SIMPLE IMAGE AUGMENTER

GROUP MA-B5

Team

Juan Jonathan
2106704894

Alifya Zhafira
2106704111

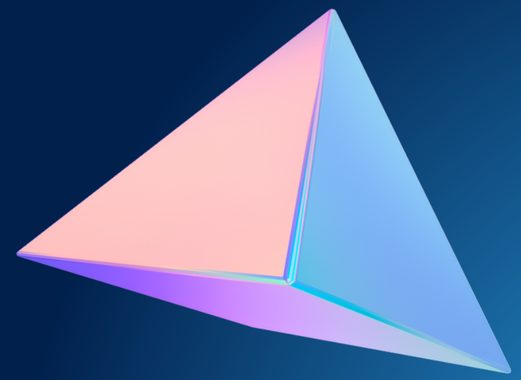
Jeffri
2106705070

Zaki Ananda
2106705474

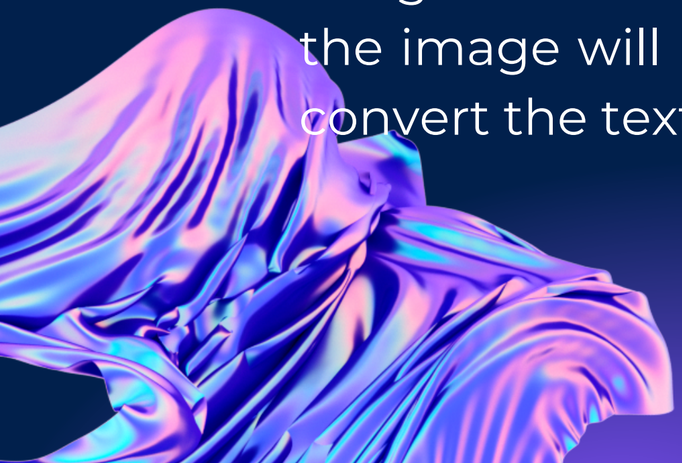
Background

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 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.

Description



Simple Image Augmenter aims to perform basic augmentation such as mirroring, rotating, flipping, and adjusting the brightness of an image. The hardware have 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 process the augmentation on the image read from the text file. After the user is satisfied with the augmentation, the image will be writted to another text file where another python script will convert the text file to an image.

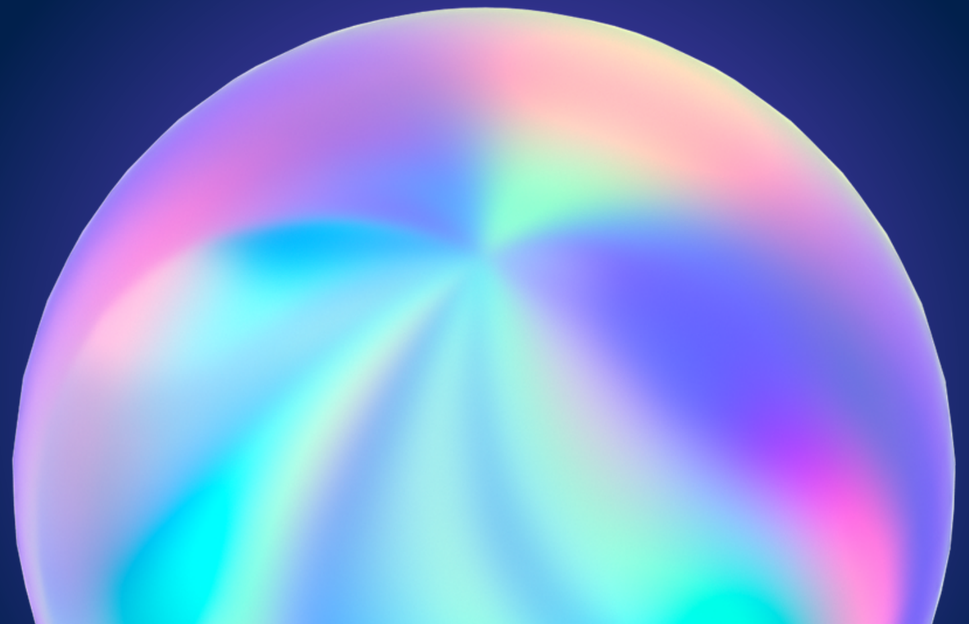


OBJECTIVES

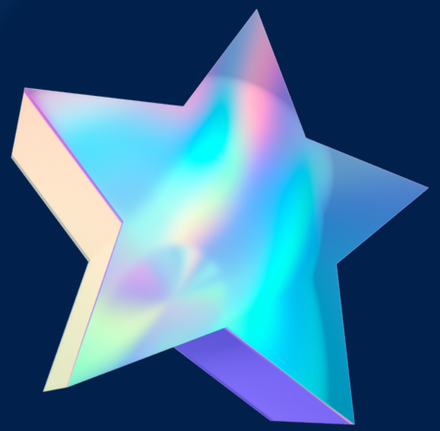
Augment the image based on the user’s input

Read an image in the form of a text file

Write the image back to a text file.



