Lab 3: Line Drawer

Introduction

In this lab, you will build a hardware implementation of *Bresenham's line drawing algorithm* in Verilog/SystemVerilog for the DE1-SoC board. The final circuit will allow the user to draw many lines on a VGA monitor using the switches to select starting and ending X/Y coordinates and colours.

Algorithm

Bresenham's line algorithm is an approach to place pixels on a monitor screen in such a way as to approximate a line. Drawing a line on a screen requires colouring pixels between the centers of two pixels, (x_1, y_1) and (x_2, y_2), such that they resemble a line as closely as possible. Consider the example in Figure 1.

We want to draw a line between points $(1, 1)$ and $(12, 5)$. The squares represent pixels that can be coloured. To draw a line using pixels, we have to follow the line and for each column colour the pixel closest to the line. To form a line between points $(1, 1)$ and $(12, 5)$ we colour the shaded pixels in the figure. We can use algebra to determine which pixels to colour. This is done using the end points and the slope of the line. The slope of the line is $(y_2 - y_1)/(x_2 - x_1) = \frac{4}{11}$. Starting at point $(1, 1)$ we move along the x-axis and compute the y-coordinate for the line as follows:

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

Thus, for column $x = 2$, the y-location of the pixel is $\frac{4}{11} + 1 = 1\frac{4}{11}$. Because pixel locations are defined by integer values we round the y coordinate *down* to the nearest integer, and determine that in column $x = 2$ we should colour the pixel at $y = 1$. We perform this computation for each column between x_1 and x_2.

The approach of moving along the x-axis has a drawback when a line is steep. A steep line spans more rows than columns, so if the line-drawing algorithm moves along the x-axis to compute the y-coordinate for each column there will be gaps in the line. For example, a vertical line has all points in a single column, so the algorithm would fail to draw it properly. To remedy the problem we can alter the algorithm to move along the y-axis when a line is steep. With this generalization made, the algorithm is shown in Figure 2.

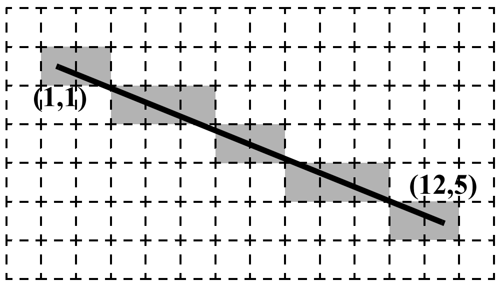

Figure 1: An example of drawing a line between points $(1, 1)$ and $(12, 5)$

```
function Bresenhams_line_algorithm($x_0, y_0, x_1, y_1$)
{
    bool steep = $abs(y_1 - y_0) > abs(x_1 - x_0)$;
    int deltax, deltay, error, ystep, x, y;

    /* Preprocessing inputs */
    if (steep) {
        swap($x_0$, $y_0$)
        swap($x_1$, $y_1$)
    }
    if $(x_0 > x_1)$ {
        swap($x_0$, $x_1$)
        swap($y_0$, $y_1$)
    }

    /* Setup local variables */
    deltax = $x_1 - x_0$;
    deltay = $abs(y_1 - y_0)$;
    error = $-(deltax/2)$; /* This is a right shift by 1 */
    y = $y_0$;
    if $(y_0 < y_1)$
        ystep = 1;
    else
        ystep = -1;

    /* Draw a line */
    for $(x = x_0; x <= x_1; x++)$ {
        if (steep)
            plot(y,x);
        else
            plot(x,y);
        error = error + deltay;
        if $(error > 0)$ {
            y = y + ystep;
            error = error - deltax;
        }
    }
}
```

Figure 2: Bresenham's line algorithm pseudocode

Rather than using floating point mathematics to compute the slope, only integer mathematics are used, with addition, subtraction, and shifting operations only. This is mostly because to use floating point mathematics in a digital circuit requires a lot of work to create a floating point unit. It is also beyond the scope of this lab.

The *error* variable is an example of a fractional quantity that is stored as an integer. It keeps track of the remaining distance, in pixels, to the next integer pixel boundary on the y axis. This quantity is between -1 and 0, and is represented as a fraction with *error* as the numerator, and the denominator implicitly, all the time, assumed to be the constant *deltax*. With that in mind, the algorithm's treatment of the variable may become more clear.

When the algorithm draws a line, it's really drawing a line between the *centers* of the starting and ending pixels, and painting every pixel that the line touches. Therefore, at the initial location, there is $\frac{1}{2}$ a pixel of distance to the next integer pixel boundary on the y axis. By initializing *error* to $-\frac{1}{2 \cdot deltax}$, the fraction it represents becomes exactly $-\frac{1}{2}$. Similarly, when *error* is incremented by *deltay*, it's really being incremented by the slope, $\frac{deltay}{deltax}$. When subtracting *deltax* from *error*, it's really representing the subtraction of $\frac{deltax}{deltax}$, or 1 whole pixel.

Implementation

In the above algorithm the function called `plot(x,y)` draws a pixel at (x, y) on the screen. This particular function will be facilitated by a VGA Adapter provided to you in the starter kit. The VGA Adapter provided has inputs `x`, `y`, `color`, `CLOCK_50` and `plot`. When the input plot toggles high and the positive edge of `CLOCK_50` occurs, the VGA Adapter draws a pixel with a 3-bit colour at location (x, y) on the screen. The valid (x, y) coordinates are between $(0, 0)$ and $(335, 209)$. There are also a number of outputs that are meant to directly drive particular pins on the FPGA. These pins are connected to a digital-to-analog converter chip (the ADV7123) outside the FPGA that controls a monitor through the VGA cable. A shell file with an instantiated VGA Adapter is already provided for you in the starter kit. Outputs of the VGA Adapter are already connected correctly, so you need not worry about them.

