FPGA Implementation of a Median Filter

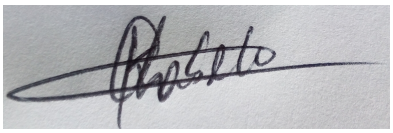
Prepared by:
Best Nkumeleni (NKHBES001)
Kananelo Chabeli (CHBKAN001)
Malefetsane Lenka(LNKMA001)
Rumbidzai Mashumba (MSHRUM006)

Prepared for:
EEE4120F
Department of Electrical Engineering
University of Cape Town

December 22, 2024

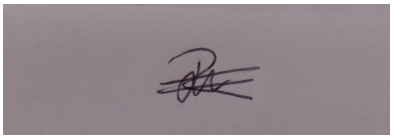
Declaration

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the IEEE convention for citation and referencing. Each contribution to, and quotation in, this report from the work(s) of other people has been attributed, and has been cited and referenced.
3. This report is my own work.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as their own work or part thereof.



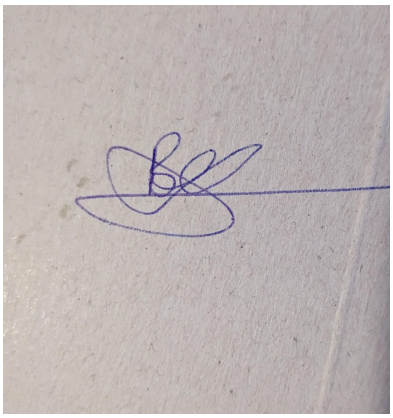
December 22, 2024

Kananelo Chabeli — Date



December 22, 2024

Rumbidzai Mashumba — Date



December 22, 2024

Best Nkhumeleni

Date

m.lenka

December 22, 2024

Malefetsane Lenka

Date

Contents

List of Figures

Chapter 1

Introduction

1.1 Background Theory

Advancement in technology has positioned image processing as crucial component of several fields such as medical imaging machine learning, automotive, and computer vision. Despite their broad application, image processing algorithms face several challenges. The most common type of noise that corrupt images is called salt-and-pepper or spiky noise. This noise is often incurred during digital image acquisition where by some pixels in the image are digitized to the two extremes, maximum and minimum [2],[1]. The need to remove noise in image is thus critical because noise corrupts the useful information in the image [2].

There have been many image processing algorithms that have been proposed in the literature and are generally classified as linear and nonlinear [2]. Nonlinear filtering algorithms are often desired because of their robustness and denoising power [1]. Median filtering is a widely used technique for its ability to preserve important image features while reducing the effects of noise. The filter replaces each pixel value with the median value of neighbouring pixels in a chosen window.Figure 1.1 illustrates this algorithm.

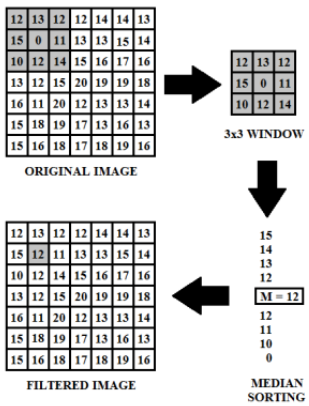


Figure 1.1: Illustration of median filter [1]

.

In [2], the authors investigates the effect of window size on denoising. They found that for small window sizes, the filter performed well for low noise densities but poorly when noise density was high.

Also, as can be seen from figure 1.1, the filter involves several mathematical operations such as sorting and two-dimensional computation. These operations usually require excessive computing speeds and memory. Thus, FPGA implementation is a crucial alternative for the implementation of image filtering algorithms in real life. Authors in [1] argue that SRAM-based FPGA offer the highest performance.

This report presents an in-depth analysis of the median filtering algorithm, its implementation in hardware, and the comparative performance evaluation across different computational platforms. Through this exploration, we aim to provide insights into the efficacy and versatility of hardware-accelerated median filtering in addressing noise reduction challenges within the realm of digital imagery.

1.2 Objectives

Therefore, the objectives of the project are to:

- Quantify the performance gains achieved by FPGA-accelerated median filter in comparison to a software-based and OpenCV-based implementation.
- Implement and optimize median filter and edge detection algorithm using Verilog on FPGA or simulated.
- Evaluate the performance of the median filter in terms of processing speed, and overall image quality improvement.

1.3 Scope & Limitation

The project has the following scope:

- The report is based on FPGA implementation of median filter
- The solution must be run on FPGA or simulated.
- The solution must also be capable of handling PNG, JPEG,n and JPG images of arbitrary size.
- The solution must realize the filter with a window approach. The window size must be left adjustable user parameter.

The project has the following limitations:

- The report must be completed in four weeks.
- The solution will be implemented Nexys A7 100T FPGA board.
- There is no financial provision for this project.

1.4 Report Outline

The report starts by giving a comprehensive literature review in chapter 2 followed by prototype design in chapter 3. The simulation and benchmark results are provided in chapter 4 from which conclusions are drawn and recommendations are made based on the conclusions in chapter 6.

1.4.1 git repo

All the code reported on can be found here:

https://github.com/BestNkhumeleni/Yoda$_p roject$

Chapter 2

Literature Review

Image filtering is a fundamental operation in image processing, used for various tasks such as noise reduction, edge enhancement, and image sharpening. There exists a broad array of filtering techniques, which can be broadly categorized into linear and non-linear filters. This review explores the landscape of image filtering, focusing particularly on median filtering and its implementation on (FPGA).

2.1 Image Filtering Techniques

2.1.1 Linear Filtering Techniques

Mean Filter

The mean filter, also known as an averaging filter, is a simple linear filter used primarily for reducing noise and smoothing images. It replaces a pixel's value with the average of its neighbors. While effective for noise reduction, it tends to blur sharp edges [3].

Gaussian Filter

A Gaussian filter is a type of linear filter used for smoothing (blurring) images and reducing noise. It applies a convolution operation with a Gaussian function, which weights neighboring pixels according to a normal distribution, offering smoother noise reduction but with an edge-blurring effect [4].

Wiener Filter

The Wiener filter is an adaptive filter used for noise reduction and signal restoration. It operates on the principle of minimizing the mean square error between the estimated and desired signals. While it is statistically optimal for specific noise models, it requires knowledge of the noise characteristics, which may not always be available [5].

2.1.2 Non-Linear Filtering Techniques

Median Filter

The median filter is a non-linear digital filtering technique commonly used to remove noise while preserving edges. Unlike linear filters such as the mean filter, the median filter replaces each pixel's value with the median value of the neighboring pixels. This approach effectively removes outliers and noise while maintaining the overall structure and detail of the image [6].

2.2 Median Filter Typologies

The standard median filter is a type of non-linear filter that sorts pixel values within a neighborhood defined by the filter mask and replaces the central pixel with the median value of the sorted group [7]. An improvement on the standard median filter is the multi-level median filter, introduced by Nieminen and Neuvo, which enhances the noise reduction properties of the standard median filter [8].

2.3 FPGA Realization of Median Filter

Bates and Nooshabadi demonstrated that their FPGA implementation could process an incoming video data stream at a maximum rate (partially dependent on the programmable logic device) of around 30 MS/s [9]. They used a Virtex-II Pro development board with an xc2vp30ff896-6 device to implement and test their FPGA implementation, finding that the standard median filter consumed nearly twice as many resources as the multi-level median filter [7].

