



**DIGITAL SYSTEM DESIGN FINAL PROJECT REPORT
DEPARTMENT OF ELECTRICAL ENGINEERING
UNIVERSITAS INDONESIA**

SIMPLE IMAGE AUGMENTER

GROUP PSD-B5

Alifya Zhafira Ananda	2106704111
Jeffri	2106705070
Juan Jonathan	2106704894
Zaki Ananda	2106705474

PREFACE

It is a pleasure to thank those who made this project possible. We owe our deepest gratitude to our lecturer, Ruki Harwahyu., ST., MT., M.Sc., Ph.D. and Yan Maraden., ST., MT., M.Sc. for their wonderful lessons in the class, also keeping us engaged and learning.

We would also like to thank Digital Laboratory for providing the opportunity to work on such an interesting topic, especially to our assistant laboratory as our advisor, Miranty Anjani, whose gentle guidance made this project a possibility.

Finally, we want to thank our family and friends for the unbelievable amount of support and love shown to us throughout our second year of college journey.

Depok, December 10, 2022

Group PSD-B5

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION	4
1.1 BACKGROUND	4
1.3 OBJECTIVES	5
1.4 ROLES AND RESPONSIBILITIES	5
CHAPTER 2: IMPLEMENTATION	6
2.1 EQUIPMENT	6
2.2 IMPLEMENTATION	7
CHAPTER 3: TESTING AND ANALYSIS	15
3.1 TESTING	15
3.2 RESULT	16
3.3 ANALYSIS	17
CHAPTER 4: CONCLUSION	19
REFERENCES	20
APPENDICES	21
Appendix A: Project Schematic	21
Appendix B: Documentation	22

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

The aim of this project is to design and implement a Simple Image Augmenter in VHDL. Theoretically, an FPGA will excel at executing processes after its hardware has been described as such. FPGA will also perform much better at specific tasks like this compared to a General Purpose Computer. As a result, an FPGA will excel at performing simple computations but a lot at a time because of its ability to perform tasks concurrently.

1.2 PROJECT DESCRIPTION

Simple Image Augmenter aims to perform basic augmentation such as mirroring, rotating, flipping, and adjusting the brightness of an image. The hardware has capabilities of reading and writing an image complemented by a python script with the ability to convert an image to a text file. Each augmentation is defined as a state in a Finite State Machine. Each state will be triggered by a rising edge of the next state. These triggers can be assumed as buttons. After being triggered, the FPGA processes the augmentation on the image read from the text file. After the user is satisfied with the augmentation, the image will be written to another text file where another python script will convert the text file to an image.

1.3 OBJECTIVES

The objectives of this project are as follows:

1. Read an image in the form of a text file
2. Augment the image based on the user's input
3. Write the image back to a text file.

1.4 ROLES AND RESPONSIBILITIES

The roles and responsibilities assigned to the group members are as follows:

Roles	Responsibilities	Person
Project Lead	<ul style="list-style-type: none"> - Writing the python script to convert an image to a text file and vice versa. - Write the VHDL procedure of adjusting brightness - Write the VHDL procedure of reading and writing the image 	Juan Jonathan
Role 2	<ul style="list-style-type: none"> - Write the VHDL procedure of mirror X - Write the VHDL top-level entity 	Alifya Zhafira Ananda
Role 3	<ul style="list-style-type: none"> - Write the VHDL procedure of rotate - Write the VHDL testbench code 	Zaki Ananda
Role 4	<ul style="list-style-type: none"> - Write the VHDL procedure of mirror Y 	Jeffri

