

**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 |

**Table of Contents**

[2 Tasks 2](#_Toc130339934)

[2.1 Task 1: Demo 2](#_Toc130339935)

[2.1.1 Verilog Code 2](#_Toc130339936)

[2.1.2 RTL Viewer 3](#_Toc130339937)

[2.1.3 Hardware Demonstration 3](#_Toc130339938)

[2.2 Task 2: Fill the Screen 3](#_Toc130339939)

[2.2.1 Verilog Code 4](#_Toc130339940)

[2.2.2 RTL Viewer 6](#_Toc130339941)

[2.2.3 State Machine Viewer 7](#_Toc130339942)

[2.2.4 Hardware Demonstration 7](#_Toc130339943)

[2.3 Task 3: Bresenham Line Algorithm 7](#_Toc130339944)

[2.3.1 Code 9](#_Toc130339945)

[2.3.2 RTL Viewer 11](#_Toc130339946)

[2.3.3 State Machine Viewer 11](#_Toc130339947)

[2.3.4 Hardware Demonstration 12](#_Toc130339948)

[2.4 Challenge Task 12](#_Toc130339949)

[2.4.1 Code 12](#_Toc130339950)

[2.4.2 RTL Viewer 13](#_Toc130339951)

[2.4.3 State Machine Viewer 13](#_Toc130339952)

[2.4.4 Hardware Demonstration 13](#_Toc130339953)