2.4 Conclusion

Image filtering is crucial in image processing, with median filtering standing out for its edge-preserving noise reduction capabilities. FPGA implementations provide a hardware-centric approach for efficient median filter execution, and ongoing research continues to explore further optimization techniques. As the demands of image processing grow, the development of more sophisticated filtering methods and their hardware implementations remains an active area of research.

Chapter 3

System Design

3.1 System Specifications

Based on the objectives in chapter **??**, the following are specifications of the prototype.

- Must write and read image files to and from memory.
- Must median filter input image file given the window size.

3.2 Prototype Design

To meet above specifications, the median filtering algorithm was designed and implemented in Verilog,and simulated it running on FPGA. The overall median filtering system is divided into three modules(Memory, Controller,and Filter) and operates in two main states(IDLE or RUN). The following sections provide detailed description of each module.

3.2.1 Controller Module

The Controller module serves as the central managing unit of the system. It determines the timing for the start of filtering, when to read pixels, and the memory addresses from which to read. The primary objective of this module is to compute the memory addresses for image pixels and write the filtered pixels to the output file. The module is configured with the following parameters:

```
parameter DATA_WIDTH = 24;
parameter BUS_WIDTH = 32;
parameter WINDOW_SIZE = 3;
parameter IMAGE_WIDTH = 512;
parameter IMAGE_HEIGHT = 512;
parameter OUTPUT_FILENAME = "../imgHex.txt";
```

By default, these parameters are set to handle a 512x512 RGB image. The Controller communicates with the Memory module using `MEM_ADDR` and `MEM_RW`. The `MEM_ADDR` is a `BUS_WIDTH`-bit wide address bus that specifies the address from which the **Filter** module reads pixel values. Moreover, `MEM_RW` is a 2-bit signal that determines whether to read ($0b10$) or write ($0b11$).

Additionally, the Controller interfaces with the **Filter** module through three lines: `FILT_DATA`, which carries the output filtered pixel value; `FILT_DNE`, a signaling line indicating that the filtering is complete

and the data on FILT_DATA line is valid; and FILT_EN, an enable line that keeps the Filter module active as long as this line is set HIGH. Figure 3.1 shows the state machine of the Controller module, which

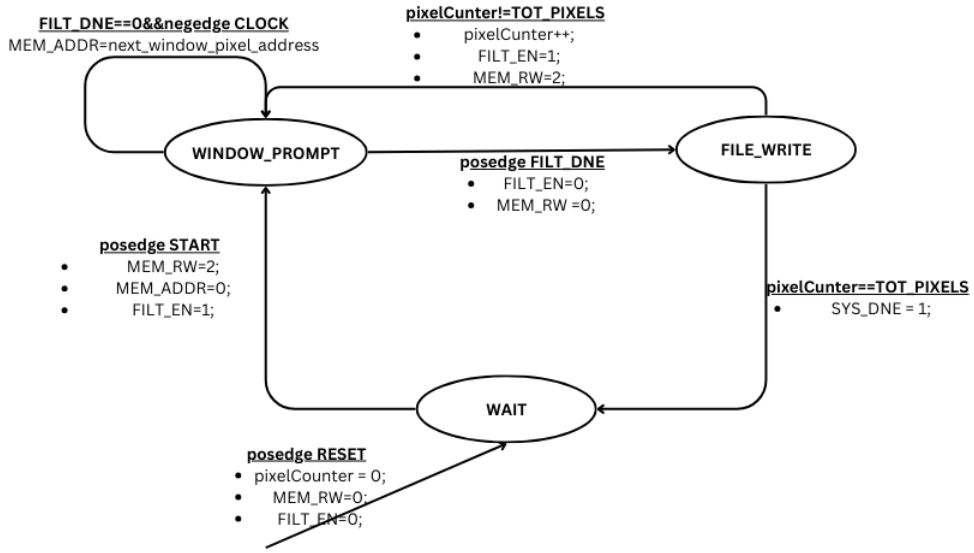


Figure 3.1: State Machine of the Controller Module

operates in three states: WAIT, WINDOW_PROMPT, and FILE_WRITE. The module enters the WAIT state upon reset and remains there until a pulse is received on the START line. Upon detecting a positive pulse on START, the Controller moves into the WINDOW_PROMPT state, where it calculates the memory address for the next pixel in the filtering window until all window pixels have been read by the **Filter Module**. After exiting this state, it transitions to FILE_WRITE upon receiving a positive pulse on FILT_DNE. In this state, it disables the filter, captures the value from FILT_DATA, and writes it to the output file. The Controller remains in the FILE_WRITE state for approximately one clock cycle, during which it calculates the address of the next pixel to be filtered, if any, re-enables the filter, or sends a positive pulse on SYS_DNE to signal that the entire image has been filtered. The methodology for calculating pixel addresses is described later in this chapter.

3.2.2 Memory Module

The Memory module is an array-based RAM implementation whose size is adjustable based on the image size. It is declared with the following syntax: `module Memory(Mem_CLK, Mem_RST, Mem_RW, Mem_ADDR, Mem_IDR, Mem_ODR, Mem_DRDY);` The module has the following parameters:

```
// Defaults to 24 bits (for a three-channel image)
parameter DATA_WIDTH = 24;
// Defaults to 512512 pixels.
parameter DEPTH = 512512;
// Width of the address, defaults to 32-bit bus address
parameter ADDR_WIDTH = 32;
parameter FILENAME = "test_hex.txt";

reg [DATA_WIDTH-1:0] MEMORY [0:DEPTH-1]; // MEMORY BLOCK
```

This module is designed to load the image file specified by `FILENAME` into the `MEMORY` buffer upon reset. Read and write operations are processed at the rising edge of `Mem_CLK` depending on the value of `Mem_RW`. The READ operation is indicated by setting `Mem_RW` to `0b10` during the rising edge of the clock, while the WRITE operation is indicated by `0b01`. `Mem_ADDR` is the address bus used for reading or writing data. During a READ operation, the requested data is loaded onto `Mem_ODR` followed by a pulse on `Mem_DRDY` which the Filter module uses to read pixel values.

3.2.3 Filter Module

The Filter module carries the actual median filtering process and is declared with: `module Filter(Filt_CLK, Filt_RST, Filt_EN, Filt_MEMRDY, Filt_MEMDATA, Filt_RES, Filt_DNE);` and has parameters:

```
parameter DATA_WIDTH = 24;
parameter WINDOW_SIZE =3;
```

The module operates in three states:`WAIT`, `WINDOW_READ` and `WINDOW_SORT`. The state diagram of the module is shown in figure 3.2.

