Lab 6

Donovan Clay (ID: 2276005), Cameron Jennings (ID: 2029631)

December 9, 2023

CSE 371

Contents

1 Design Procedure

In this lab, we implemented a system that has the ability to display animations on a VGA terminal using the DE1_SoC's computer video-out port, this system is also known as a VGA controller. Many of the modules needed were provided as starter code, therefore we were tasked with designing and programming the modules responsible for drawing lines and creating a system to output an animation to the VGA terminal.

1.1 Task #1

The first task in this lab was to download the starter code for the project and become familiar with it. We began with an overview of the background information. The VGA terminal is a 640x480 resolution and the configuration of the pixels are determined by the pixel buffer module provided. The pixel buffer controller outputs data in sequential order, starting from the top left corner and proceeding horizontally, then vertically until reaching the bottom right corner. To change the contents of a pixel, simply pass a coordinate to the buffer along with the color of the pixel. The last step in this task was to run the simulation on LabsLand to ensure that the starter code would output as expected.

The screen began in a rainbow gradient and once the system started it became a black screen as expected.

1.2 Task #2

The next step was to draw a line on the VGA terminal using the line_drawer module. The purpose of this module was to receive a start point and an endpoint with x and y coordinates, every clock cycle the system outputs the coordinates of a point along the line connecting the input points so that (x0, y0) slowly increments to (x1, y1). The line between the two points will be drawn on the VGA terminal by the line_drawer module outputting each coordinate to the pixel buffer with an associated color, in this case white.

To implement the line_drawer module, we used the Bresenham algorithm. The system begins in the start state which sets all the values of my_x0, my_y0, my_x1, and my_y1. The values that are hard coded in the module may not be the same as the corresponding inputs. The idea our implementation runs with is that x0 is always incremented by 1 every clock cycle and y0 is only incremented when the error of y is greater than one. For example, a slope of 1/2 would wait two increments of x before y is increased because the error is finally greater than or equal to 1, the error would be (0, .50, 1) which we would then increase y. Technically if the slope of the line is less than one then the (x0, y0) and (x1, y1) values match input. If the slope of the line is greater than one, then the inputs become (y0, x0) and (y1, x1) so that y0 is now in the x place and gets increased every cycle and x0 is now responsible for incrementing when the error is greater than one. Considering the system only works when the slope is less than one, this ensures that it also works for when slope is greater than 1. A variable called is_steep tracks this value swap and updates other parts of the code to ensure the system still works when this happens. The last case for the start is that if x0 > x1, the line goes to the left instead of the right, in this case the x0 becomes x1 and y0 becomes y1, and vice versa, then the system runs as it typically would. These conditionals and swapping of values ensure that our system works for all cases.

Once the system is in the draw state, the coordinates of the line begin to be incremented and output to the pixel buffer. The is_steep variable described before determines if the system should use delta_x or delta_y to track error and updates accordingly. Then the system begins to increase x every clock cycle and y when the error is greater than 1. The swapping in the start state and the variable ensure that the entire system works regardless of the orientation of the system. Once x0 and x1 are equal or x0 is greater, the system switches to the done state. In this state it signals to the top-level controller that the line is done and the top-level then coordinates its next state.

1.3 Task #3

The final step of the lab was to use the line_drawer to make an animation of our choice. To implement our animation, we thought it would be best to provide our line_drawer module with x0, y0, x1, y1 values output in order by a ROM. The following diagram shows the encoding of the data stored in the ROM.

The system begins in a start state which resets the address counter and the line_drawer module. Then it moves to the stall state for a clock cycle before moving to the "drawline" state. The purpose of the stall state is to wait for the ROM to output it's data. In the draw line state, it turns off the reset signal so the line_drawer module will start and stays in the state until the module signals that it is done drawing the line. Then the next state of the system is the finished_line state, which increases the count of the address accessed by the ROM and resets the line_drawer again to prepare for the next line input. The ROM holds data which draws 3 lines and then erases the lines and then draws the 3 lines again with the y coordinates shifted down by one. This makes it appear as if the lines are moving down the screen. The system continues to do this until all the addresses of the ROM have been read and output, then it cyles back to the start state to start at the beginning of the ROM.

