EE 371, Lab 3

Display Interface

Lab Objectives

In this lab we will learn how to display images. We will use the DE1-SoC Computer's video-out port to display images on a VGA terminal.

Background Information

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. 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. 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 640 ×480, 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.

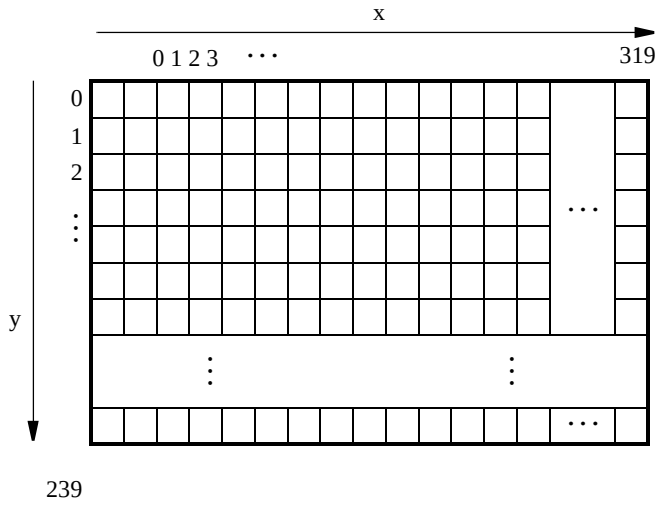


Figure 1: Pixel buffer coordinates.

Figure 2*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 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.

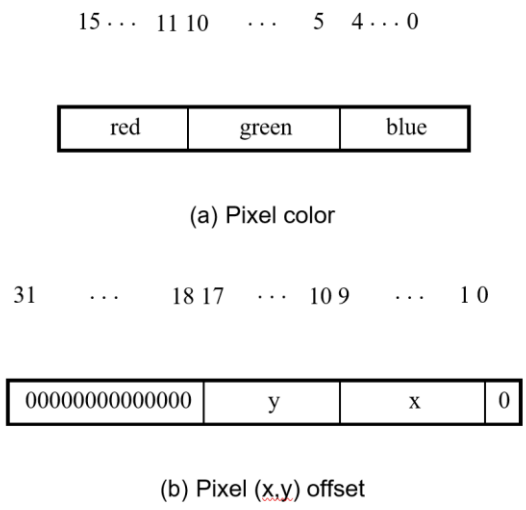


Figure 2: Pixel values and addresses.

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, as described below.

Double Buffering

As mentioned above, a pixel buffer controller reads data out of the pixel buffer so that it can be displayed on the VGA screen. This pixel buffer controller includes a programming interface in the form of a set of registers, as illustrated in Figure 3. The register at address 0xFF203020 is called the *Buffer* register, and the register at address 0xFF203024 is the *Backbuffer* register. Each of these registers stores the starting address of a pixel buffer. The Buffer register holds the address of the pixel buffer that is displayed on the VGA screen. As mentioned above, in the default configuration of the DE1-SoC Computer this Buffer register is set to the address 0xC8000000, which points to the start of the FPGA on-chip memory. The default value of the Backbuffer register is also 0xC8000000, which means that there is only one pixel buffer. But software can modify the address stored in the Backbuffer register, thereby creating a second pixel buffer. An image can be drawn into this second buffer by writing to its pixel addresses. This image is not displayed on the VGA monitor until a pixel buffer *swap* is performed, as explained below.

A pixel buffer swap is caused by writing the value 1 to the Buffer register. This write operation does not directly modify the content of the Buffer register, but instead causes the contents of the Buffer and Backbuffer registers to be swapped. The swap operation does not happen right away; it occurs at the end of a VGA screen-drawing cycle, after the last pixel in the bottom-right corner has been displayed. This time instance is referred to as the *vertical synchronization* time, and occurs every 1/60 seconds. Software can poll the value of the *S* bit in the *Status* register, at address 0xFF20302C, to see when the vertical synchronization has happened. Writing the value 1 into the Buffer register causes *S* to be set to 1. Then, when the swap of the Buffer and Backbuffer registers has been completed *S* is reset back to 0. The *Status* register contains additional bits of information, shown in Figure 3, but these bits are not needed for this exercise. Also, the programming interface includes a *Resolution* register, shown in the figure, that contains the X and Y resolution of the pixel buffer(s).

Address	31 . . . 24	23 . . . 16	15 . . . 8	7 . . . 4	3	2	1	0	
0xFF203020	front buffer address								Buffer register
0xFF203024	back buffer address								Backbuffer register
0xFF203028	Y		X						Resolution register
0xFF20302C	m	n	Unused	B	Unused	A	S		Status register

Figure 3: Pixel buffer controller registers.

In a typical application the pixel buffer controller is used as follows. While the image contained in the pixel buffer that is pointed to by the Buffer register is being displayed, a new image is drawn into the pixel buffer pointed to by the Backbuffer register. When this new image is ready to be displayed, a pixel buffer swap is performed. Then, the pixel buffer that is now pointed to by the Backbuffer register, which was already displayed, is cleared and the next new image is drawn. In this way, the next image to be displayed is always drawn into the "back" pixel buffer, and the "front" and "back" buffer pointers are swapped when the new image is ready to be displayed. Each time a swap is performed software has to synchronize with the VGA controller by waiting until the *S* bit in the Status register becomes 0.