The `WAIT` state is entered upon reset and remains there until a pulse is received on `Filter_EN` input

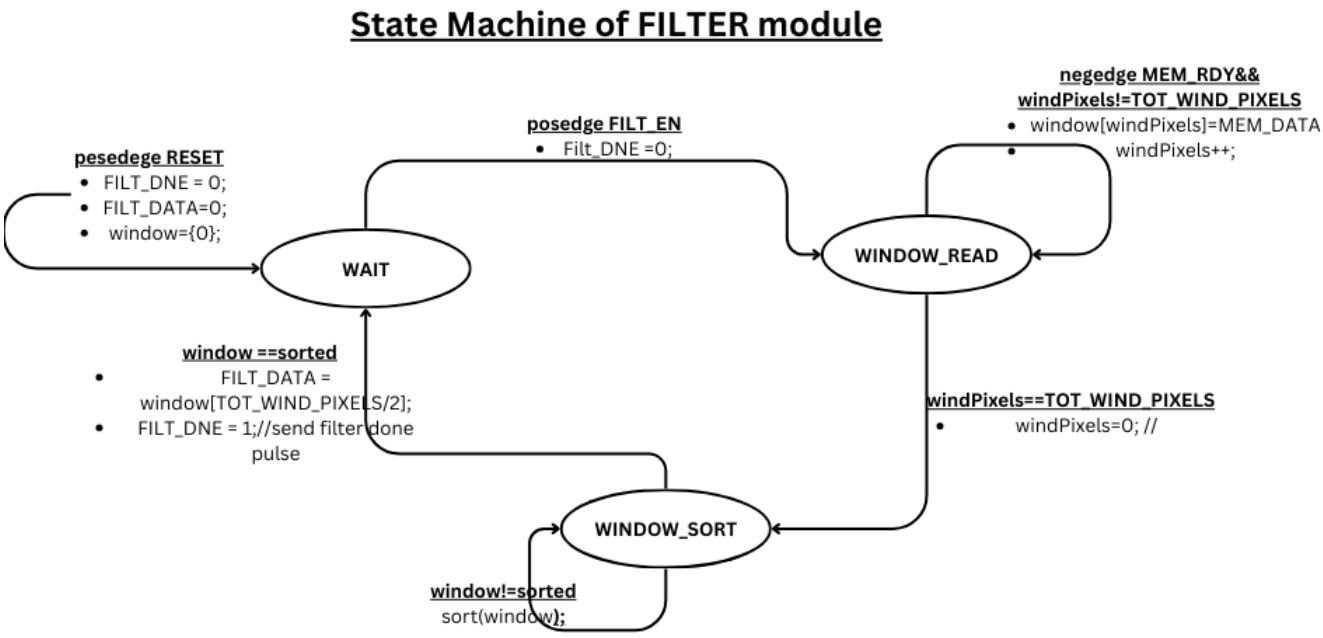


Figure 3.2: State Machine of the Filter Module

port in which case it transition into `WINDOW_READ`. The module remains in this state and reads window pixel values from memory on each positive pulse of `MEM_DRDY` line. This data is transmitted from Memory using `Filt_MEMDATA` data bus and added into `window` internal buffer,until all window pixels has been read. The system then transitions into `WINDOW_SORT` in which case it sorts the `window` buffer and extracts the median value. This median value is loaded onto `Filt_RES` data bus and a positive pulse is sent on `Filt_DNE`. The window buffer is sorted using bubblesort and is implemented as follows:

```
function sortWindow(input integer mxm);
reg [DATA_WIDTH-1:0] temp;
```

```
integer k, j ; // iterators
begin
    for( k = 0;k< WINDOW_SIZE*WINDOW_SIZE-1; k= k+1) begin
        for(j=0; j < WINDOW_SIZE*WINDOW_SIZE-1-k; j= j+1) begin
            if(window[j] > window[j+1]) begin
                temp = window[j];
                window[j] = window[j+1];
                window[j+1] = temp;
            end
        end
    end
    sortWindow =0;
end
endfunction
```

3.2.4 Integrated System

In this section we provide details of how the aforementioned modules are integrated together achieve the outlined system objectives. The overall system block is shown in figure 3.3. The system is started

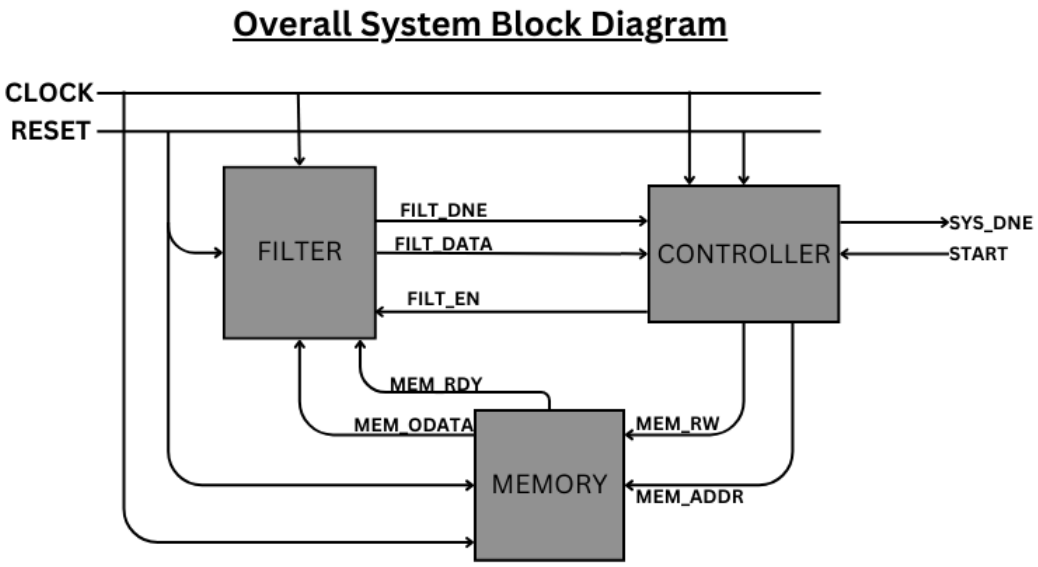


Figure 3.3: Overall System Block Diagram

by sending a positive pulse on `START` line. **Controller** then computes the address of the starting pixel of the filter window, and sends a pulse on `FILT_EN` line to enable the filter. The each subsequent negative edge of the clock, the **Filter** module reads the data value on `MEM_ODATA` bus upon receipt of a pulse on `MEM_RDY`. At the same time, the **Controller** module computes the address of the next pixel in that particular window. When all pixels within the window has been read by **Filter** module, it then sorts the window pixel and writes the resulting median pixel on `FILT_DATA` data bus,and sends indication to the Controller on `FILT_DNE` line. The controller then grabs the value on `FILT_DATA` and writes it to the output file given at the start of the simulation.

3.3 FPGA

This module was designed to transmit filtered image data from FPGA to the PC at a 9600 baud rate or 115200 baud rate. The UART module has 4 states: IDLE, START, SEND_BIT, and STOP. The machine remains in the IDLE state if the data available flag is low. In the START state, the TX line is held at 0 for a number of clock cycles per bit. Then follows SEND_BIT, where a byte to be transmitted is sent bit by bit. In the STOP state, the TX line is held high for a number of clocks per bit, and then the machine goes back to IDLE.
Figure 3.4 shows the state transition diagram.

