Abstract

In this lab we learned to display image by using the DE1-SoC computer’s video-out port on a VGA terminal. This lab also teaches to convert a software pseudo code to a System Verilog code which ensures a better understanding System Verilog. In this process, we also learned how pixel buffer and double buffering work and later implement a line drawing algorithm using pixel buffer.

Introduction

The DE1-SoC computer includes a video-out port with a VGA controller that can be connected to a standard VGA monitor. The VGA controller supports a screen resolution of 640×480. The image that is displayed by the VGA controller is derived from two sources: a pixel buffer, and a character buffer. We only used the pixel buffer in this lab hence we did not implement anything using the character buffer. Through the procedures mentioned in this report, I was able to create an line drawer algorithm for hardware. Doing this gave me a better understanding of how hardware logic works and its implementation.

**Pixel Buffer**

The pixel buffer for the video-out port holds the data (color) for each pixel that is displayed by the VGA controller. As illustrated in Figure 1, the pixel buffer provides an image resolution of 320×240 pixels, with the coordinate 0,0 being at the top-left corner of the image. Since the VGA controller supports the screen resolution of 640x480, each of the pixel values in the pixel buffer is replicated in both the x and y dimensions when it is being displayed on the VGA screen.

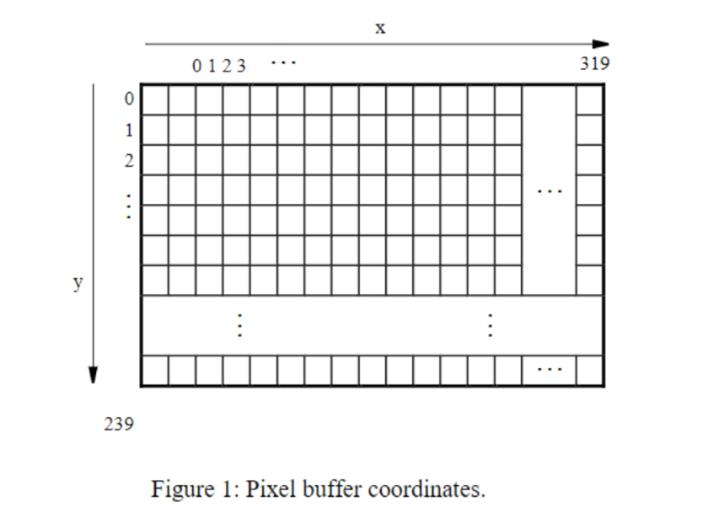
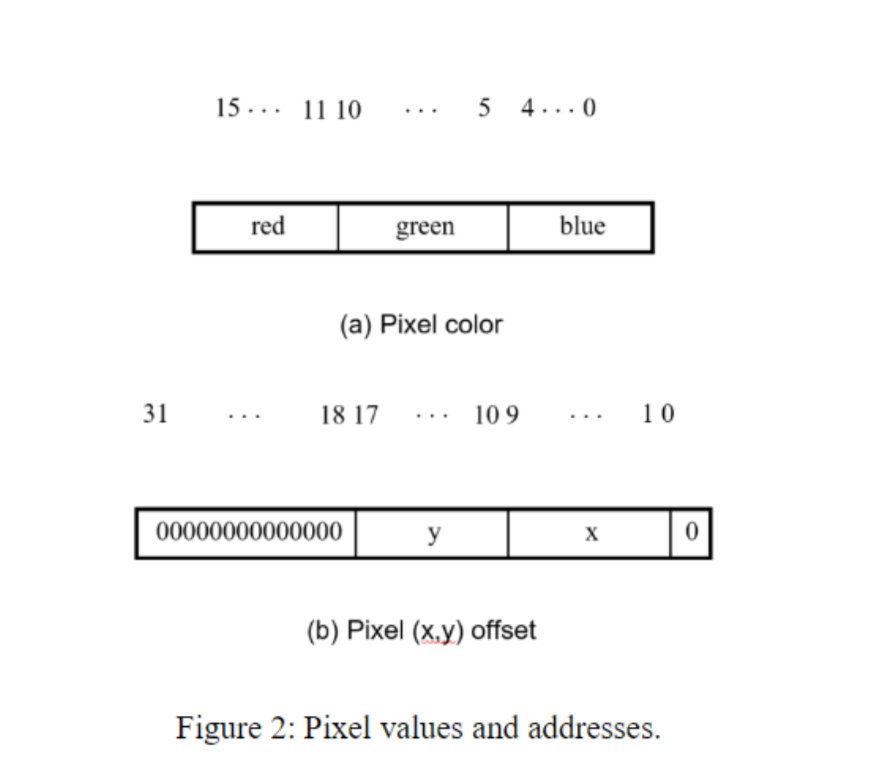
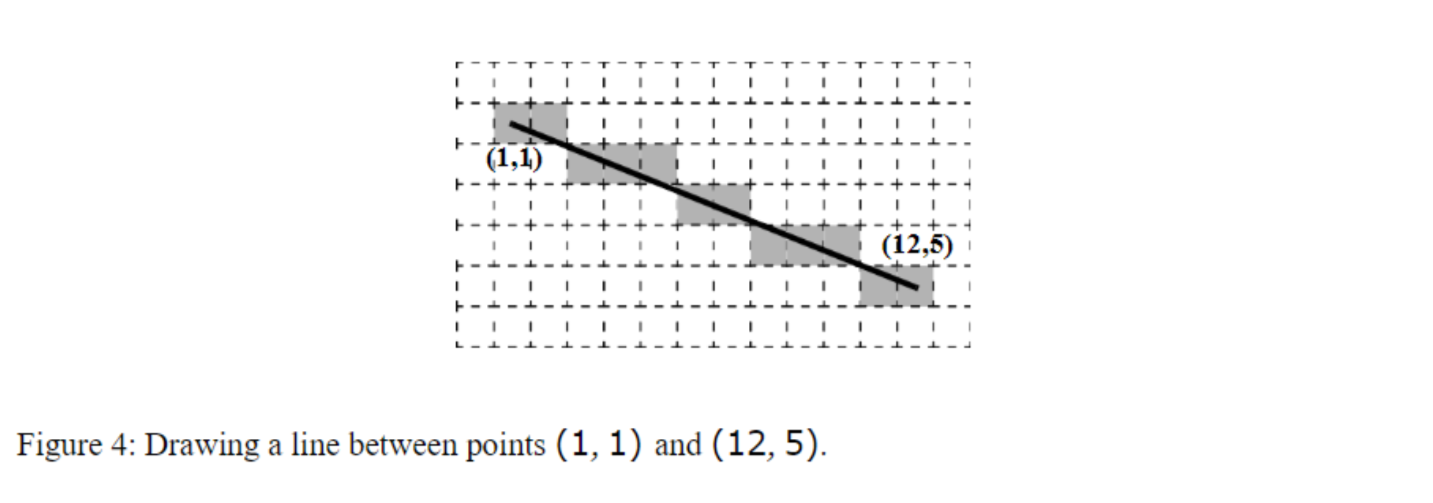


Figure 2a shows that each pixel color is represented as a 16-bit half word, with five bits for the blue and red components, and six bits for green. As depicted in part b of Figure 2, pixels are addressed in the pixel buffer by using the combination of a base address and an x, y offset. In the DE1-SoC Computer the default address of the pixel buffer is 0xC8000000, which corresponds to the starting address of the FPGA on-chip memory. Using this scheme, the pixel at location 0,0 has the address 0xC8000000, the pixel 1,0 has the address base + (00000000 000000001 0)2 = 0xC8000002, the pixel 0,1 has the address base+ (00000001 000000000 0)2= 0xC8000400, and the pixel at location 319,239 has the address base +(11101111 100111111 0)2 = 0xC803BE7E.

