reg <= add\_reg[2] - add\_reg[0];  end  assign current\_in\_shifted = {current\_in, 1'b0};  assign sobel\_Y\_out = (  sobel\_Y\_reg[10]==1)?~sobel\_Y\_reg[9:0]:sobel\_Y\_reg[9:0];  endmodule |

Module *sequential\_sobel*

Module *sequential\_sobel* acts as the upper module for *sequential\_sobel\_X* and *sequential\_sobel\_Y*, and yields the result of the sobel operator. The module takes in the current and adjacent pixel value for each input (a total of three), and outputs a Sobel value for each clock cycle. The module is divided into two parts, the part containing the sub-modules, as well as a sum of Sobel results part. The design of module *sequen*tial\_sobel is shown in Figure 19.

A screenshot of a computer

Description automatically generated

Figure 19. Design of module sequential\_sobel

First, the three pixel values are taken into the module, where they are passed into the submodules of *sequential\_sobel\_X* and *sequential\_sobel\_Y*. The module also has input and output ports for the sub-module *sequential\_sobel\_X* to link and transmit to and from adjacent modules for the intermediate sum of weights. The output of the two sub-modules are then saved into a temporary register, ready to be fetched into the next part.

In the second part of the module, the results of X and Y Sobel are being added together to result in a final sobel output value. There are multiple ways to evaluate such a final output, with the most prominent ways being the Manhattan distance and Euclidean distance. Euclidean distance, as its name suggests, is computes by taking the Euclidean sum of the values, which takes the square root of the sum of squares of the input values [15]. Manhattan distance is being calculated by only adding the magnitude of the two values together, without regard to the orientation of values, where X and Y Sobel edges are perpendicular to each other. Manhattan distance has its name from the city blocks of Manhattan, New York, where the distance is calculated by the sum of magnitude only due to the perpendicular design of streets.

Currently, we implemented the addition of the two Sobel values using Manhattan distance, as it uses relatively small resources compared to Euclidean distance, which requires taking square root. Taking square root not only uses much more hardware resources, but also more time as most efficient algorithms for finding square root value, such as the Heron’s method [16]. We will try and explore efficient algorithms for taking square roots in the future.

The source code and the RTL schematic is shown below.

A screenshot of a computer

Description automatically generated

Figure 20. RTL schematic of module sequential\_sobel

|  |
| --- |
| Source code of module *sequential\_sobel*  module sequential\_sobel(  inout [7:0] current\_inout,  input [7:0] left\_in, right\_in,  input [9:0] left\_intermediate, right\_intermediate,  output [9:0] current\_intermediate,  output [7:0] sobel\_out,  input clk  );  wire [9:0] sobel\_X\_out, sobel\_Y\_out;  reg [9:0] sobel\_X\_reg, sobel\_Y\_reg;  reg [10:0] sobel\_reg;  sequential\_sobel\_X X\_filter(.current\_in(current\_inout), .left\_intermediate(left\_intermediate), .right\_intermediate(right\_intermediate),  .current\_intermediate(current\_intermediate), .sobel\_X\_out(sobel\_X\_out), .clk(clk));  sequential\_sobel\_Y Y\_filter(.current\_in(current\_inout), .left\_in(left\_in), .right\_in(right\_in), .sobel\_Y\_out(sobel\_Y\_out), .clk(clk));    always @(posedge clk) begin  sobel\_X\_reg <= sobel\_X\_out;  sobel\_Y\_reg <= sobel\_Y\_out;  sobel\_reg <= sobel\_X\_reg + sobel\_Y\_reg;  end  assign sobel\_out = sobel\_reg[10:3];  endmodule |

Module *sobel\_filter\_scalable*

Different from the median filter module in 2.2.1, the *sequential\_sobel* module can not work with a single module, due to the fact that the computation of Sobel X value requires intermediate weighted sum values from neighboring modules. This module provides a flexible generation of sub-module *sequential\_sobel*, which acts as a computational block for Sobel filter. The filter accepts input of size and an output of size two smaller than the input, as the size of kernel of Sobel filter is three. This module generates number of modules corresponding to the size of input, where the three major inputs are being mapped to the current and adjacent pixel input. There is also a temporary linkage called intermediate that connects the intermediates of different computational units together, which are used for Sobel X output calculations. The final outputs are being mapped to output array.

The RTL schematic and source code of a sample module with input size of five is shown below. The size can be easily changed to scale the module by changing the value SIZE on row two of the source code.

