COL215 HW Assignment-2

EDGE DETECTOR

Jahnabi Roy : 2022CS11094
Anamika : 2022CS11098

15th October, 2023

1 AIM

- To generate RAM/ROM using Vivado's memory generator and populate it with the image provided
- Perform image gradient operation on the image stored
- Display the image stored in the FPGA memory before and after the transformation.

2 MODULE IMPLEMENTED

2.1 IMAGE GRADIENT VHDL MODULE

Designed a module that read the image from the block RAM and calculate the image gradient as is being described below.

2.1.1 Calculation of Gradient

- Filter stride along the row of the image to calculate the resultant pixels in the output image.
- The output pixel value is the sum of the element-wise multiplications of the filter value and the input image pixel.
- Input image pixel at location (i,j) is denoted by I(i, j), output pixel as O(i, j).

$$O(i, j) = 1 * I(i, j - 1) - 2 * I(i, j) + 1 * I(i, j + 1)$$

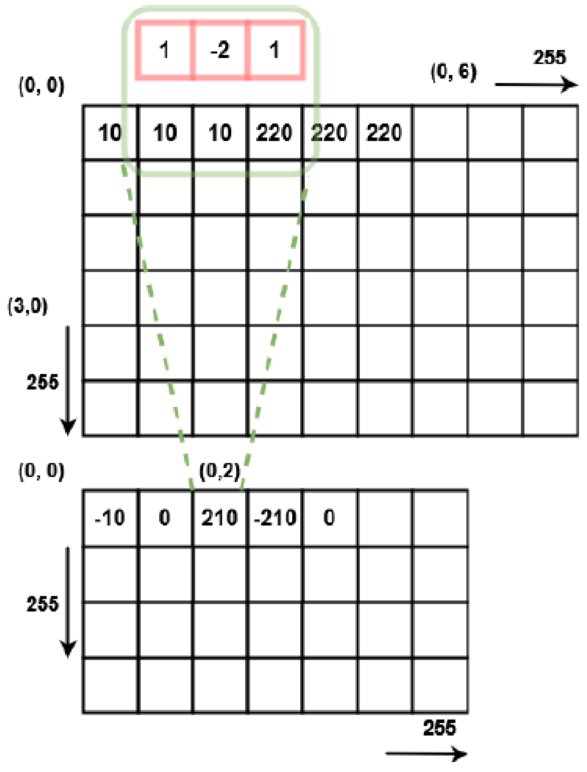


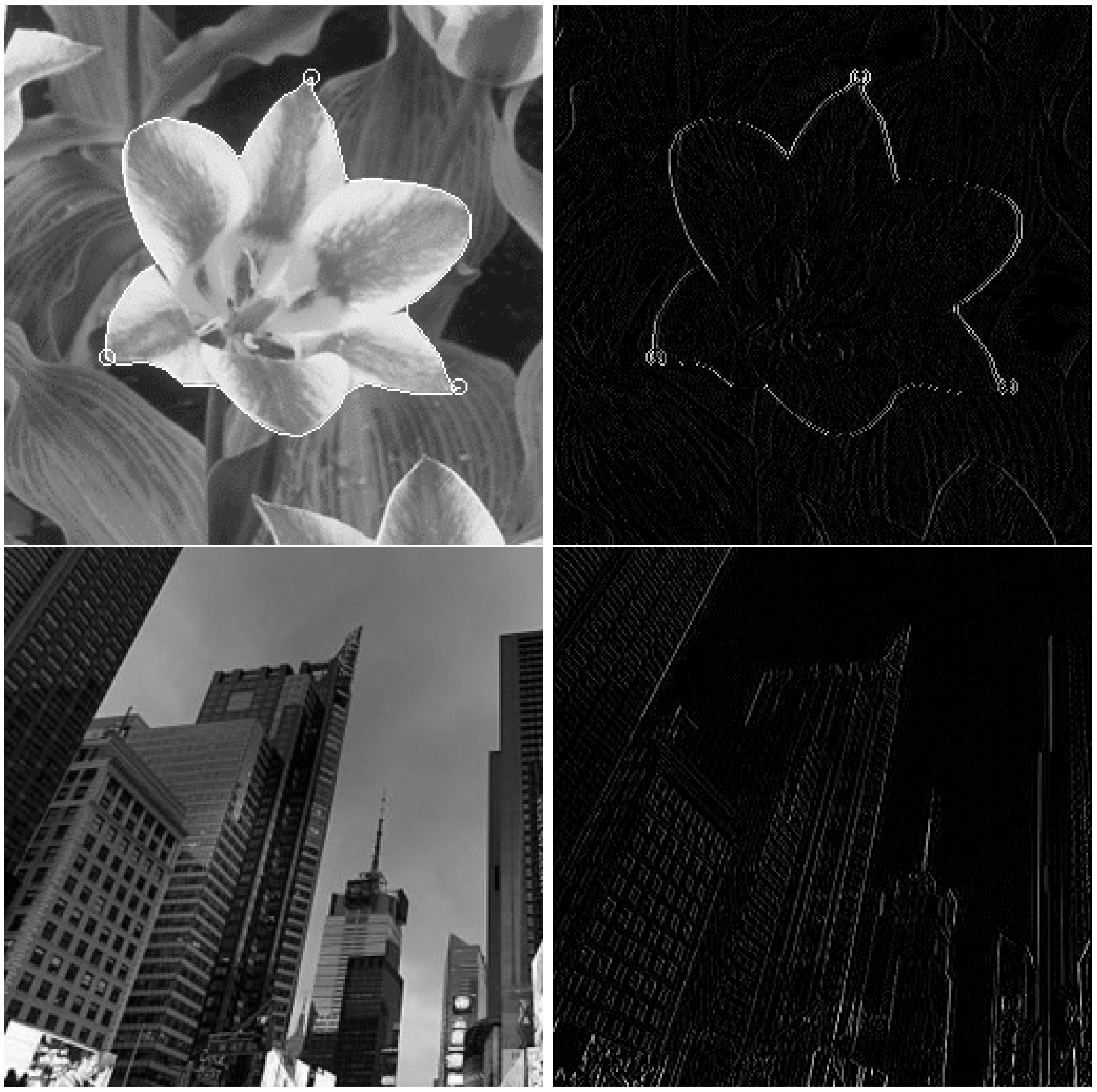
Figure: Illustration to perform 1D operation on grayscale image (Source: Assignment3 pdf)

NOTE: When the column index is j < 0 or j > 255, then assume the pixel value as 0. This is applicable in edge cases of the matrix. Also, in the final output image, any negative pixel values need to be clamped to 0 and values greater than 255 need to be clamped to 255 to ensure the values are stored in 8 bit unsigned format.

The resultant image is stored into new block RAM and displayed on the VGA monitor.

2.2 OUR UNDERSTANDING USING PYTHON

To understand the working of the gradient operation using the filter (1,-2,1), we made a python program that took a .png file of 256x256 pixels as input and gave an output of gradient calculated image. It detected the edges of the grayscale image. The algorithm is the same as above and we used the Pillow (PIL) library to convert the image into matrix, compute and convert it back to an image format. We have attached the code and along with the submission for your reference. Some of the images we tried are shown below :