Table 1. Roles and Responsibilities

CHAPTER 2

IMPLEMENTATION

2.1 EQUIPMENT

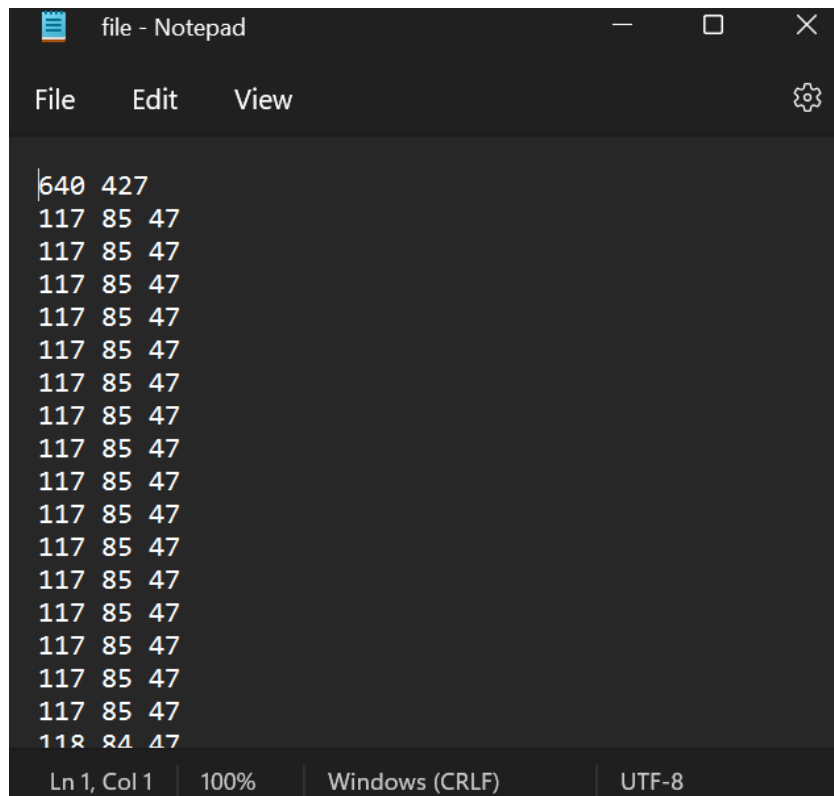
The tools that are going to be used in this project are as follows:

- VSCode
- ModelSim
- Intel Quartus Prime

2.2 IMPLEMENTATION

The python script complements the FPGA's ability to write and read images by reading an image's resolution and writing it in the first line of a text file. After that, the script will read each pixel's red, green, and blue channel values and write it in the second line and after. Each channel is separated with a whitespace while each pixel is separated by a new line.

The argparse module is used to run the script with specific arguments which in this case is the image and the text file's path. The usage of the python script can be seen in the README on our github. The PILLOW module or more commonly known as Python Image Loader helps us to load the image and convert it to a 2 dimensional array of tuples with the tuples containing the values of red, green, and blue channels ranging from 0 to 255. Hence, it is possible to take each element of the tuple and concatenate them separated with a whitespace. Each pixel will be written according to a nested for loop with ranges according to the image's resolution. For example, the script reads a 600x400 image so the first line will contain '600 400' and each line after that will contain each pixel, resulting in a total of 240000 lines containing each pixel of the image.



```
file - Notepad
File Edit View
640 427
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
117 85 47
118 84 47
Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8
```

Fig 1. Example of an image converted to a text file

VHDL can read text files with the library `std.textio` and `std_logic_textio`. A buffer must be defined in the form of the filepath of the text file. Each line of the input buffer will be read with the `readline` function resulting in a line being read exactly as the format of the text file. The format is “{R} {G} {B}”, as such other than reading the integers, the whitespaces must also be read. Hence, each line will be read by the following lines :

```
for i in 1 to height loop
  for j in 1 to width loop
    readline(input_buf, read_line);
    read(read_line, red); -- Read red channel
    read(read_line, spaces); -- Read space
    read(read_line, green); -- Read green channel
    read(read_line, spaces); -- Read space
    read(read_line, blue); -- Read blue channel
    Img(i - 1, j - 1, 2) <= red;
    Img(i - 1, j - 1, 1) <= green;
    Img(i - 1, j - 1, 0) <= blue;
  end loop;
end loop;
```

Fig 2. Algorithm used to read each line

Img here is a 3 dimensional matrix with the first 2 dimensions representing the width and height of the image while the 3rd dimension contains 3 elements which are the R, G, and B channels.