**A computer screen shot of a computer

Description automatically generated**

Figure 21. RTL schematic of module sobel\_filter\_scalable

|  |
| --- |
| Source code of module sobel\_filter\_scalable  module sobel\_filter\_scalable  # (parameter SIZE = 5) ( // change SIZE to vary system size  input [7:0] arr\_in [SIZE-1:0],  output [7:0] arr\_out [SIZE-3:0],  input clk  );  wire [7:0] intermediate [SIZE-1:0];  genvar i;  generate  for(i=0; i<SIZE; i=i+1) begin  sequential\_sobel sobel(.current\_in(arr\_in[i]), .left\_in(arr\_in[i-1]), .right\_in(arr\_in[i+1]),  .current\_intermediate(intermediate[i]), .left\_intermediate(intermediate[i-1]), .right\_intermediate(intermediate[i+1]),  .sobel\_out(arr\_out[i-1]), .clk(clk));  end  endgenerate  endmodule |

**Task 3:**

**Aim:** Verify the correctness of Sobel filter

**Member in charge:** Haven

**Work Description:**

Currently under progress

**Task 4:**

**Aim:** Measure the performance of the implemented Sobel filter

**Member in charge:** Haven

**Work Description:**

To be executed

### **2.2.4 Otsu’s method for grayscale to black-and-white**

**Aim:** To design and implement Otsu’s method that can be updated concurrently and compute final weighting by combining concurrent parts, and integrate into the system.

**Tasks:**

Design the algorithm for concurrent pipelining of Otsu’s algorithm

Haven

Implement the parallelized Otsu’s algorithm

Haven

Verify the correctness of the implemented Otsu’s algorithm

Gordon

Measure the performance of the implemented Otsu’s algorithm

Gordon

Combine the Otsu’s algorithm with the rest of the system

Issac

**Progress:** Under design

### **2.2.5 Color to grayscale**

**Aim:** To implement color-to-grayscale and integrate into the system.

**Tasks:**

Implement color to grayscale filter

Haven

Measure the performance of the implemented color-to-grayscale filter

Haven

Combine the color to grayscale filter with the rest of the system

Haven

**Progress:** Under implementation

### **2.2.6 Input and Output**

To design and implement interface for image input and output, and integrate into the system.

Tasks:

Implement I/O for the system

Haven

Verify correctness of the whole system

Issac

Measure the performance of the whole system

Gordon

**Progress:** Awaiting completion of previous parts

# **SECTION 3— Project Planning**

## **3.1 Project Schedule**

The project Schedule is included in the next pages.

Table 5. Project Schedule

| **Objective Statements** | **Task** | Group Member in charge | wk1-3 | wk4-6 | wk7-9 | wk10-12 | wk13-15 | wk16-18 | wk19-21 | wk22-24 | wk25-27 | wk28-30 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***Median filter for noise reduction*** |  |  |  |  |  |  |  |  |  |  |  |  |
|  | **Design of median filter** | **Issac** |  |  |  |  |  |  |  |  |  |  |
|  | **Implement the computational block of the median filter algorithm** | **Issac** |  |  |  |  |  |  |  |  |  |  |
|  | **Implement the median filter algorithm by using the computational blocks** | **Issac**  **Gordon** |  |  |  |  |  |  |  |  |  |  |
|  | **Verify the correctness of the median filter** | **Issac** |  |  |  |  |  |  |  |  |  |  |
|  | **Measure the performance of the implemented median filter** | **Haven** |  |  |  |  |  |  |  |  |  |  |
| **Data cache for fetching and storing** |  |  |  |  |  |  |  |  |  |  |  |  |
|  | **Implement the data cache** | **Gordon** |  |  |  |  |  |  |  |  |  |  |
|  | **Verify the correctness of data cache** | **Gordon** |  |  |  |  |  |  |  |  |  |  |
|  | **Combine data cache with the sobel filter** | **Issac** |  |  |  |  |  |  |  |  |  |  |
| **Sobel filter for edge detection** |  |  |  |  |  |  |  |  |  |  |  |  |
|  | **Design of optimized sobel filter** | **Issac** |  |  |  |  |  |  |  |  |  |  |
|  | **Implementation of sobel filter** | **Issac** |  |  |  |  |  |  |  |  |  |  |
|  | **Verify the correctness of sobel filter** | **Haven** |  |  |  |  |  |  |  |  |  |  |
|  | **Measure the performance of the implemented sobel filter** | **Haven** |  |  |  |  |  |  |  |  |  |  |
|  | **Combine the sobel filter with the rest of the system** | **Issac** |  |  |  |  |  |  |  |  |  |  |
| **Otsu’s method for grayscale to black-and-white** |  |  |  |  |  |  |  |  |  |  |  |  |
|  | **Design of the algorithm for concurrent pipelining of Otsu’s algorithm** | **Haven** |  |  |  |  |  |  |  |  |  |  |
|  | **Implement the parallelized Otsu’s algorithm** | **Haven** |  |  |  |  |  |  |  |  |  |  |
|  | **Verify the correctness of the implemented Otsu’s algorithm** | **Gordon** |  |  |  |  |  |  |  |  |  |  |
|  | **Measure the performance of the implemented Otsu’s algorithm** | **Gordon** |  |  |  |  |  |  |  |  |  |  |
|  | **Combine the Otsu’s algorithm with the rest of the system** | **Issac** |  |  |  |  |  |  |  |  |  |  |
| **Color to grayscale** |  |  |  |  |  |  |  |  |  |  |  |  |
|  | **Implement color to grayscale filter** | **Issac** |  |  |  |  |  |  |  |  |  |  |
|  | **Measure the performance of the implemented color-to-grayscale filter** | **Gordon** |  |  |  |  |  |  |  |  |  |  |
|  | **Combine the color to grayscale filter with the rest of the system** | **Haven** |  |  |  |  |  |  |  |  |  |  |
| **Input and Output** |  |  |  |  |  |  |  |  |  |  |  |  |
|  | **Implement I/O for the system** | **Haven** |  |  |  |  |  |  |  |  |  |  |
|  | **Verify correctness of the whole system** | **Gordon** |  |  |  |  |  |  |  |  |  |  |
|  | **Measure the performance of the whole system** | **Issac** |  |  |  |  |  |  |  |  |  |  |