2.2.1 OUR APPROACH

- We stored the input image in a read-only memory (ROM) and store the output image in random-access memory (RAM).
- The image is just a 256x256 2-D matrix with each pixel of 8 bit. These were stored at address 0 (0000_{16}) to address 65535 (1111_{16}). We need to traverse the complete matrix to perform the gradient operation. We kept a check on the row in which we are by the index i and the column by the index j. The address of the pixel was then given by $256 * i + j$.
- We passed the address to the input port of ROM to retrieve the data that has been stored at that address in ROM and stored the data in variables.
- We used three variables **p1**, **p2** and **p3** to store the pixel value at the three addresses needed to perform the gradient operation. And we passed the value of the p2 to p1 and p3 to p2 and read a new pixel in p3. Thus, optimising the process of reading the pixels from ROM.
- We started off with i and j being 0 and each time a pixel is read we increase j by 1 if j is not equal to 255 because j=255 marks the last pixel corresponding to a line hence, when j=255 and that pixel is also read we set j=0 and increase i by 1 due to initialisation of new line.
- The total number of pixels that have been read and operated is stored in a signal count and the reading operation from the ROM is continued till the value of count is less than 65536.

- The edge cases were handled carefully by taking the pixel to be zero when the column is either greater than 255 or less than 0.
- We then passed the RAM address in one clock cycle and then stored the output pixel at this address in the next clock cycle.
- To give a breakdown of the processes happening at each clock cycle:

  If the pixel count is less that 65536, then the image processing process is continued as per the following.