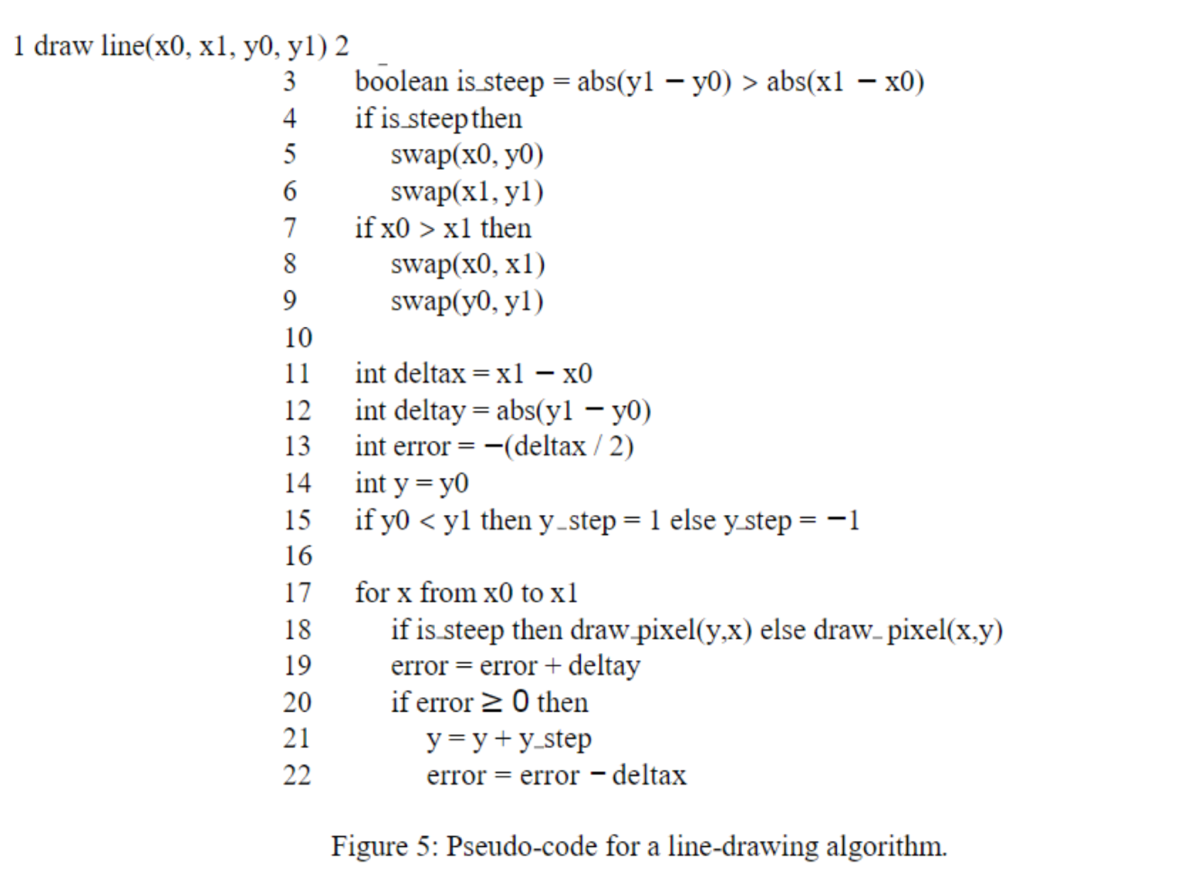


We can create an image by writing color values into the pixel addresses as described above. A dedicated pixel buffer controller reads this pixel data from the memory and sends it to the VGA display. The controller reads the pixel data in sequential order, starting with the pixel data corresponding to the upper-left corner of the VGA screen and proceeding to read the whole buffer until it reaches the data for the lower-right corner. This process is then repeated, continuously. You can modify the pixel data at any time, by writing to the pixel addresses. Writes to the pixel buffer are automatically inter leaved in the hardware with the read operations that are performed by the pixel buffer controller.