## **3.2 Budget**

Table 6. Budget

|  |  |
| --- | --- |
| **Items\*** | Cost |
| **Total** | $0 |

# **References**

|  |  |
| --- | --- |
| [1] | *M. Rouse, Ed. (8 Dec. 2016), “Application-specific integrated circuit,” Techopedia, https://www.techopedia.com/definition/2357/application-specific-integrated-circuit-asic (accessed Aug. 31, 2023)..* |
| [2] | *“What is an FPGA? Field Programmable Gate Array,” Xilinx, https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html (accessed Aug. 31, 2023)..* |
| [3] | *Intel, “What Is a GPU? Graphics Processing Units Defined,” Intel, 2023. https://www.intel.com/content/www/us/en/products/docs/processors/what-is-a-gpu.html (accessed Aug. 31, 2023).* |
| [4] | *“CUDA GPUs - compute capability,” NVIDIA Developer, https://developer.nvidia.com/cuda-gpus (accessed Aug. 31, 2023)..* |
| [5] | *“What is ASIC? - ASIC Cost,” Arm | The Architecture for the Digital World. https://www.arm.com/zh-TW/glossary/asic (accessed Nov. 19, 2023)..* |
| [6] | *“Understanding FPGA Architecture,” Xillinx. https://www.xilinx.com/htmldocs/xilinx2017\_4/sdaccel\_doc/odz1504034293215.html.* |
| [7] | *N. Beckmann, “Dataflow Architecture: Pure, Hybrid, and Spatial,” CMU School of Computer Science, https://www.cs.cmu.edu/~15740-f20/ (accessed Jan. 8, 2024).* |
| [8] | *pktparticle, “Computer Organization: Von Neumann architecture,” GeeksforGeeks, https://www.geeksforgeeks.org/computer-organization-von-neumann-architecture/ (accessed Jan. 8, 2024).* |
| [9] | *"Course Syllabus - ELEC2350 Introduction to Computer Organization and Design (Fall 21-22)," HKUST Department of Electronic and Computer Engineering,* https://seng.hkust.edu.hk/sites/default/files/IMCE/UG/Course%20Syllabus/Fall\_2021-2022/ELEC2350\_Fall%2021-22.pdf (accessed Jan. 8, 2024). |
| [10] | *“Course Syllabus - ELEC3310 Digital Fundamentals and System Design,” Department of Electronic and Computer Engineering, https://ece.hkust.edu.hk/sites/default/files/Course%20Syllabi\_202104/ELEC3310.pdf (accessed Jan. 8, 2024).* |
| [11] | *“Course Syllabus – ELEC4320 FPGA based Design: From Theory to Practice,” Department of Electronic and Computer Engineering, https://ece.hkust.edu.hk/sites/default/files/Course%20Syllabi\_202104/ELEC4320%20.pdf (accessed Jan. 8, 2024).* |
| [12] | *“Course Syllabus – ELEC4130 Digital Image Processing,” Department of Electronic and Computer Engineering, https://ece.hkust.edu.hk/sites/default/files/2023-07/ELEC4130.pdf (accessed Jan. 8, 2024).* |
| [13] | *V. Bevara and P. K. Sanki (4 Jan. 2020), “A new fast and efficient 2-D median filter architecture,” Department of ECE, SRM University-Andhra Pradesh, Guntur, India [Online]. Available: https://www.ias.ac.in/article/fulltext/sadh/045/0192..* |
| [14] | *R. Fisher, S. Perkins, A. Walker, and E. Wolfart, “Sobel edge detector,” Feature Detectors - Sobel Edge Detector, https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm (accessed Jan. 8, 2024).* |
| [15] | E. Gregersen, “Euclidean distance - Definition, Formulas and Facts,” Encyclopædia Britannica, https://www.britannica.com/science/Euclidean-distance (accessed Jan. 8, 2024). |
| [16] | G. Swarankar, “Heron algorithm or Babylonian method (square root),” Medium, https://medium.com/@gauravswarankar/heron-algorithm-or-babylonian-method-square-root-14fb599db5d7 (accessed Jan. 8, 2024). |

# **APPENDICES**

## **Appendix A — Meeting Minutes**

Meeting 1:

Date: 22/07/2023

Time: 3pm

Location: Mong Kok

Attendees: Haven, Issac, Gordon

Minutes taken by: Gordon

* All members decided the topic of this final year project to be design and implement a system for image processing on a FPGA board
* All members discussed the details of the proposal and decided to finish the introduction and methodology by September 1st
* Haven have sent an email to the supervisor, Professor Zhang, to have a meeting discussing about the technical detail of the system and the proposal