  1. *Clock Cycle 1* :

     Here, we check if we are at the boundaries of the matrix. If yes, we change the row to the next one. Else, we change it to the next column of the same row. We also update the pixel count by 1.
  2. *Clock Cycle 2* :

     Here, we assign the rom address of that particular pixel.
  3. *Clock Cycle 3* :

     Now, we check the column we are in. If at the leftmost edge, we increment the value we will set p1 variable as 0 as per the algorithm explained above. Else, we assign the previous value of p2 to p1.
  4. *Clock Cycle 4* :

     Here too we calculate the value of variable p2 with the logic that at the edge of the matrix, it reads the value stored else it is assigned the value of variable p3.
  5. *Clock Cycle 5 and 6*:

     Here, we assign the next rom address.
  6. *Clock Cycle 7*:

Now, we calculate the value of the variable p3. If at the rightmost edge, we need to assign as 0 as mentioned above in the algorithm else we assigned the stored value at that address.

7. *Clock Cycle 8 and 9* :

Here, we finally calculate the gradient and also, consider the cases of greater than 255 and less than 0 to maintain unsigned 8 bit value.

8. *Clock Cycle 10* :

Here we assign the outpixel 8 bit value of the calculated gradient.

9. *Clock Cycle 11* :

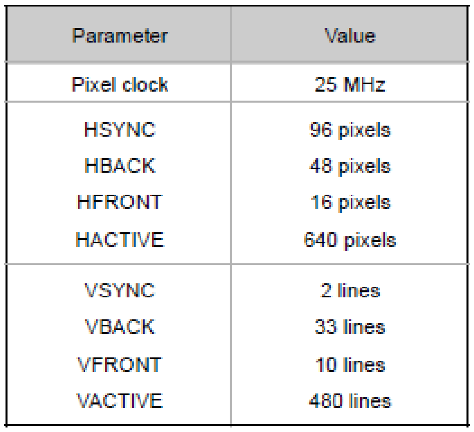
Now, we assign the outpixel value to ram input.

2.3 DISPLAY CONTROLLER

The image stored in the RAM needs to be displayed on a monitor for that we need to design this module. This module derives the externally connected VGA Display. Only when the electron beam is travelling ”forward” (from left to right and from top to bottom), and not when it is reset back to the left or top border of the display, is information displayed. Therefore, a significant portion of the possible display time is wasted during ”blanking” intervals when the beam is stabilized and reset to start a new horizontal or vertical display pass. The display resolution is determined by the size of the beams, the frequency at which the beam may be tracked across the display, and the frequency at which the electron beam can be adjusted.

In this assignment we implemented (640*480@60Hz) the following timings.

Table 1: Timing specification for 640x480@60Hz



Parameter	Value
Pixel clock	25 MHz
HSYNC	96 pixels
HBACK	48 pixels
HFRONT	16 pixels
HACTIVE	640 pixels
VSYNC	2 lines
VBACK	33 lines
VFRONT	10 lines
VACTIVE	480 lines

So, the total number of pixels in one horizontal line (active+blanking) will be:

$$\mathbf{HTOT = HACTIVE + HBACK + HFRONT + HSYNC}$$

$$800 = 640 + 48 + 16 + 96$$

Total number of lines in one frame will be :

$$\mathbf{VTOT = VACTIVE + VBACK + VFRONT + VSYNC}$$

$$525 = 480 + 33 + 10 + 2$$

Total number of pixels in one frame are, 420000 which has to be displayed in a fixed amount of time that is 16.66 ms (60 Hz or 60 frames per seconds). Thus, time period for the clock for one pixel will be (25.2 Mhz):