# Tasks

## 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.

### 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

### RTL Viewer

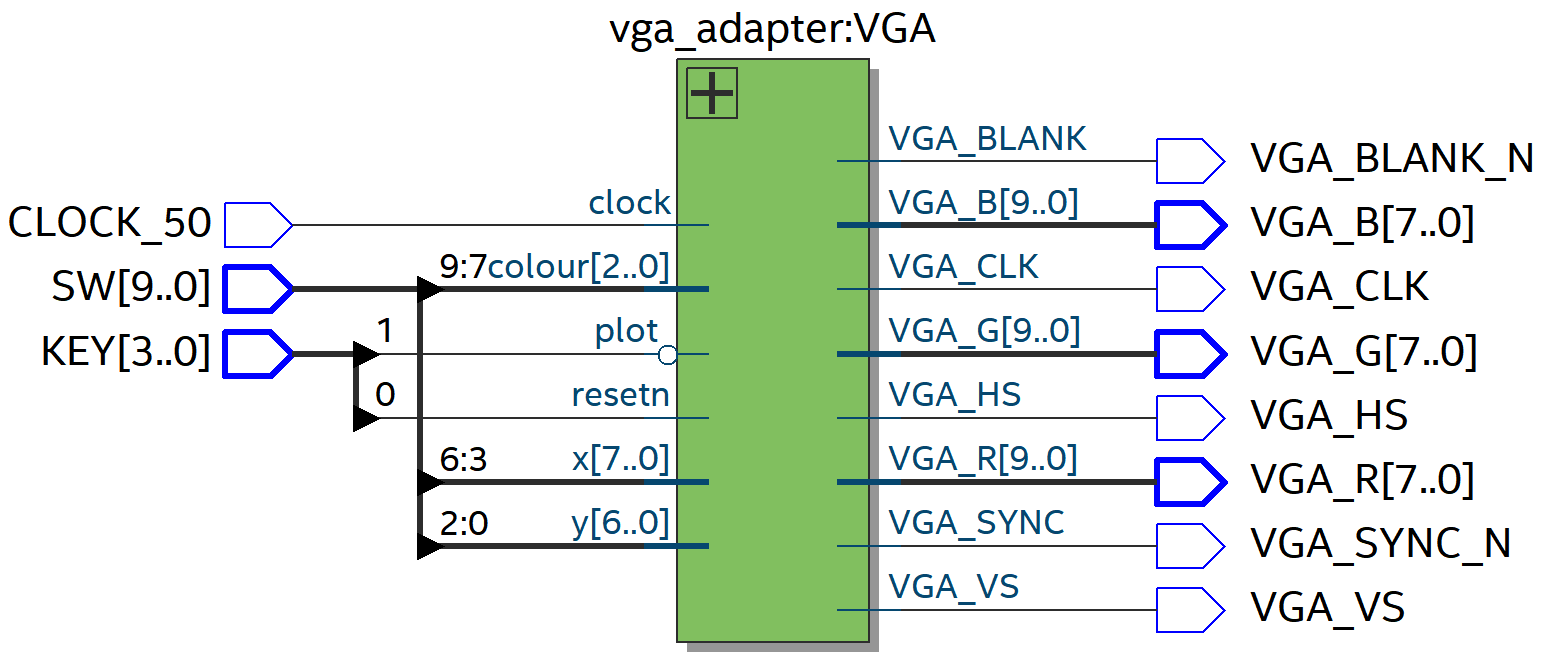


Figure : Task 1: RTL Viewer

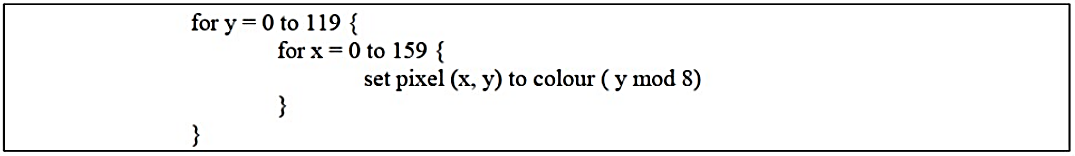
### Hardware Demonstration



Figure : Task 1: Hardware Demo

## 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:



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.