Drawing

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

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 $slope = (y_2 - y_1)/(x_2 - x_1) = 4/11$. Starting at point $(1, 1)$ we move along the x axis and compute the y coordinate for the line as follows:

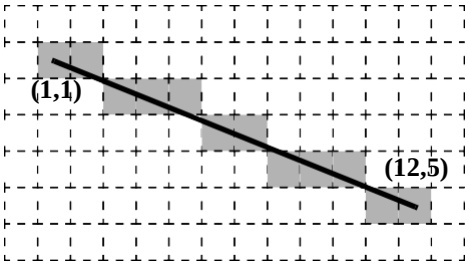


Figure 4: Drawing a line between points $(1, 1)$ and $(12, 5)$.

$$y = y_1 + slope \times (x - x_1)$$

Thus, for column $x = 2$, the y location of the pixel is $1+\frac{4}{11}\times(2-1) = 1\frac{4}{11}$. 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 $y = 1 + \frac{4}{11}\times(3-1) = 1\frac{8}{11}$ 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. With this change, we can implement a line-drawing algorithm known as *Bresenham's algorithm*. Pseudo-code for this algorithm is given in Figure 5. 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 1 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 each *y* value. The version of the algorithm shown in Figure 5 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.

```
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
```

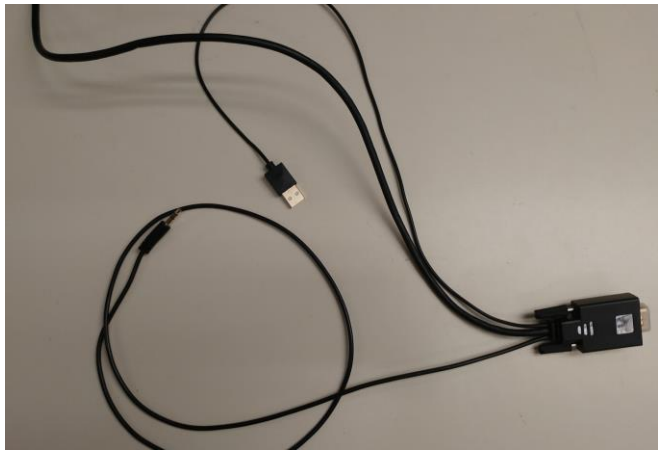
Figure 5: Pseudo-code for a line-drawing algorithm.

Task 1

Your task for lab 3 is to implement Bresenham's line algorithm in SystemVerilog and compile it onto your 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 38,400-byte 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. Ensure the project compiles and produces an output on the VGA monitor.
    a. There are several monitors in both EEB 137 and EEB 361 which have VGA-to-HDMI adaptors installed. They look like this:



The HDMI cables of the adapters are already connected to the monitors. They also have a 3.5mm audio jack, which you won't need, and a USB cable, which powers the adapter. Then, the VGA port connects to your board like this:

b. With the adapter connected and powered, switch the monitor from displaying mDP to HDMI.
c. Compile the project and load it onto your DE1 board. If everything is working correctly, the monitor should be black. If there's a message on it stating "No HDMI/MHL Cable" then something is not connected properly.

Task 2

Implement Bresenham's line algorithm.

Some notes about the line_drawer.sv file:

1. The file takes inputs *x0, y0, x1, y1* corresponding to the coordinate pairs (x0, y0) and (x1, y1)
2. On positive edges of the input clock *clk*, the outputs *x* and *y* are coordinate pairs on the line between (x0, y0) and (x1, y1). On any given clock cycle, *x* and *y* should increment at most one pixel.
3. 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* and is a bit longer than the *x* and *y* inputs/ouputs.

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.

Task 3

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 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

- Submit a lab report that includes the procedures and results obtained in the lab. Suggestions on what to include follow:
  - Abstract: A brief overview of the entire report
  - Introduction: What the lab is (specifications or background info, etc.
  - Procedure/Results/Analysis:
    - Steps to complete the lab
    - Description of each module (explain how they work, point out code, etc.)
    - Simulations of each module with comments on important things to notice about them.
    - Overall description of what the finished product was/how it behaved
    - Discussions of any problems had while completing the lab, or unsolved errors
  - Conclusion: A final summary, reflection on the lab or what was learned, etc.
- Include any hurdles or challenges (if any) that you faced in finishing this lab and how you overcome them.
- Submit the SystemVerilog code for all tasks and include screenshots for the ModelSim waveforms of all modules.
- Submit the Flow Summary (produced during compilation) of compiling your system.
- In your report, include the number of hours (estimated) it took to complete this lab, including reading, planning, design, coding, debugging, testing, etc. Everything related to the lab (in total).
- Submit your report and programs to Canvas. No hard copies. Submit a pdf file as well as a compressed folder of your source files.