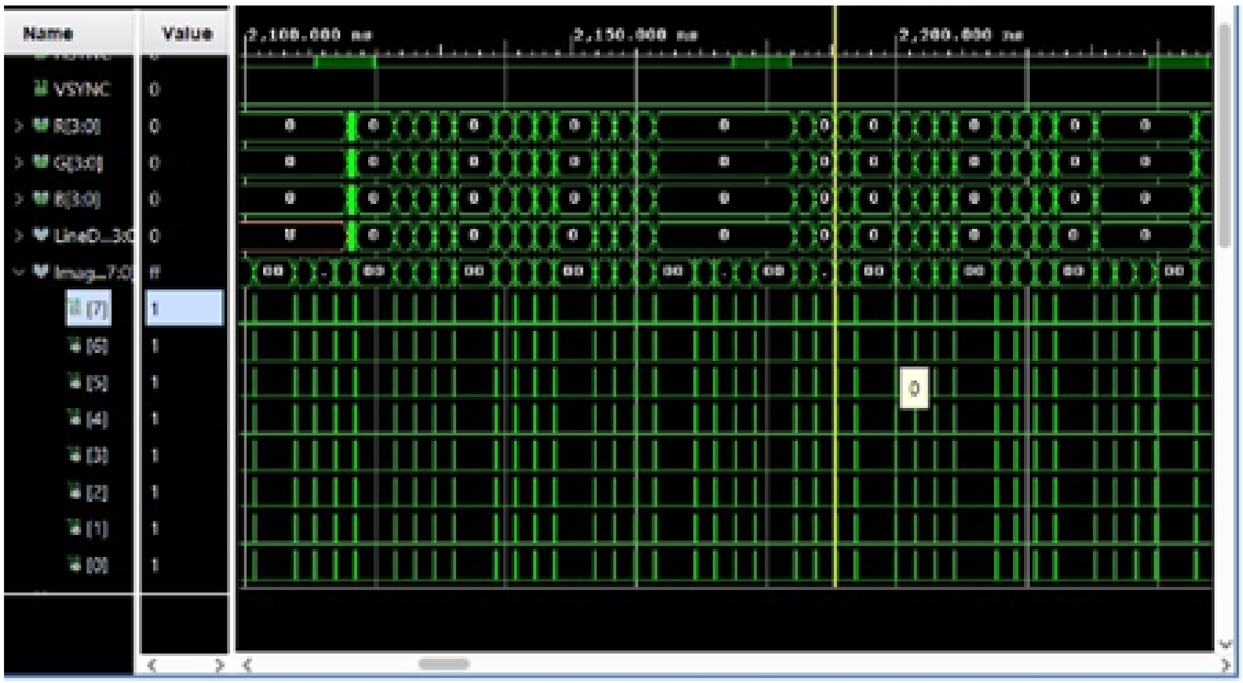
$$420000 * 60 = 25200000$$

The horizontal pixel counter will keep track of pixels in the line, input will be pixel clock and reset signal. Output will be pixel count (Hcoordinate) and line complete signal (indicating

end of the line, which will serve as an enable signal to the vertical counter). Counter will start from 0 and increment till Hcoordinate is equal to HTOT then reset the count to 0.
The vertical line counter will keep track of lines in a frame, input will be pixel clock, reset and enable signal, Output will be line count (Vcoordinate). Counter will start from 0 and increment till Vcoordinate is equal to VTOT then reset the count to 0.
Based on the Hcoordinate, HSYNC signal will be driven HIGH for the pixel count 0 to HACTIVE + HFRONT and changed to LOW for duration HYSNC. Similarly, based on Vcoordinate, VSYNC will be driven HIGH from line 0 to VACTIVE + VFRONT and LOW for VSYNC duration.
First, to display grayscale image using VGA RGB format, each colour line of RGB should be set to the same value that is R=G=B. But basys3 has 4-bit width per colour line, so image 8-bit value should be truncated to 4-bit value via discarding lower 4 bits (i.e. using 4 MSB bits). This will reduce the 256 colour level to 16 colour level. For example, if pixel at location (0,0) has 8-bit value 10110001 then video data in RGB format will be 1011 1011 1011 (12-bit RGB)
As mentioned in the pdf, the above logic is implemented in the VGA display controller.

3 OUTPUT DISPLAY AND WAVEFORM

Simulation waveform :