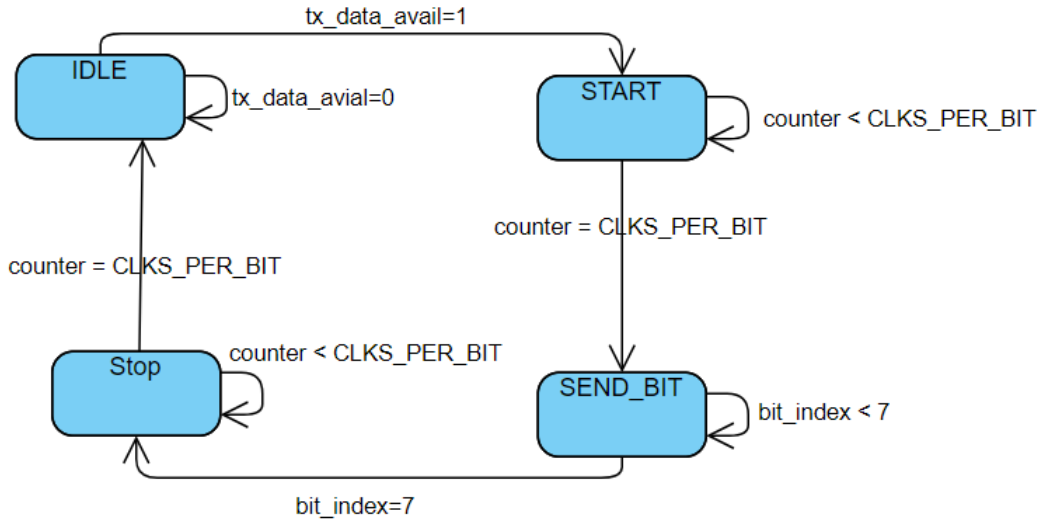


Figure 3.4: UART transmitter state machine

3.4 Performance Evaluation

This process was carried out to evaluate the performance of Verilog implementation of median filter in comparison to a well-established golden standard. The evaluation concentrated on two main aspects: the accuracy and the speed of the filtering algorithm.

3.4.1 Benchmarking Suit

Golden measure

The chosen golden measure for benchmarking was OpenCV. It is a widely used image-processing library. Using OpenCV provided optimised and robust image processing algorithms. This made it an ideal reference point.

Rationale for chosen Golden Standard

The following features are the reason OpenCV was chosen:

- OpenCV is well tested and it has been widely used in academia and in the industry. It has been proven to be accurate and reliable.
- OpenCV has been optimised and this allows it to provide a high-performance baseline for comparison.
- OpenCV is easy to use. This is because a lot of supporting resourcing are available for use. This ease of use made the benchmarking efficient.

Chapter 4

Testing and Results

4.1 Unit Test Procedures

This sections details the unit test procedure carried out to ensure that each individual module works as intended.

4.1.1 UTP01: Memory Read and Write

This test procedure is aimed at validating that memory module can be written and be read from. The expectation is that any write or read request should be serviced by the memory module on the rising edge of the clock. Additionally, for a read request, the module should send `MEM_DRDY` pulse to indicate the data has been retrieved from memory. The test procedure is passed if, for a write operation, the value on `MEM_IDR` is written to memory address value on `MEM_ADDR` bus. For read operation, this test is passed if the value in, memory at address `MEM_ADDR` is loaded onto `MEM_ODR` data bus, and a pulse is send non `MEM_DRDY` line.

Testing Procedure: Create *Memory_tb.v* file which will be used as a testbench for testing *Memory.v* module. Instantiate the module and set the parameters appropriately. write a series of data streams to memory and read the back. Analyse the results on gtkwave forms to confirm if they are as expected.

4.1.2 UTP02: Filter read window pixels from Memory

This test procedure is designed to ensure that the *Filter.v* module can read window pixels from memory. It should be conducted after the passing of UTP01. The test is passed if pixels within the window of the specific centre pixel can be read from memory.

Testing Procedure Create a separate testbench called *Filter_tb.v* and in it, instantiate *Memory.v* and *Filter.v* modules. Load memory with data from external data files. Analyse signal lines on gtkwave analyse to confirm that it runs as expected.

4.1.3 UTP03: Sort Window Pixels

This procedure is a follow-up procedure carried out immediately after a pass on UTP02. It has been designed to validate if the *Filter.v* can correctly sort pixels in the window buffer read from the memory. Thus, it must be conducted after a pass of UTP02.

Testing Procedure With the same testbench used for UTP02, add a the following for loop in *Filter.v* module after the sorting the `window` buffer:

```
for(i =0; i< TOT_WIND_PIXELS; i= i+1)begin
    \$display("\%x th pixel: \%x",i,window[i]);
end
```

Observe the pixels as displayed on the console if they appear in a sorted manner.

4.1.4 UTP04: Select Median Pixel

This test procedure was designed to validate the *Filter.v* module can accurately select the median window pixel. It should be carried out after a pass to both UTP02 and UTP03. The test is passed if the data written on `Filt_DATA` is the median pixel and a pulse is sent on `Filt_DNE`.

Testing Procedure Do the test procedures as in UTP02 and UTP03. Use gtkwave analyser to confirm that data written on `Filt_DATA` data bus is the median value, and that there is a pulse on `Filt_DNE` line.

4.1.5 UTP05: UART transmitter

This test procedure is designed to validate that the FPGA can transmit data to the pc.

Testing procedure Test the module in simulation using Vivado. Then write a Python script to read data. Use SW0 to SW7 on Nexys A7-100T as the data byte and SW8 as data available flag. Synthesise the code, rum implementation, generate bitstream and load bit files to FPGA. Run Python code and observe output. FPGA must be running at `100MHz` and UART baud rate should be `9600bps`.

4.1.6 Summary of Unit Test Procedures

Table 4.1 provides the summary of the unit test procedures described above.

Table 4.1: Summarising Unit Test Procedures

Test ID	Description	Pass/Fail
UTP01	Memory Read/Write	pass
UTP02	Read window pixels	pass
UTP03	Sort window pixels	pass
UTP04	Compute and Return Median	pass
UTP05	Send data to pc via UART	pass

4.2 Integration Test Procedures

The following integration test procedures are designed to validate the integration of the system as a whole, validating that is filters the image as expected.

4.2.1 ITP01:Filtering Synthesized Image

This test procedure was designed to validate that the system filters the image by using a custom image of 5x5 size and a window size of 3. This synthesized dataset is shown in table 4.2.

Testing Procedure: Load the data in table 4.2 into a text file and read it into the image using `readmemh()` system functions. Within a testbench *Integrate_tb.v* instantiate all modules and collect simulation data, dumping it to the external dump file. Analyse this data to ensure that the signal line carry data at expected times.

4.2.2 ITP02: Filtering Real-Time Image

The test procedure was developed to assess if the implemented modules can filter the real-time image. To be able to process edge pixels effectively, the image processed needs to be zero-padded with a layer of zeros depending on the filtering window size. Additionally, Verilog does not have a system function that can read the image file as it is. Thus, we first needed to convert the image file from a file format of, PNG for example, into a hexadecimal text file. This was done by an additional C++ program. This test is passed if the filtered image is correctly formatted as compared to filtering made by the golden standard or any well-known filtering tool.

