NATIONAL UNIVERSITY OF SCIENCES & TECHNOLOGY

Digital System Design (EE-421)

Assignment # 2

VGA Controller and Drawing Lines

Submission Details

Submitted to:	Dr. Rehan Ahmed
Submitted by:	Muhammad Umer
Class:	BEE-12C
Semester:	6th
Dated:	23/3/2023
CMS ID:	345834

NUST
School of Electrical Engineering
and Computer Science (SEECS)

1 Table of Contents

2 Tasks

2.1 Task 1: Demo

The VGA Adapter core was created at the University of Toronto for a course similar to ours. The following describes enough for you to use the core; more details can be found on University of Toronto's web page: https://www.eecg.utoronto.ca/~jayar/ece241_07F/vga/.

This task is not worth any marks, but you should do it to ensure that everything else is working (e.g. your VGA Cable is good) before starting the main task below. Make sure you include the Adaptor Core files in your project: vga_adapter.v, vga_controller.v, vga_address_translator.v and vga_pll.v. And remember to set up your pin assignments for your board.

2.1.1 Verilog Code

```
module demo (
    // Standard
    input         CLOCK_50,
    input   [3:0] KEY,
    input   [9:0] SW,
    // VGA
    output        VGA_BLANK_N,
    output  [7:0] VGA_B,
    output        VGA_CLK,
    output  [7:0] VGA_G,
    output        VGA_HS,
    output  [7:0] VGA_R,
    output        VGA_SYNC_N,
    output        VGA_VS
);

    // Task 1
    vga_adapter VGA (
        .resetn(KEY[0]),
        .clock(CLOCK_50),
        .colour(SW[9:7]),
        .x(SW[6:3]),
        .y(SW[2:0]),
        .plot(~(KEY[1])),
        // DAC Signals
        .VGA_R(VGA_R),
        .VGA_G(VGA_G),
        .VGA_B(VGA_B),
        .VGA_HS(VGA_HS),
        .VGA_VS(VGA_VS),
        .VGA_BLANK(VGA_BLANK_N),
        .VGA_SYNC(VGA_SYNC_N),
        .VGA_CLK(VGA_CLK)
    );

    defparam VGA.RESOLUTION = "160x120";
    defparam VGA.MONOCHROME = "FALSE";
    defparam VGA.BITS_PER_COLOUR_CHANNEL = 1;
    defparam VGA.BACKGROUND_IMAGE = "D:/IntelFPGA/projects/assign_2/demo/image.colour.mif";

endmodule
```