EQUIPMENT

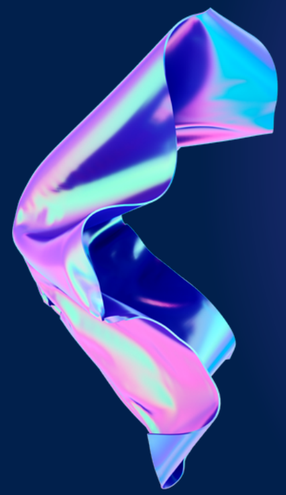


VSCode

A code editor

ModelSim

Simulator



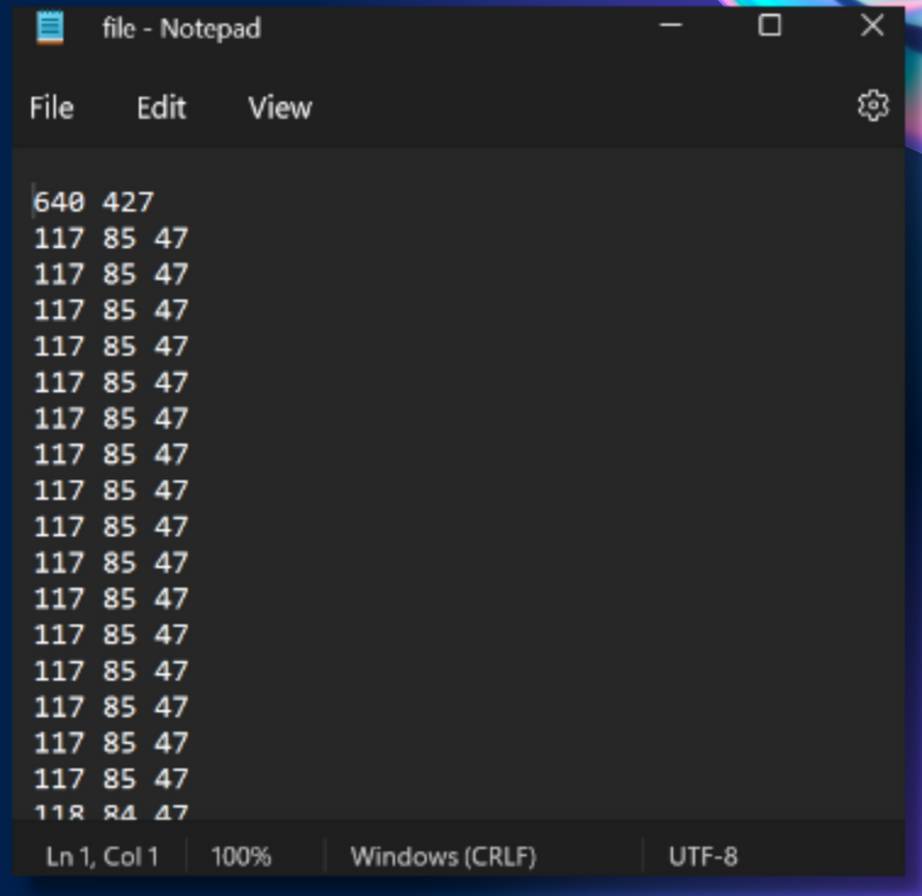
Intel Quartus Prime

Analysis and Synthesis

IMPLEMENTATION

IMAGE TO TEXT FILE

The python script complements the FPGA's ability to write and read image 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 writes it in the second line and after. Each channel is separated with a whitespace while each pixel is separated by a new line.



READ EACH LINE

```
for i in 1 to height loop
    for j in 1 to wdth 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;
```

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

WRITE TEXT FILE

```
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;
```

Writing the text file works similar 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 values 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
        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;
```

ADJUST BRIGHTNESS

Procedure to adjust brightness works in a very simple way. 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.

MIRROR X

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. The image can be mirrored horizontally by storing and replacing the RGB value with the temporary array helps.

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

```
procedure mirrorY(
    constant wdth : in integer;
    constant height : in integer;
    signal Img : inout matrix
) is
    -- type RGB is array(2 downto 0) of Integer range 0 to 25
    -- variable i, j : integer range 0 to 1999; -- counter fo
    variable temp : RGB; -- temporary variable to store RGB v
begin
    for i in 1 to height loop
        for j in 1 to wdth / 2 loop
            temp(2) := Img(i - 1, j - 1, 2); -- Storing Red v
            temp(1) := Img(i - 1, j - 1, 1); -- Storing Green
            temp(0) := Img(i - 1, j - 1, 0); -- Storing Blue
            Img(i - 1, j - 1, 2) <= Img(i - 1, wdth - j, 2);
            Img(i - 1, j - 1, 1) <= Img(i - 1, wdth - j, 1);
            Img(i - 1, j - 1, 0) <= Img(i - 1, wdth - j, 0);
            Img(i - 1, wdth - j, 2) <= temp(2); -- Replacing
            Img(i - 1, wdth - j, 1) <= temp(1); -- Replacing
            Img(i - 1, wdth - j, 0) <= temp(0); -- Replacing
        end loop;
    end loop;
end procedure mirrorY;
```

MIRROR Y

The for loop iterates through the height of the image and then half the width of the image. The pixel's RGB values were swapped, from left to right, then top to bottom. The left pixels' RGB values, were stored into a temporary values first, then the right pixels replace the left pixels, and the temporary array replace the right pixels.

ROTATE

The procedure to rotate an image clockwise is to move the first (topmost) row of pixel into the last (rightmost) column of pixel, the second row of pixel into the second to last colum 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.