Table 1. Action Items for Next Meeting

|  |  |  |
| --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** |
| Proposal 1.1 | Aug 15 | Haven |
| Proposal 1.3.1 | Aug 15 | Gordon |
| Proposal 1.3.2 | Aug 15 | Issac |
| Proposal 2.1 | Aug 15 | Issac |

Next Meeting: 23-08-2023/3pm /HKUST

Meeting 2:

Date: 23/08/2023

Time: 3pm

Location: HKUST, Room 2449

Attendees: Prof. Zhang, Haven, Issac, Gordon

Minutes taken by: Gordon

* Section 1 of the proposal has been mostly finished.
* Section 2 of the proposal is undergoing a good progress
* Prof. Zhang has suggested that the proposal should mainly focus on the methodology and it should be written with details and diagrams to explain the system clearly.
* Issac has explained the theory of the system to Prof. Zhang and she gave some ideas for the system, including what hardware or algorithm could be implemented.
* Haven has some enquiries regarding the proposal and suggested arranging a regular meeting with Prof. Zhang, in which a bi-weekly regular meeting is agreed among the group.

Table 1. Action Items from Previous Meeting

|  |  |  |  |
| --- | --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** | **Status** |
| Proposal 1.1 | Aug 15 | Haven | 80% |
| Proposal 1.3.1 | Aug 15 | Gordon | Completed |
| Proposal 1.3.2 | Aug 15 | Issac | Completed |
| Proposal 2.1 | Aug 15 | Issac | 50% |

Table 2. Action Items for Next Meeting

|  |  |  |
| --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** |
| Proposal 1.2 | Sept 1 | Issac |
| Proposal 2.1.3 | Sept 1 | Issac |
| Proposal 2.1.4.1 | Sept 1 | Issac |
| Proposal 2.1.4.2 | Sept 1 | Haven |
| Proposal 2.2 | Sept 1 | Issac |
| Proposal 3.1 | Sept 1 | Gordon |

Next meeting: 04-09-2023/11am/HKUST

Meeting 3:

Date: 04/09/2023

Time: 11am

Location: HKUST, LSK room1032

Attendees: Prof. Zhang, Haven, Issac, Gordon

Minutes taken by: Gordon

* First Draft of proposal being reviewed by Professor Zhang, where she is satisfied with the most part of the proposal.
* Prof. Zhang suggests minor changes to the proposal, particularly the background of introduction and the project schedule.
* All group members agree to amend the proposal ASAP and submit it to FYPMS a few days before the official deadline.

Table 1. Action Items from Previous Meeting

|  |  |  |  |
| --- | --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** | **Status** |
| Proposal 1.2 | Sept 1 | Issac | Completed |
| Proposal 2.1.3 | Sept 1 | Issac | Completed |
| Proposal 2.1.4.1 | Sept 1 | Issac | Completed |
| Proposal 2.1.4.2 | Sept 1 | Haven | Completed |
| Proposal 2.2 | Sept 1 | Issac | Completed |
| Proposal 3.1 | Sept 1 | Gordon | Completed |

Table 2. Action Items for Next Meeting

|  |  |  |
| --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** |
| Proposal 1.1 | Sept 9 | Issac |
| Proposal 3.1 | Sept 9 | Gordon |

Next meeting: mid Oct/3pm/HKUST

Meeting 4:

Date: 12/10/2023

Time: 3pm

Location: HKUST, room 2466

Attendees: Haven, Issac, Gordon

Minutes taken by: Gordon

* The proposal is uploaded to the FYP management system and approved by Professor Zhang, where she has reminded us to follow the schedule.
* Issac has designed the median filter structure, details will be written on the monthly report#1
* Gordon will implement the median filter and aims to finish it before the end of November.
* Issac and Haven are working on the design of sobel filter together, the progress might be slowed down due to the incoming mid-term exams.

Table 1. Action Items from Previous Meeting

|  |  |  |  |
| --- | --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** | **Status** |
| proposal section 1 | Sept 9 | Issac | Completed |
| proposal section 2 | Sept 9 | Haven | Completed |
| proposal section 3 | Sept 9 | Gordon | Completed |

Table 2. Action Items for Next Meeting

|  |  |  |
| --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** |
| Monthly report#1 | Oct 25 | Issac |
| Implement the computational block of the median filter algorithm | Nov 15 | Gordon |
| implement the median filter algorithm by using the computational blocks | Nov 15 | Gordon |
| Design of Sobel Filter | Nov 30 | Issac, Haven |

Next meeting: mid Nov/3pm/HKUST

Meeting 5:

Date: 15/11/2023

Time: 3pm

Location: HKUST, room 2466

Attendees: Haven, Issac, Gordon

Minutes taken by: Gordon

* Monthly report #1 is approved by Professor Zhang and she has reminded us to make a clearer plan of migration to the systolic array and we should also include a detailed work distribution of all group members in the next monthly report.
* Gordon has finished the computational blocks of the median filter and its performance will be measured by Issac.
* Issac has finalized the design of Sobel Filter but Haven needs more time to implement it.