Writing the text file works similarly except we need to create the line that needs to be written. A buffer with the type line will be used to create a line with each value concatenated with whitespace and then written per line to the text file specified by the filepath.

```
for i in 1 to height loop
  for j in 1 to wdth loop
    write(out_line, integer'image(Img(i - 1, j - 1, 2)) & " " &
          integer'image(Img(i - 1, j - 1, 1)) & " " &
          integer'image(Img(i - 1, j - 1, 0)), left, 11); -- Write R G B
    writeline(output_buf, out_line);
  end loop;
end loop;
```

Fig 3. Algorithm used to write to text file.

Procedure to adjust brightness works in a very simple way. Take for example we want to increase the brightness of the image by 20%, then we simply need to multiply each R, G, and B values of each pixel by 120% or 1.2. The reverse is also true for darkening an image. For example we want to darken the image by 30%, then we simply need to multiply each R, G, and B values of each pixel by 70% or 0.7.


```

for i in 1 to height loop
  for j in 1 to width loop
    if(Img(i - 1, j - 1, 2) = 0 and bright /= 0) then
      Img(i - 1, j - 1, 2) <= 1;
    end if;
    if(Img(i - 1, j - 1, 1) = 0 and bright /= 0) then
      Img(i - 1, j - 1, 1) <= 1;
    end if;
    if(Img(i - 1, j - 1, 0) = 0 and bright /= 0) then
      Img(i - 1, j - 1, 0) <= 1;
    end if;
    Img(i - 1, j - 1, 2) <= Img(i - 1, j - 1, 2) * bright / 100;
    Img(i - 1, j - 1, 1) <= Img(i - 1, j - 1, 1) * bright / 100;
    Img(i - 1, j - 1, 0) <= Img(i - 1, j - 1, 0) * bright / 100;
    if(Img(i - 1, j - 1, 2) > 255) then
      Img(i - 1, j - 1, 2) <= 255;
    end if;
    if(Img(i - 1, j - 1, 1) > 255) then
      Img(i - 1, j - 1, 1) <= 255;
    end if;
    if(Img(i - 1, j - 1, 0) > 255) then
      Img(i - 1, j - 1, 0) <= 255;
    end if;
  end loop;
end loop;

```

Fig 4. Algorithm used to adjust the brightness of the image.

Since R, G, and B channels cannot go below 0 or above 255, then some error handling needs to be done. Hence, we need to check if any R, G, or B channel is 0 when the brightness factor is not 0. After that, we need to check if any channel goes beyond 255, if there are any, then it is reduced to 255.

Another feature that we provide in our program is to mirror an image. There are two types of this feature. They are mirror x, which is mirroring the image vertically and mirror y, which is mirroring the image horizontally. It is like a flip feature in any photo editor software.

```

procedure mirrorX(
    constant width : in integer;
    constant height : in integer;
    signal Img : inout matrix
) is
    variable temp : RGB;
begin
    for j in 1 to width loop
        for i in 1 to height / 2 loop
            temp(2) := Img(i - 1, j - 1, 2);
            temp(1) := Img(i - 1, j - 1, 1);
            temp(0) := Img(i - 1, j - 1, 0);
            Img(i - 1, j - 1, 2) <= Img(height - i, j - 1, 2);
            Img(i - 1, j - 1, 1) <= Img(height - i, j - 1, 1);
            Img(i - 1, j - 1, 0) <= Img(height - i, j - 1, 0);
            Img(height - i, j - 1, 2) <= temp(2);
            Img(height - i, j - 1, 1) <= temp(1);
            Img(height - i, j - 1, 0) <= temp(0);
        end loop;
    end loop;
end procedure mirrorX;

