EE/CSE 371:
Design of Digital Circuits and Systems

Lab3: Display Interface

Lab Objectives

In this lab, we will learn how to display images from an FPGA. We will use the LabsLand DE1-SoC VGA video-out port to display images on a computer monitor.

Background Information

The DE1-SoC FPGA 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. Only the pixel buffer will be used in this exercise, hence we will not discuss the character buffer.

Pixel Buffer

The pixel buffer for the video-out port holds the data (color) for each pixel that is displayed by the VGA controller. The pixel buffer provides an image resolution of 640 × 480 pixels, with the coordinate 0,0 being at the top-left corner of the image.

Figure 1 (*a)* shows that each pixel color is represented as a 16-bit halfword, with five bits for the blue and red components, and six bits for green. As depicted in Figure 1 (b), pixels are addressed in the pixel buffer by using the combination of a *base* address and an (*x, y)* offset.

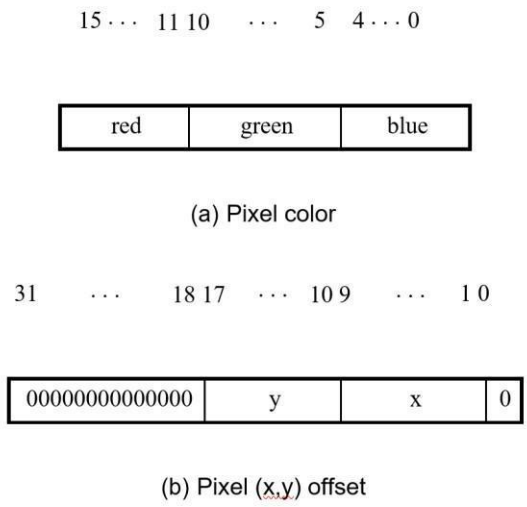

Figure 1: Pixel values and addresses.

You 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 interleaved in the hardware with the read operations that are performed by the pixel buffer controller.

It is also possible to prepare a new image for the VGA display without changing the content of the pixel buffer, by using the concept of *double-buffering*. In this scheme two pixel buffers are involved, called the *front* and *back* buffers. You don't need to worry about double-buffering for this lab.

Drawing

In this lab, you will learn how to implement a simple line-drawing algorithm in hardware.

Drawing a line on a screen requires coloring pixels between two points (x_1, y_1) and (x_2, y_2), 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 $slope = \frac{y_2 - y_1}{x_2 - x_1} = \frac{4}{11}$. Starting at point (1, 1) we move along the *x* axis and compute the *y* coordinate for the line as follows:

$$y = y_1 + slope * (x - x_1)$$

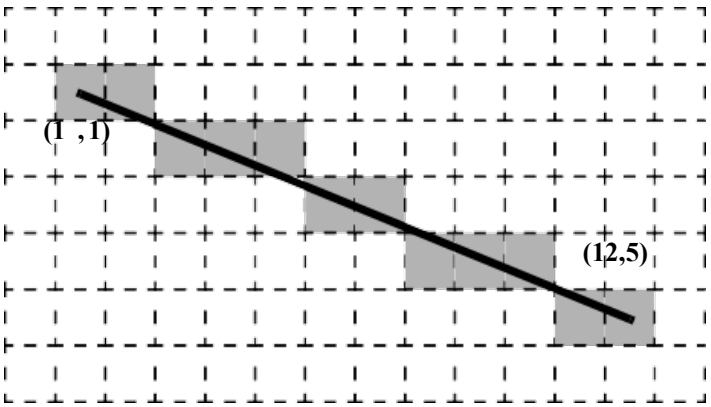

Figure 2: Drawing a line between points (1,1) and (12, 5).

Thus, for column $x = 2$, the *y* location of the pixel is $1 + \frac{4}{11}(2 - 1) = \frac{15}{11}$. 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 $y = 1 + \frac{4}{11}(3 - 1) = \frac{19}{11}$, and round the result to $y = 2$. Similarly, we perform such computations for each column between x_1 and x_2.

The approach of moving along the *x* axis has drawbacks when a line is steep. A steep line spans more rows than it does columns, and hence has a slope with absolute value greater than 1. In this case, our calculations will not produce a smooth-looking line. Also, in the case of a vertical line we cannot use the slope to make a calculation. To address this problem, we can alter the algorithm to move along the *y* axis when a line is steep.

The algorithm described above is called Bresenham's line algorithm. Pseudo-code for an implementation of this algorithm is given in Figure 3. The first 15 lines of the algorithm make the needed adjustments depending on whether or not the line is steep. Then, in lines 17 to 22 the algorithm increments the *x* variable one step at a time and computes the *y* value. The *y* value is incremented when needed to stay as close to the ideal location of the line as possible. Bresenham's algorithm calculates an *error* variable to decide whether or not to increment the *y* value. The version of the algorithm shown in Figure 3 uses only integers to perform all calculations. To understand how this algorithm works, you can read about Bresenham's algorithm in a textbook or by searching for it on the internet.