Table 1. Action Items from Previous Meeting

|  |  |  |  |
| --- | --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** | **Status** |
| Monthly report#1 | Oct 25 | Issac | completed |
| Implement the computational block of the median filter algorithm | Nov 15 | Gordon | completed |
| Implement the median filter algorithm by using the computational blocks | Nov 15 | Gordon | completed |
| Design of Sobel Filter | Nov 30 | Issac, Haven | 60% |

Table 2. Action Items for Next Meeting

|  |  |  |
| --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** |
| Measure the performance of the implemented median filter | Nov 30 | Issac |
| Design of Sobel filter | Nov 30 | Issac,Haven |
| Implementation of Sobel filter | Dec 30 | Haven |

Next meeting: Will be decided after final exam

Meeting 6:

Date: 27/12/2023

Time: 7pm

Location: Discord

Attendees: Haven, Issac, Gordon

Minutes taken by: Gordon

* Issac has measured the performance of the median filter implemented by Gordon and it is fine to use in the next stage.
* The design of Sobel filter is finished but Haven finds it difficult to be implemented, Issac will help him on that matter.
* Gordon will start working on implementing the data cache and aims to finish it before January.
* Haven will do the color to grayscale filter and aims to finish it before 15 February.
* The progress is slightly behind schedule

Table 1. Action Items from Previous Meeting

|  |  |  |  |
| --- | --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** | **Status** |
| Measure the performance of the implemented median filter | Nov 30 | Issac | completed |
| Design of Sobel filter | Nov 30 | Issac, Haven | completed |
| Implementation of Sobel filter | Dec 30 | Haven | 40% |

Table 2. Action Items for Next Meeting

|  |  |  |
| --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** |
| Implementation of Sobel filter | Dec 30 | Issac, Haven |
| Implement color to grayscale filter | Feb 15 | Haven |
| Implement the data cache | Jan 30 | Gordon |

Next meeting: 5/1/2024/4pm/HKUST

Meeting 6:

Date: 5/1/2023

Time: 1pm

Location: Discord

Attendees: Haven, Issac, Gordon

Minutes taken by: Gordon

* The communication tutor has reminded us about some formatting mistakes we have in our draft of progress report, which deadline is 10/1/2024 11:59pm
* The implementation of Sobel filter is completed
* Issac has decided what model of FPGA board should be used and planned to seek advice from Professor Zhang in the next meeting.
* Gordon decided to finish the data cache after deciding which board will be used.

Table 1. Action Items from Previous Meeting

|  |  |  |  |
| --- | --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** | **Status** |
| Implementation of Sobel filter | Dec 30 | Issac, Haven | Completed |
| Implement the data cache | Jan 30 | Gordon | 10% |
| Implement color to grayscale filter | Feb 15 | Haven | 25% |

Table 2. Action Items for Next Meeting

|  |  |  |
| --- | --- | --- |
| **Action Item to be completed** | **By when** | **By whom** |
| Progress Report | 10 Jan | Group |
| Mid-term poster | 10 Jan | Group |
| Implement color to grayscale filter | Feb 15 | Haven |
| Implement the data cache | Feb 15 | Gordon |

Next meeting: 12/1/2024/4pm/HKUST

## **Appendix B — Group Members’ Contributions**

LUK, Pak Him (Issac)

Completed work:

* Research on median filter algorithm
* Research on spatial dataflow architecture
* Design of whole system incorporating spatial dataflow architecture
* Design of sequential median filter algorithm and its sub-modules
* Implementation of module *three\_sort* and *median\_filter\_scalable*
* Design of sequential Sobel filter algorithm and its sub-modules
* Implementation of module *sequential\_sobel* and *sobel\_filter\_scalable*
* Testbench of module *sequential\_sobel*
* Design of color to grayscale algorithm

Undergoing work:

* Preliminary Design and implementation of Otsu’s method
* Analysis of FPGA board used for prototyping

SHUM Kwan Ho (Gordon)

So far in this final year project, I have done literature reviews on hardware acceleration project that have been done or currently developing by other parties, including a review on CUDA, an Application Programming Interface developed by NVIDIA, which aims to perform data acceleration on their own GPU. At the early stage of this project, I have planned the working schedule together with my groupmates, which help us keep track of our process in these months.

After the start of fall semester, while dealing with course works, I have implemented the sequential median filter module designed by Issac in Verilog and used multiple sub modules to make pipelined dataflow, I have also used this design in the ELEC4320 project. I have written the module *sequential\_nine\_median,* sub module *column\_sort,* and their testbenches. This median filter module will be further developed to improve the performance and collaborate with other parts of our project.

I am responsible for taking the meeting minutes, while also drew most of the diagrams. In the following months I will implement the data cache after deciding which FPGA board will be used in our project.