```

Fig 5. Algorithm used to mirror the image vertically

The for loop iterates through the width of the image and then iterates through the height of the image. The for loop then assigns a value to each pixel in order from left to right, top to bottom. First, we store the red, green, and blue (RGB) values of the original top side pixels to a temporary array. Then, we start replacing the RGB value of the top side pixels with the bottom side pixels. After that, we also replaced the RGB value of the bottom side pixels with the temporary array. So, the image can be mirrored horizontally by storing and replacing the RGB value with the temporary array helps.

```

procedure mirrorY(
    constant width : in integer;
    constant height : in integer;
    signal Img : inout matrix
) is
    -- type RGB is array(2 downto 0) of Integer range 0 to 255;
    -- variable i, j : integer range 0 to 1999; -- counter for pixels
    variable temp : RGB; -- temporary variable to store RGB values
begin
    for i in 1 to height loop
        for j in 1 to width / 2 loop
            temp(2) := Img(i - 1, j - 1, 2); -- Storing Red value
            temp(1) := Img(i - 1, j - 1, 1); -- Storing Green value
            temp(0) := Img(i - 1, j - 1, 0); -- Storing Blue value
            Img(i - 1, j - 1, 2) <= Img(i - 1, width - j, 2);
            Img(i - 1, j - 1, 1) <= Img(i - 1, width - j, 1);
            Img(i - 1, j - 1, 0) <= Img(i - 1, width - j, 0);
            Img(i - 1, width - j, 2) <= temp(2); -- Replacing Red value
            Img(i - 1, width - j, 1) <= temp(1); -- Replacing Green value
            Img(i - 1, width - j, 0) <= temp(0); -- Replacing Blue value
        end loop;
    end loop;
end procedure mirrorY;

```

Fig 6. Algorithm used to mirror the image horizontally

The for loop iterates through the height of the image and then half the width of the image. Inside of the loop body, the pixel's RGB values were swapped, from the first row of pixels, from the leftmost swapped with the rightmost, gradually progressing to the center pixel(s), then the last row of pixels. The swapping uses a temporary array to store the left pixels' RGB values, which eventually replace the right pixels just as the right pixels replace the left pixels.

```

procedure rotate(
    signal width : inout integer;
    signal height : inout integer;
    signal Img : inout matrix
)
is
    variable tmp : integer;
begin
    for i in 1 to height loop
        for j in 1 to width loop
            Img(j - 1, height - i, 2) <= Img(i - 1, j - 1, 2);
            Img(j - 1, height - i, 1) <= Img(i - 1, j - 1, 1);
            Img(j - 1, height - i, 0) <= Img(i - 1, j - 1, 0);
        end loop;
    end loop;
    tmp := width;
    width <= height;
    height <= tmp;
end procedure rotate;

```

Fig 7. Algorithm used to rotate the image clockwise

The procedure to rotate an image clockwise is to move the first (topmost) row of pixels into the last (rightmost, now height - i) column of pixel, the second row of pixel into the second to last column of pixel, and so on. The size of the image also needs to be adjusted by swapping the width and height of the original image.

```

process (Rd, Wr, Mx, My, Rt, AdBr, Bright, clk) is
variable tmp: integer;
begin
    inputs <= Rd & Mx & My & Rt & AdBr & Wr;
    case present is
        when S0 => -- Wait for Input
            if(inputs = "100000") then
                nxt <= S1;
            elsif(inputs = "010000") then
                nxt <= S2;
            elsif(inputs = "001000") then
                nxt <= S3;
            elsif(inputs = "000100") then
                nxt <= S4;
            elsif(inputs = "000010") then
                nxt <= S5;
            elsif(inputs = "000001") then
                nxt <= S6;
            else
                nxt <= present;
            end if;

        when S1 => -- Read Image
            if(Rd = '0') then
                readImage(Img, w, h);
                RES <= Img;
                RES_W <= w;
                RES_H <= h;
                nxt <= S0;
            end if;

        when S2 => -- Mirror X
            if(Mx = '0') then
                mirrorX(w, h, Img);
                nxt <= S0;
            end if;

        when S3 => -- Mirror Y
            if(My = '0') then
                mirrorY(w, h, Img);
                nxt <= S0;
            end if;

        when S4 => -- Rotate
            if(Rt = '0') then
                rotate(w, h, Img);
                tmp := w;
                w <= h;
                h <= tmp;
                nxt <= S0;
            end if;

        when S5 => -- Add Brightness
            if(AdBr = '0') then
                adjustBrightness(Bright, w, h, Img);
                nxt <= S0;
            end if;

        when S6 => -- Write Image
            if(Wr = '0') then
                writeImage(w, h, Img);
                nxt <= S0;
            end if;

        when others =>
            nxt <= present;
    end case;
end process;

