Task 1: Understanding the VGA Adapter Core

In the initial phase of our project, we embarked on comprehending the intricacies of the VGA Adapter Core—a pivotal module essential for the rendering of geometric shapes on VGA displays. This core, a sophisticated piece of hardware designed at the University of Toronto, serves as the backbone for our objective to implement a circle-drawing algorithm on a VGA screen.

We directed our efforts towards integrating this VGA Adapter Core with our own circuitry. The core is adept at handling fundamental operations crucial for graphic technologies, which not only includes the rendering of circles but also the drawing of various other geometric figures. Recognizing its significance in the graphics rendering pipeline akin to those used in desktop PCs, gaming consoles, and virtual reality headsets, we diligently studied the core's specifications and operational parameters.

To facilitate our understanding and application, we referred to detailed documentation and resources available on the University of Toronto's website. This allowed us to gain a comprehensive understanding of the core's capabilities and limitations, thereby enabling us to effectively incorporate it into our design framework.

In our design, the VGA Adapter Core was configured to manipulate a display grid of 160x120 pixels. We meticulously connected our circuit with the core, ensuring that our hardware could command the pixel plotting process with precision. Through this integration, our system was capable of controlling the VGA interface to execute the shape-drawing functions, essentially turning theoretical concepts into tangible visual outputs on the VGA screen.

The culmination of these efforts was the successful rendering of circles on the VGA display, which not only demonstrated our

circuit's functionality but also validated the adaptability and robustness of the VGA Adapter Core in a practical application setting.

Task 2: Fill The Screen

In the second phase of our project, we successfully developed and implemented a component to interact with the VGA Adapter Core, with the specific objective of filling the VGA screen with a continuous spectrum of colors. Our component was designed around a simple Finite State Machine (FSM) that methodically plotted pixels on the screen in sequential order, assigning each row a distinct color that cycled every 8 rows. This task hinged on our ability to manipulate the VGA Adapter Core—one pixel at a time—using the FSM algorithm provided in the task description.

Our datapath module was constructed to manage the pixel coordinates, with an always block triggered on the positive edge of the CLOCK_50 signal or the negative edge of the reset. This careful sequencing ensured that each pixel coordinate was processed correctly, resetting to (0,0) upon a reset signal and otherwise iterating through the display's pixels in a raster scan fashion.

Concurrently, the controller module, tied intrinsically to the datapath, defined our color filling logic. It implemented a state machine that switched between states based on the plot signal. Upon activation, the controller would select a color value based on the current x-coordinate, cycling through a range of colors to produce a repeating pattern every 8 pixels. This approach guaranteed that each row would render in a unique color, thereby achieving the intended fill effect.

To encapsulate our design, we created the top-level module, VGA_Uploader, which served as the orchestrator of our project. This module instantiated both the controller and the datapath

and continuously drove the plot signal to maintain persistent pixel updates. The VGA_Uploader module also interfaced directly with the vga_adapter, ensuring that the VGA display received the appropriate synchronization signals and color data for each pixel.

Our team conducted extensive testing on the DE board to validate our design's functionality. The board was connected using a USB cable, a VGA cable, and a VGA-capable display. We paid special attention to the instructions regarding the setup to avoid any potential issues with VGA signal transmission, especially noting the VGA connection limitations related to the DE board and laptop.

The testing phase was crucial for our design refinement, and we employed the use of Modelsim for debugging our component outputs. With the help of simulation tools, we could efficiently identify and resolve any discrepancies in our FSM and pixel plotting logic.

After rigorous testing and debugging, we were pleased to observe that our project met the task's requirements: our FPGA successfully filled the VGA screen with rows of colors, cycling every 8 rows as per the specified algorithm. This accomplishment not only showcased our FPGA programming capabilities but also demonstrated a practical application of the VGA Adapter Core in a graphical display context.

Here's a breakdown of the code:

Module: Datapath

```
module datapath(CLOCK_50, rst, plot, x, y);
    input CLOCK_50, rst, plot;
    output reg [7:0] x;
    output reg [6:0] y;
```

```
    always @(posedge CLOCK_50 or negedge rst) begin
        if (rst == 0) begin
            x <= 0; y <= 0;
        end else if (plot == 1) begin
            if (y == 119) begin
                y <= 0;
                if (x == 159) begin
                    x <= 0;
                end else
                    x <= x + 1;
            end else
                y <= y + 1;
        end else begin
            x <= x;
            y <= y;
        end
    end
endmodule
```

The datapath module is responsible for generating the (x,y) pixel coordinates that are sent to the VGA Adapter. The coordinates are driven by CLOCK_50 and reset on a negative edge of rst. If plot is asserted, the coordinates increment to cover the screen size, which is 160x120 pixels (hence the conditional checks for x == 159 and y == 119). The coordinates wrap around to start at (0,0) after reaching the bottom-right corner of the screen.