Testing Procedure: The testing procedure is the same at ITP01. However, the real image might be too big to analyse on the waveform analyser,as such the final image pixels are written to an external file and converted back to a PNG image using a C++ program developed.

4.3 Benchmarking Test Procedures

4.3.1 BTP01: Golden Standard Timing

Implementing OpenCV and the median filter was done on the same standard PC. This procedure was developed to obtain the benchmarking results of the selected golden standard.

Testing Procedure: The following is the procedure followed to obtain timing data of the openCV median filtering.

- Image loading using the *imread* function.
- Median filtering using the *medianBlur* function.
- Execution time Measurement is done by incorporating a timer into the code.
- Output Verification to see if the image is changing as expected with different filtering windows.

4.3.2 BTP02:Verilog Simulation Timing

BTP02 is a benchmarking test procedure developed to obtain the timing data of Verilog simulation of the median filter.

Testing Procedure: The following steps were taken:

- Image loading using using the memory module mentioned in chapter 3.
- Median filtering using the filtering module mentioned in 3.
- Timing was done by measuring the number of clock cycles taken to filter an image and multiplying by time unit of 1ns to find simulation time in seconds.
- Output Verification to see if the image is changing as expected with different filtering windows.

4.4 Test Data

Table 4.2 shows a small 5x5 3-channel image used for unit testing of the median. This image data is loaded into the memory at compile time, and read by the filter and controller modules to filter some of the pixels. The largest window size that can be used is 5, with centre pixel of table 4.2 being the pixel filtered.

Table 4.2: 5x5 image pixels used for testing

	0	1	2	3	4
0	defe45	33efe1	edeeff	defffc	23deda
1	acbeff	ddedf1	34fede	ef342e	23dec5
2	efefe1	fefe4f	defeb2	cedefe	23efec
3	edfe34	edecee	2343fd	ddfecd	23fbcd
4	45defe1	edfed3	efede1	dbcd23	de2314

4.5 Unit Test Results

4.5.1 Controller Module Testing

Figure 4.1: Controller Unit test

Figure 4.1 shows the input and output registers of the controller change when:

The controller is initialized:

```
// Initialize Inputs
        Control_RST = 0;
        Control_STRT = 0;
        Control_FDNE = 0;
        Control_FDATA = 0;
```

The controller is reset:

```
// Reset the module
        #(CLOCK_PERIOD);
        Control_RST = 1;
        #(CLOCK_PERIOD);
        Control_RST = 0;
```

When the filtering process has been started:

```
// Start the filtering process
        #(CLOCK_PERIOD);
        Control_STRT = 1;
        #(CLOCK_PERIOD);
        Control_STRT = 0;
```

Send a done signal to the controller:

```
// Send a done signal to the controller
        #(CLOCK_PERIOD);
        Control_FDNE = 1;
        #(CLOCK_PERIOD);
        Control_FDNE = 0;
```

4.5.2 Filter Module Test

Figure 4.2 shows how the output and input registers and wires of the filter change as and when these factors are changed:

```
// Initialize Inputs
        Filt_RST = 0;
```

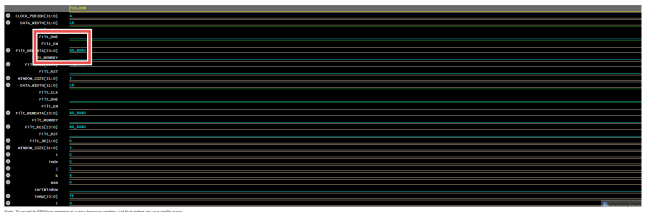


Figure 4.2: Filter unit test

```
        Filt_EN = 0;
        Filt_MEMRDY = 0;
        Filt_MEMDATA = 0;

        // Apply reset
        #(CLOCK_PERIOD);
        Filt_RST = 1;
        #(CLOCK_PERIOD);
        Filt_RST = 0;

        // Enable the filter
        #(CLOCK_PERIOD);
        Filt_EN = 1;

        // Send window data
        send_window_data;
```

After a filter is applied the data is sent through the send window wire. The highlighted section in figure 4.2 shows when the filter is enabled.

Memory Module Test

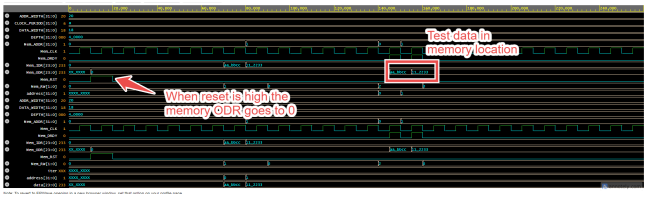


Figure 4.3: Memory unit test

```
// Initialize Inputs
        Mem_RST = 0;
        Mem_RW = 0;
        Mem_IDR = 0;
        Mem_ADDR = 0;

        // Dump waveforms to VCD file
        $dumpfile("dump.vcd");
        $dumpvars(0, memoryTB);

```

```
        // Apply reset
        #(CLOCK_PERIOD);
        Mem_RST = 1;
        #(CLOCK_PERIOD);
        Mem_RST = 0;

        // Wait for memory to initialize
        #(5 * CLOCK_PERIOD);

        // Write data to memory
        write_memory(32'h00000000, 24'hAABBCC);
        write_memory(32'h00000001, 24'h112233);

        // Wait for write operations to complete
        #(5 * CLOCK_PERIOD);

        // Read data from memory
        read_memory(32'h00000000);
        read_memory(32'h00000001);
```

In the figure above (4.5.2) dummy data is written into the the memory and then read to simulate information transfer.

4.6 UART Test Results

4.6.1 UART Simulation

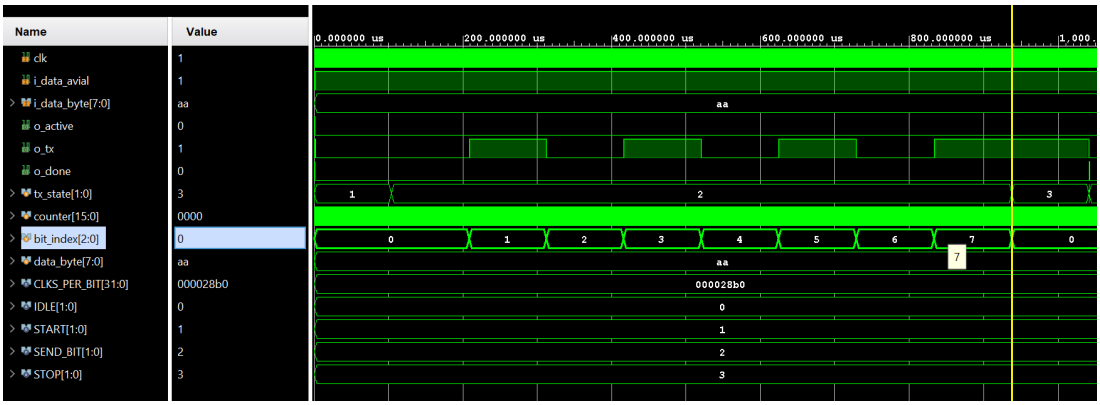


Figure 4.4: UART transmitter simulation

Figure 4.4 shows the UART transmitter simulation results in Vivado. This figure shows that state transition is occurring correctly as data `0xAA` is transmitted correctly when the data available flag is high. Also, the data is transmitted least significant bit first as per UART protocol data frame specification.