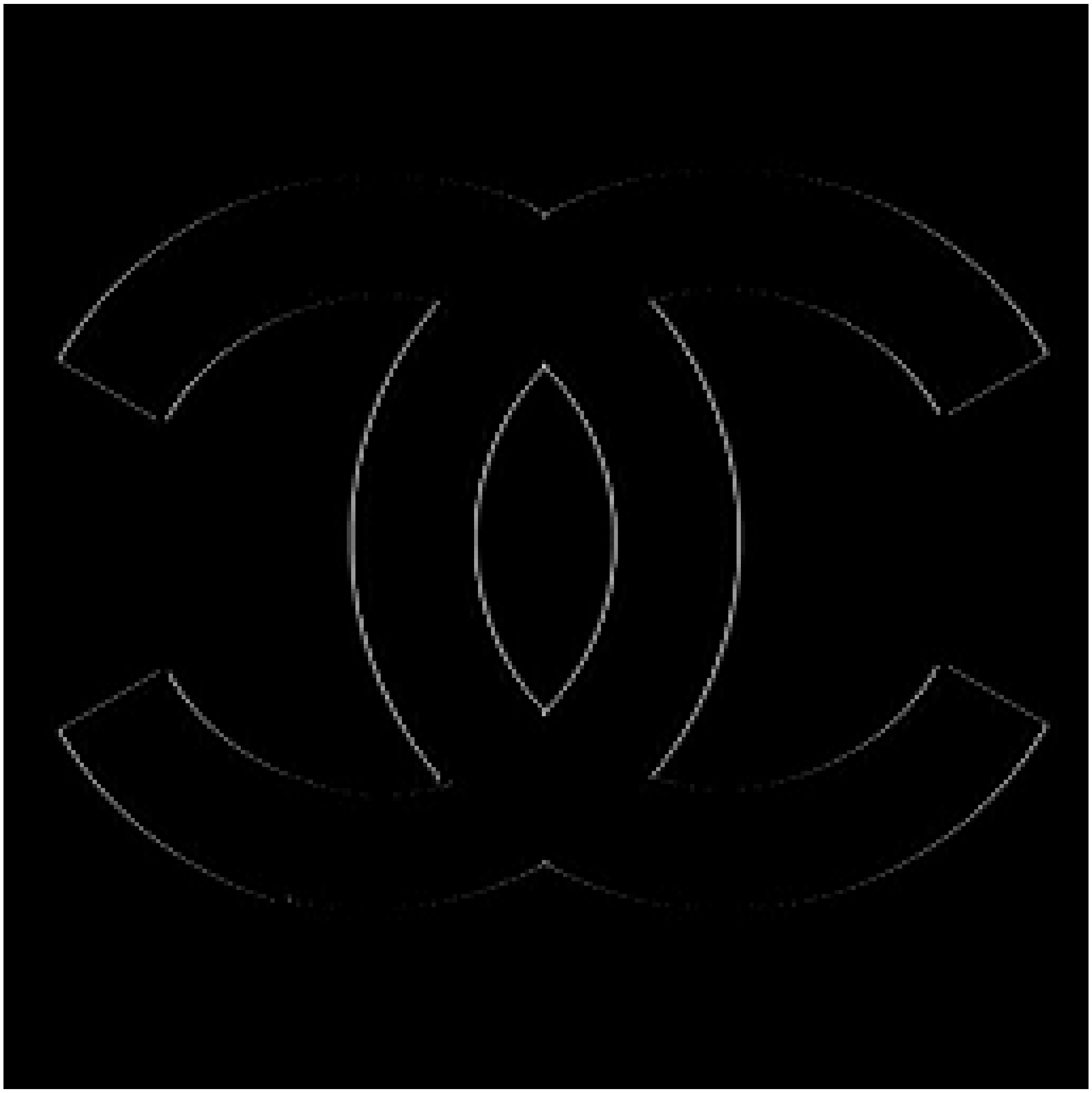


Original Image :

Output Image :

Original Image :