Module: Controller

```
module controller(plot, reset, CLOCK_50, x, y, colour);
    input CLOCK_50, reset, plot;
```

```
output [7:0] x;
output [6:0] y;
output reg[2:0] colour;
reg clr;

datapath dp (CLOCK_50, reset, plot, x, y);

parameter S0 = 1'b0, S1 = 1'b1;

//Defining the states
reg [1:0]present_state, next_state;

always @(*) begin
//Checking the cases for the states
case(present_state)
    S0:
    begin
        if (plot == 1)
            next_state = S1;
        else
            next_state = S0;
    end
    S1:
    begin
        if(clr == 1)
        begin
            colour = 3'b000;
            next_state = S0;
```

```
            end
            else
            begin
                colour = x % 8;
                next_state = S1;
            end
        end
        default: next_state = S0;
    endcase
    end

    //State changer
    always@(posedge CLOCK_50 or negedge reset) begin
        if (~reset) begin
            clr <= 1;
            present_state <= S0;
        end
        else if (plot == 1) begin
            clr <= 0;
            present_state <= next_state;
        end else
            present_state <= S0;
    end

endmodule
```

The controller module incorporates the FSM that controls the coloring of the pixels. It defines two states, S0 and S1. The FSM remains in S0 when not plotting (plot != 1), and transitions to

S1 when it is ready to plot (plot == 1). In S1, the colour is determined based on the x-coordinate (x % 8), creating a cycle of 8 colors for the rows. The clr signal seems to be used to clear the color (set it to zero), but there is no logic in this snippet to change clr from outside the module.

Top Level Module: VGA Uploader

```
module VGA_Uploader(CLOCK_50,KEY,VGA_R, VGA_G,
VGA_B,VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N,
VGA_CLK);

    // Define our inputs and outputs
    input CLOCK_50;
    input [3:0] KEY;
    output [9:0] VGA_R;
    output [9:0] VGA_G;
    output [9:0] VGA_B;
    output VGA_HS;
    output VGA_VS;
    output VGA_BLANK_N;
    output VGA_SYNC_N;
    output VGA_CLK;

    // Define our wires
```

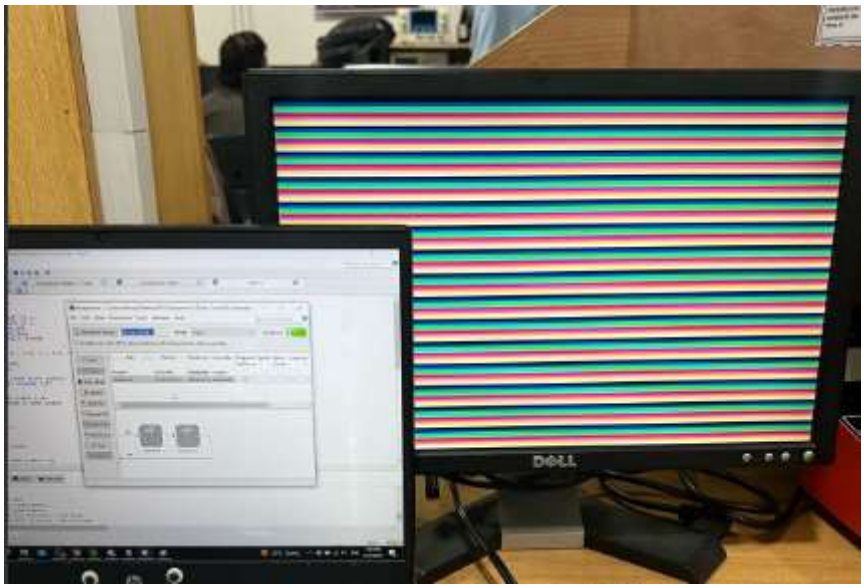
```
    wire [2:0] color;
    wire [7:0] x;
    wire [6:0] y;

    // We keep plot signal always as 1
wire rst = KEY[3];
    wire plot = 1'b1;

controller fill (plot, rst, CLOCK_50, x, y, colour);

    // Using the Va
vga_adapter VGA (
    .resetn(KEY[0]),
    .clock(CLOCK_50),
    .colour(color),
    .x(x),
    .y(y),
    .plot(plot),
            /* Signals for the DAC to drive the monitor. */
    .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;

endmodule
```

The VGA_Uploader module, developed in Verilog, serves the purpose of interfacing with VGA displays, handling a clock signal and a 4-bit key input to output VGA-compatible signals. These outputs encompass the essential VGA signals for color (red, green, blue), synchronization (horizontal and vertical sync), along with a blanking signal, sync signal, and a dedicated VGA clock signal. This setup ensures that the module can directly control a VGA display, dictating the color and positioning of pixels on the screen for visual output.