### Verilog Code

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

### RTL Viewer

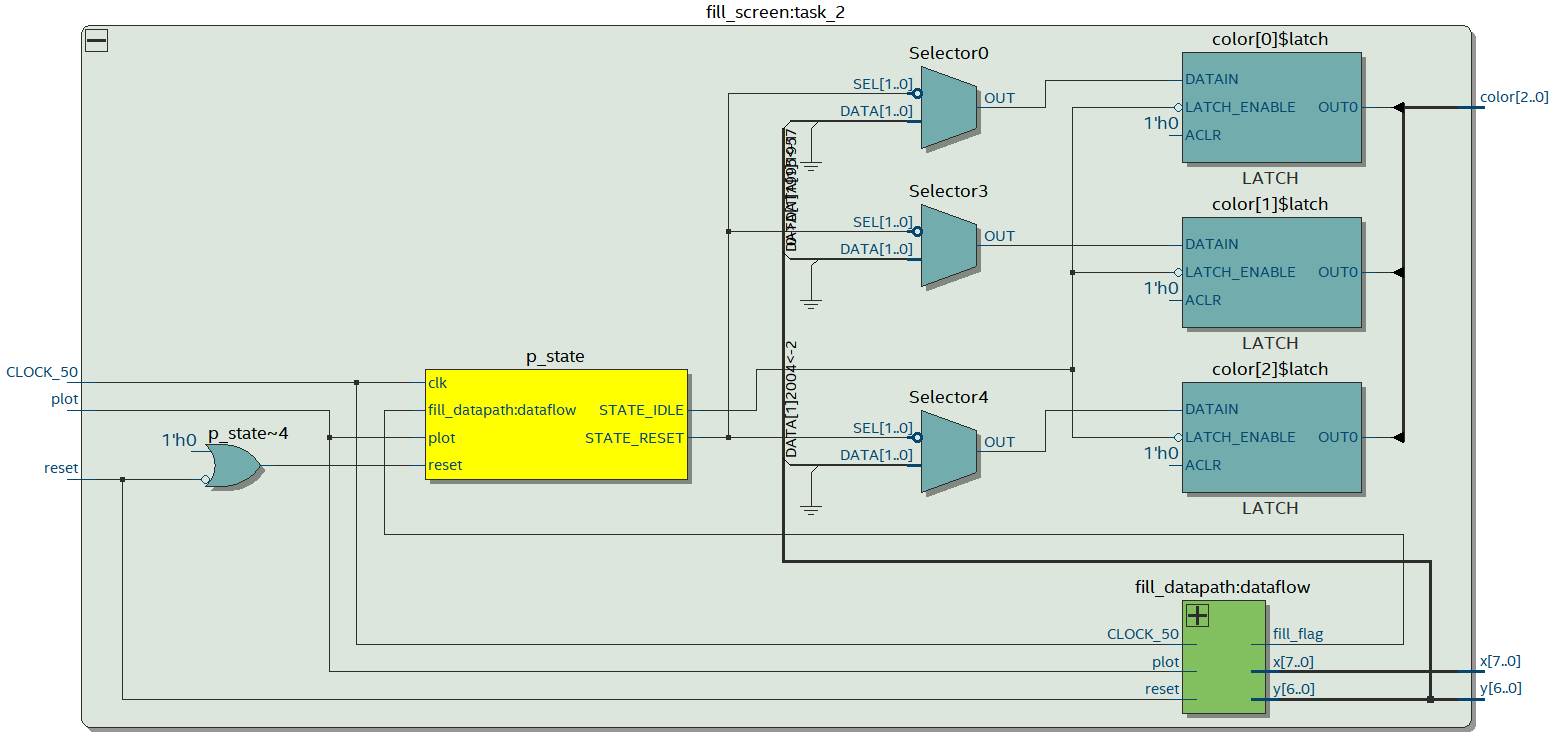


Figure : Task 2: RTL Viewer

### State Machine Viewer

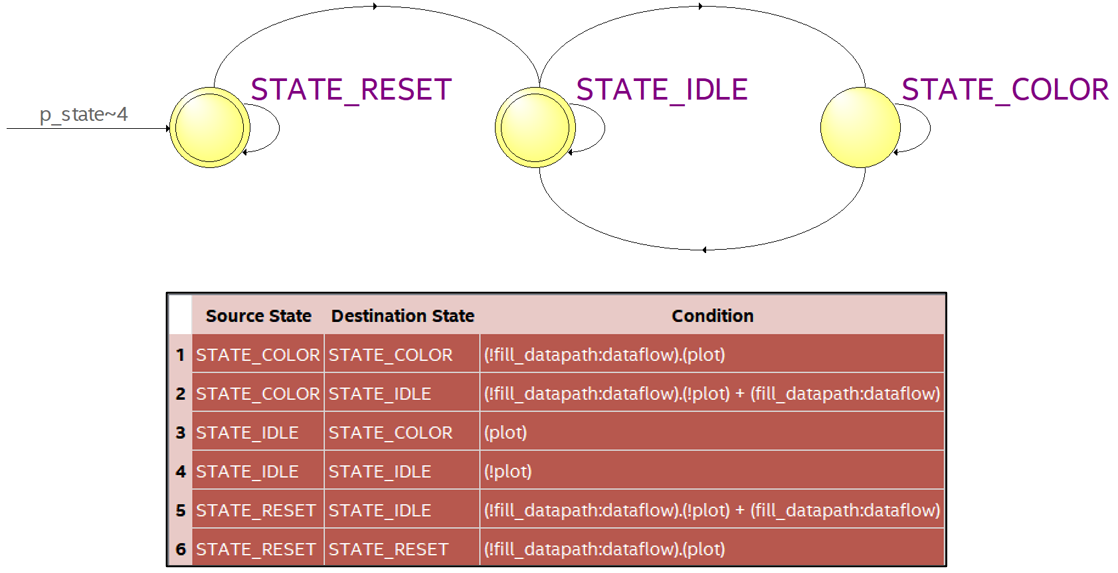


Figure : Task 2: FSM

