EE 371 Lab3 Report: HDMI-to-VGA Line Drawer

Ziyi Huang

ECE, Embedded System

University of Washington

April 28, 2019

Introduction

The purpose of this lab is to control the FPGA to display images on a VGA terminal. Three procedures were performed. The rest of report will describe in details about the procedures, the results, and the problems faced & feedback of the lab.

Background

The VGA monitor used in this lab supported a resolution of 640x480. Its display could be derived from a pixel buffer or a character buffer. However, only the pixel buffer was used for the scope of this lab.

The pixel buffer used by the FPGA video-out port held data with a resolution of 320x240 pixels. Therefore, on the actual display, each pixel was replicated twice - once in x and once in y direction. The pixel map of the buffer is depicted in Figure **??**.

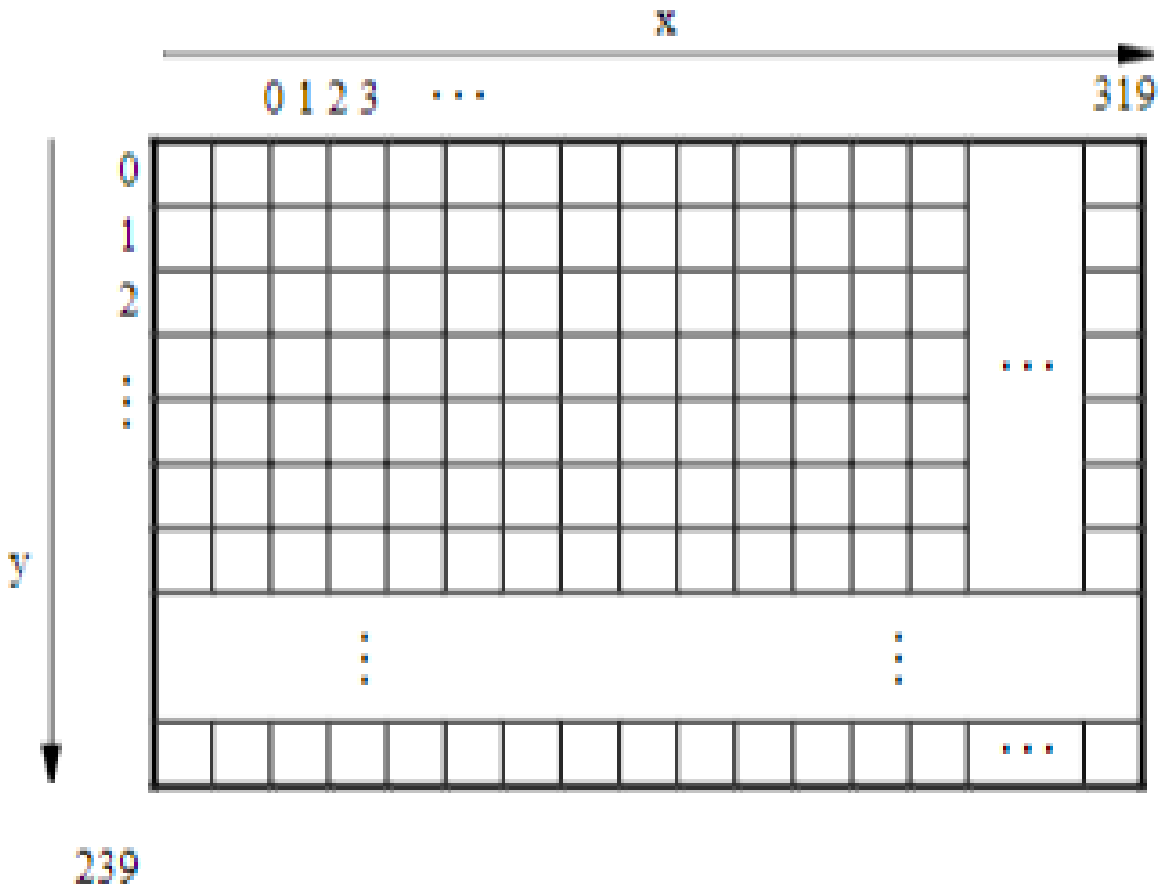

Figure 1: The pixel map of the pixel buffer

The block diagram of the final system is presented in Figure 2

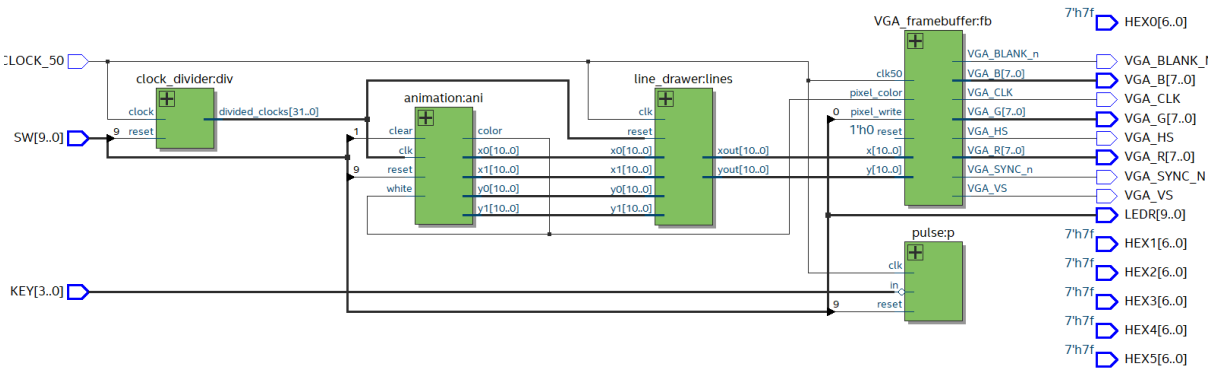

Figure 2: The block diagram of the final system

Procedure

Task 1

The goal of this task is to warm up with the spec and make sure the VGA sample file works correctly. It required downloading the provided starter kit from the course website and directly synthesizing them to the FPGA. Then, it required connecting the HDMI-to-VGA adapter to the FPGA video-out port. Finally, it required making sure that the monitor turned on and showed black back light.

Task 2

The goal of this task is to implement the Bresenham's drawing algorithm for monitors in SystemVerilog. The difficulty lied in converting the sequential logic of C++ pseudo-code into hardware wirings of VHDL. The pseudo-code used in presented in Figure 3. Many test cases were developed to verify that the algorithm functioned correctly.

The algorithm could be divided into three stages: the pre-processing, the updates, and the finalization. The first step made adjustments to the coordinates if the line was steep or extended backwards. Its goal was to generate the optimal coordinates to smoothen the line and made sure that the drawing always went from x0 to x1. The next step increment x by 1 while adjusted the y increment according to feedbacks. The last step flipped the x and y coordinates if necessary, i.e. when the line was steep. This step was caused by the pre-processing step, where the algorithm

flipped x and y to generate a smooth line.