4.6.2 FPGA verification results

Python UART transmitter test code

```
# Used to test uart_tx module
# better compared to serial terminal
import serial
# Configure the serial port
ser = serial.Serial(
    port='COM4',   # Change this to your UART port, e.g., COM1, COM2, etc.
    baudrate=9600, # Change this to match your UART baudrate
    timeout=1      # Set timeout to 1 second
)
try:
    while True:
        # Read one byte from UART
        data = ser.read(1)

        # Check if data is received
        if data:
            # Convert the byte to integer (LSB first)
            value = int.from_bytes(data, byteorder='little')
            print("Received:", value)
        else:
            print("No data received")

except KeyboardInterrupt:
    ser.close()  # Close the serial port
```

Figure 4.5 shows FPGA configuration when the data available flag is held low as illustrated by the red highlighted switch. Python test program output in figure 4.6 verifies that there is do data being transmitted when the flag is held low. Moreover, UART receiver led (the receiver from USB to UART chip which is connected to FPGA transmitter) is off hence verifying that UART is in an idle state. FPGA successfully transmitted `0b11111111` indicated by SW0-SW7 on figure 4.7. Also UART status led is on. Finally python output on figure 4.8 verifies that number `255` which is `0b11111111` in binary is recieved.

4.7 Integration Results

This section presents results obtained for carrying on integration test procedures discussed in section 4.2. The image data shown in table 4.2 was written to a text file and padded with a layer of zeros suitable for filtering with a window size of 3x3.

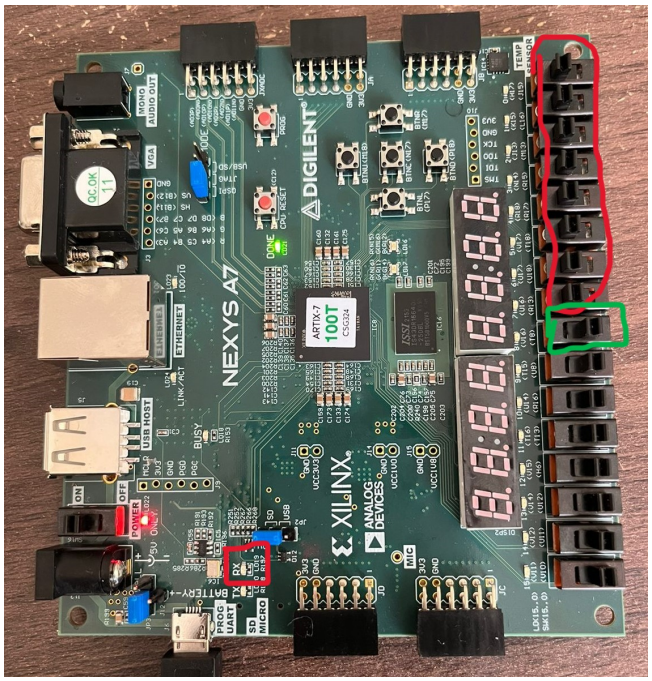


Figure 4.5: FPGA when SW8 is off

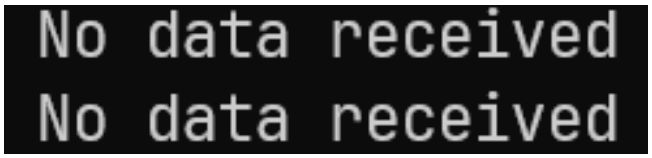


Figure 4.6: Python output when SW8 is off

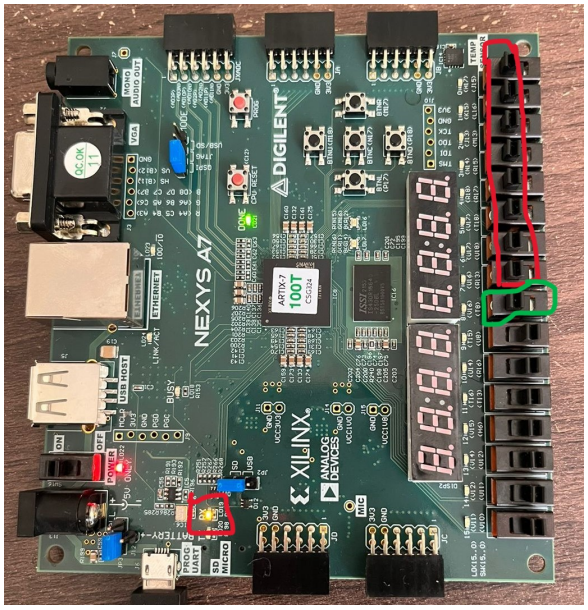


Figure 4.7: FPGA when SW8 is on

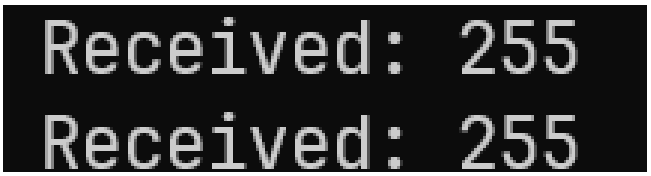


Figure 4.8: Python output when SW8 is on

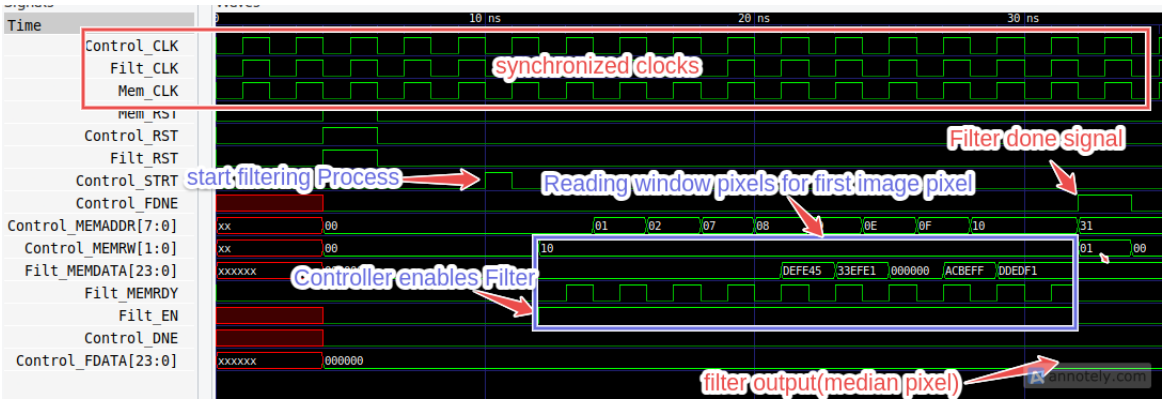


Figure 4.9: Filtering of 1st Image Pixel

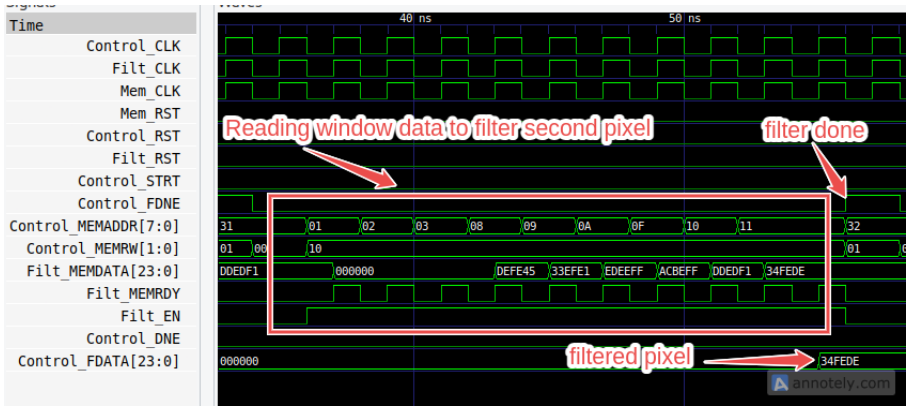


Figure 4.10: Filtering Results of 2nd Window Pixel.