### Hardware Demonstration

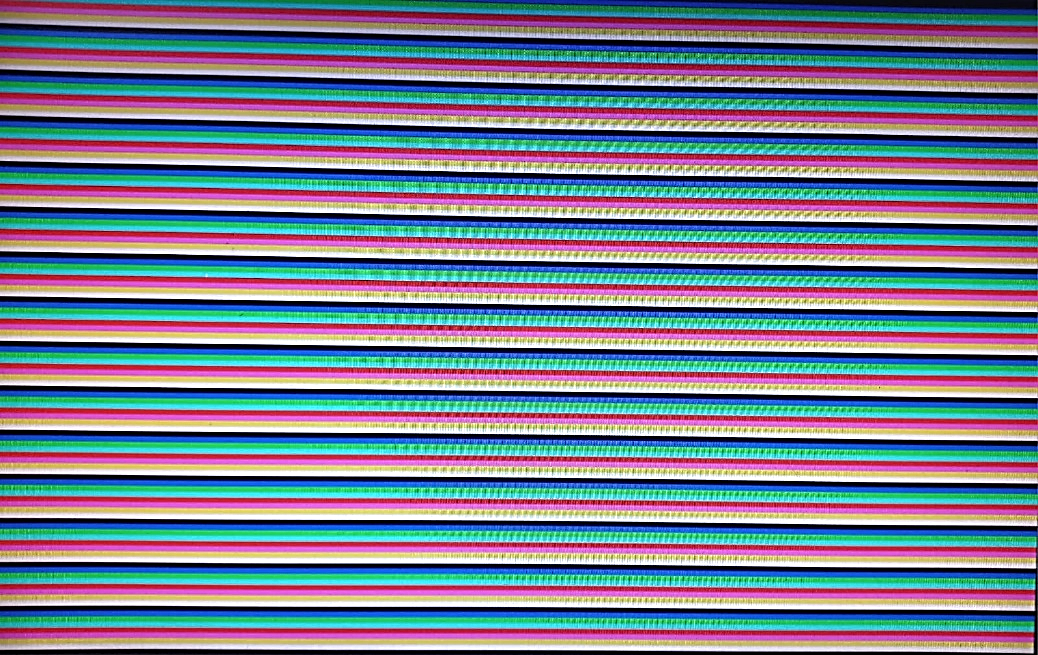


Figure : Task 2: Hardware Demonstration

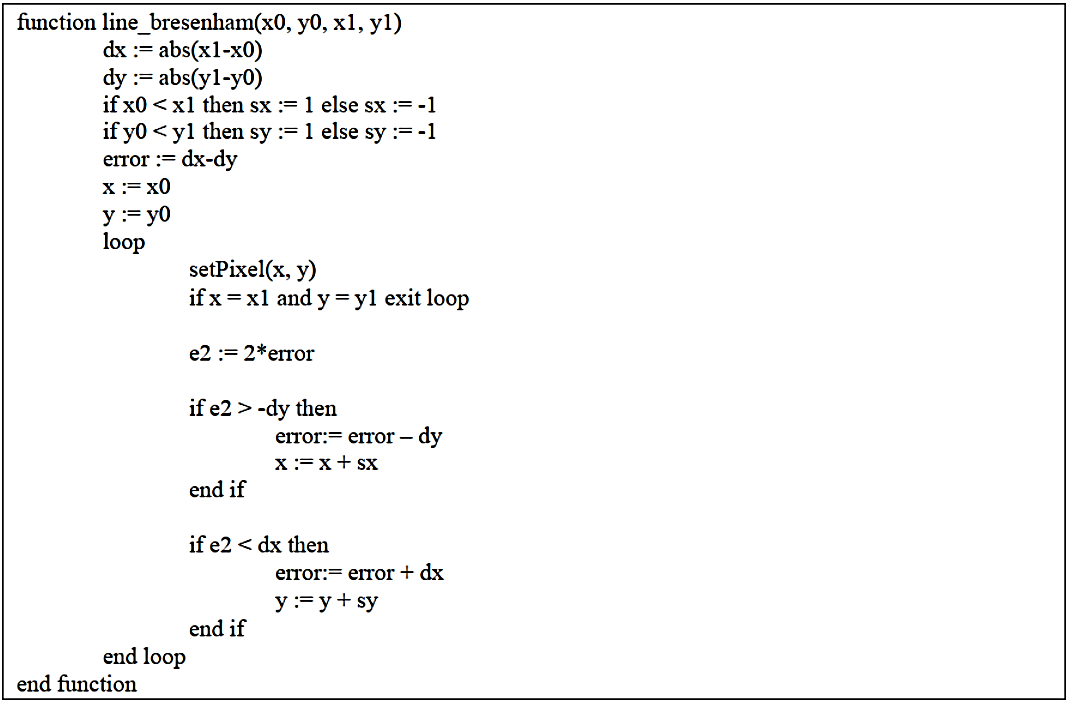
## Task 3: Bresenham Line Algorithm

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):