The implementation of the pre-processing step demonstrated some similarities and differences between VHDL and higher-level languages. I used functions in SystemVerilog to implement the abs(), which could then be called in the same ways as in the higher-level languages. To do the swapping, I used 6 intermediate registers in a combinational logic, simply because that the values of the input registers could not be modified.

Before the update steps, I created even more registers to store necessary information, such as error, deltax, and deltay. An additional error_next register was created because I found it improved the readability of my sequential logic. After the setup, I used a always_ff to do the update, which reset/initialized the x and the y to x0 and y0 only when reset was signaled. This is a standard way of implementing D-flip-flops, but in this case it caused problems. The reason was that my x and y would not be initialized if the system didn't start in the reset state and hence would feed noisy inputs to the pixel drawer module, causing it to draw random lines. But overall, this implementation allowed the x and y to update correctly and synchronously.

The last step was just a simple combinational logic that assign x output to y and y output to x if the line was steep. Again, this was because the first step flipped them in this way to smoothen the drawing.

Seven test cases were implemented to verify the algorithm. The ModelSim simulations would be presented in the Results section.

```
1 draw line(x0, x1, y0, y1)
2
3     boolean is_steep = abs(y1 − y0) > abs(x1 − x0)
4     if is_steep then
5         swap(x0, y0)
6         swap(x1, y1)
7     if x0 > x1 then
8         swap(x0, x1)
9         swap(y0, y1)
10
11    int deltax = x1 − x0
12    int deltay = abs(y1 − y0)
13    int error = −(deltax / 2)
14    int y = y0
15    if y0 < y1 then y_step = 1 else y_step = −1
16
17    for x from x0 to x1
18        if is_steep then draw_pixel(y,x) else draw_ pixel(x,y)
19        error = error + deltay
20        if error ≥ 0 then
21            y = y + y_step
22            error = error − deltax
```

Figure 3: Pseudo-code of Bresenham's Drawing Algorithm

Task 3

The goal of this task is to animate a line moving around the VGA screen. Lines of different steepness should be demonstrated. A screen clear control should also be supported. The difficulty lied in finding a way of erasing the line after drawing it, as well as synchronizing two clocks so that the lines were drawn fast enough while the animation changed slow enough.