In this lab, you also learned how to implement a simple line-drawing algorithm. Drawing a line on a screen requires coloring pixels between two points (x1, y1) and (x2, y2), such that the pixels represent the desired line as closely as possible. Consider the example in Figure 4, where we want to draw a line between points (1, 1) and (12, 5). The squares in the figure represent the location and size of pixels on the screen. As indicated in the figure, we cannot draw the line precisely—we can only draw a shape that is similar to the line by coloring the pixels that fall closest to the line’s ideal location on the screen. We can use algebra to determine which pixels to color. This is done by using the end points and the slope of the line. The slope of our example line is slope = (y2- y1) / (x2 - x1) = 4 / 11. Starting at point (1, 1) we move along the x axis and compute the y coordinate for the line as follows:



Thus, for column x = 2, the y location of the pixel is 1 + 4 × (2 – 1) = 14. Since pixel locations are defined by integer values we round the y coordinate to the nearest integer, and determine that in column x = 2 we should color the pixel at y = 1. For column x = 3 we perform the calculation y = 1 + 4 × (3 − 1) = 18, and round the result to y = 2. Similarly, we perform such computations for each column between x1 and x2. We were also provided with pseudo code for software which is shown in Figure 5.



Procedures

* Task 1
  + Downloaded the required files from Canvas.
  + Understood how do VGA\_framebuffer.sv and DE1-SoC.sv work.
  + Compiled the project on Quartus to expect a black screen.
* Task 2
  + Understood the pseudo code in Figure 5 and what the input and output ports are in line\_drawer.sv.
  + While write the Bresenham’s algorithm I followed these approaches:
    1. Assumed x0 = y0 or x1 = y1 and use the line\_drawer.sv file to draw perfectly straight lines.
    2. Assumed that (x0, y0) will be (0, 0) and x1 = y1. That means, designed an algorithm that only draws perfectly diagonal lines from the origin.

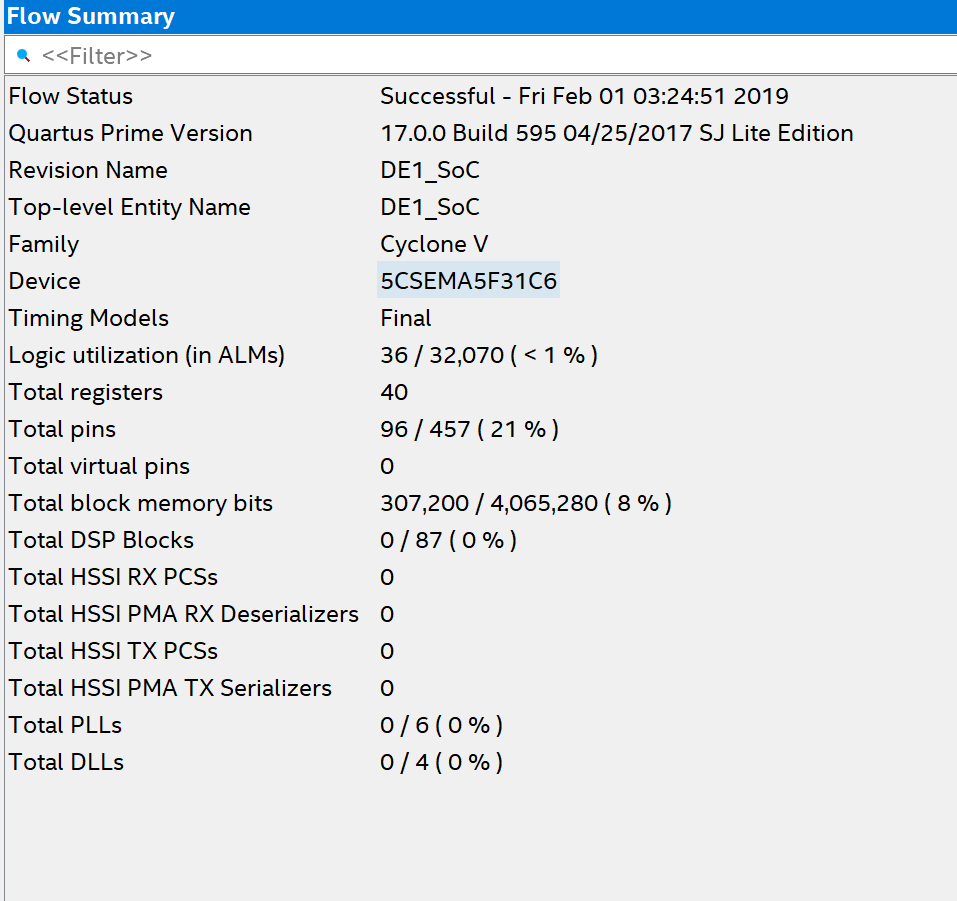
3. Modified my algorithm to draw perfectly diagonal lines from any arbitrary starting point.