```
procedure rotate(
    signal wdth : 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  wdth 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 := wdth;
            wdth <= height;
            height <= tmp;
end procedure rotate;
```

```
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;
```

TOP–LEVEL ENTITY

The first step is to define the signals that will be used for each pixel. Then, 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. For a special case, S0 was made for error handling when there are two inputs or more at the same time.

TESTING

```
clock: process
variable end_time : time := 2800 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;
```

```
--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);
```

Firstly, we encountered some limitations when synthesizing the image, where bounds of the for loop cannot be defined at runtime. 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.

TESTBENCH

The testbench architecture is divided into two processes: one for clock, and one for testing. 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.

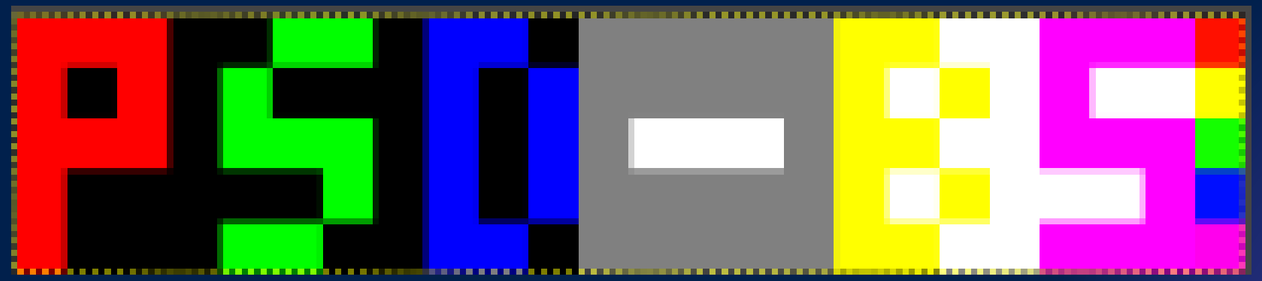
```
clock: process
variable end_time : time := 2000 ns;
variable current_time : time := 0 ns;
begin
        while current_time < end_time loop
                clk <= '0';
                wait for clk_time/2;

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

```
--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);
```