2.1.2 RTL Viewer

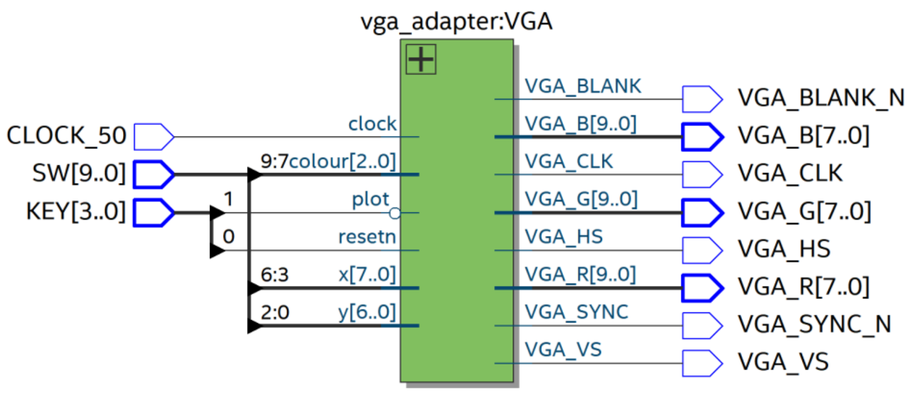


Figure 1: Task 1: RTL Viewer

2.1.3 Hardware Demonstration

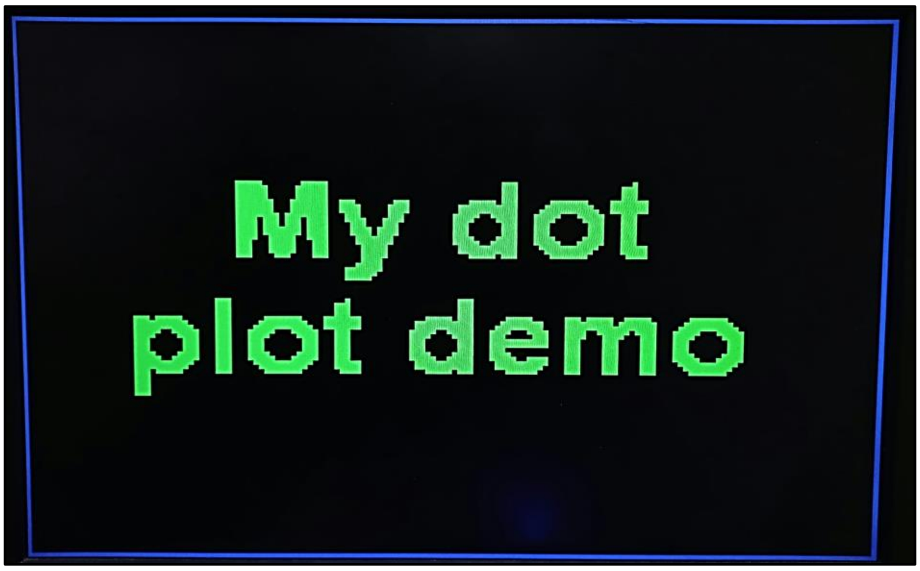


Figure 2: Task 1: Hardware Demo

2.2 Task 2: Fill the Screen

You will create a new component that interfaces with the VGA Adaptor Core. It will contain a simple FSM to fill the screen with colors. This is done by writing to one pixel at a time in the VGA Adapter core. Each row will be set to a different color (repeating every 8 rows). Since you can only set one pixel at a time, you will need an FSM that does something like this:

```
for y = 0 to 119 {
        for x = 0 to 159 {
                set pixel (x, y) to colour ( y mod 8)
        }
}
```

Create an FSM that implements the above algorithm. Your design should have an asynchronous reset which will be driven by KEY (3). You don't need to use KEY (0) or any of the switches in this task. Note that your circuit will be clocked by CLOCK_50.

Test your design on the DE board. You need your DE board with a USB cable, a VGA cable, and a VGA-capable display. Most new LCD displays have multiple inputs, including DVI (digital) and VGA (analog). Note: the VGA connection on your laptop is an OUTPUT, so do not connect your laptop's VGA port to your DE board.

