Bert Zhao
EE 371

Abstract: The purpose of this lab is to introduce the video output of the DE1-SoC. We learn the concepts behind a frame buffer and how data is sent from the FPGA to the monitor via the VGA port. The DE1-SoC outputs video at a resolution of 640x480, however the pixel buffer only stores 320x240 pixels. This is because the VGA controller handles the scaling from 320x240 to 640x480. Throughout this lab, we will be addressing the pixel buffer and therefore will be selecting points only from the 320x240 pixel space.

Introduction: The main task of this lab is to learn how to produce an image on the DE1-SoC and display it on a monitor. The images in this lab will all be lines. To draw our lines in this lab, we will implement Bresenham's line drawing algorithm. This line drawing algorithm is specifically designed to optimize drawing lines on a bitmap image. The fundamental concept behind this algorithm is to work only with integer values because we are picking pixels. This means we need some method to process error to determine the most accurate pixel to correspond with our line. The second part of this lab is using our line drawing algorithm to create a simple animation. We essentially draw many sets of lines to give the illusion that there is a moving image on the screen. We also implement a clear screen function.

Procedure

Task 1: This preliminary part of this lab is to perform the necessary setup for the rest of the lab. We download the code from canvas which includes the skeleton code for the line drawing algorithm and the framebuffer. We also test the hardware by using a VGA cable to send the output from the FPGA to a monitor.

Task 2: In this first major part of the lab, we implement Bresenham's line drawing algorithm. The algorithm takes a beginning and endpoint and determines the most accurate pixels to pick in between the points to simulate a line. The most essential parts of this algorithm is to determine the appropriate directions to draw the line, and to create an error calculation to determine whether to select the pixel in the current column or row or move to the next. The module outputs a single point per clock cycle until the line is completely drawn, at which point it sends out a complete signal and starts over. The complete signal is essential in the next part of the lab when creating an animated image. There is also a reset, which when high, locks the line drawer at the starting point.

Task 3: The second major part of the lab is creating an animated output. This essentially requires drawing many lines in succession to give the illusion of a moving image. We draw one line at a time, making use of the complete signal to indicate when we are ready to pick new endpoints. The new point picker operates on a much slower clock than line drawer and pixel buffer because otherwise the image would complete so quickly that the refresh rate of the monitor cannot keep up the outputs of frames from the buffer. We also implement a screen clear function, which sets every pixel to black and allows us to restart our animation. The clear screen is triggered by KEY2 and the animation restart is triggered by KEY3.

Results and Analysis: The line drawer was tested in ModelSim prior to its implementation for an output to the monitor. The specific cases that need to be tested are when the line needs to be drawn right to left and down to up. The Bresenham line drawing algorithm is designed to handle these cases where as a simpler line drawing algorithm might not. The flow summary is also included below, and the most

important feature of note are the 307,200 total memory block bits used by the frame buffer once the output is scaled to the final resolution of 640x480. We also use SW9 as the reset for point positions and SW8 as the toggle between clear and enable line drawing. Everything operates on the 50MHz clock with the exception of the animation as mentioned previously and clear screen. The clear screen has issues when operating on a clock speed greater than 12.5MHz. The simulation screenshots and the completed animation are included below.

ModelSim Screenshots:

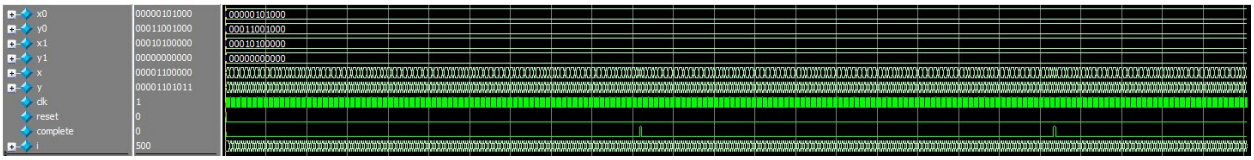

Figure 1: line_drawer ModelSim screenshot

Flow Summary:

Flow Summary

<<Filter>>

Flow Status	Successful - Thu Oct 25 18:14:54 2018
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	DE1_SoC
Top-level Entity Name	DE1_SoC
Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	218 / 32,070 (< 1 %)
Total registers	164
Total pins	96 / 457 (21 %)
Total virtual pins	0
Total block memory bits	307,200 / 4,065,280 (8 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 2: Flow Summary

Completed Animation:

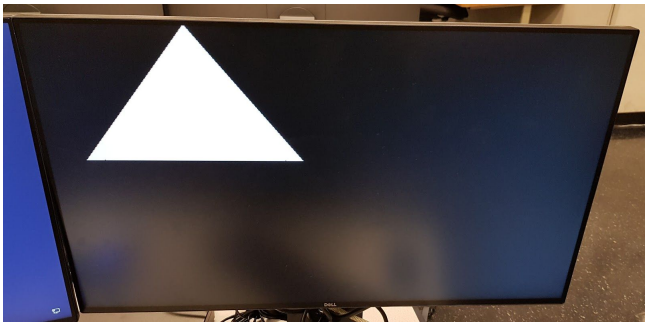

Figure 3: Finished animation

Conclusion: Overall, this lab extends the knowledge of memory and buffers we have previously learned to the frame buffer and video output of the DE1-SoC. We draw lines in this lab to become familiar with the idea of manipulating a video output. We will continue to use the video output in future labs and the final project, so it is important to become familiar with the concepts covered in this lab. From start to finish, this lab took about 10 hours to complete.

Appendix:

```
module DE1_SoC (HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, KEY, LEDR, SW, CLOCK_50,
   VGA_R, VGA_G, VGA_B, VGA_BLANK_N, VGA_CLK, VGA_HS, VGA_SYNC_N, VGA_VS);

   output logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
   output logic [9:0] LEDR;
   input logic [3:0] KEY;
   input logic [9:0] SW;

   input CLOCK_50;
   output [7:0] VGA_R;
   output [7:0] VGA_G;
   output [7:0] VGA_B;
   output VGA_BLANK_N;
   output VGA_CLK;
   output VGA_HS;
   output VGA_SYNC_N;
   output VGA_VS;

   assign HEX0 = '1;
   assign HEX1 = '1;
   assign HEX2 = '1;
   assign HEX3 = '1;
   assign HEX4 = '1;
   assign HEX5 = '1;
   assign LEDR = SW;

   // Generate clk off of CLOCK_50, whichClock picks rate.
   logic [31:0] clk;
   parameter whichClock = 10; // clock for drawing
   parameter whichClock2 = 1; // clock for clear screen
   clock_divider cdiv (CLOCK_50, clk);

   logic reset, clear, compD, compC;
   assign reset = ~KEY[3]; // reset the animation
   assign clear = ~KEY[2]; // clear screen

   logic [10:0] xd0, yd0, xd1, yd1, xDraw, yDraw; // points for drawing
   logic [10:0] xc0, yc0, xc1, yc1, xClear, yClear; // points for clear screen
   logic [10:0] x, y; // points into frame buffer

   VGA_framebuffer fb(.clk50(CLOCK_50), .reset(1'b0), .x, .y,
            .pixel_color(~clear), .pixel_write(1'b1),
            .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS,
            .VGA_BLANK_n(VGA_BLANK_N), .VGA_SYNC_n(VGA_SYNC_N));

   // Line drawer for animation
   line_drawer lines(.clk(CLOCK_50), .reset, .x0(xd0), .y0(yd0), .x1(xd1), .y1(yd1), .x(xDraw), .y(yDraw), .complete(compD));

   // Point picker for animation
   always_ff @(posedge clk[whichClock]) begin
      if (reset) begin
         xd0 <= 160;
         yd0 <= 0;
         xd1 <= 40;
         yd1 <= 200;
      end else if (compD && (xd1 < 280)) begin
         xd1 += 1;
      end else if (compD && (xd1 == 280)) begin // finish at single point (280, 0)
         xd0 <= 280;
         yd0 <= 0;
         xd1 <= 280;
         yd1 <= 0;
      end
   end

   // Line drawer for clear screen
   line_drawer clear_screen(.clk(CLOCK_50), .reset(1'b0), .x0(xc0), .y0(yc0), .x1(xc1), .y1(yc1), .x(xClear), .y(yClear), .complete(compC));
```

```
   // Point picker for clear screen
   always_ff @(posedge clk[whichClock2]) begin
      if (compC) begin
         if (xc1 >= 640) begin
            xc0 <= 0;
            yc0 <= 0;
            xc1 <= 0;
            yc1 <= 480;
         end else begin // sweeping vertical line
            xc0 += 1;
            xc1 += 1;
         end
      end
   end
```

```
    always_comb begin
        if (clear) begin
            x = xClear;
            y = yClear;
        end else begin
            x = xDraw;
            y = yDraw;
        end
    end

    initial begin
        // first line from (160, 0) to (40, 200)
        xd0 = 160;
        yd0 = 0;
        xd1 = 40;
        yd1 = 200;

        // vertical line at left side
        xc0 = 0;
        yc0 = 0;
        xc1 = 0;
        yc1 = 480;
    end
endmodule

// divided_clocks[0] = 25MHz, [1] = 12.5Mhz, ... [23] = 3Hz, [24] = 1.5Hz, [25] = 0.75Hz, ...
module clock_divider (clock, divided_clocks);
    input logic clock;
    output logic [31:0] divided_clocks;

    initial begin
        divided_clocks <= 0;
    end

    always_ff @(posedge clock) begin
        divided_clocks <= divided_clocks + 1;
    end
endmodule
```