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

RESULT

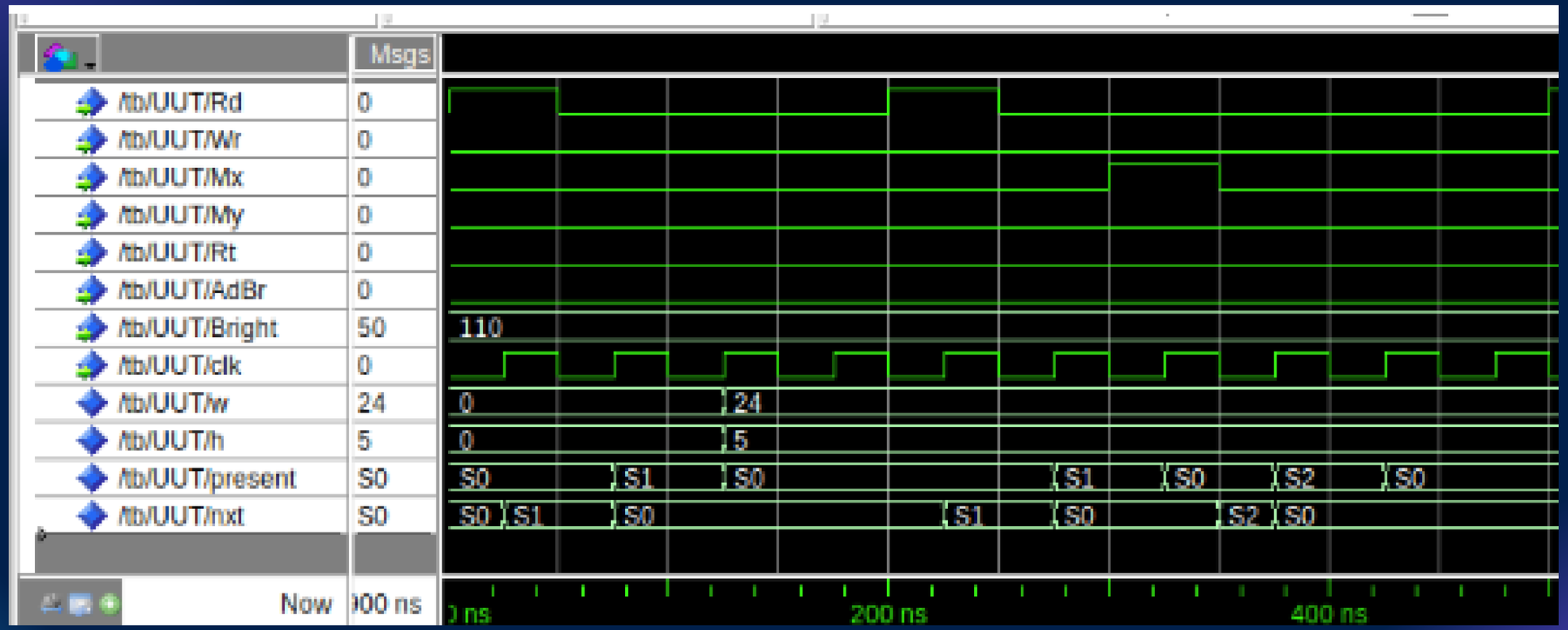


Fig 12. Testbench waveform at 0 – 500ns

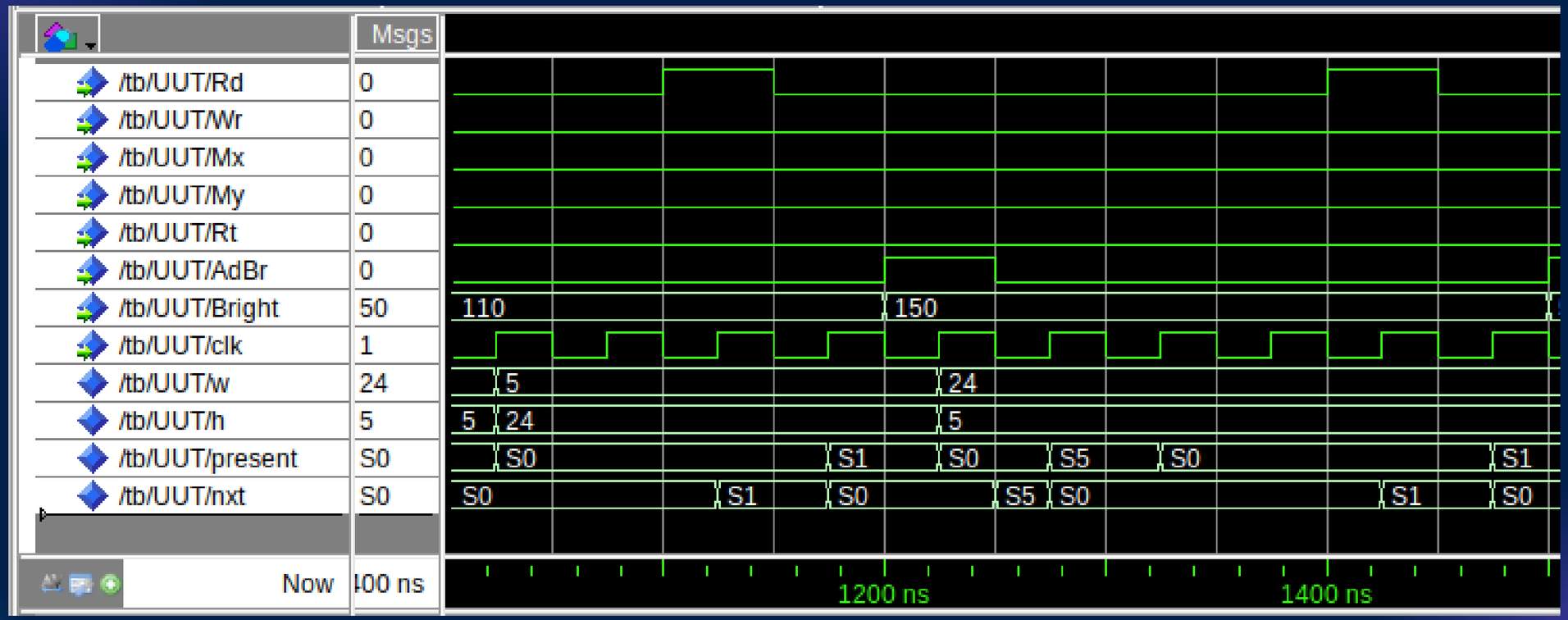


Fig 12. Testbench waveform at 500 – 1000ns