2.2.1 Verilog Code

```verilog
module task_2 (
    // Standard
    input         CLOCK_50,
    input   [3:0] KEY,
    input   [9:0] SW,
    // VGA
    output        VGA_BLANK_N,
    output  [7:0] VGA_B,
    output        VGA_CLK,
    output  [7:0] VGA_G,
    output        VGA_HS,
    output  [7:0] VGA_R,
    output        VGA_SYNC_N,
    output        VGA_VS
);

    wire [2:0] color;
    wire [7:0] x; wire [6:0] y;
    wire reset = KEY[3];
    wire plot = 1'b1;

    fill_screen task_2 (
        .CLOCK_50(CLOCK_50),
        .reset(reset),
        .plot(plot),
        .color(color),
        .x(x),
        .y(y)
    );

    vga_adapter VGA (
        .resetn(1'b1),
        .clock(CLOCK_50),
        .colour(color),
        .x(x),
        .y(y),
        .plot(plot),
        // DAC Signals
        .VGA_R(VGA_R),
        .VGA_G(VGA_G),
        .VGA_B(VGA_B),
        .VGA_HS(VGA_HS),
        .VGA_VS(VGA_VS),
        .VGA_BLANK(VGA_BLANK_N),
        .VGA_SYNC(VGA_SYNC_N),
        .VGA_CLK(VGA_CLK)
    );
    defparam VGA.RESOLUTION = "160x120"; defparam VGA.MONOCHROME = "FALSE";
    defparam VGA.BITS_PER_COLOUR_CHANNEL = 1;
    //  defparam VGA.BACKGROUND_IMAGE = "D:/IntelFPGA/projects/assignment_2/image.colour.mif";

endmodule

module fill_screen (
    input CLOCK_50,
    input reset,
    input plot,
    output reg [2:0] color,
    output [7:0] x,
    output [6:0] y
);

    // defining states; may contain redundant states (subject to change)
    parameter STATE_RESET = 2'b00, STATE_IDLE = 2'b01, STATE_COLOR = 2'b10;
    parameter SCREEN_WIDTH = 160;
    parameter SCREEN_HEIGHT = 120;

    reg [1:0] p_state, n_state;
    wire fill_flag;

    fill_datapath dataflow (.CLOCK_50(CLOCK_50), .reset(reset), .plot(plot),
                            .x(x), .y(y), .fill_flag(fill_flag));

    defparam dataflow.xmax = SCREEN_WIDTH;
    defparam dataflow.ymax = SCREEN_HEIGHT;

    always @(*) begin
        case (p_state)
            STATE_IDLE: begin
                if (plot == 1) begin
                    n_state = STATE_COLOR;
                end else n_state = STATE_IDLE;
            end
            STATE_COLOR: begin
                color    = y % 8;
                if (fill_flag == 1)
                    n_state = STATE_IDLE;
                else
                    n_state = STATE_COLOR;
            end
            STATE_RESET: begin
                color    = 3'b000;
                if (fill_flag == 1)
                    n_state = STATE_IDLE;
                else
                    n_state = STATE_RESET;
            end
            default: n_state = STATE_IDLE;
        endcase
    end

    // state hop register
    always @(posedge CLOCK_50, negedge reset) begin
        if (reset == 0) begin
            p_state <= STATE_RESET;
        end else if (plot == 1) begin
            p_state <= n_state;
        end else p_state = STATE_IDLE;
    end
```

```
endmodule

module fill_datapath (input CLOCK_50,
                  input reset,
                  input plot,
                  output reg fill_flag,
                  output reg [7:0] x,
                  output reg [6:0] y);

    // define constants for screen resolution
    parameter xmax = 160;
    parameter ymax = 120;

    always @(posedge CLOCK_50, negedge reset) begin
        if (reset == 0) begin
            x <= 0;
            y <= 0;
            fill_flag <= 0;
        end else if (plot == 1) begin
            fill_flag <= 0;
            if (x == xmax - 1) begin
                x <= 0;
                if (y == ymax - 1) begin
                    y <= 0;
                    fill_flag <= 1;
                end else y <= y + 1;
            end else x <= x + 1;
        end else begin
            x <= x;
            y <= y;
        end
    end


endmodule
```

2.2.2 RTL Viewer

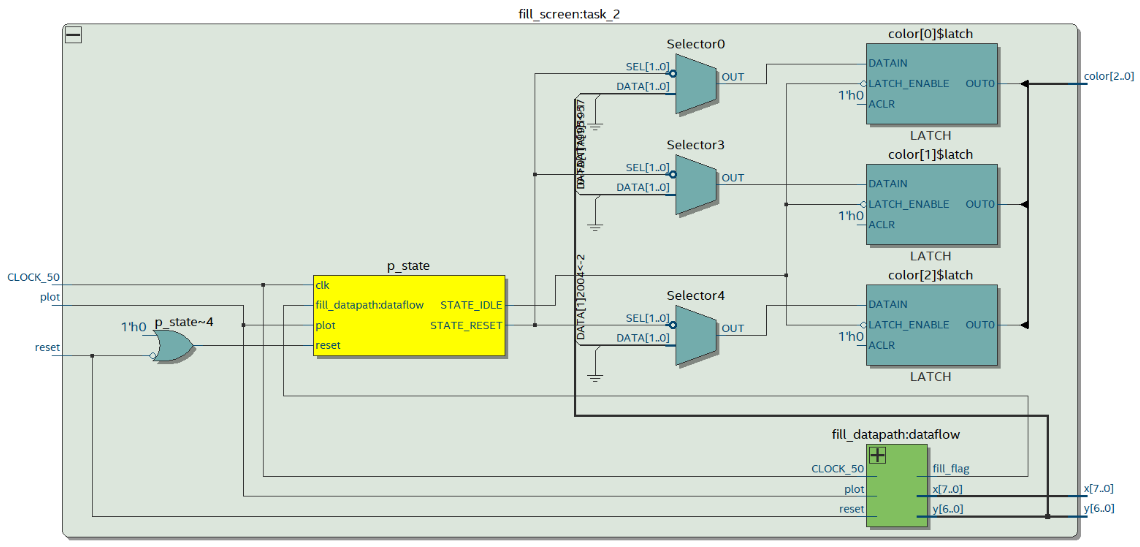
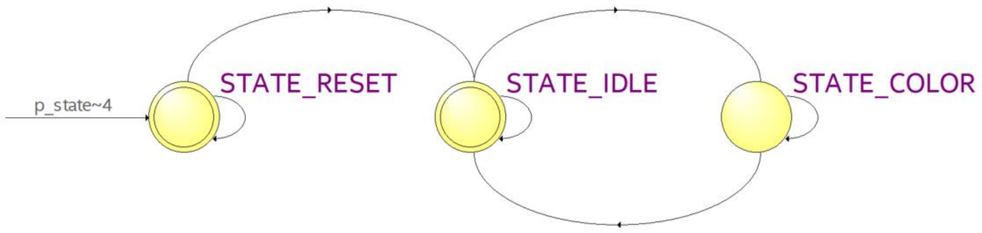
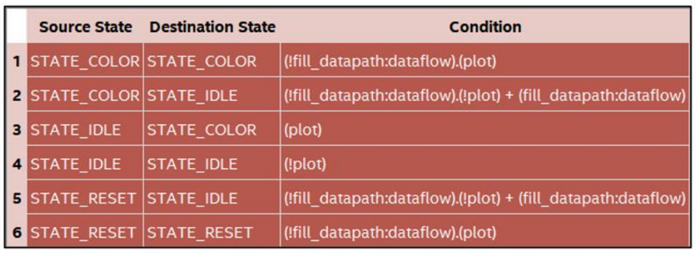