The circuit you are to design is supposed to take an (x, y) coordinate and colour from a user and draw a line from point (x_0, y_0) to point (x, y). Once the line is drawn the circuit is to update (x_0, y_0) to be (x, y), and allow the user to input another point (x, y) and colour. As a result a user should be able to draw a set of connected lines by providing the coordinates of subsequent line endpoints. Assume that initially $(x_0, y_0) = (0, 0)$.

To help you complete the lab, a system level diagram is provided in Figure 3.

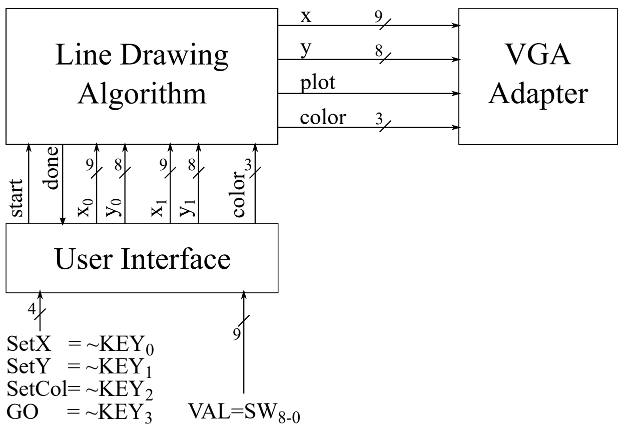

Figure 3: System Diagram

The user uses the first three `KEY` switches to set x, y, and the 3-bit colour, with `SW[8:0]` (using however many bits required) being used to supply the values. `KEY[3]` initiates the process of drawing a line and then updating (x_0, y_0) to the last-entered (x, y).

You are to create the Line Drawing Algorithm (LDA) and User Interface module separately. Your LDA will be re-used in Labs 4 and 6, and it will not be driven by human-controlled buttons or switches. It accepts the endpoints of the line and a colour, and begins drawing when it sees the `start` signal go high. When it finishes drawing the line, it raises the `done` signal to 1 for one clock cycle.

Both the LDA and UI should contain separate *datapath* and *control* sub-modules. As a general design principle, a datapath contains the hardware necessary to perform computations: registers to hold algorithm values, arithmetic and multiplexers to transform or select data, and comparators to test the values of signals. The corresponding control module is a finite state machine that interacts with the datapath and tells it *what* to do *when.* This is accomplished by the FSM controlling the datapath via small control signals, usually 1 up to a few bits in width – for example, the enable signals of datapath registers, or the select signals of datapath muxes. In turn, the datapath sends the 1-bit results of comparators back to the FSM so it can make control decisions. Algorithms, especially for the LDA, will require many clock cycles to complete. It's the job of the state machine to sequence this. Plan carefully which steps of an algorithm happen during which clock cycle.

Simulation

Although it is not required as a deliverable for this lab, you are strongly encouraged to test and debug your LDA and UI modules in ModelSim before testing them on the board. The LDA algorithm is complicated enough such that bugs are inevitable on your early attempts. Here are a few guidelines to help make simulation easier.

First, as has been mentioned since Lab 1, avoid simulating the entire system. The VGA adapter is not easily simulate-able, and even if it was, its output is meaningless to you. Instead, create a testbench to simulate just the LDA, just the UI, or both. That way, you can observe the `x/y/color/plot` signals to see if the right pixels are being drawn.

Included with this lab is a file called `vga_bmp.sv` that defines a module `vga_bmp` that you can instantiate *only in a testbench.* It receives the `x/y/color/plot` that would normally go to a real VGA adapter, and instead, writes them to a 336x210 BMP image in your simulation directory. Simply instantiate it in your testbench and connect it to the output of your LDA module. At the end of your testbench's main `initial` block, call the module's `write_bmp()` task as follows:

```
vga_bmp vga_bmp_inst ( /* connect x/y/color/plot signals */ );

initial begin
  // drive LDA or LDA+UI inputs, etc
  //...

  // Write current contents of simulated VGA to vga.bmp file
  vga_bmp_inst.write_bmp();
  $stop();
end
```

NOTE: vga_bmp is for **simulation only**!. Do not compile with this file to program your FPGA. You will ned to use the actual VGA adapters files for your design to work on the FPGA.

Marking Scheme

- Preparation: Write the complete Verilog/SystemVerilog code for the full system, including the LDA and UI modules (8 marks)

- In lab: Demonstrate your hardware on the DE1-SoC board and your station's monitor (8 marks)

Frequently Asked Questions

In this section, questions from previous years have been collected and answered. Please check here first before posting a question on piazza.

Q1: Should the values of (x_1, y_1) stay constant throughout drawing the line?
A1: You do not need to make any assumptions about the values of (x_1, y_1) while drawing the line. Your FSM should only read the inputs while the *valid* signal is 1. The values of (x_1, y_1) can be ignored at all other times.

Q2: Should we handle the cases where (x_1, y_1) are larger than the sizes of our registers?
A2: No, your design does not need to account for this case.

Q3: When error=0, should the line be horizontal or should it be vertically offset by 1 pixel?
A3: The provided algorithm is undefined for error=0, so either solution would be acceptable.