```

Fig 8. Algorithm used to access the provided features

The top-level entity implements an instance of the function SimpleImageAugmenter_functions.vhd to allow a user to use the features. The first step is to define the signals that will be used for each pixel: Img (matrix) represents the original image; w and h represent the width and height of each pixel respectively. We declare six types of states, which is to allow a user to access the feature that they want to use based on their input. Take for example, if we want to mirror an image horizontally(mirror X), it will go for S2 and after they execute the mirror X procedure, the next state is S0, which is a state to wait for another input. For a special case, S0 was made for error handling when there are two inputs or more at the same time.

CHAPTER 3

TESTING AND ANALYSIS

3.1 TESTING

Firstly, we encountered some limitations when synthesizing the image, where bounds of the for loop cannot be defined at runtime. This is because VHDL is a hardware descriptive language, as such it cannot synthesize for loops with bounds defined at runtime. Modelsim for simulation purposes however, works just fine.

```
clock: process
variable end_time : time := 2000 ns;
variable current_time : time := 0 ns;
begin
    while current_time < end_time loop
        clk <= '0';
        wait for clk_time/2;

        clk <= '1';
        wait for clk_time/2;
        current_time := current_time + clk_time;
    end loop;
    wait;
end process;
```

Fig 9. Clock process at testbench code

```
--TEST: mirrorX
Rd <= '1';
wait for clk_time;
Rd <= '0';
wait for clk_time;
Mx <= '1';
wait for clk_time;
Mx <= '0';
wait for 3*clk_time;

TST_IMG := RES;
TST_H := RES_H;
TST_W := RES_W;
COR_PATH := new string'(path_mirrorX);

tb_arch.verify(TST_IMG, TST_W, TST_H, COR_PATH);
```

Fig 10. Validity test on mirror X operation
at testbench code

For testing the most top-level implementation, we use a VHDL testbench code. The testbench architecture is divided into two processes: one for clock, and one for testing every augmentation procedure. The clock process simulates a 20MHz clock that stops at the 40th cycle. while the test process stimulates the top-level entity to read the image file and perform various augmentation. The result is then compared with the correct, externally-augmented image file. An exception is made for the brightness adjustment augmentation, where the testbench simply checks whether the RGB value is at least the same, or higher/lower—depending on whether the operation increase or decrease the brightness—than the normal image file, and to check whether the RGB value exceeds the set boundary (0 to 255).

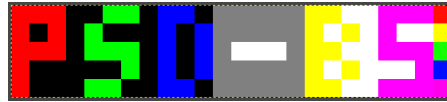


Fig 11. Test image used at the testbench VHDL code

For the test image, we used an image containing pixelated phrase “PSD-B5” to simulate the average case scenario. We set the image size to be small, which is 24 x 5 px, to simplify troubleshooting when the verification procedure detected a mismatch between the resulting and the expected image file.

3.2 RESULT

Below is the result of the aforementioned testbench, as simulated by ModelSim up to the 2000ns or the 40th clock cycle.