Figure 3: Task 2: RTL Viewer

2.2.3 State Machine Viewer





	Source State	Destination State	Condition
1	STATE_COLOR	STATE_COLOR	(!fill_datapath:dataflow).(plot)
2	STATE_COLOR	STATE_IDLE	(!fill_datapath:dataflow).(!plot) + (fill_datapath:dataflow)
3	STATE_IDLE	STATE_COLOR	(plot)
4	STATE_IDLE	STATE_IDLE	(!plot)
5	STATE_RESET	STATE_IDLE	(!fill_datapath:dataflow).(!plot) + (fill_datapath:dataflow)
6	STATE_RESET	STATE_RESET	(!fill_datapath:dataflow).(plot)

Figure 4: Task 2: FSM

2.2.4 Hardware Demonstration

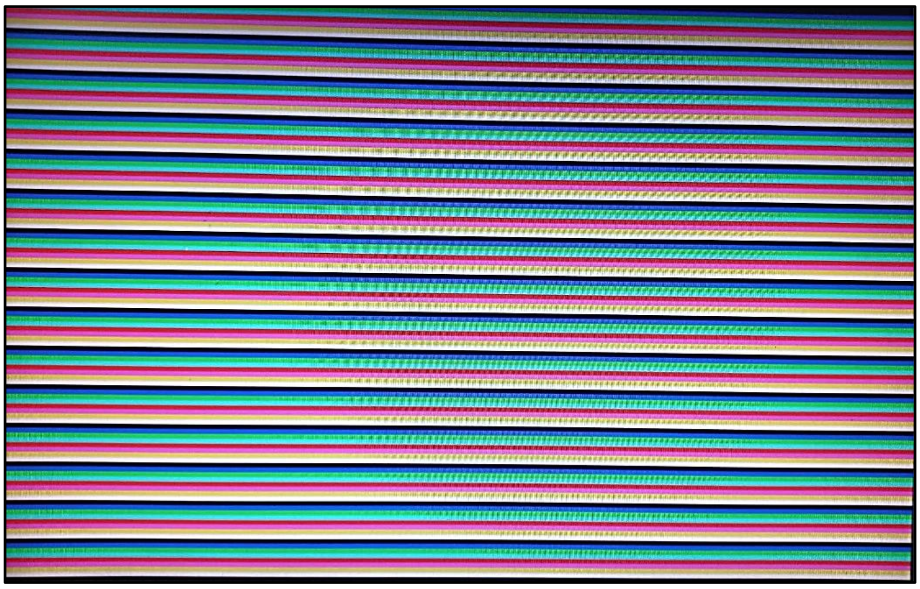


Figure 5: Task 2: Hardware Demonstration

2.3 Task 3: Bresenham Line Algorithm

The Bresenham Line algorithm is a hardware (or software!) friendly algorithm to draw lines between arbitrary points on the screen. The algorithm is efficient: it contains no multiplication or division (multiplication by 2 can be implemented by a shift-register that shifts right). Because of its simplicity and efficiency, the Bresenham Line Algorithm can be found in many software graphics libraries, and in graphics chips. The basic algorithm is as follows (taken from Wikipedia):