The other part of task #3 was making the reset function which "erases" everything from the screen. Since we know the dimensions of the display never changes, we chose to implement this by drawing 480 horizontal lines with the color off. The x-coordinates of the erasing lines will always be the same so we only needed a counter for the y-value of the lines. For the erasing, the state logic for starting the line drawer module and waiting for it to finish is exactly the same. The only difference is we increment a y-value, rather than a read address.

The most important part of the top level module is controlling the line drawing module. It uses the module to both run the animation and "erase" the screen when SW[0] is on. So, the top level module has logic to manage the read address of the ROM, reseting the line drawer, and the y-coordinate for "erasing". It also includes a clock divider to slow down the speed that it goes through the ROM's addresses. However, the line drawer still draws at the full 50MHz speed.

2 Results

The system we implemented in this lab has the ability to output pixel configurations to a VGA terminal as well as draw lines on the terminal to create an animation. The VGA driver code in this lab was provided, so we only implemented and tested two modules, the line_drawer and the top module. However, there was not much testing of the VGA terminal output as completion of the third task could be confirmed with a simple eye-test of the final output. The only module that did require testing was the line_drawer module.

2.1 Line-Drawer Testbench

These test benches show the input values of x0, y0, x1, y1 and process of the line drawer module incrementally moving from (x0, y0) to (x1, y1). In all the testbenches, we can see the line drawer's output starts at either (x0, y0) or (x1, y1) and finishes at the opposing coordinate. These test benches thoroughly exhaust all the different cases of lines the line drawing module can be asked to draw.

2.2 DE1_SoC

As stated in the section above, the top-level did not require any testing as the output could be viewed simply with an eye-test. The output that was given when we ran our system in LabsLand was the following.

The animation outputs in a legible manner the word/name "CAM".

2.3 Flow Summary

DE1_SoC

Fitter Status : Successful - Thu Dec 7 23:12:26 2023
Quartus Prime Version : 17.1.0 Build 590 10/25/2017 SJ Lite Edition
Revision Name : DE1_SoC1
Top-level Entity Name : DE1_SoC1
Family : Cyclone V
Device : 5CSEMA5F31C6
Timing Models : Final
Logic utilization (in ALMs) : 905 / 32,070 (3 %)
Total registers : 637
Total pins : 118 / 457 (26 %)
Total virtual pins : 0
Total block memory bits : 1,228,820 / 4,065,280 (30 %)
Total RAM Blocks : 151 / 397 (38 %)
Total DSP Blocks : 5 / 87 (6 %)
Total HSSI RX PCSs : 0
Total HSSI PMA RX Deserializers : 0
Total HSSI TX PCSs : 0
Total HSSI PMA TX Serializers : 0
Total PLLs : 1 / 6 (17 %)
Total DLLs : 0 / 4 (0 %)

Analysis & Synthesis Status : Successful - Thu Dec 7 23:11:37 2023
Quartus Prime Version : 17.1.0 Build 590 10/25/2017 SJ Lite Edition
Revision Name : DE1_SoC1
Top-level Entity Name : DE1_SoC3
Family : Cyclone V
Logic utilization (in ALMs) : N/A
Total registers : 588
Total pins : 118
Total virtual pins : 0
Total block memory bits : 1,228,820
Total DSP Blocks : 5
Total HSSI RX PCSs : 0
Total HSSI PMA RX Deserializers : 0
Total HSSI TX PCSs : 0
Total HSSI PMA TX Serializers : 0
Total PLLs : 1
Total DLLs : 0

3 Experience Report

This lab was harder, it took a while for us to figure out how to properly set the values for y_step and initializing the values of my_x0, my_y0, my_x1, my_y1. Those were the main bugs because the lab spec outlines code for sequential code, but we have to translate it into something combinational.

This lab took approximately 23 hours, broken down as follows:

Design: 15 minutes

Coding: 12 hours

Debugging: 7 hours

Write up: 3 hours