### Code

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;

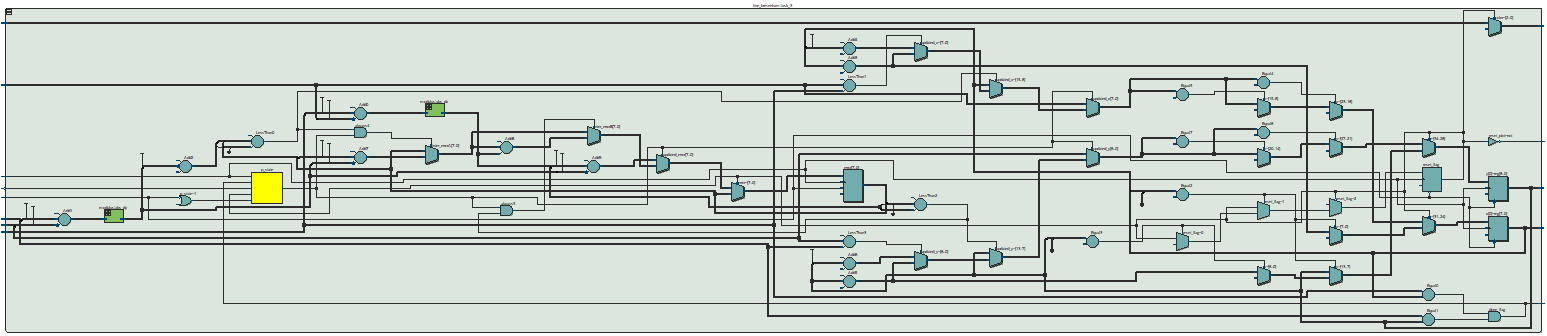
    end

    assign reset\_plot = ~reset\_flag;

    assign done\_plot = done\_flag;