In this task, you will implement a circuit that behaves as follows:

1. The switch KEY(3) is an asynchronous reset. When the machine is reset, it will start clearing the screen to black. Hint: Task2 is basically clearing the screen if you set all pixels to black. Clearing the screen will take at least 160*120 cycles.
2. Once the screen is cleared, your circuit will idle. At this point, the user can set switches 17 downto 3, which indicates a point on the screen, and switches 2 downto 0, which indicates a colour. Specifically, SW(17 downto 10) will be used to encode the X coordinate of the point and SW(9 downto 3) will encode the Y coordinate of the pixel. Finally, SW(2 downto 0) will indicate one of 8 possible colours used to draw the line. IMPORTANT: Restrict user entered coordinates to be within (0,0) to (159, 119). If you don't you will see some unexpected behavior (strange patterns being drawn instead of a line). For example, if the user set the switches to indicate a value of 124 for X coordinate, just clip it to 119.
3. When the user presses KEY(0), the circuit will draw a line. Draw from the centre of the screen (location 80,60) to the position specified by the user. Of course, this will take multiple cycles; the number of cycles depends on the length of the line.
4. Once the line is done, the circuit will go back to step 3, allowing the user to choose another point and color. Do not clear the screen between iterations. At any time, the user can press KEY(3), the asynchronous reset, to go back to the start and clear the screen. The reset signal on the VGA Core does not clear the screen. That's why you need to do it manually in step 1. But this also means that you don't have to do anything special to retain previously drawn lines on the screen.

The basic algorithm is as follows (taken from Wikipedia):

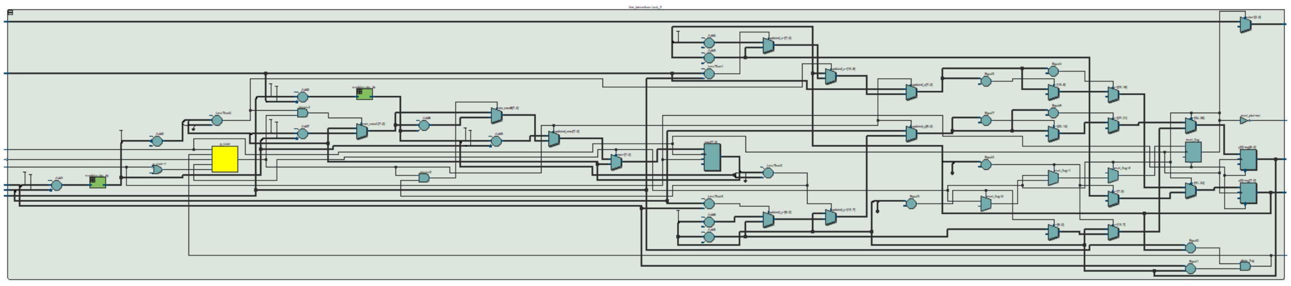
```
function line_bresenham(x0, y0, x1, y1)
        dx := abs(x1-x0)
        dy := abs(y1-y0)
        if x0 < x1 then sx := 1 else sx := -1
        if y0 < y1 then sy := 1 else sy := -1
        error := dx-dy
        x := x0
        y := y0
        loop
                setPixel(x, y)
                if x = x1 and y = y1 exit loop

                e2 := 2*error

                if e2 > -dy then
                        error:= error – dy
                        x := x + sx
                end if

                if e2 < dx then
                        error:= error + dx
                        y := y + sy
                end if
        end loop
end function
```