4.8 Benchmarking Results

4.8.1 Bench marking tests

To benchmark the two filtering methods had to be tested under the similar conditions. This was done by using the same images, applying the same filtering window sizes and running the tests on the same PC. This made sure that any differences in performance were due to the filtering algorithms rather than changing testing environments.

Filtering window

To get a good idea of performance the filtering widow was varied. This variation helped in understanding how the filters perform under different window sizes. Each image was processed using a $5x5, 9x9, 15x15$ window. It was expected that for both filters, increasing window size increasing smoothing. The larger the window the blurrier the image.

Image size

To evaluate the performance of the filtering algorithms different image sizes were used in the benchmarking. This was helpful in understanding how size affects the performance and scalability of the filter. It was expected that with increasing size the execution time would increase too due to high numbers of pixels to be processed.

4.9 Benchmark Results

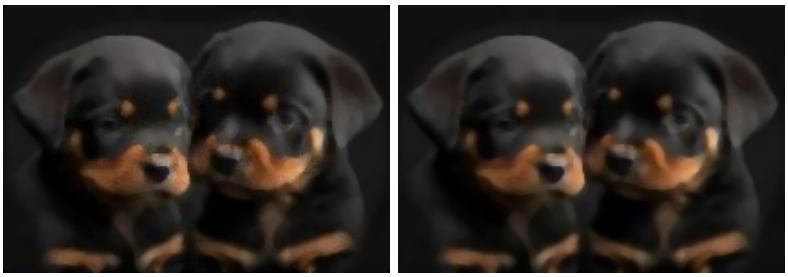
This section deals with the analysis of the results obtained after the implementation of the two filters. First, let us consider the implication of increasing the image size. As shown in Figure 3.1, when the size increases the execution time for both the golden measure and the user-designed median filter algorithm increases. The time increase for the golden measure however is less drastic. The following images show

Window size	Image size	OpenCV time (s)	Verilog time(s)
5	234x348	0.0011	0.00456
15	234x348	0.092	0.0137
5	148x214	0.001	0.0017
15	148x214	0.003	0.0144

Table 4.3: Execution time for different image sizes

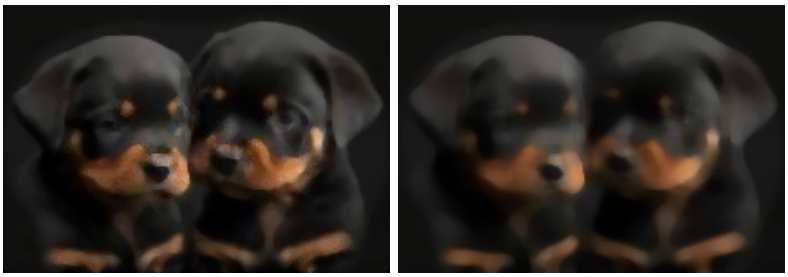
the image outputs of the different filters with Figure 3.4 and 3.8 as the reference original images.

Figure 4.11: Original dog image



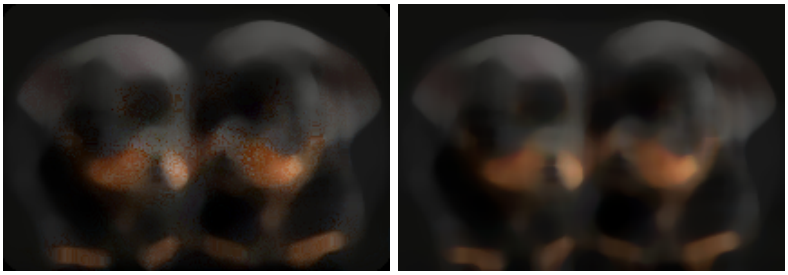
(a) Verilog, window: 5 (b) OpenCV, window: 5

Figure 4.12: Filtered images, window:5



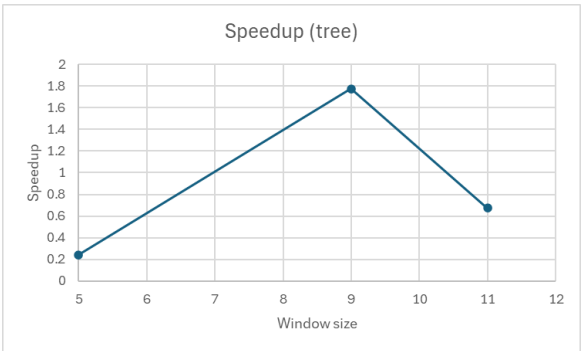
(a) Verilog, window: 11 (b) OpenCV, window: 11

Figure 4.13: Filtered images, window:11

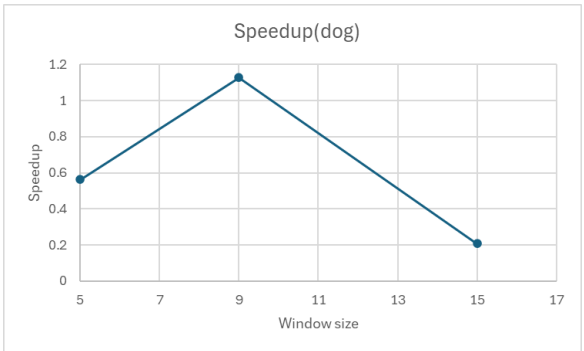


(a) Verilog, window: 15 (b) EOpenCV, window: 15

Figure 4.14: Filtered images, window:11



(a) Varying window size (tree)



(b) Varying window size (dog)

Figure 4.15: Filtered images, window:11

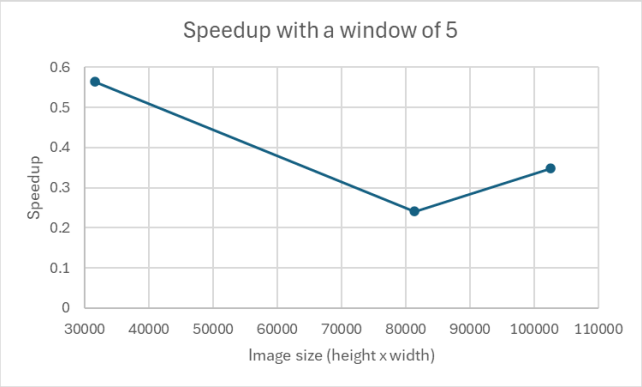


Figure 4.16: Speedup with constant window but changing size

Chapter 5

Discussion

5.1 Comments of Unit Test Results

5.1.1 Controller module

The unit test for the controller module correctly resets and sets all values to 0 when the reset pin is enabled. It also initiates the transfer and filtering process when triggered, as expected, as shown in Figure 4.1.