Chio Yat Hei (Haven)

I have implemented the design of sobel filter X kernel and Y kernel sequentially. Which consisted of module sequential\_sobel\_X and sequential\_sobel\_Y. I have been studying ELEC4320 and working as a team with my FYP project groupmate Gordon for our course project. It is to implement an image recognition program as well. While dealing with course works I have been discussing and building the idea and model of the Sobel filter with Issac and sooner implement it in code . Other than putting the Sobel filter kernel in FYP project into this project I decide not to do so due to the bug in the filter of FYP is not solved. I have solved this problem after the course. sequential\_sobel\_X and sequential\_sobel\_Y as an upper module was designed and the lower module sequential\_sobel was designed by Issac. I have been working with him on the idea of designing the kernels and I in charge of implementing them. I was also in charge of poster designs. In the future I will be testing and record the result of this filter and modify it for improvement and will start implementation color to grayscale filter.

**Appendix C – Deviation(s) from the proposal and supporting reason(s)**

Page 3: Corrected some mistakes made in the literature review section in the proposal.

Page 8: Included introduction of spatial dataflow architecture

Page 13: Gordon and Haven have swapped the job of implementing median filter and Sobel filter.

Page 14: Schedule of Data cache for fetching and storing is delayed since the implementation of Sobel filter is not yet finished.

**Appendix D – Source code**

module three\_sort(

input [7:0] A\_in,

input [7:0] B\_in,

input [7:0] C\_in,

output [7:0] L\_out,

output [7:0] M\_out,

output [7:0] S\_out,

input clk

);

// intermediate registers

reg A\_GT\_B, B\_GT\_C, C\_GT\_A;

reg [7:0] tempA, tempB, tempC;

always @ (posedge clk) begin

A\_GT\_B <= A\_in > B\_in;

B\_GT\_C <= B\_in > C\_in;

C\_GT\_A <= C\_in > A\_in;

tempA <= A\_in; tempB <= B\_in; tempC <= C\_in;

end

assign L\_out = A\_GT\_B? (C\_GT\_A?tempC:tempA) :(B\_GT\_C?tempB:tempC);

assign M\_out = A\_GT\_B? (B\_GT\_C?tempB: (C\_GT\_A?tempA:tempC) ) :(B\_GT\_C? (C\_GT\_A?tempC:tempA) :tempB);

assign S\_out = A\_GT\_B? (B\_GT\_C?tempC:tempB) : (C\_GT\_A?tempA:tempC);

endmodule

module three\_sort\_tb(

);

reg [7:0] tb\_A\_in, tb\_B\_in, tb\_C\_in;

reg tb\_clk;

wire [7:0] tb\_L\_out, tb\_M\_out, tb\_S\_out;

reg correct;

three\_sort uut(.A\_in(tb\_A\_in), .B\_in(tb\_B\_in), .C\_in(tb\_C\_in), .L\_out(tb\_L\_out), .M\_out(tb\_M\_out), .S\_out(tb\_S\_out), .clk(tb\_clk));

integer i, j, k;

initial begin

for(k=0;k<256;k=k+1)begin

tb\_C\_in = k;

for(j=0;j<256;j=j+1)begin

tb\_B\_in = j;

for(i=0;i<256;i=i+1)begin

tb\_A\_in = i;

tb\_clk = 1'b0;

#5;

tb\_clk = 1'b1;

#5;

end

end

end

end

always @(\*) begin

if (tb\_L\_out>=tb\_M\_out && tb\_M\_out>=tb\_S\_out)

correct = 1'b1;

else

correct = 1'b0;

end

endmodule

module column\_sort(

input [7:0] A\_in,

output [7:0] L\_out,

output [7:0] M\_out,

output [7:0] S\_out,

input clk

);

reg [7:0] B\_in, C\_in;

three\_sort sorter (.A\_in(A\_in), .B\_in(B\_in), .C\_in(C\_in), .L\_out(L\_out), .M\_out(M\_out), .S\_out(S\_out), .clk(clk));

always @ (posedge clk) begin

B\_in <= A\_in;

C\_in <= B\_in;

end

endmodule

module column\_sort\_tb(

);

reg [7:0] tb\_A\_in;

reg tb\_clk;

wire [7:0] tb\_L\_out, tb\_M\_out, tb\_S\_out;

column\_sort dut (.A\_in(tb\_A\_in), .L\_out(tb\_L\_out), .M\_out(tb\_M\_out), .S\_out(tb\_S\_out), .clk(tb\_clk));

initial begin

tb\_clk = 1'b0;

forever #1 tb\_clk = ~tb\_clk;

end

initial begin

tb\_A\_in = 8'd0;

#2;

tb\_A\_in = 8'd9;

#2;

tb\_A\_in = 8'd15;

#2;

tb\_A\_in = 8'd12;

#2;

tb\_A\_in = 8'd8;

#2;

tb\_A\_in = 8'd10;

#2;

tb\_A\_in = 8'd7;

#2;

#2;

tb\_A\_in = 8'd7;