2.3.1 Code

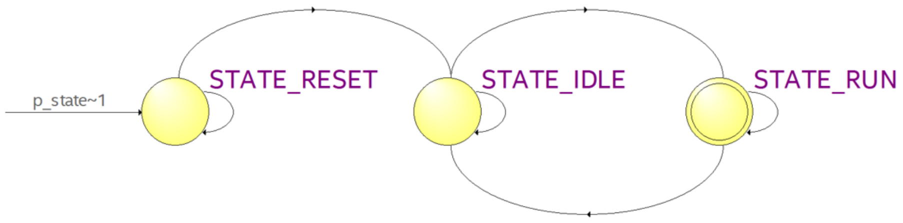
```verilog
module line_bresenham (
    input [2:0] color_in,
    input CLOCK_50,
    input reset,
    input plot,
    input [7:0] x0,
    input [7:0] x1,
    input [6:0] y0,
    input [6:0] y1,
    output reg [7:0] x,
    output reg [6:0] y,
    output reg [2:0] color,
    output reset_plot,
    output done_plot
);

    // defining states; may contain redundant states (subject to change)
    parameter STATE_RESET = 2'b00, STATE_IDLE = 2'b01, STATE_RUN = 2'b10, STATE_DONE = 2'b11;
    parameter SCREEN_WIDTH = 160;
    parameter SCREEN_HEIGHT = 120;

    reg [1:0] p_state, n_state;

    // X & Y Variables
    reg [7:0] updated_x;
    reg [7:0] updated_y;

    // Error Variables
    wire signed [15:0] e2, updated_error;
    reg signed [15:0] error, inter_errorA, inter_errorB;
    wire signed [7:0] dx, dy;
    wire signed [7:0] dx_calc = x1 - x0;
    wire signed [7:0] dy_calc = y1 - y0;

    wire plot_flag, done_flag, idle_flag, loop_flag;
    reg reset_flag;
    assign plot_flag = plot;

    ///////////// FLAG DEFINITIONS /////////////
    assign done_flag = ((x == x1) && (y == y1));
    assign loop_flag = (p_state == STATE_RUN);

    ///////////// FSM /////////////
    always @(*) begin
        case (p_state)
            STATE_RESET: if (reset_flag) n_state = STATE_IDLE;
            else n_state = STATE_RESET;

            STATE_IDLE: if (plot_flag) n_state = STATE_RUN;
            else n_state = STATE_IDLE;

            STATE_RUN: if (done_flag) n_state = STATE_IDLE;
            else n_state = STATE_RUN;

            default: n_state = STATE_IDLE;
        endcase
    end

    ///////////// STATE HOP /////////////
    always @(posedge CLOCK_50, negedge reset) begin
        if (reset == 0) begin
```

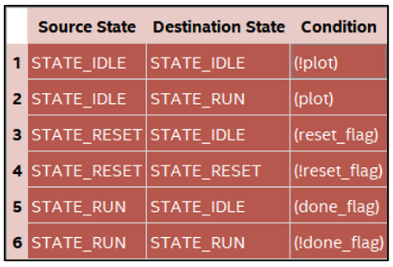
```
            p_state <= STATE_RESET;
        end
        else p_state <= n_state;
    end

    ///////////// DATA PATH /////////////
    assign dx = (dx_calc[7]) ? (-dx_calc) : (dx_calc);
    assign dy = (dy_calc[7]) ? (-dy_calc) : (dy_calc);
    assign e2 = error << 1;

    // X & Y Update
    always @(*) begin
        if (loop_flag)
            if (e2 > -dy)
                if (x0 < x1)
                    updated_x = x + 1'b1;
                else
                    updated_x = x - 1'b1;
            else
                updated_x = x;
        else
            updated_x = x0;
    end

    always @(*) begin
        if (loop_flag)
            if (e2 < dx)
                if (y0 < y1)
                    updated_y = y + 1'b1;
                else
                    updated_y = y - 1'b1;
            else
                updated_y = y;
        else
            updated_y = y0;
    end

    // Error Update
    always @(*) begin
        if (e2 > -dy && loop_flag) inter_errorA = error - dy;
        else inter_errorA = error;
    end

    always @(*) begin
        if (e2 < dx && loop_flag) inter_errorB = inter_errorA + dx;
        else inter_errorB = inter_errorA;
    end

    assign updated_error = (loop_flag) ? inter_errorB : (dx - dy);

    always @(posedge CLOCK_50, negedge reset) begin
        if (reset == 0) begin
            x <= 0;
            y <= 0;
            reset_flag <= 0;
        end
        else if (reset_flag == 0) begin
            if (x == SCREEN_WIDTH - 1) begin
                x <= 0;
                if (y == SCREEN_HEIGHT - 1) begin
                    y <= 0;
                    reset_flag <= 1;
                end else y <= y + 1'b1;
            end else x <= x + 1'b1;
        end
        else begin
            if (updated_x == 8'd160) x <= 8'd159;
            else if (updated_x == 8'd255) x <= 8'd0;
            else x <= updated_x;
            if (updated_y == 7'd120) y <= 7'd119;
            else if (updated_y == 7'd127) y <= 7'd0;
            else y <= updated_y;
            error <= updated_error;
        end
    end

    //////////// COLOR CHOICE ////////////
    always @(*) begin
        if (reset_flag == 0)
            color = 3'b000;
        else
            color = color_in;
    end

    assign reset_plot = ~reset_flag;
    assign done_plot = done_flag;

endmodule
```