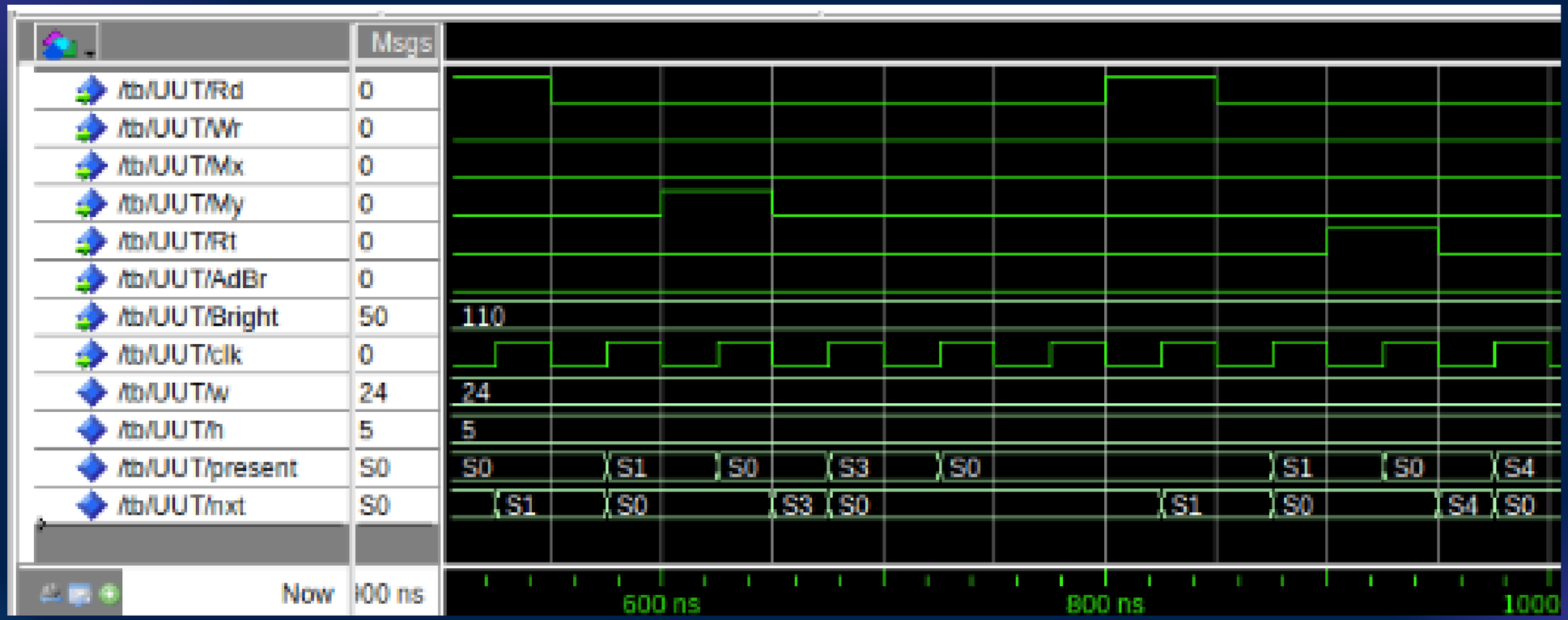


Fig 14. Testbench waveform at 1000 – 1500ns

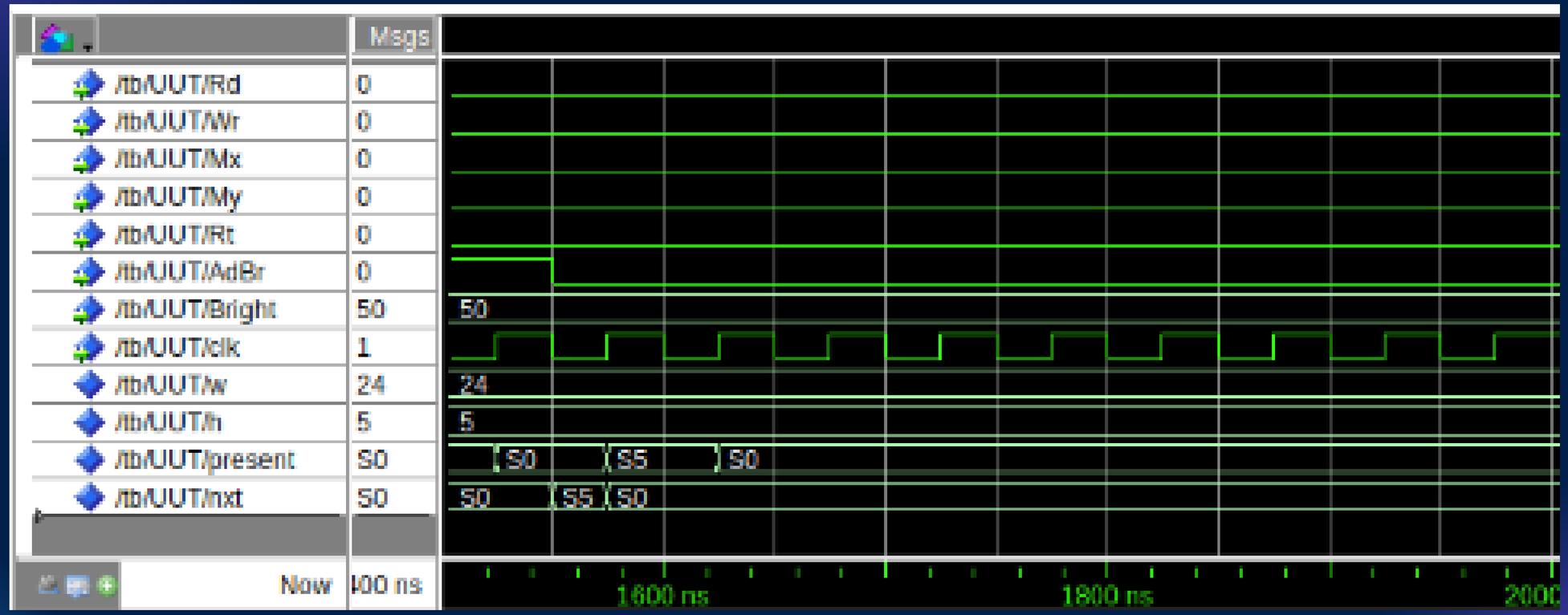


Fig 15. Testbench waveform at 1500 – 2000ns

Analysis

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. At the next rising edge, the width and height signal is updated to 24 and 5, implying that the read operation has successfully finished.

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 software did not print any report whatsoever, suggesting the test was a total success.