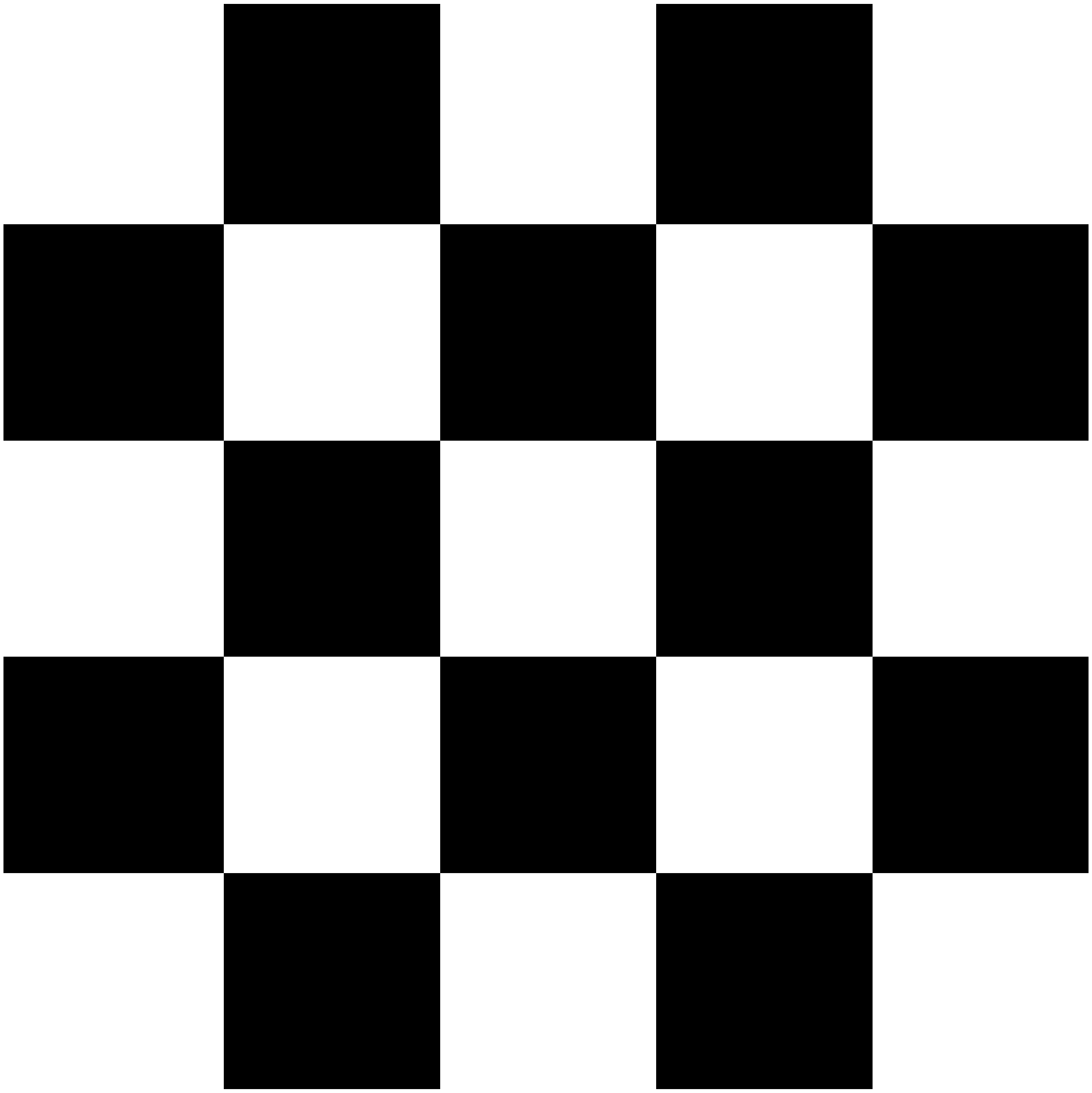
Output Image :