Internally, the module utilizes various wires and components, including a controller and a VGA adapter, to process and translate the input signals into a VGA-compliant format. The controller manages the drawing operations based on coordinates and color data, while the VGA adapter converts these operations into actual VGA signals. The module is configurable in terms of resolution, color mode, and color depth, allowing it to be tailored to specific VGA display requirements. This design makes the VGA_Uploader an adaptable tool for developers looking to integrate VGA displays into their digital projects.

Output



Task 3: Bresenham Circle Algorithm

We designed a FSM that implements the “Bresenham circle drawing” algorithm directly on the FPGA.

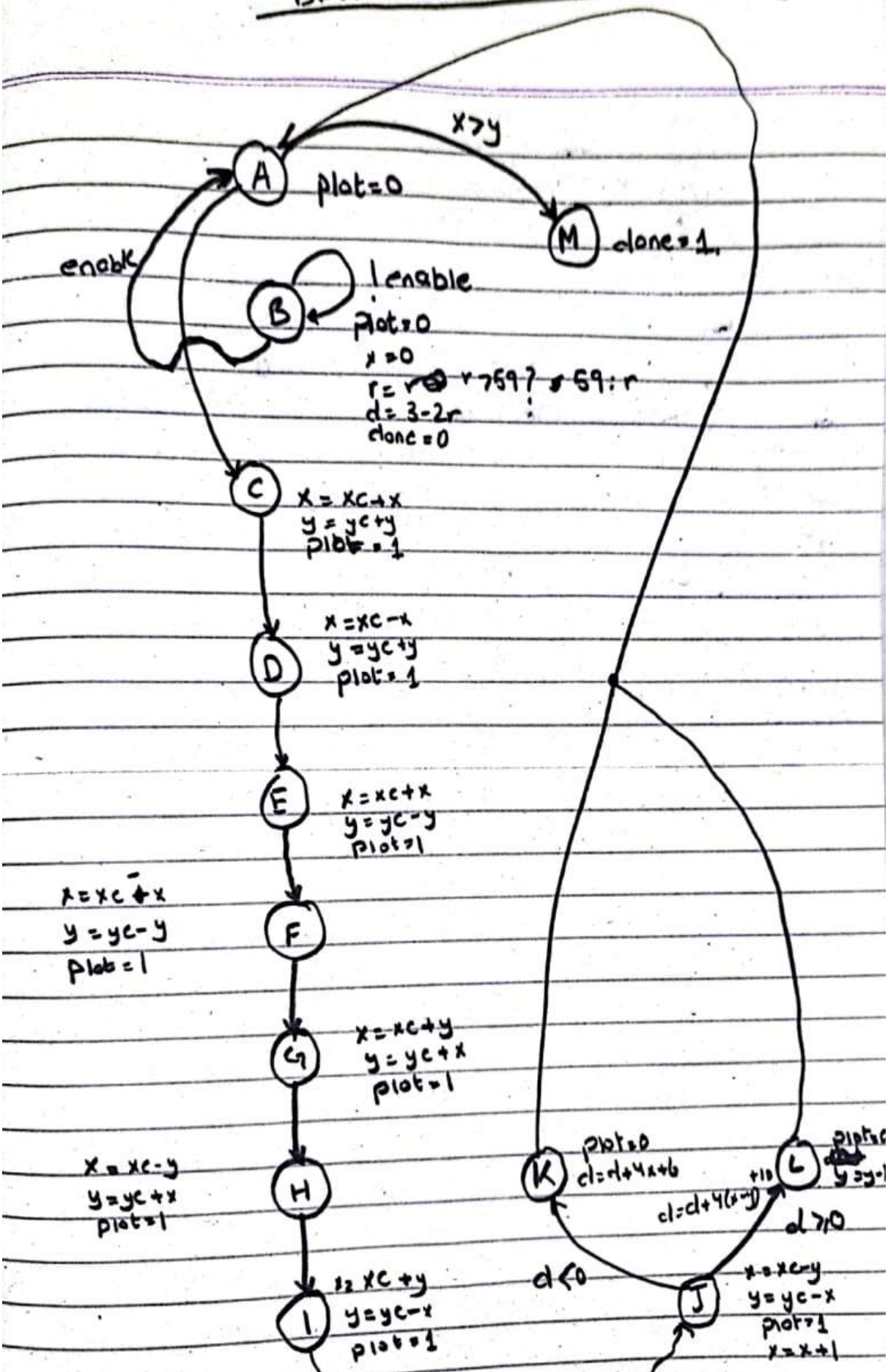
The working of this algorithm is just as in the state diagram below.