I implemented a five-stage FSM to do this task. It is presented in Figure 4. It covered five different steepness with one of them in the reversed direction (x0 < x1). It operated under a slower clock, rather than the 50MHz built-in clock used by pixel and line drawing module. The FSM was straight forward - state changed at every positive clock edge and the system generated new endpoints. However, to erase the old line, I had to consider the current color used for drawing. If it was white, the FSM would stay in the current state and changed the color to black so that a black line would be drawn at the same place at the next cycle. The screen clear

was hard at first because I wanted to use a key as the control. But using a switch made it super easy since the color just equaled to (other logic) & SW. The simulation for the animation is presented in the Results section.

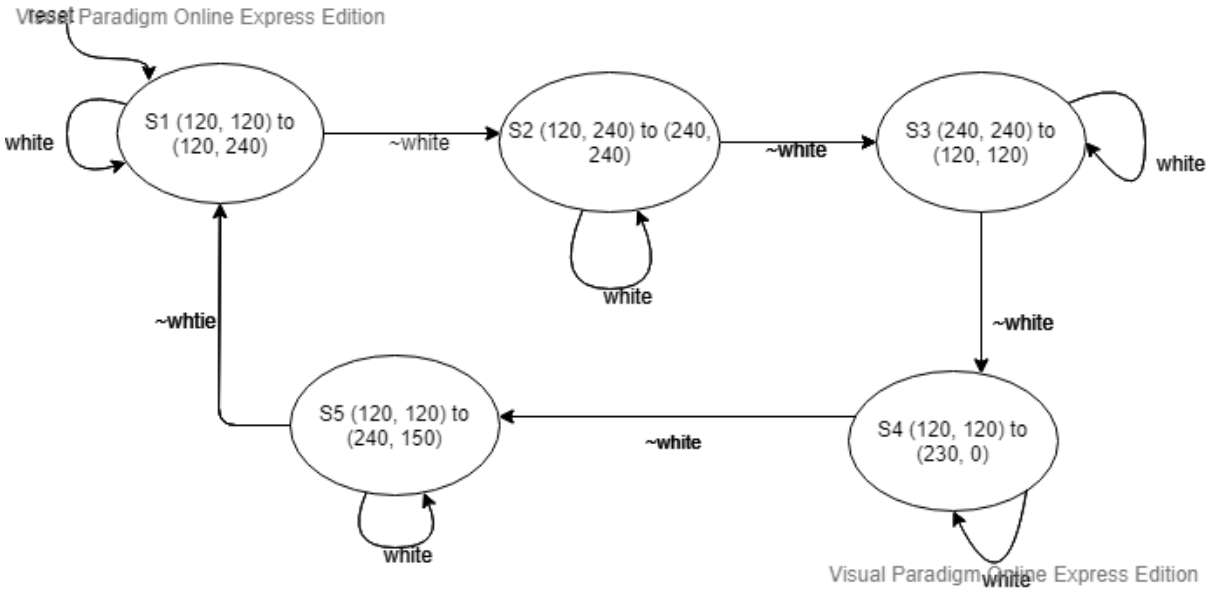

Figure 4: FSM

Results

Task 1

A turned on VGA monitor.