You may find a simple C version of Bersenham's algorithm on this page:
http://members.chello.at/easyfilter/bresenham.html
This implementation differs slightly from the pseudo-code shown in Figure 3, but the results are the same.

```
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 delta_x = x1 − x0
12   int delta_y = abs(y1 − y0)
13   int error = −(delta_x / 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
19             draw_pixel(y,x)
21        else
22             draw_pixel(x,y)
23        error = error + delta_y
24        if error ≥ 0 then
25             y = y + y_step
26             error = error − delta_x
```

Figure 3: Pseudo-code for a line-drawing algorithm.

Task 1

Your task for lab 3 is to implement Bresenham's line algorithm in SystemVerilog and compile it onto labsLand FPGA to draw lines on a computer monitor. Some files have been uploaded to Canvas to make this easier.

Perform the following:

1. Download the "lab3template.zip" file from Canvas. This is a compressed folder containing a full Quartus project with some files that will help you work with the VGA output of the DE1 board.
2. Observe that the project contains three SystemVerilog files:
   a. VGA_framebuffer.sv, a driver for the VGA port of the board. You don't need to edit or understand this file, but you might notice it uses a very large framebuffer register,

similar to what was described above. The ternary operator on the last line of this file controls the colors of the lines you'll be drawing.

b. line_drawer.sv, a skeleton file for you to add your code to implement Bresenham's algorithm.
c. DE1_SoC.sv, a top-level module which instantiates both of the above modules. This should compile and function without any editing on your part, but you are free to do whatever you want with it.

3. Implement Bresenham's line algorithm.

   Some notes about the line_drawer.sv module:

   a. The module takes inputs *x0, y0, x1, y1* corresponding to the coordinate pairs (x0, y0) and (x1, y1). Your task is to draw the line of pixels that best connects these coordinate pairs.
   b. The module outputs *x* and *y*. These are the pixels being drawn by your algorithm. These coordinates should change by at most one pixel each clock cycle.
   c. As indicated in the file, you'll need to create some local registers to keep track of things. Notice that the example is declared as *signed*. This will be a two's complement binary number, where the MSB is the sign bit. Keep in mind that you'll be doing signed arithmetic during this lab. If you display numbers as *unsigned* in ModelSim, you might end up very confused.

Bresenham's algorithm can get complicated. Ultimately, you'll want to be able to draw a line between any two arbitrary points on the monitor, regardless of whether you're drawing to the left or right, up or down, or whether the line is steep or gradual. Instead of doing this all at once, you'll probably want to work in smaller steps.

The following are suggestions on how to approach this problem, but you can complete this task in whatever way makes the most sense to you.

1. Assume x0 = x1 or y0 = y1 and use the line_drawer.sv file to draw perfectly straight lines
2. Assume that (x0, y0) will be (0,0) and x1 = y1. That is, design an algorithm that only draws perfectly diagonal lines from the origin
3. Modify your algorithm to draw perfectly diagonal lines from any arbitrary starting point
4. Modify your algorithm to handle lines with gradual slopes, such as a line from (0,0) to (100, 20)

Demonstrate that your line algorithm can generate a line between any two coordinates on the monitor. To enable the VGA display, select "VGA" for "User interface" on LabsLand.

Task 2

Modify the DE1_SoC.sv file to implement the following:

1. Use your line algorithm to draw a line on the monitor and animate it to move around the screen.
2. Implement a reset that, when activated, clears the screen by drawing every pixel to be black. You'll probably need to modify the arguments being passed to the VGA_framebuffer module to choose between drawing black or white.

Demonstrate that you are able to animate an object moving around the screen and that your reset feature clears the monitor.

Lab Demonstration and Submission Requirements

- Record a demo video for each of the tasks in this lab. The length of each video is expected to be 2-3 minutes. You will need to demonstrate the soundness of your design by executing the design on the FPGA. You do **not** need to include Modelsim simulations in your demo for this lab. You only need to demonstrate the functionality of your system. Your demo video must be a screen recording created from software like ActivePresenter or Zoom. Please refer to the grading rubric on Canvas..
- Write a Lab Report, as framed by the Lab Report Outline document on Canvas. Comment your code. Follow commenting guidelines as discussed in the Commenting Code document on Canvas. Please include the results of simulation in your lab report. Submit your lab report as a pdf file and all of your SystemVerilog files on Canvas. Please refer to the grading rubric on Canvas.