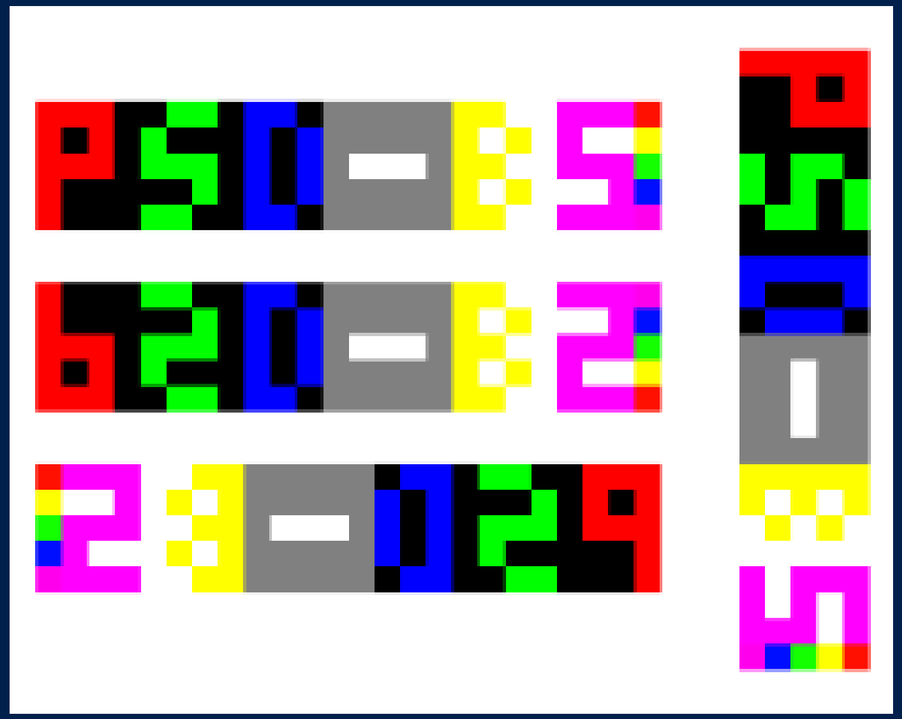


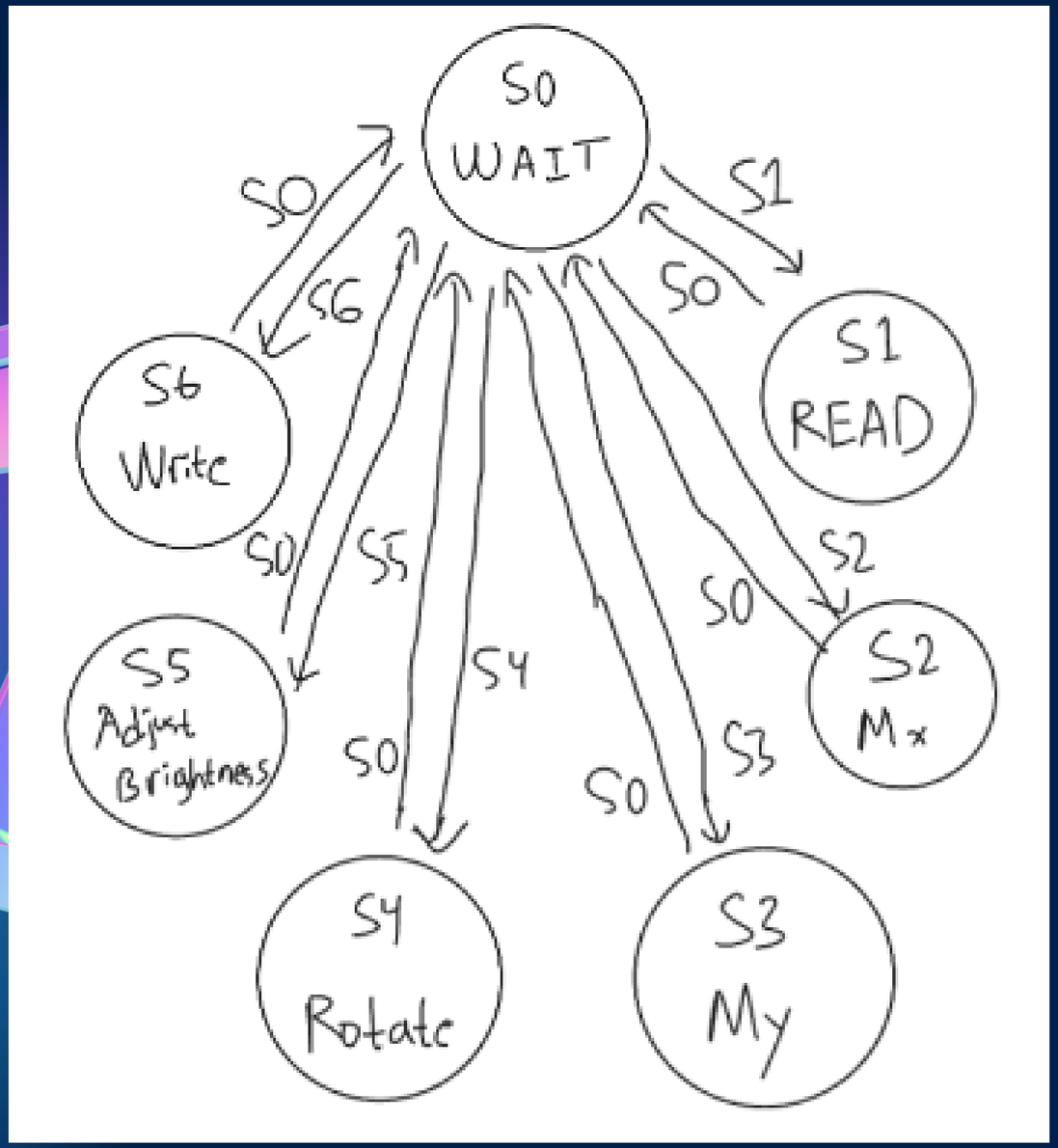
Fig 16. These images were compared with the resulting augmentation, and the software reported no mismatch. Normal image (left top), X-mirrored image (left middle), Y-mirrored image (left bottom), rotated image (right)

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.
- 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.