endmodule

### RTL Viewer



### State Machine Viewer

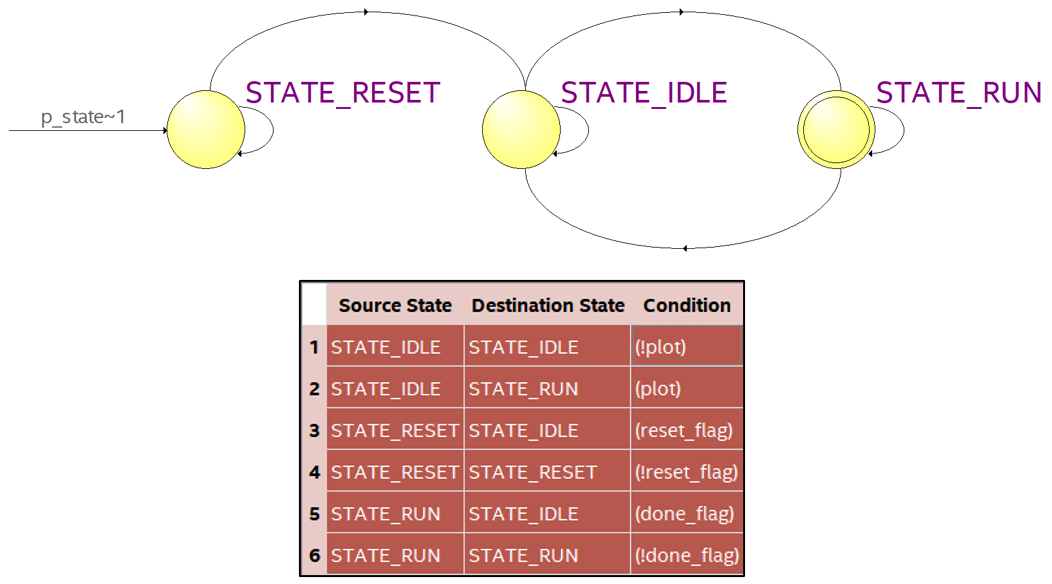


Figure : Task 3: FSM

### Hardware Demonstration

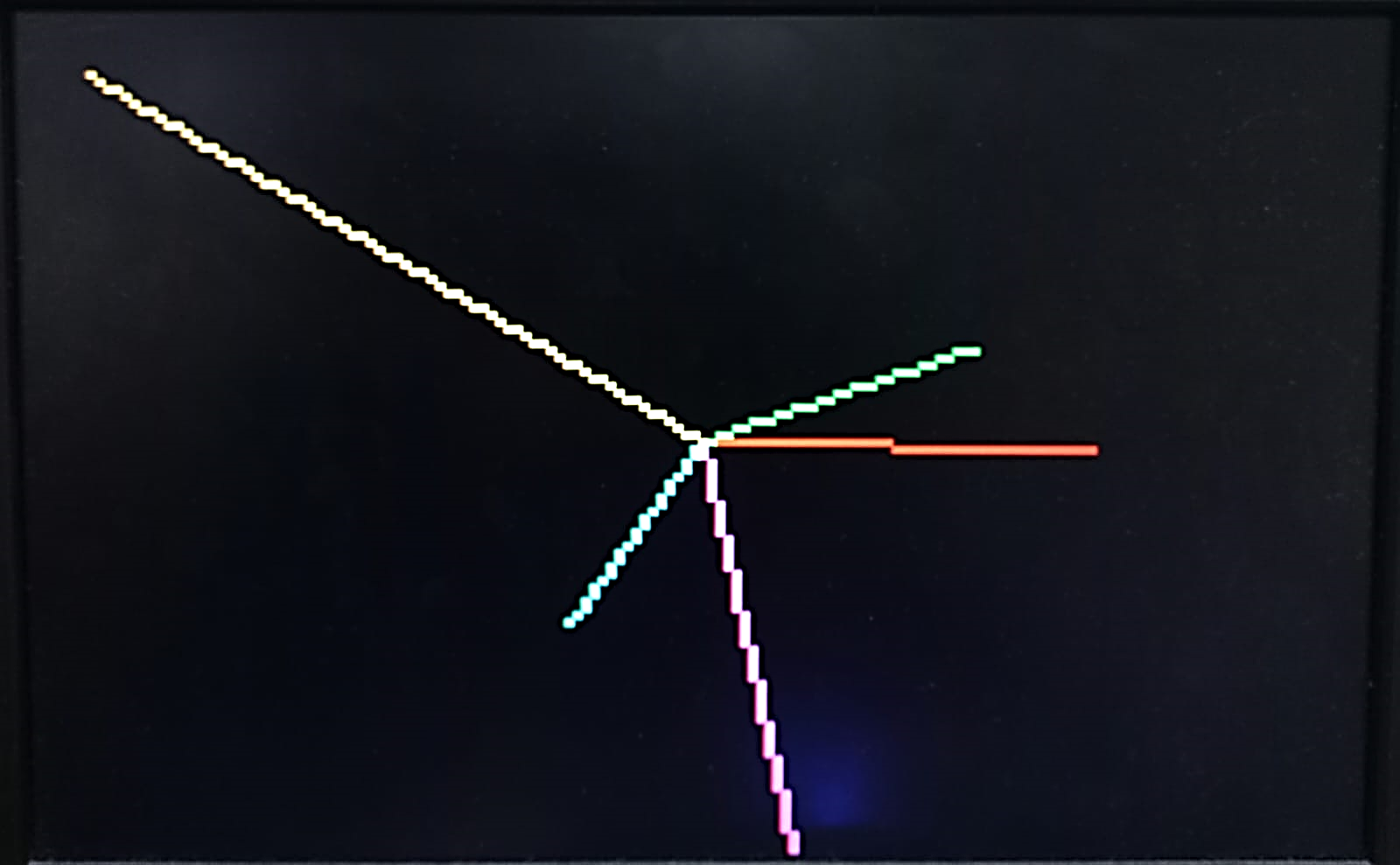


Figure : Task 3: Hardware Demonstration