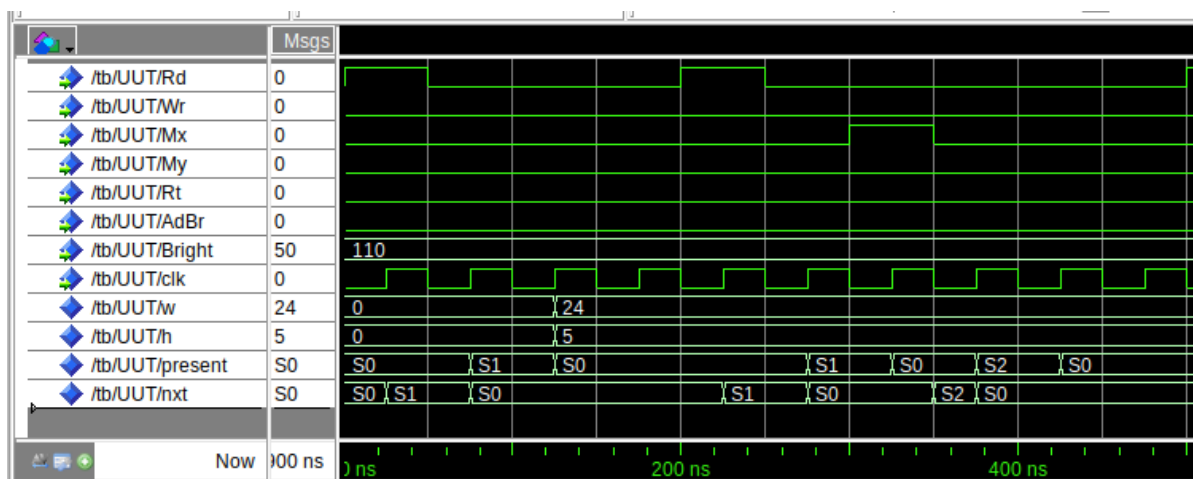


Fig 12. Testbench waveform at 0 – 500ns

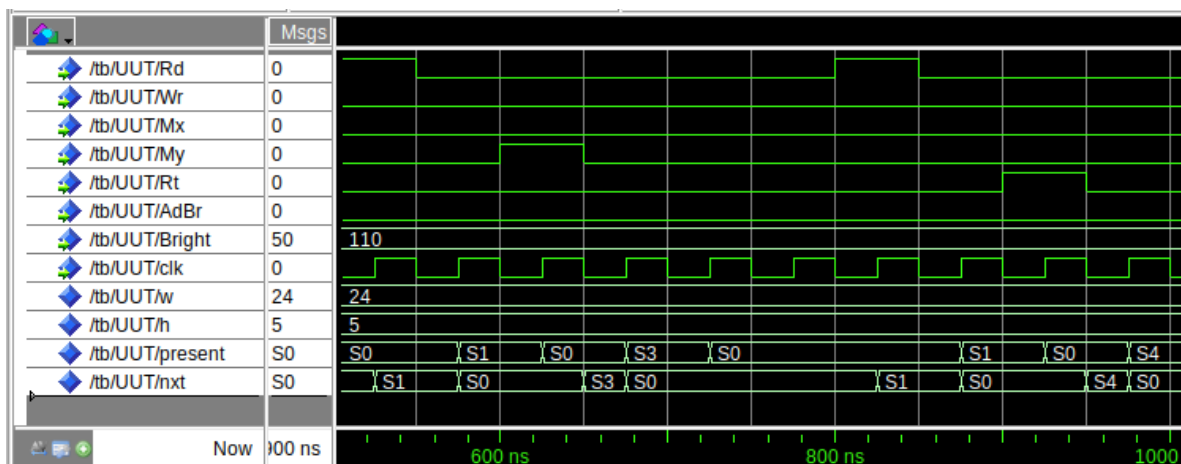


Fig 13. Testbench waveform at 500 – 1000ns

Fig 14. Testbench waveform at 1000 – 1500ns

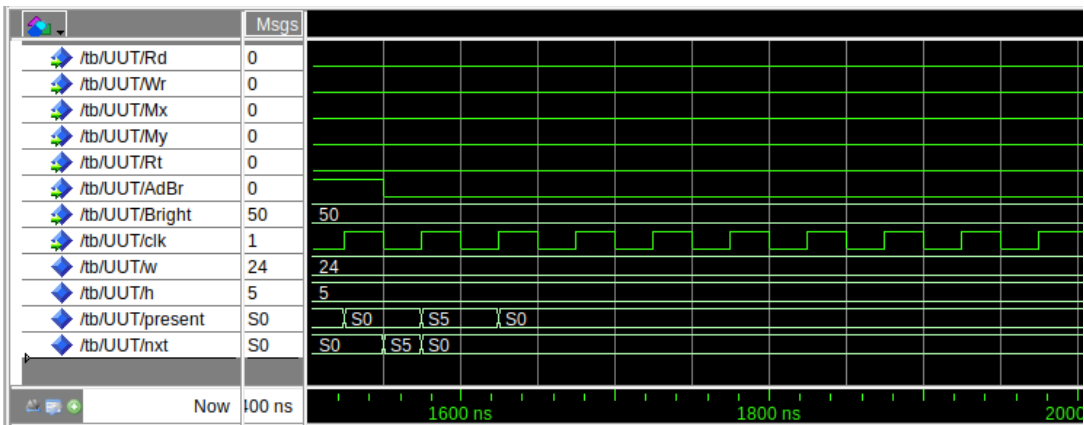


Fig 15. Testbench waveform at 1500 – 2000ns

3.3 ANALYSIS

From the start of the waveform, it can be seen that the default value was initialized correctly. The *Rd* signal is activated for a while, and then deactivated, simulating a momentary button press. The present state then switched to S1 at the next rising edge of the clock. The next state is correspondingly set to S0, back to the default state. The fact that the next state was set instantaneously suggests that the software simulates instantaneous operation. At the next rising edge, the width and height signal is updated to 24 and 5, implying that the read operation has successfully finished.

The next set of tests would stimulate read operation and various augmentation. Due to the way the testbench was set, there is very little analysis that could be done on the waveform when mirror X and mirror Y augmentation were being performed. On the next test, as the rotate augmentation was being performed, the width and height signals were swapped accordingly, suggesting that the operation was successful. The last two tests performed a brightness-adjusting (both increasing and decreasing) augmentation.

The above simulation checks the validity of the resulting augmentation using a simple assert – report code, and the software did not print any report whatsoever, suggesting the test was a total success.



Fig 16. These images were compared with the resulting augmentation, and the software reported no mismatch. Normal image (left top), X-mirrored image (left middle), Y-mirrored image (left bottom), rotated image (right)