Task 2

The main results of this task are the simulations for the test cases. The test cases were:

- Horizontal line (0, 120) to (240, 120)
- Vertical line (120, 0) to (120, 240)
- Backwards vertical line (120, 240) to (120, 0)
- Diagonal line from the origin, i.e. (0,0) to (240, 240)
- Diagonal line with an arbitrary starting point (120, 120) to (200, 200)
- Gradual-slope line that was steep (0, 0) to (120, 240)

- Gradual-slope line that was not steep (120, 120) to (200, 150)

The simulations of the test cases were included in Figure 5-11

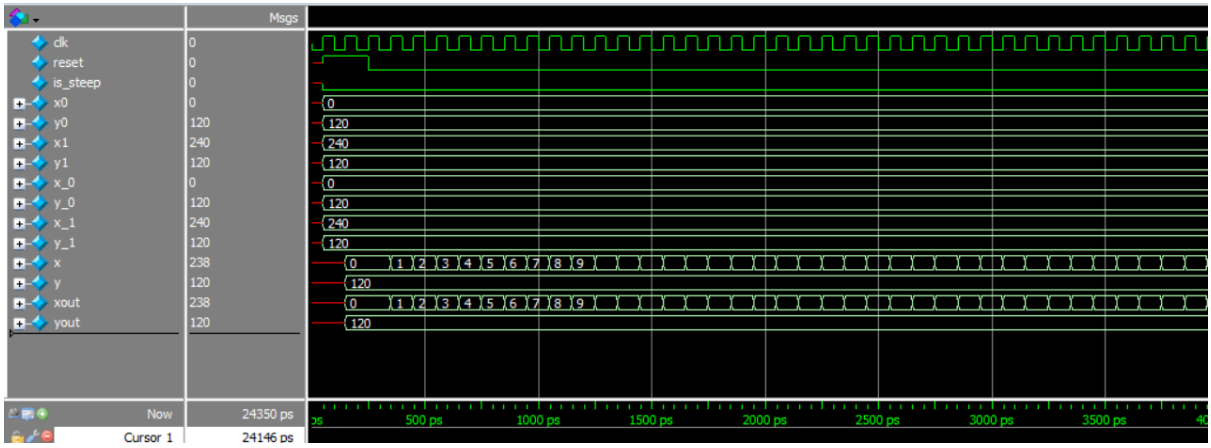

Figure 5: Simulation for VGA drawing a horizontal line

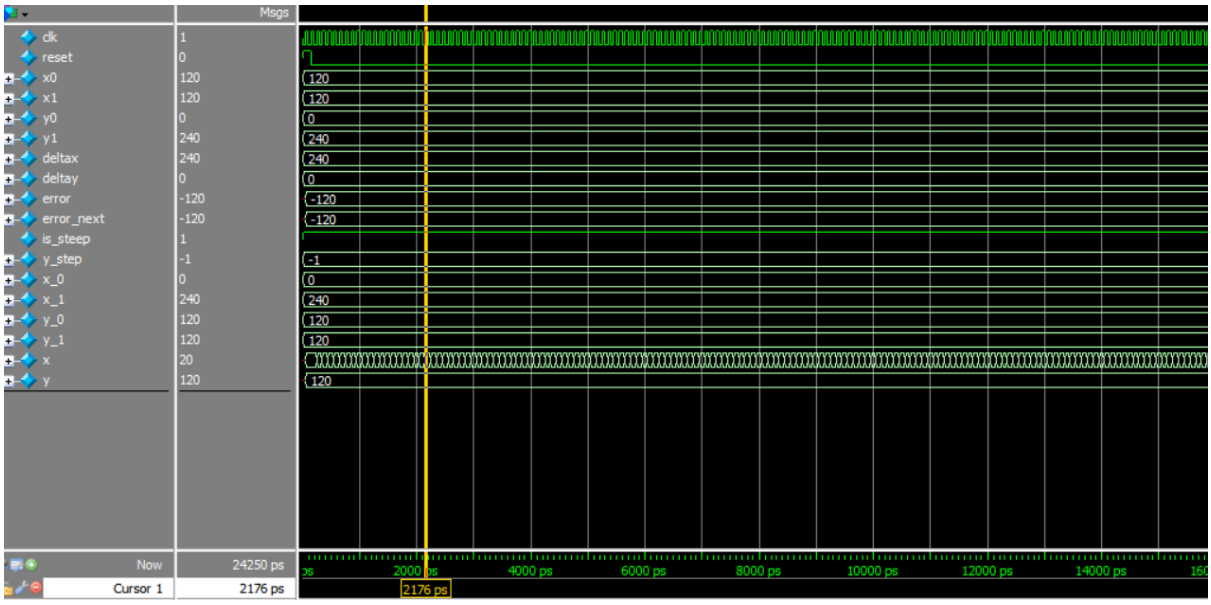

Figure 6: Simulation for VGA drawing a vertical line

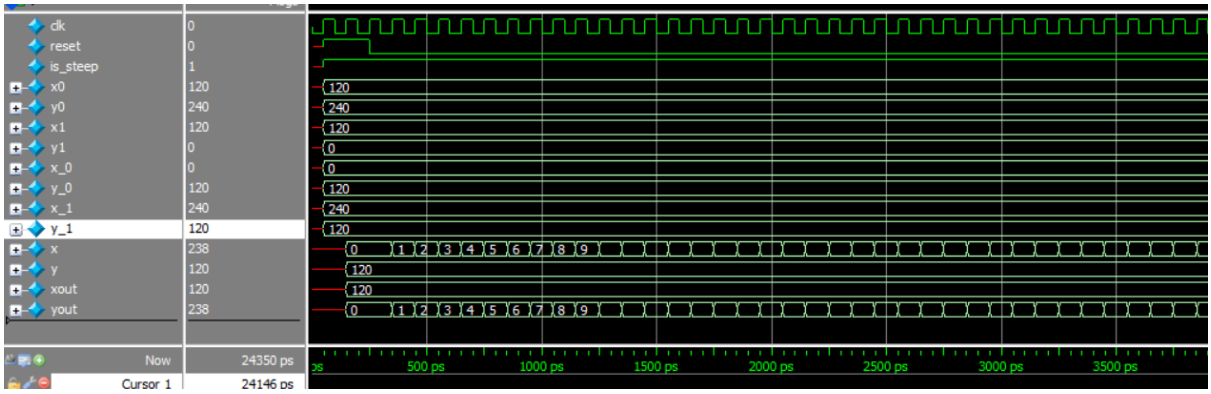

Figure 7: Simulation for VGA drawing a vertical line backwards

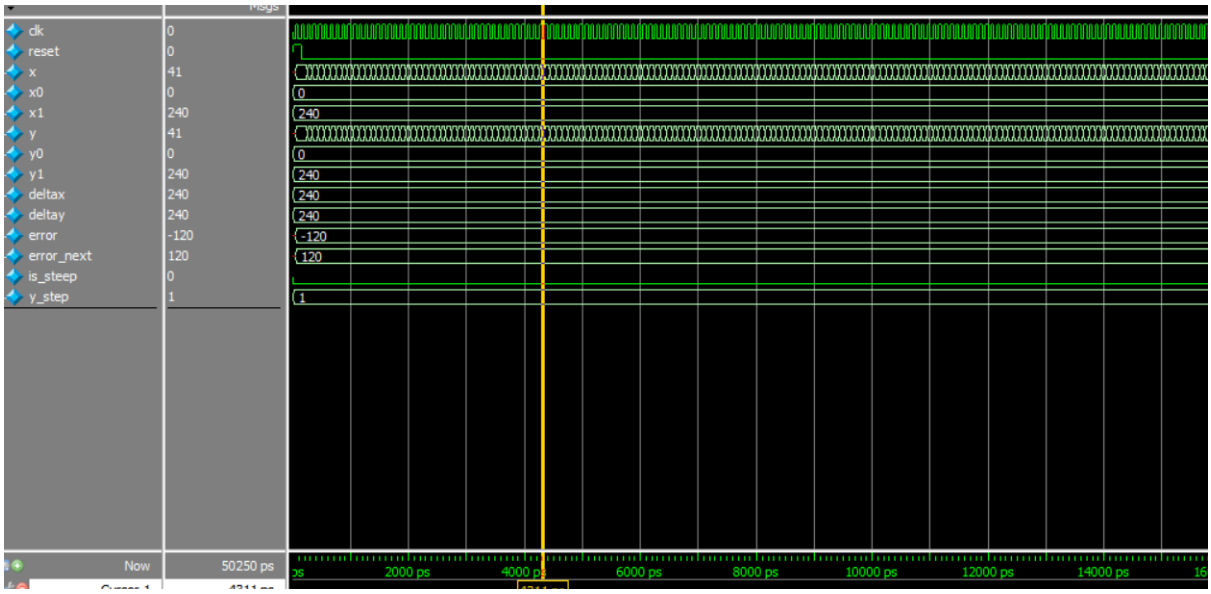

Figure 8: Simulation for VGA drawing a diagonal line

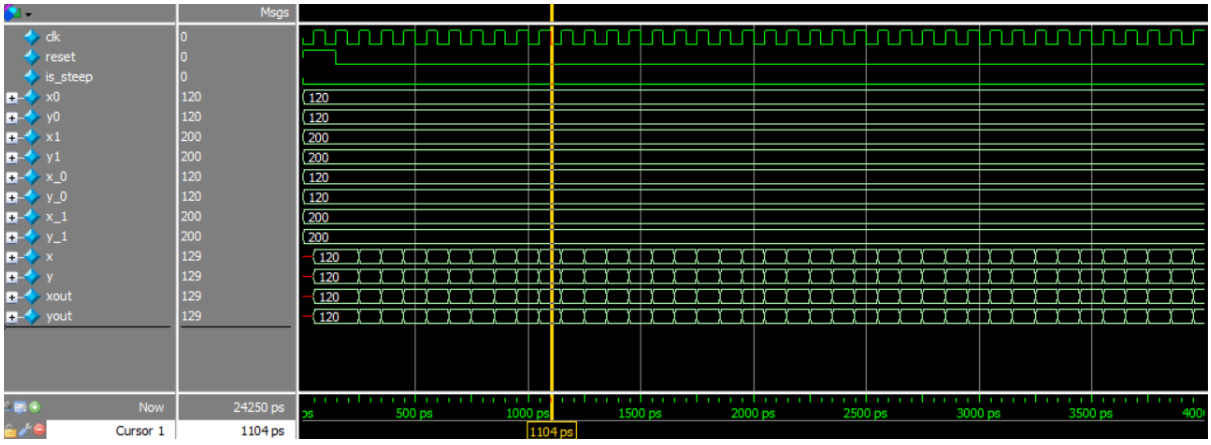