## 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).

### 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 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

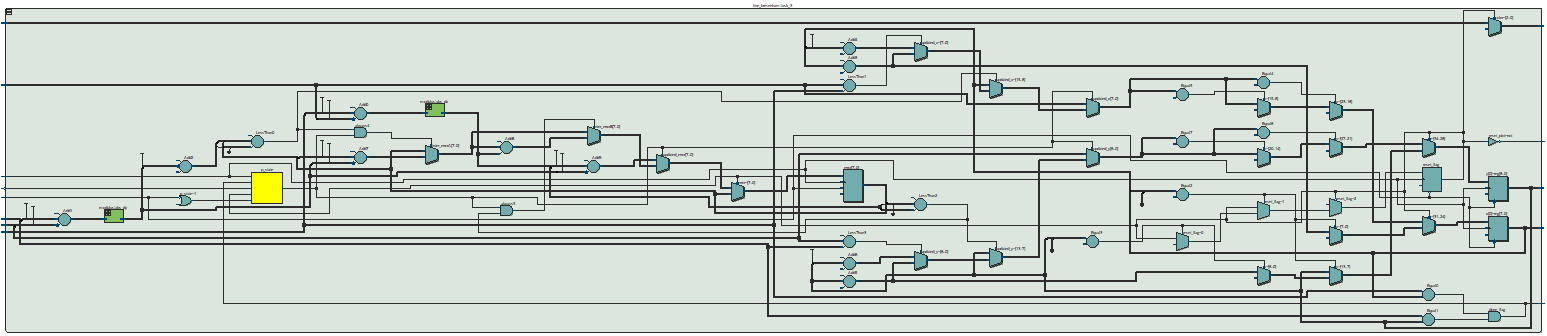
        x0 <= x1;

        y0 <= y1;