Code 1: DE1_Soc.sv

```
module line_drawer(
    input logic clk, reset,
    input logic [10:0]   x0, y0, x1, y1, //the end points of the line
    output logic [10:0]  x, y, //outputs corresponding to the pair (x, y)
    output logic complete
    );

    /*
     * You'll need to create some registers to keep track of things
     * such as error and direction
     * Example: */
    logic signed [11:0] error, dx, dy, errorTemp;
    logic right, down;

    enum {set, plot, done} ps, ns;

    // Next state logic
    always_comb begin
        case (ps)
            set:  ns = plot;
            plot: if ((x == x1) && (y == y1))
                      ns = done;
                  else
                      ns = plot;
            done: ns = set;
        endcase
    end
```

```
    // State behavior
    always_ff @(posedge clk) begin
       case (ps)
          set:  begin
                   dx = x1 - x0; // x direction
                   right = dx >= 0;
                   if (~right) // left
                      dx = -dx;

                   dy = y1 - y0; // y direction
                   down = dy >= 0;
                   if (down) // down
                      dy = -dy;

                   error = dx + dy;
                   x <= x0;
                   y <= y0;
                   complete <= 0;
                end
          plot: begin
                   errorTemp = error << 1;

                   // check for change in x
                   if ((errorTemp > dy)) begin
                      error += dy;
                      if (right) // right
                         x <= x + 1;
                      else // left
                         x <= x - 1;
                   end

                   // check for change in y
                   if ((errorTemp < dx)) begin
                      error += dx;
                      if (down) // down
                         y <= y + 1;
                      else // up
                         y <= y - 1;
                   end
                end
          done: begin
                   complete <= 1;
                   x <= x1;
                   y <= y1;
                end
       endcase
    end

    // DFFs
    always_ff @(posedge clk) begin
       if (reset)
          ps <= set;
       else
          ps <= ns;
    end
endmodule

module line_drawer_tb();
    logic [10:0] x0, y0, x1, y1, x, y;
    logic clk, reset, complete;

    line_drawer dut(clk, reset, x0, y0, x1, y1, x, y, complete);
```

```
            done: begin
                     complete <= 1;
                     x <= x1;
                     y <= y1;
                  end
         endcase
      end

      // DFFs
      always_ff @(posedge clk) begin
         if (reset)
            ps <= set;
         else
            ps <= ns;
      end
endmodule

module line_drawer_tb();
      logic [10:0] x0, y0, x1, y1, x, y;
      logic clk, reset, complete;

      line_drawer dut(clk, reset, x0, y0, x1, y1, x, y, complete);

      // Set up the clock.
      parameter CLOCK_PERIOD = 100;
      initial begin
         clk <= 0;
         forever #(CLOCK_PERIOD/2) clk <= ~clk;
      end

      integer i = 0;
      initial begin
         reset <= 1; x0 <= 40; y0 <= 200; x1 <= 160; y1 <= 0; @(posedge clk);
         reset <= 0; @(posedge clk);
         for(i = 0; i < 500; i++) begin
            @(posedge clk);
         end
         $stop;
      end
endmodule
```