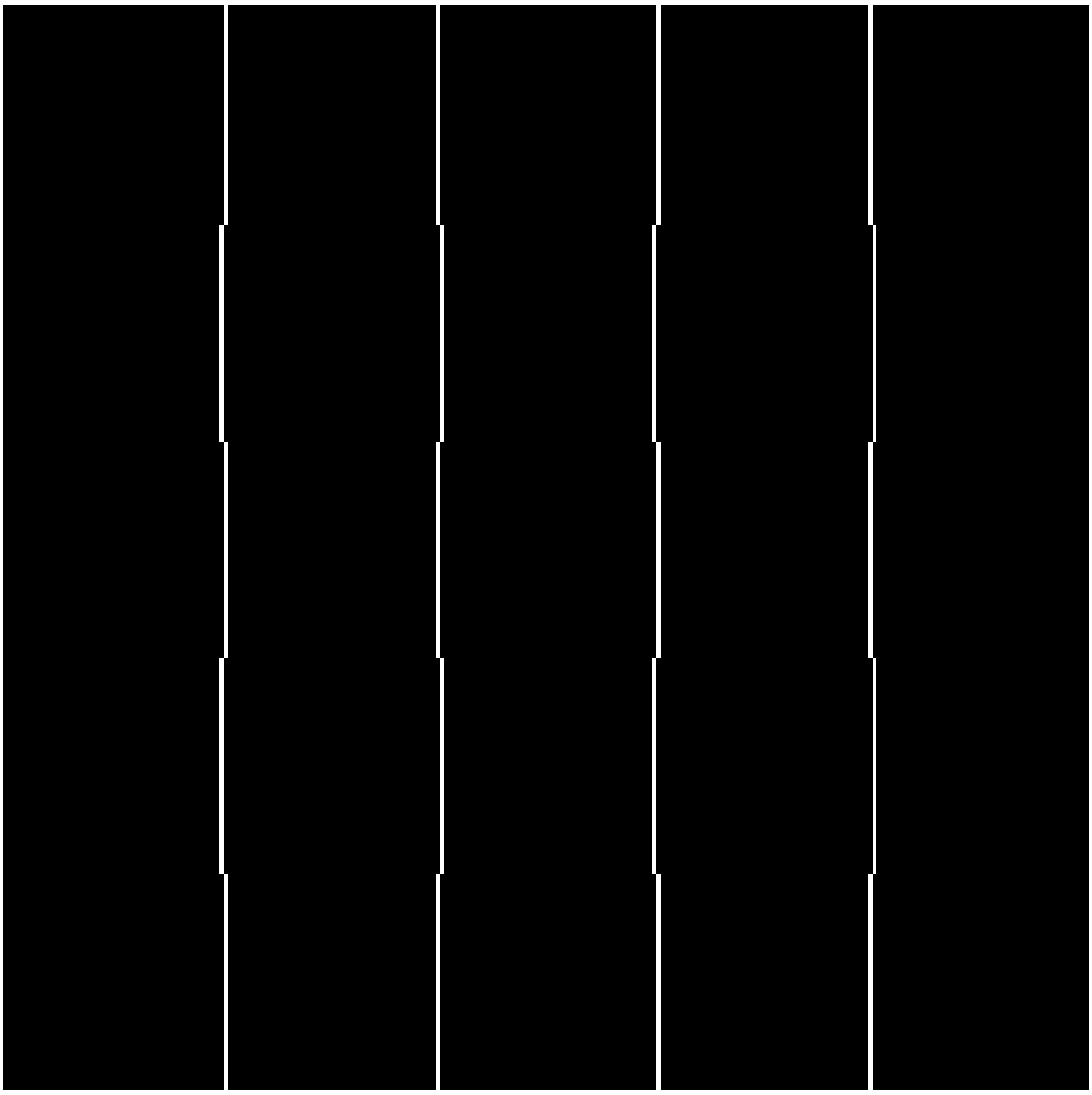
Original Image :

Output Image :

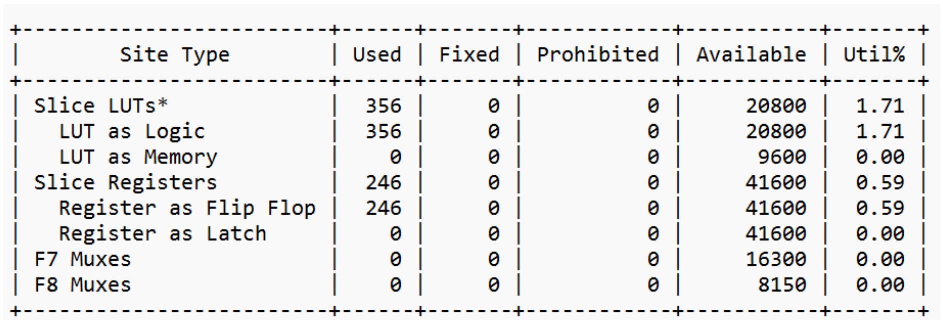
Original Image :



Output Image :



4 ATTACHMENTS AND REFERENCES



```
+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |  356 |     0 |          0 |     20800 |  1.71 |
|   LUT as Logic          |  356 |     0 |          0 |     20800 |  1.71 |
|   LUT as Memory         |    0 |     0 |          0 |      9600 |  0.00 |
| Slice Registers         |  246 |     0 |          0 |     41600 |  0.59 |
|   Register as Flip Flop |  246 |     0 |          0 |     41600 |  0.59 |
|   Register as Latch     |    0 |     0 |          0 |     41600 |  0.00 |
| F7 Muxes                |    0 |     0 |          0 |     16300 |  0.00 |
| F8 Muxes                |    0 |     0 |          0 |      8150 |  0.00 |
+-------------------------+------+-------+------------+-----------+-------+
```

LUTs

The submission of the assignment consists of six files - the constraint file `basys3.xdc`, the VHDL code file `DisplayController_gate.vhd`, the VHDL code `clock_divider.vhd`, the `DisplayController.bit` file generated after running synthesis and implementing the synthesized design, `DisplayController_utilization_synth.rpt` where we can see the LUTs utilised.

For reference, we used the references given to us in the assignment.