    end

end

### RTL Viewer



### State Machine Viewer

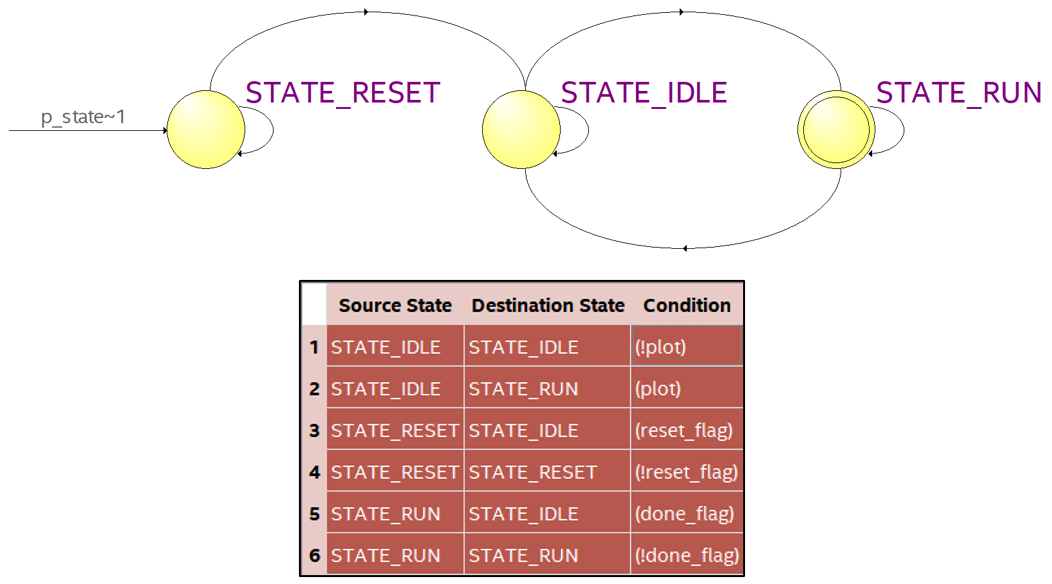


Figure : Challenge Task: FSM

### Hardware Demonstration

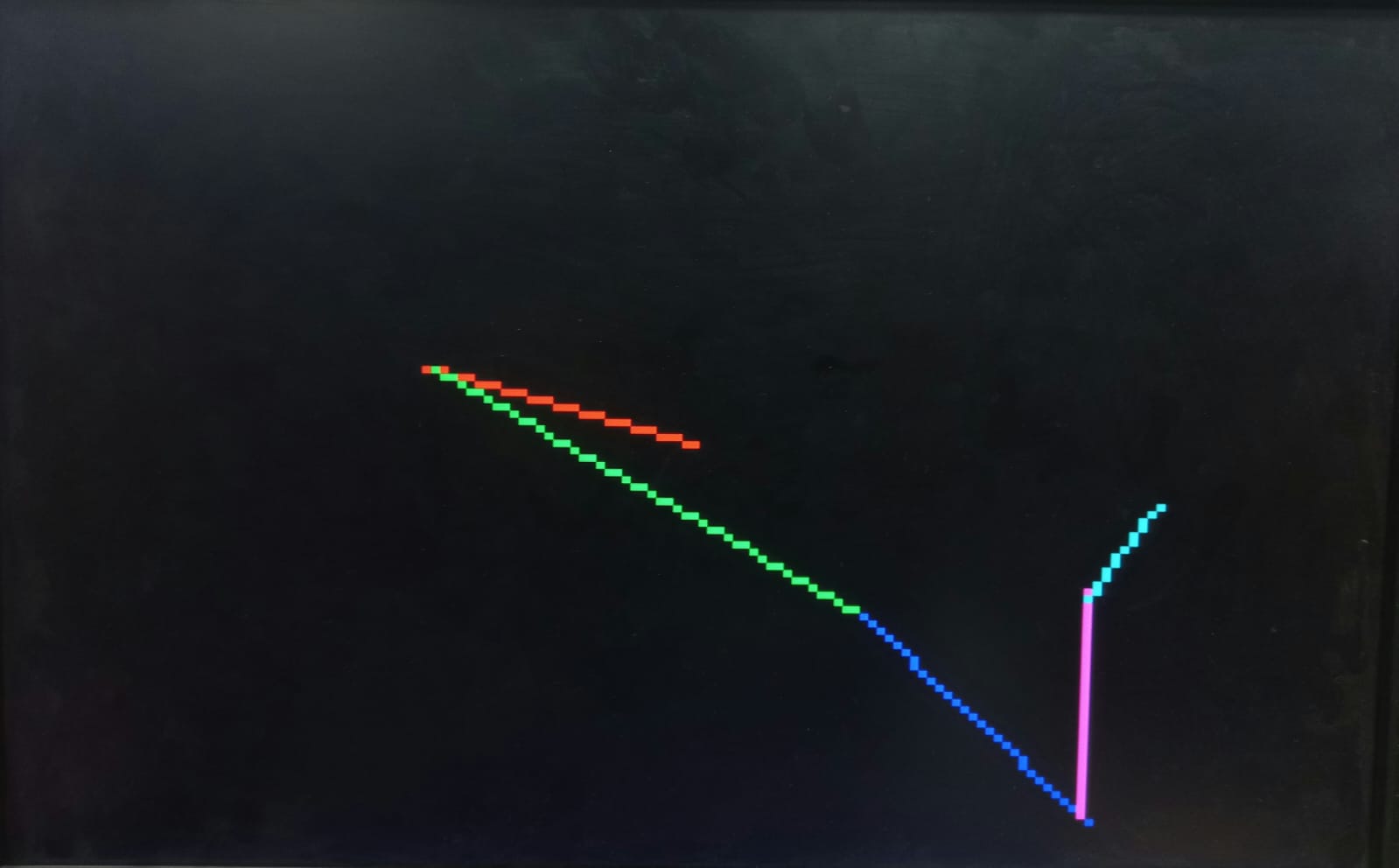


Figure : Challenge Task: Hardware Demonstration