CHAPTER 4

CONCLUSION

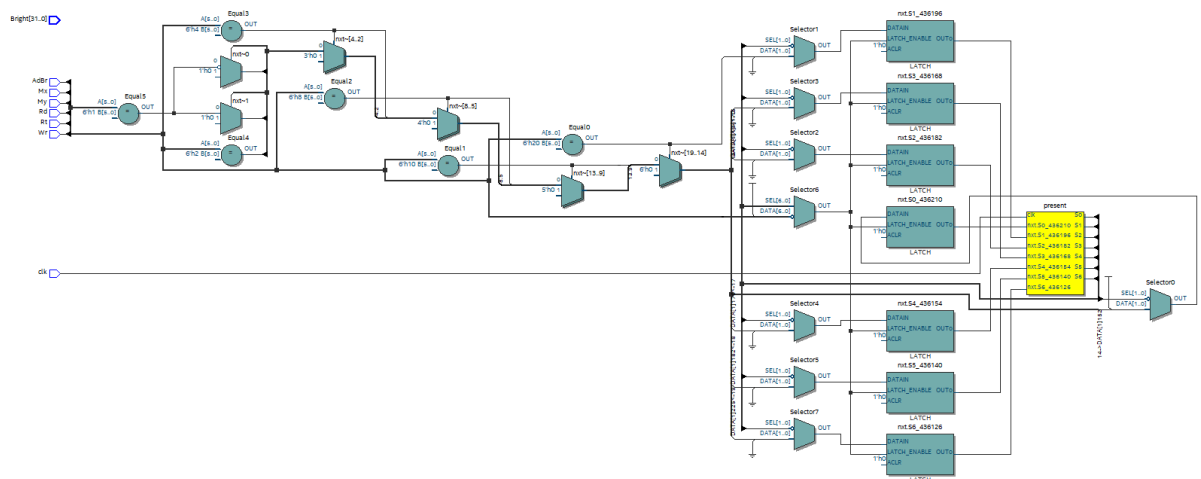
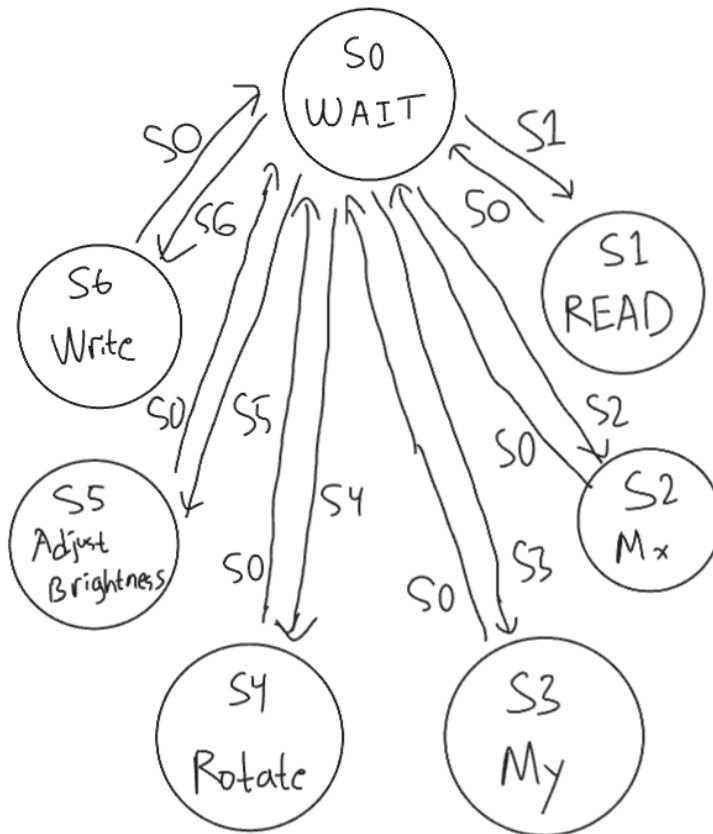
The experimental results demonstrated a simple augmentation of an image, which directly generates an image to adjust its brightness, rotates, and mirrors the image. Image processing can be done using VHDL with Python's help. VHDL cannot read an image as an image itself. The easiest way to read an image file in VHDL is to convert it to a text file with the RGB format using another programming language, such as Python. Once the image can be read in VHDL, it can be executed to apply the features that we provided. To look for the result, the image should be converted again from a text file to an image using python.

REFERENCES

- [1] M. Shawky, S. Bonnet, S. Favard and P. Crubille, "A computing platform and its tools for features extraction from on-vehicle image sequences," ITSC2000. 2000 IEEE Intelligent Transportation Systems. Proceedings (Cat. No.00TH8493), 2000, pp. 39-45, doi: 10.1109/ITSC.2000.881015.
- [2] Altium, La Jolla, CA, USA. *VHDL Language Reference*. (2008). Accessed: Dec. 10, 2022. [Online]. Available: <http://valhalla.altium.com/Learning-Guides/TR0114%20VHDL%20Language%20Reference.pdf>.
- [3] B. Mealy and F. Tappero. *Free Range VHDL*, 2013. Accessed: Dec. 10, 2022. [Online]. Available: https://ia600909.us.archive.org/16/items/free_range_vhdl/free_range_vhdl.pdf.
- [4] Mentor Graphics, Wilsonville, OR, USA. *ModelSim® Tutorial*. (2008). Accessed: Dec. 10, 2022. [Online]. Available: https://courses.cs.washington.edu/courses/csep567/10wi/labs/modelsim_tut.pdf

APPENDICES

Appendix A: Project Schematic



Appendix B: Documentation