2.3.2 RTL Viewer



2.3.3 State Machine Viewer





	Source State	Destination State	Condition
1	STATE_IDLE	STATE_IDLE	(!plot)
2	STATE_IDLE	STATE_RUN	(plot)
3	STATE_RESET	STATE_IDLE	(reset_flag)
4	STATE_RESET	STATE_RESET	(!reset_flag)
5	STATE_RUN	STATE_IDLE	(done_flag)
6	STATE_RUN	STATE_RUN	(!done_flag)

Figure 6: Task 3: FSM

2.3.4 Hardware Demonstration

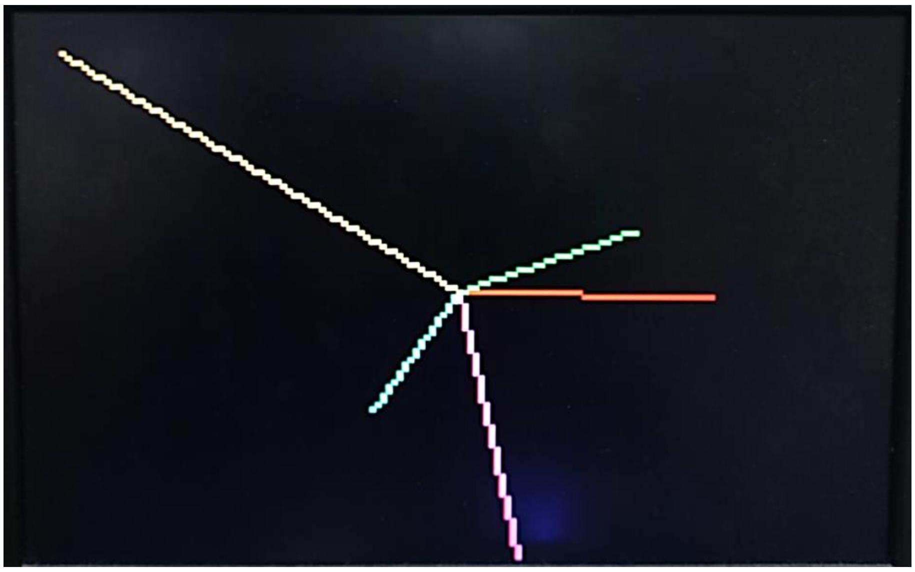


Figure 7: Task 3: Hardware Demonstration

2.4 Challenge Task

Challenge tasks are tasks that you should only perform if you have extra time, are keen, and want to show off a little bit. This challenge task is only worth 10 marks. If you don't demo the challenge task, the maximum score you can get on this assignment is 90/100 (which is still an A+).

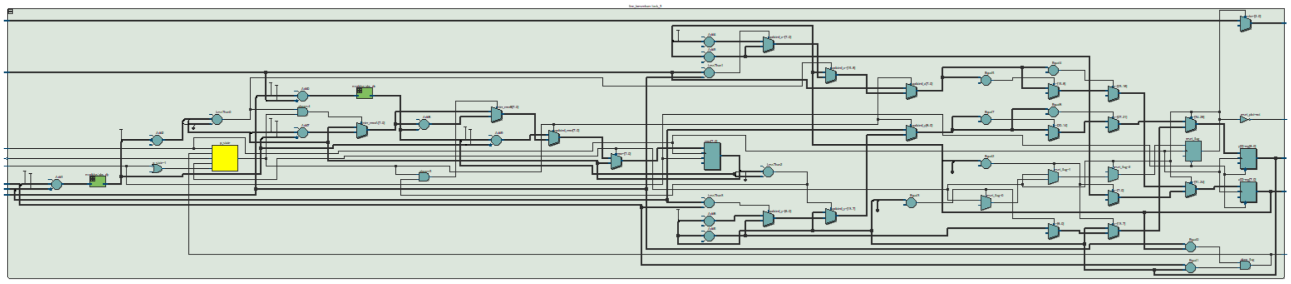
This challenge task is actually fairly easy: In the original circuit, you always connect the center of the screen (80,60) to the point specified by the user. Modify your circuit such that, instead of starting from the center, it starts from the point specified by the user in the previous iteration. So, for the first iteration, if the user specifies point (x0, y0), a line is drawn from the middle of the screen to (x0, y0). Then, in the second iteration, if the user specifies point (x1, y1), a line is drawn from (x0, y0) to (x1, y1). In iteration i, a line is drawn from point (x i-1, y i-1) to (x i, y i).