We used a module to create the circle using the algorithm. We also created the top level module that tests this module to draw a circle and control its color, and radius. Coupled with the clear module, we achieved the goal of this task.

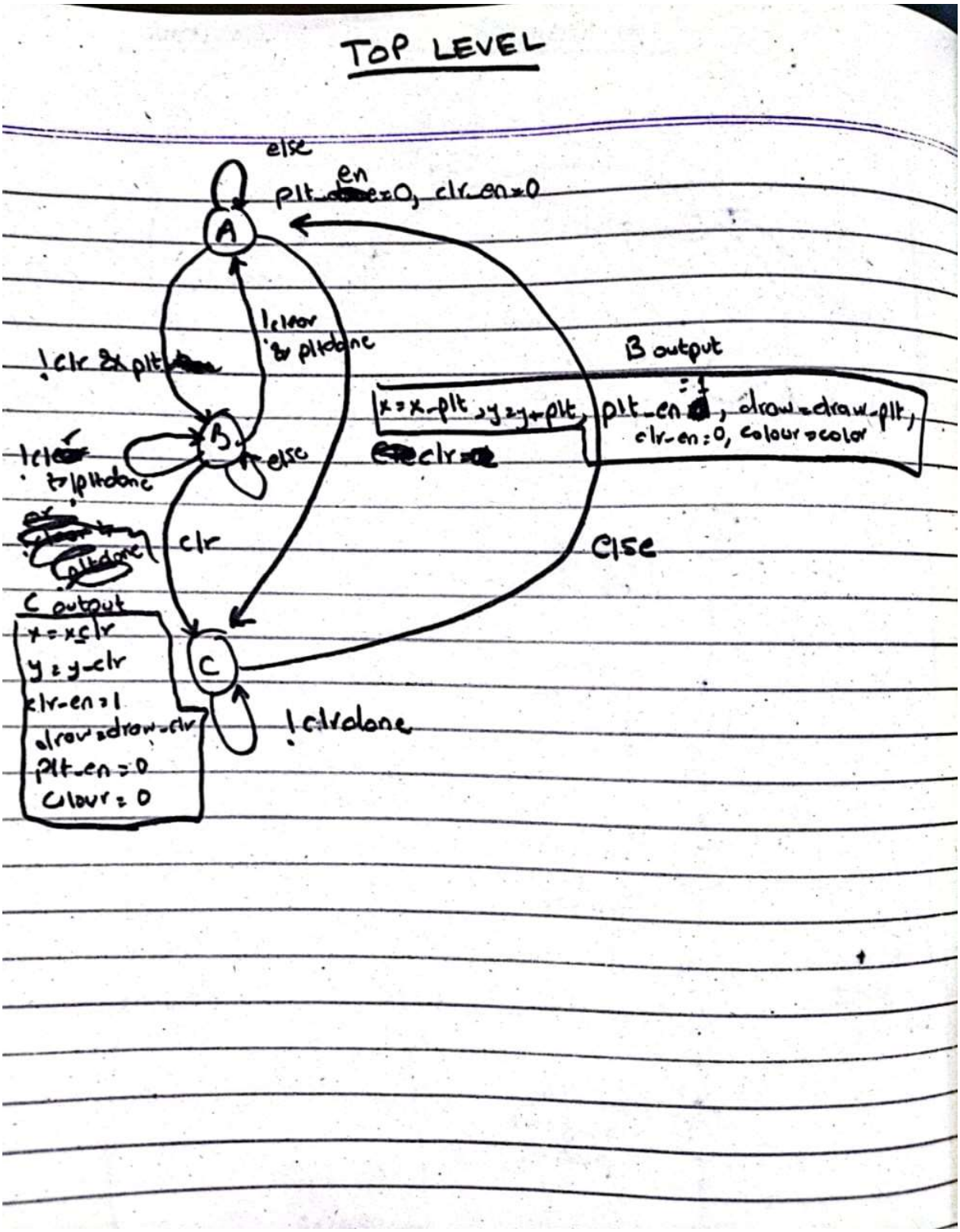
We also included the VGA adapter and designed the block of “your circuit” presented at start of the assignment.

Below are the state diagrams and you can use this link to view the demo (all source files included). We have attached some screenshots below too.

BRESENHAM. CIRCLE ALGORITHM



TOP LEVEL


else
en
plt_done=0, clr_en=0
A
!clear & plt_done
!clr & plt
B output
x = x_plt, y = y_plt
plt_en = 1, draw = draw_plt,
clr_en = 0, colour = color
B
else
clr
else
C output
x = x_clr
y = y_clr
clr_en = 1
draw = draw_clr
plt_en = 0
colour = 0
C
!clrdone