#2;

end

endmodule

module sequential\_nine\_median(

input [7:0] A\_in,

input [7:0] B\_in,

input [7:0] C\_in,

output [7:0] median\_out,

input clk

);

reg [7:0] row\_L\_reg, row\_M\_reg, row\_S\_reg, column\_LS\_reg, column\_MM\_reg, column\_SL\_reg;

wire [7:0] row\_L\_out, row\_M\_out, row\_S\_out, column\_LS\_out, column\_MM\_out, column\_SL\_out;

three\_sort row\_sorter(.A\_in(A\_in), .B\_in(B\_in), .C\_in(C\_in), .L\_out(row\_L\_out), .M\_out(row\_M\_out), .S\_out(row\_S\_out), .clk(clk));

column\_sort column\_sorter\_L (.A\_in(row\_L\_reg), .S\_out(column\_LS\_out), .clk(clk));

column\_sort column\_sorter\_M (.A\_in(row\_M\_reg), .M\_out(column\_MM\_out), .clk(clk));

column\_sort column\_sorter\_S (.A\_in(row\_S\_reg), .L\_out(column\_SL\_out), .clk(clk));

three\_sort diagonal\_sorter(.A\_in(column\_LS\_reg), .B\_in(column\_MM\_reg), .C\_in(column\_SL\_reg), .M\_out(median\_out), .clk(clk));

always @(posedge clk) begin

row\_L\_reg <= row\_L\_out;

row\_M\_reg <= row\_M\_out;

row\_S\_reg <= row\_S\_out;

column\_LS\_reg <= column\_LS\_out;

column\_MM\_reg <= column\_MM\_out;

column\_SL\_reg <= column\_SL\_out;

end

endmodule

module sequential\_nine\_median\_tb(

);

reg [7:0] tb\_A\_in, tb\_B\_in, tb\_C\_in;

reg tb\_clk;

wire [7:0] tb\_median\_out;

sequential\_nine\_median dut(.A\_in(tb\_A\_in), .B\_in(tb\_B\_in), .C\_in(tb\_C\_in), .median\_out(tb\_median\_out), .clk(tb\_clk));

initial begin

tb\_A\_in = 7'd0;

tb\_B\_in = 7'd1;

tb\_C\_in = 7'd2;

#2;

tb\_A\_in = 7'd3;

tb\_B\_in = 7'd4;

tb\_C\_in = 7'd5;

#2;

tb\_A\_in = 7'd6;

tb\_B\_in = 7'd7;

tb\_C\_in = 7'd8;

#2;

tb\_A\_in = 7'd0;

tb\_B\_in = 7'd1;

tb\_C\_in = 7'd2;

#2;

tb\_A\_in = 7'd3;

tb\_B\_in = 7'd4;

tb\_C\_in = 7'd5;

#2;

tb\_A\_in = 7'd6;

tb\_B\_in = 7'd7;

tb\_C\_in = 7'd8;

end

initial begin

tb\_clk = 0;

forever #1 tb\_clk = ~tb\_clk;

end

endmodule

module median\_filter\_scalable

# (parameter SIZE = 10) ( // change SIZE to vary system size

input [7:0] arr\_in [SIZE-1:0],

output [7:0] arr\_out [SIZE-3:0],

input clk

);

genvar i;

generate

for (i=1;i<SIZE-1;i=i+1) begin

sequential\_nine\_median median(.A\_in(arr\_in[i-1]), .B\_in(arr\_in[i]), .C\_in(arr\_in[i+1]), .median\_out(arr\_out[i-1]), .clk(clk));

end

endgenerate

endmodule

module sequential\_sobel\_X(

input [7:0] current\_in,

input [9:0] left\_intermediate,

input [9:0] right\_intermediate,

output [9:0] current\_intermediate,

output [9:0] sobel\_X\_out,

input clk

);

reg [7:0] shift\_add\_reg [1:0];

reg [9:0] add\_reg;

reg [10:0] sobel\_X\_reg;

wire [8:0] shift\_add\_reg\_0\_shift;

always @(posedge clk) begin

shift\_add\_reg[0] <= current\_in;

shift\_add\_reg[1] <= shift\_add\_reg[0];

add\_reg <= current\_in + shift\_add\_reg\_0\_shift + shift\_add\_reg[1];

sobel\_X\_reg <= right\_intermediate - left\_intermediate;

end

assign shift\_add\_reg\_0\_shift = {shift\_add\_reg[0], 1'b0};

assign current\_intermediate = add\_reg;

assign sobel\_X\_out = (sobel\_X\_reg[10] == 1)?~sobel\_X\_reg[9:0]:sobel\_X\_reg[9:0];

endmodule

module sequential\_sobel\_X\_tb(

);

reg [7:0] tb\_current\_in;

reg [9:0] tb\_left\_intermediate, tb\_right\_intermediate;

reg tb\_clk;

wire [9:0] tb\_sobel\_X\_out;

wire [9:0] tb\_current\_intermediate;

initial begin

tb\_clk = 1'b0;

forever #1 tb\_clk = ~tb\_clk;

end

initial begin

tb\_current\_in = 8'd5; tb\_left\_intermediate = 10'd5; tb\_right\_intermediate = 10'd10;

#2 tb\_current\_in = 8'd10;