4. Modified my algorithm to handle lines with gradual slopes, such as a line from (0, 0) to (100, 20).

* Task 3
  + Modified the DE1\_SoC.sv file to implement the following:
    1. Used my line algorithm to draw a line on the monitor and animate it to move around the screen.
    2. Tried to implement a reset that, when activated, clears the screen by drawing every pixel to be black.

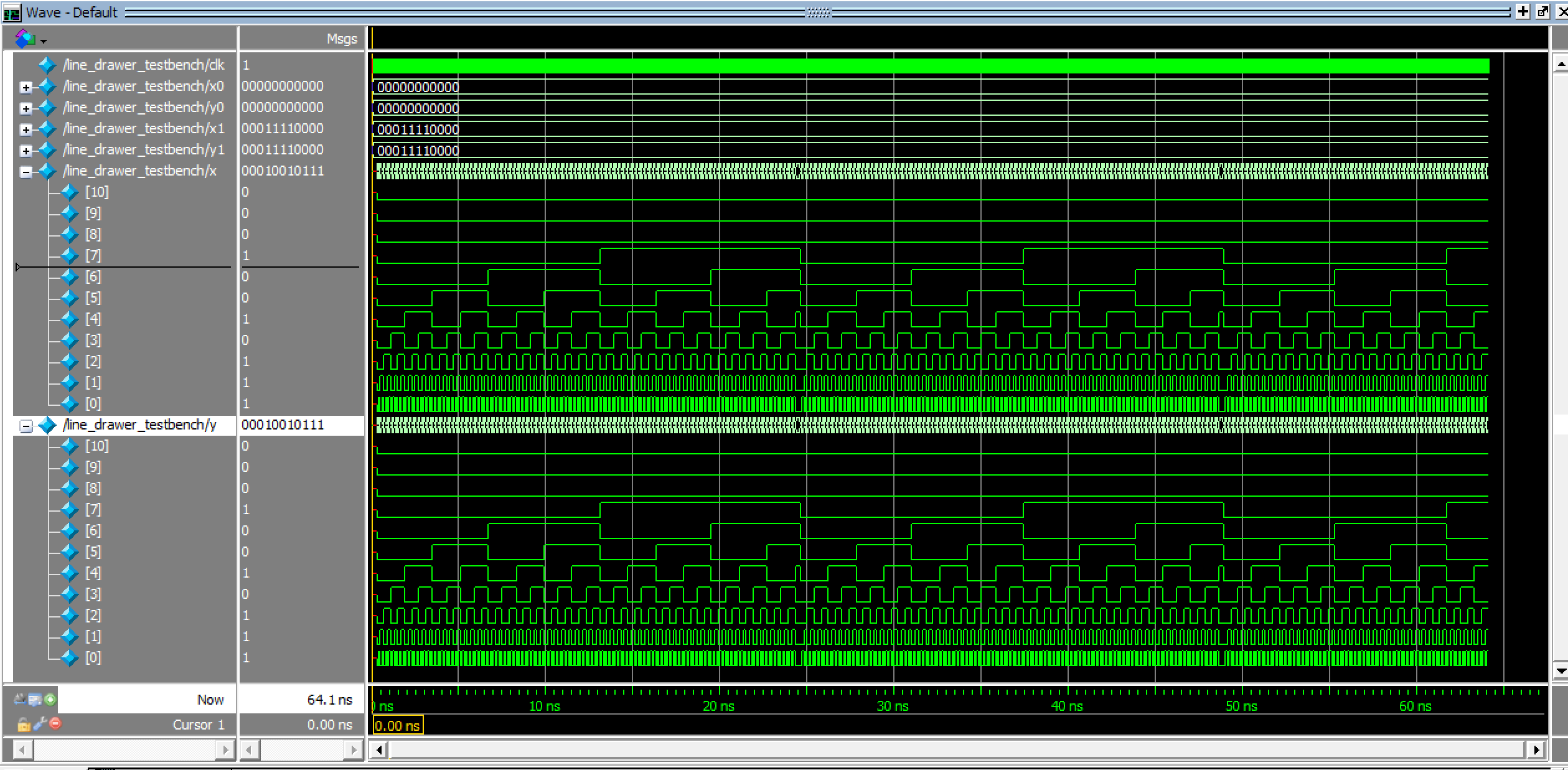
Results

* Task 1:
  + No problems were faced during this task.
  + Flow Summary

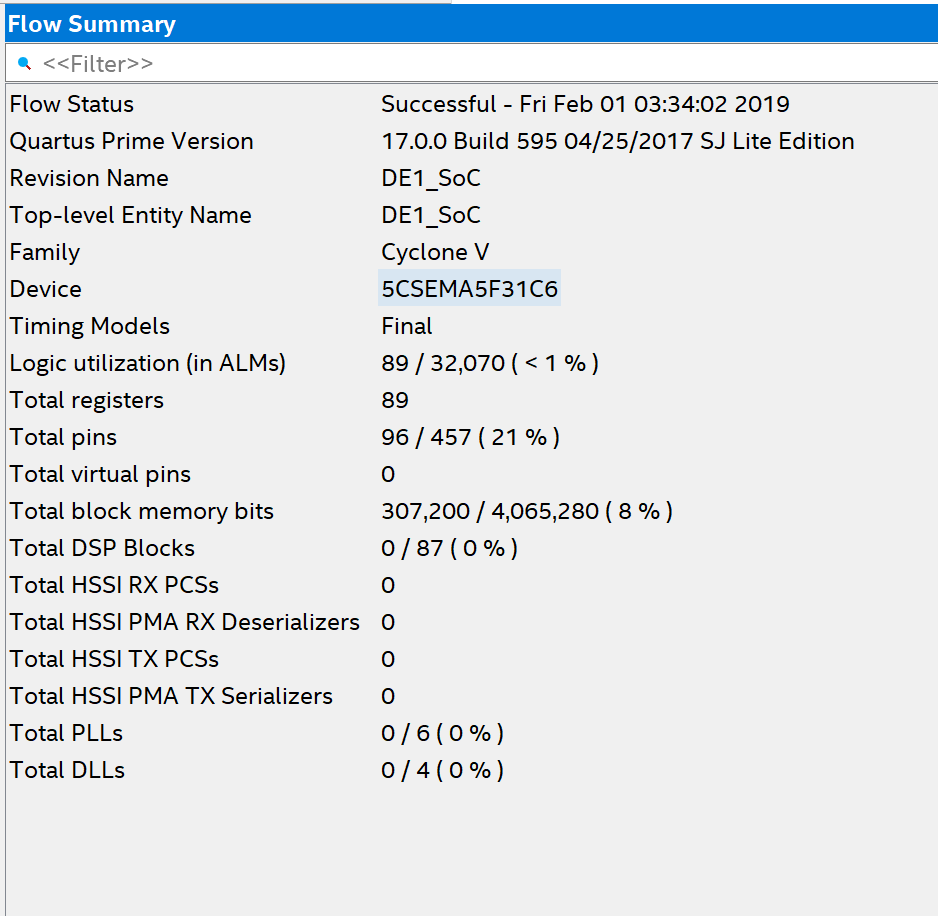


* Task 2:

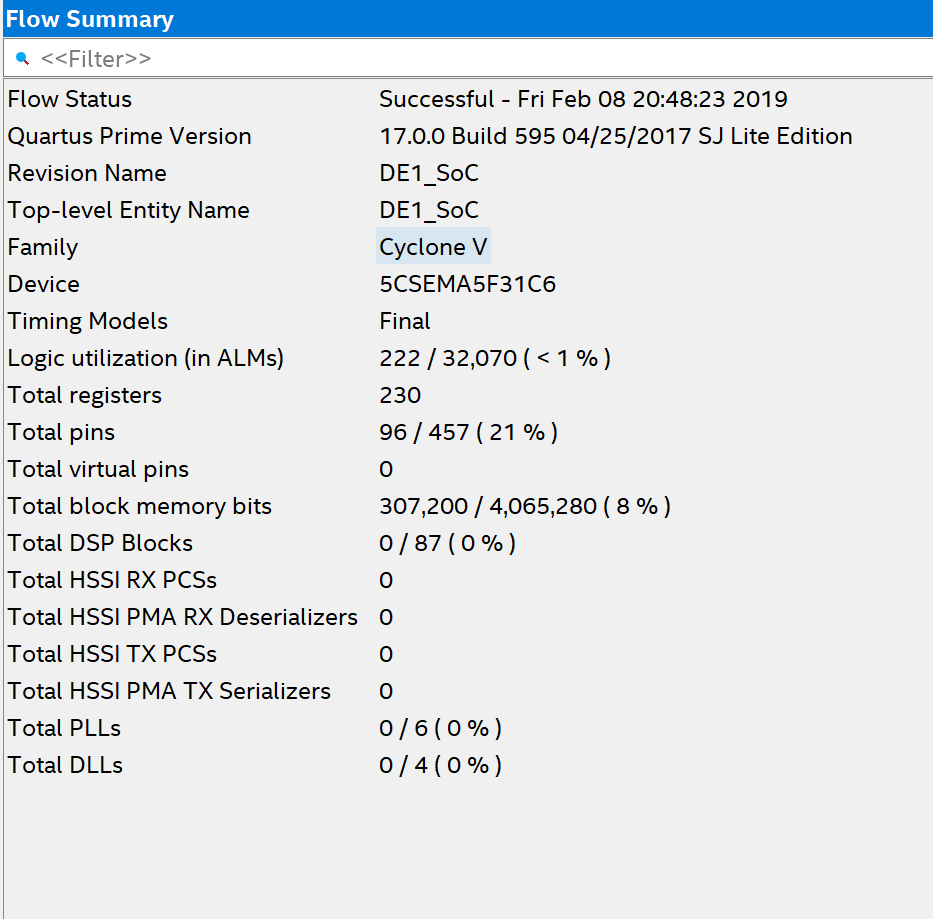
* + It took a lot of time to figure out that I have to use states to do this problem. I kept getting constant driver error for x , y and error.
  + line\_drawer\_testbench waveform:
    - It showed expected output as both x and y coordinates are the same and they all lie on the line from (0, 0) to (240, 240). The only output we are getting here are coordinates (1, 1), (2, 2), (3, 3) and so on till (240, 240).



* + Flow Summary



* Task 3
  + All the waveforms were the same as the previous task.
  + I couldn’t implement the reset feature because whenever I tried to call my reset module it threw an error or gave a not expected output of random lines on the monitor.
  + Flow Summary



Analysis

This lab gave us an introduction on pixel buffer and Bresenham’s line algorithm in System Verilog. The final product took two input coordinates and then colored specific pixels which fell on the line between the coordinates. The line algorithm working in several states. In the first state, I assign the coordinates to temporary variable by checking which coordinate value is the highest. With that I also change the value of delta\_x and delta\_y which later helps me to add and subtract it with the error variable. In the second state, I assign the error = -delta\_x/2 and initialize the temp output coordinates. In the third state, which is the last state, I assign the output coordinates to the temp variable and increment one in them so that the program checks all the coordinates. I also update error accordingly in this state. The commented code in line\_drawer.sv is the code I first wrote working on the approaches mentioned in the lab specifications.

Conclusion

This lab took a total time of 30 hours including reading, planning, design, coding, debugging, testing etc. Problems were faced during creating line\_drawer.sv and implementing the reset feature. In this lab, we learned about pixel buffer, double buffering and how to implement the pixel buffer. Not only this we implemented a fairly complex algorithm in hardware from a software based pseudo code, which will further help to develop better logic for the upcoming labs.