5.1.2 Filter Module

As shown in Figure 4.2, when the filter is provided with dummy data and the enable pin is held high, it returns the filtered data during the send data window, filtering the data as expected.

5.1.3 Memory Module test

As shown in Figure 4.3, the memory module stores dummy data when the write memory command is issued, and this data can be accessed and read using the read memory command. When the reset pin is enabled, both buses are then set to 0, as expected.

5.2 Comments on Integration Test Results

For integration test procedure, the goal was to validate if the integrated system is able to filter the image. The procedures consisted of using a synthesized image and a real-time image. The pixels of 5x5 synthesized are shown in table 4.2. These data was written to a text file and padded with a layer of zeros to permit filtering of 3x3 window size. The simulation results are shown in figure 4.9 and figure 4.10. In particular, figure 4.9 shows the simulation results when filtering the first image pixel, `0xdefe45`. With this pixel as the centre of the window and considering the zero layer for 3x3 window size. The window pixels needed for this are: `0x000000`,`0x000000`,`0x000000`,`0x000000`,`0xdefe45`,`0x33efe1`,`0x000000`,`0xacbeff`, `0xddedf1`. If these pixels are sorted and a median is selected we will get `0x000000` as the median pixel. This is clearly highlighted in figure 4.9. Additionally, figure 4.10 shows the section of the results when filtering the second window pixel, `0x33efe1` in table 4.2. These results shows that the image filter implemented correctly reads window data and correctly obtains the the median.

5.3 Comments on Benchmark Results

Changing the window size significantly changes the execution time. By keeping the image size constant the window size because the primary factor of performance. From Table 4.3 it can be seen that for a window of 5, the execution times are faster than when the window size is 15. This is because of the increased complexity as a larger size results in more pixels being considered and sorted.

Figures 4.9- 4.12 show the side-by-side comparisons of the Verilog implementation and the OpenCV implementation. From these images, it can be seen that the results of the two filters are consistent showing that the Verilog algorithm is indeed accurately filtering the images.

Figure 4.13 shows the speedup graph with (a) depicting the speedup of the filtering algorithm used on an image of a tree with varying window sizes and (b) illustrating the speedup of the algorithm on an image of a dog under the same conditions. The results are not as expected, instead the graphs illustrate fluctuations in performance. This is an indication of irregularities in speedup instead of a smooth predictable pattern.

Figure 4.14 is the speedup graph for varying image sizes. A window of 5 was used to maintain consistency and eliminate other factors that may alter the results. It was expected that as the size increases the speed up also increases consistently but just like the graphs mentioned above there are irregularities in the speed up.

5.4 Comments on FPGA Simulation Results

As observed in figure 4.6 and 4.8 UART succefully transmitted `0xFF` and `0x00` .Moreover, the UART module only sends data when data available flag is high which is expected. While UART module worked, the overall system did not work on FPGA. One of the main limitations on Nexys A7-100T is the lack of System on Chip (SoC) which makes it harder to develop memory-intensive applications smoothly. Xilinx offers a soft processor IP called Microblaze, however, Microblaze does not come with any peripheral connected and only offers up to `128KB` RAM. Moreover, there is no free version of the SD card controller IP for Microblaze which could have been a much easier option. Also adding peripherals individually is more work than using vendor IP for the dedicated processor on the chip than the soft processor. Besides this, the median filter is inherently difficult to implement on FPGA as it requires sorting and sorting unrolls into lots of logic elements which therefore leads to timing violations on standard clock speed. Moreover, algorithm like median of medians which tries to optimise median finding are inherently recursive which is not desirable in Verilog. Finally, during implementation, Vivado occasionally ignored some elements of the sorted array as only the middle element is required for output.

Chapter 6

Conclusions and Recommendations

This report detailed the design and implementation of a median filter on an FPGA platform. The literature review on image filtering techniques is presented in chapter 2, followed by a detailed system design procedure in chapter 3. Some of the objectives of the project were to implement a median filter on FPGA platform and analyse its performance. The system was coded in Verilog HDL and an attempt to run on Nexys A7100T board, but failed due to reasons explained in chapter 5. However, this can be overcome by using more powerful FPGA boards like Digilent Zybo Z7 which has SoC on board. Moreover, more robust hardware descriptive languages like VHDL or System Verilog can be used as instead of Verilog as they are more expressive and robust than Verilog.

The unit tests demonstrated that the submodules and the overall system behave as expected. The memory module stores information and makes it accessible at all times, resetting when prompted. The filter processes data correctly, and the controller successfully triggers the other modules as intended.

Glossary

FPGA Field Gate Programmable Array

SoC System on Chip

UART Universal Asynchronous Receiver Transmitter

Bibliography

[1] L. A. Aranda, P. Reviriego, and J. A. Maestro, "Error detection technique for a median filter," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2219–2226, 2017.

[2] S. Sudan, "Median filter performance based on different window sizes for salt and pepper noise removal in gray and rgb images," *International Journal of Signal Processing, Image Processing and Pattern Recognition*, vol. 8, no. 10, pp. 343–352, 2015.

[3] B. Justusson, "Median filtering: Statistical properties," *Two-dimensional digital signal prcessing II: transforms and median filters*, pp. 161–196, 2006.

[4] G. Deng and L. Cahill, "An adaptive gaussian filter for noise reduction and edge detection," in *1993 IEEE conference record nuclear science symposium and medical imaging conference*. IEEE, 1993, pp. 1615–1619.

[5] J. Benesty, S. Makino, J. Chen, J. Benesty, J. Chen, Y. Huang, and S. Doclo, "Study of the wiener filter for noise reduction," *Speech enhancement*, pp. 9–41, 2005.

[6] L. Yin, R. Yang, M. Gabbouj, and Y. Neuvo, "Weighted median filters: a tutorial," *IEEE Transactions on circuits and systems II: analog and digital signal processing*, vol. 43, no. 3, pp. 157–192, 1996.

[7] Y. Hu and H. Ji, "Research on image median filtering algorithm and its fpga implementation," in *2009 WRI global Congress on intelligent systems*, vol. 3. IEEE, 2009, pp. 226–230.

[8] A. Nieminen and Y. Neuvo, "Comments on‘ theoretical analysis of the max/median filter' by gr arce and mp mclaughlin," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, no. 5, pp. 826–827, 1988.

[9] G. Bates and S. Nooshabadi, "Fpga implementation of a median filter," in *TENCON '97 Brisbane - Australia. Proceedings of IEEE TENCON '97. IEEE Region 10 Annual Conference. Speech and Image Technologies for Computing and Telecommunications (Cat. No.97CH36162)*, vol. 2, 1997, pp. 437–440 vol.2.