Code 2: line_drawer.sv

```
/*
 * Black-and-white VGA Framebuffer
 *
 * Stephen A. Edwards, Columbia University
 */

module VGA_framebuffer(
 input logic        clk50, reset,
 input logic [10:0] x, y, // Pixel coordinates
 input logic        pixel_color, pixel_write,

 output logic [7:0] VGA_R, VGA_G, VGA_B,
 output logic       VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 *HCOUNT 1599 0             1279       1599 0
 *
 * __________|    Video      |____________|  Video
```

```
* |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
*       _____________________             _____________
* |____|       VGA_HS          |____|
*/

  parameter HACTIVE      = 11'd 1280,
            HFRONT_PORCH = 11'd 32,
            HSYNC        = 11'd 192,
            HBACK_PORCH  = 11'd 96,
            HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC + HBACK_PORCH; //1600

  parameter VACTIVE      = 10'd 480,
            VFRONT_PORCH = 10'd 10,
            VSYNC        = 10'd 2,
            VBACK_PORCH  = 10'd 33,
            VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC + VBACK_PORCH; //525

  logic [10:0]              hcount; // Horizontal counter
  logic                endOfLine;

  always_ff @(posedge clk50 or posedge reset)
    if (reset)          hcount <= 0;
    else if (endOfLine) hcount <= 0;
    else            hcount <= hcount + 11'd 1;

  assign endOfLine = hcount == HTOTAL - 1;

  // Vertical counter
  logic [9:0]               vcount;
  logic                endOfField;

  always_ff @(posedge clk50 or posedge reset)
    if (reset)          vcount <= 0;
    else if (endOfLine)
      if (endOfField)   vcount <= 0;
      else              vcount <= vcount + 10'd 1;

  assign endOfField = vcount == VTOTAL - 1;

  // Horizontal sync: from 0x520 to 0x57F
  // 101 0010 0000 to 101 0111 1111
  assign VGA_HS = !( (hcount[10:7] == 4'b1010) & (hcount[6] | hcount[5]));
  assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

  assign VGA_SYNC_n = 1; // For adding sync to video signals; not used for VGA


  // Horizontal active: 0 to 1279     Vertical active: 0 to 479
  // 101 0000 0000  1280          01 1110 0000  480
  // 110 0011 1111  1599          10 0000 1100  524
  logic                blank;
  assign blank = ( hcount[10] & (hcount[9] | hcount[8]) ) |
       ( vcount[9] | (vcount[8:5] == 4'b1111) );

  // Framebuffer memory: 640 x 480 = 307200 bits

  logic                framebuffer [307199:0];
  logic [18:0]            read_address, write_address;

  assign write_address = x + (y << 9) + (y << 7) ; // x + y * 640
  assign read_address = (hcount >> 1) + (vcount << 9) + (vcount << 7);

  logic                pixel_read;
```

```
    always_ff @(posedge clk50) begin
        if (pixel_write) framebuffer[write_address] <= pixel_color;
        if (hcount[0]) begin
          pixel_read <= framebuffer[read_address];
          VGA_BLANK_n <= ~blank; // Keep blank in sync with pixel data
        end
    end

    assign VGA_CLK = hcount[0]; // 25 MHz clock: pixel latched on rising edge

    assign {VGA_R, VGA_G, VGA_B} = pixel_read ? 24'hFF_FF_FF : 24'h0;

endmodule
```

Code 3: VGA_frambuffer.sv