#2 tb\_current\_in = 8'd15; tb\_left\_intermediate = 10'd15; tb\_right\_intermediate = 10'd10;

end

sequential\_sobel\_X dut(.current\_in(tb\_current\_in), .current\_intermediate(tb\_current\_intermediate), .left\_intermediate(tb\_left\_intermediate), .right\_intermediate(tb\_right\_intermediate), .sobel\_X\_out(tb\_sobel\_X\_out), .clk(tb\_clk));

endmodule

module sequential\_sobel\_Y(

input [7:0] current\_in,

input [7:0] left\_in,

input [7:0] right\_in,

output [9:0] sobel\_Y\_out,

input clk

);

wire [8:0] current\_in\_shifted;

reg [9:0] add\_reg [2:0];

reg[10:0] sobel\_Y\_reg;

always @ (posedge clk)begin

add\_reg[0] <= current\_in\_shifted + left\_in + right\_in;

add\_reg[1] <= add\_reg[0];

add\_reg[2] <= add\_reg[1];

sobel\_Y\_reg <= add\_reg[2] - add\_reg[0];

end

assign current\_in\_shifted = {current\_in, 1'b0};

assign sobel\_Y\_out = (sobel\_Y\_reg[10]==1)?~sobel\_Y\_reg[9:0]:sobel\_Y\_reg[9:0];

endmodule

module sequential\_sobel\_Y\_tb(

);

reg [7:0] tb\_current\_in, tb\_left\_in, tb\_right\_in;

wire [9:0] tb\_sobel\_Y\_out;

reg tb\_clk;

sequential\_sobel\_Y dut(.current\_in(tb\_current\_in), .left\_in(tb\_left\_in), .right\_in(tb\_right\_in), .sobel\_Y\_out(tb\_sobel\_Y\_out), .clk(tb\_clk));

initial begin

tb\_clk = 1'b0;

forever #1 tb\_clk = ~tb\_clk;

end

initial begin

tb\_left\_in = 1; tb\_current\_in = 2; tb\_right\_in = 3;

#2

tb\_left\_in = 4; tb\_current\_in = 5; tb\_right\_in = 6;

#2

tb\_left\_in = 0; tb\_current\_in = 0; tb\_right\_in = 0;

#2

tb\_left\_in = 14; tb\_current\_in = 15; tb\_right\_in = 16;

end

endmodule

module sequential\_sobel(

input [7:0] current\_in,

input [7:0] left\_in, right\_in,

input [9:0] left\_intermediate, right\_intermediate,

output [9:0] current\_intermediate,

output [7:0] sobel\_out,

input clk

);

wire [9:0] sobel\_X\_out, sobel\_Y\_out;

reg [9:0] sobel\_X\_reg, sobel\_Y\_reg;

reg [10:0] sobel\_reg;

sequential\_sobel\_X X\_filter(.current\_in(current\_in), .left\_intermediate(left\_intermediate), .right\_intermediate(right\_intermediate),

.current\_intermediate(current\_intermediate), .sobel\_X\_out(sobel\_X\_out), .clk(clk));

sequential\_sobel\_Y Y\_filter(.current\_in(current\_in), .left\_in(left\_in), .right\_in(right\_in), .sobel\_Y\_out(sobel\_Y\_out), .clk(clk));

always @(posedge clk) begin

sobel\_X\_reg <= sobel\_X\_out;

sobel\_Y\_reg <= sobel\_Y\_out;

sobel\_reg <= sobel\_X\_reg + sobel\_Y\_reg;

end

assign sobel\_out = sobel\_reg[10:3];

endmodule

module sequential\_sobel\_tb(

);

reg [7:0] tb\_current\_inout, left\_in, right\_in;

reg [9:0] tb\_left\_intermediate, tb\_right\_intermediate;

wire [9:0] current\_intermediate;

wire [7:0] tb\_sobel\_out;

reg tb\_clk;

sequential\_sobel dut(.current\_inout(tb\_current\_inout), .left\_in(tb\_left\_in), .right\_in(tb\_right\_in),

.left\_intermediate(tb\_left\_intermediate), .right\_intermediate(tb\_right\_intermediate), .current\_intermediate(tb\_current\_intermediate),

.sobel\_out(tb\_sobel\_out), .clk(tb\_clk));

initial begin

tb\_clk = 1'b0;

forever #1 tb\_clk = ~tb\_clk;

end

endmodule

module sobel\_filter\_scalable

# (parameter SIZE = 5) ( // change SIZE to vary system size

input [7:0] arr\_in [SIZE-1:0],

output [7:0] arr\_out [SIZE-3:0],

input clk

);

wire [7:0] intermediate [SIZE-1:0];

genvar i;

generate

for(i=0; i<SIZE; i=i+1) begin

sequential\_sobel sobel(.current\_in(arr\_in[i]), .left\_in(arr\_in[i-1]), .right\_in(arr\_in[i+1]),

.current\_intermediate(intermediate[i]), .left\_intermediate(intermediate[i-1]), .right\_intermediate(intermediate[i+1]),

.sobel\_out(arr\_out[i-1]), .clk(clk));

end

endgenerate

endmodule