2.4.1 Code

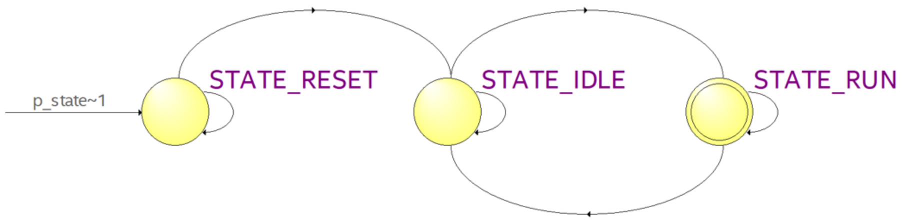
All of the top level and Bradenham code is same as Task 3, however, the following backpropagation block has been implemented based on when the circuit is either reset, or when the logic $(x == x_1)$ & $(y == y_1)$ is true, coined the term done_flag/done_plot.

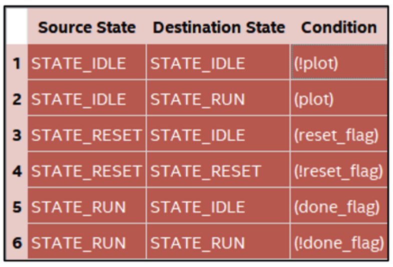
```
assign done_plot = done_flag;
always @(posedge done_plot, negedge reset) begin
    if (reset == 0) begin
        x0 <= 8'd80;
        y0 <= 7'd60;
    end else begin
        x0 <= x1;
        y0 <= y1;
    end
end
```

2.4.2 RTL Viewer



2.4.3 State Machine Viewer





	Source State	Destination State	Condition
1	STATE_IDLE	STATE_IDLE	(!plot)
2	STATE_IDLE	STATE_RUN	(plot)
3	STATE_RESET	STATE_IDLE	(reset_flag)
4	STATE_RESET	STATE_RESET	(!reset_flag)
5	STATE_RUN	STATE_IDLE	(done_flag)
6	STATE_RUN	STATE_RUN	(!done_flag)

Figure 8: Challenge Task: FSM

2.4.4 Hardware Demonstration

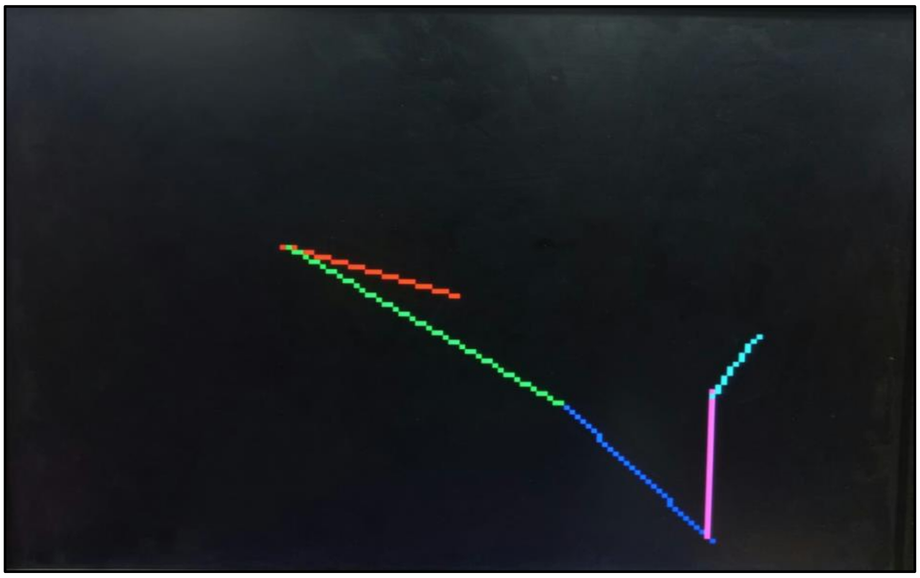


Figure 9: Challenge Task: Hardware Demonstration