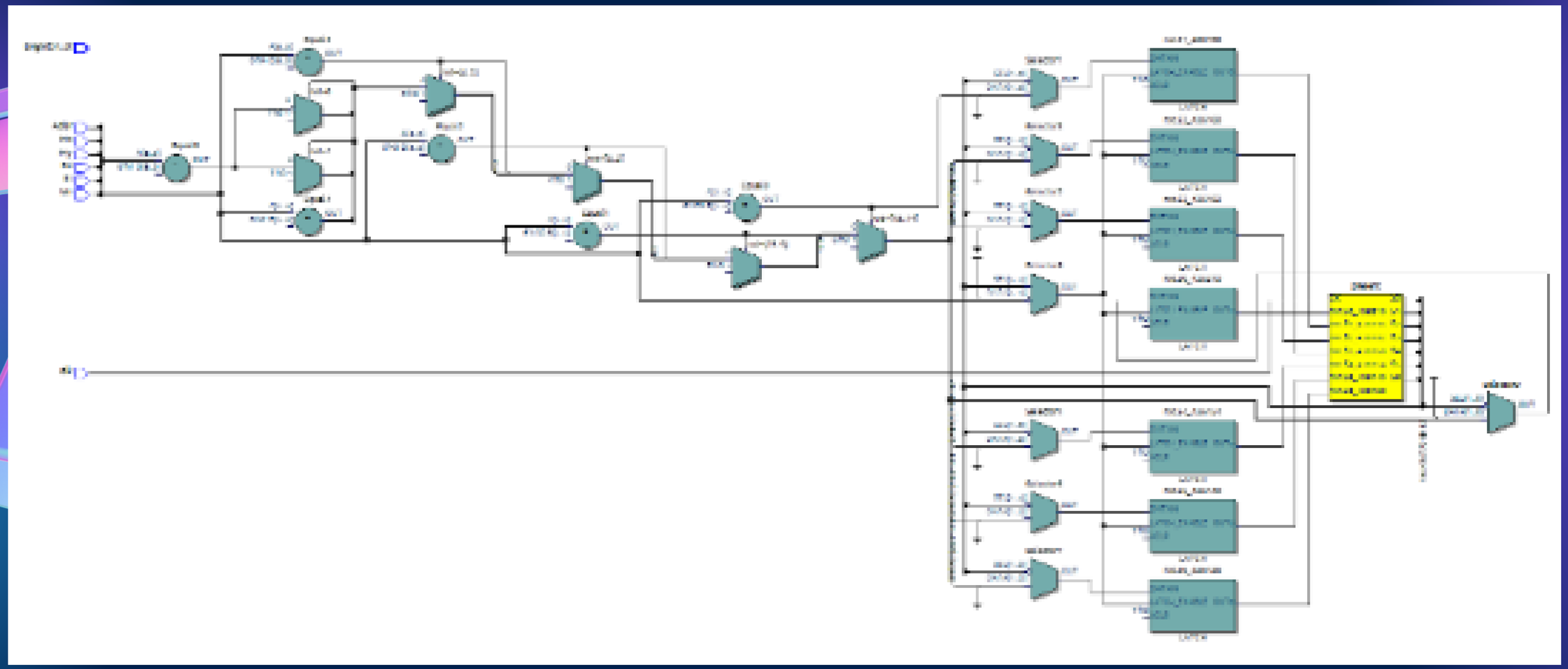
References

- 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.
- 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.
- 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.
- 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

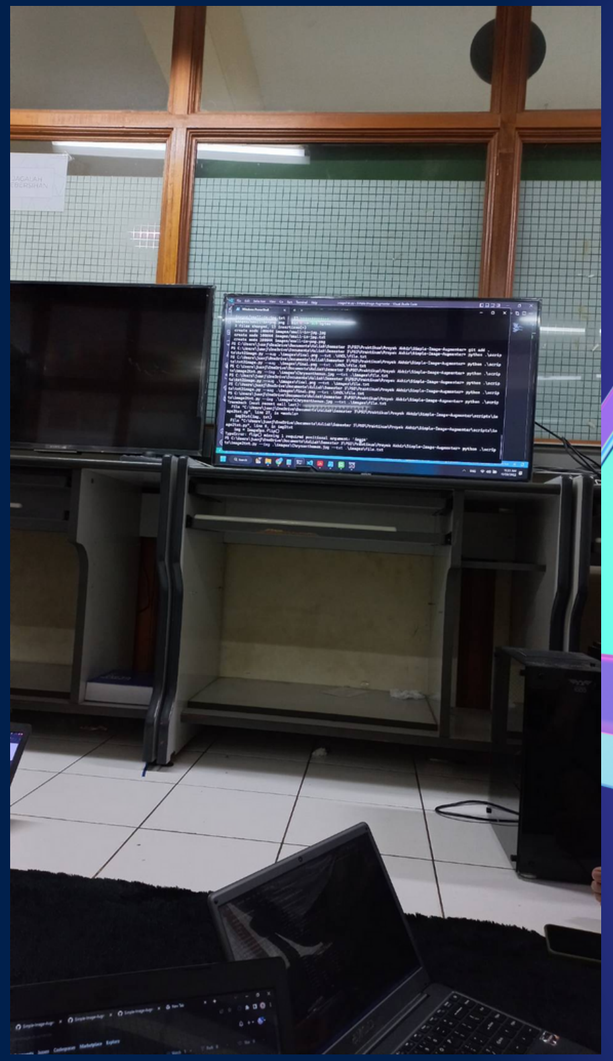


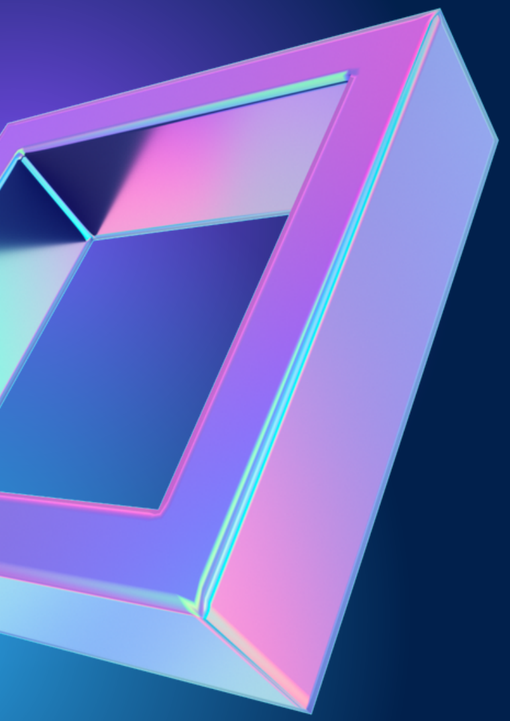
Appendix A:
Project Schematic
(Finite State Machine)

Appendix A: Project Schematic (Synthesis RTL Schematic)



Appendix B: Documentation





Thank You