Figure 9: Simulation for VGA drawing a diagonal line with arbitrary endpoints

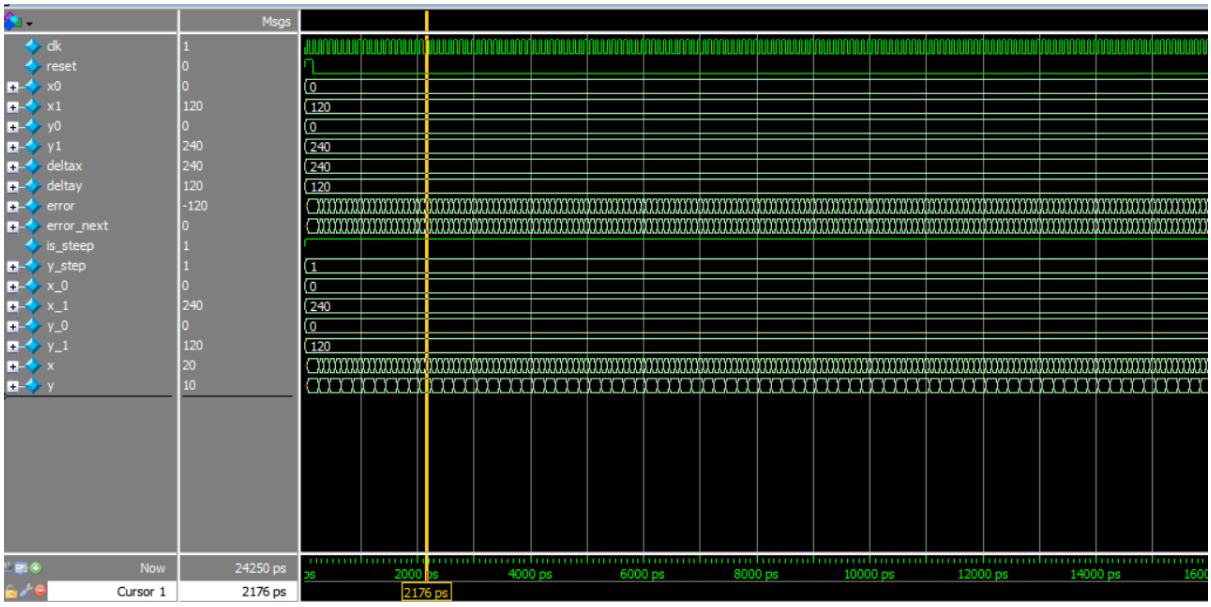

Figure 10: Simulation for VGA drawing a steep line

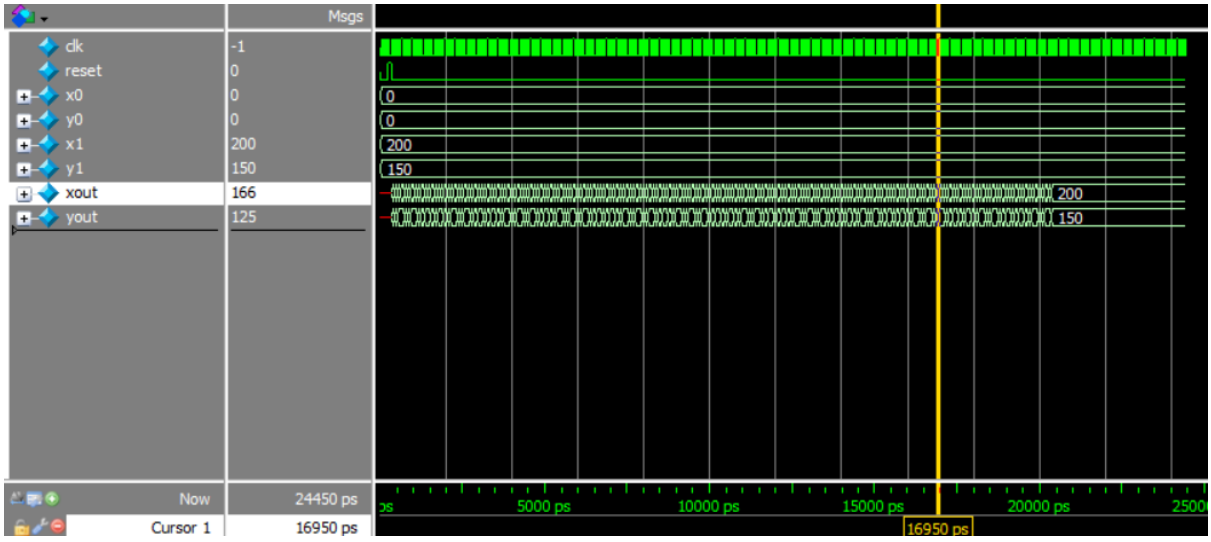

Figure 11: Simulation for VGA drawing a not steep line

Task 3

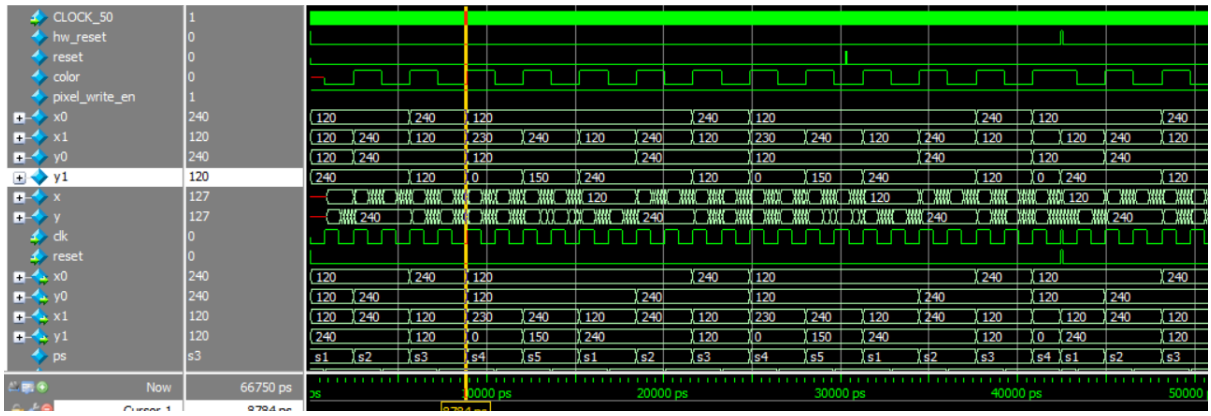

Figure 12: Simulation for the animation

Problem Faced & Feedback

Overall, this lab is very fun since it reveals the difference between software and hardware, as well as clock issues. Also, image display is very fun. The difficulties for me include initializing the D-flip-flops and using keys to change values. For the first problem, I decided to force the user to start the system at the reset state; for the second problem, I decided to use switches instead of keys because switch inputs are more consistent than the keys. Around 20 hours were spent on this lab. My tips would be thinking about the synchronicity and metastability when programming the hardware.