```
/***********************************

animator.sv

***********************************/

/*
 * Connor Aksama
 * 01/30/2023
 * CSE 371
 * Lab 3
*/

/**
 * Outputs the endpoints of the current line to draw on the display and a signal indicating whether
 * the scren should be cleared.

 * Inputs:
 *   clk [1 bit] - the clock to use for this module
*   reset [1 bit] - the reset signal for this module; resets to the first frame when raised

 * Outputs:
 *   x0 [10 bit] - The x coordinate of the first point of the line to draw
 *   y0 [9 bit] - The y coordinate of the first point of the line to draw
 *   x1 [10 bit] - The x coordinate of the second point of the line to draw
 *   y1 [9 bit] - The y coordinate of the second point of the line to draw
 *   clear [1 bit] - Signal to indicate when the screen should be cleared
*/
module animator#(
    parameter frame_time = 833333
    )(
    output logic [9:0] x0, x1
    ,output logic [8:0] y0, y1
    ,output logic clear
    ,input logic clk, reset
    );

    logic next_frame, next_frame_q;
    logic [31:0] frame_num;

    // Counter to time the duration of each frame
    // timer_out -> number of cycles current frame is active
    // inc on every cycle; no dec; use same clock, reset on frame switch/reset
    logic [31:0] timer_out;
    counter_32b timer (.count(timer_out), .inc(1'b1), .dec(1'b0), .clk, .reset(reset | next_frame));

    // Counter to keep track of the current frame number
    // frame_num -> current frame number
    // inc on every frame switch; no dec; use same clock, reset on end of sequence/reset
    logic frame_reset;
    counter_32b frame_counter (.count(frame_num), .inc(next_frame_q), .dec(1'b0), .clk, .reset(reset | frame_reset));

    // Determine if timer expired, switch to next frame logic
    always_comb begin

        // Check for next frame
        next_frame = (timer_out >= frame_time - 1) && !next_frame_q;

        // Clear screen every other frame
        clear = ~frame_num[0];

        // End of sequence reached
        frame_reset = (frame_num >= 100);

        // Frames
        case (frame_num[31:1])

            0:begin x0=420;y0=240;x1=220;y1=240;end  1:begin x0=419;y0=243;x1=221;y1=237;end  2:begin x0=419;y0=246;x1=221;y1=234;end  3:begin x0=419;y0=249;x1=221;y1=231;end
            4:begin x0=419;y0=252;x1=221;y1=228;end  5:begin x0=418;y0=255;x1=222;y1=225;end  6:begin x0=418;y0=258;x1=222;y1=222;end  7:begin x0=417;y0=261;x1=223;y1=219;end
            8:begin x0=416;y0=264;x1=224;y1=216;end  9:begin x0=416;y0=267;x1=224;y1=213;end  10:begin x0=415;y0=270;x1=225;y1=210;end  11:begin x0=414;y0=273;x1=226;y1=207;end
            12:begin x0=412;y0=276;x1=228;y1=204;end  13:begin x0=411;y0=279;x1=229;y1=201;end  14:begin x0=410;y0=282;x1=230;y1=198;end  15:begin x0=409;y0=285;x1=231;y1=195;end
            16:begin x0=407;y0=288;x1=233;y1=192;end  17:begin x0=406;y0=290;x1=234;y1=190;end  18:begin x0=404;y0=293;x1=236;y1=187;end  19:begin x0=402;y0=296;x1=238;y1=184;end
            20:begin x0=400;y0=298;x1=240;y1=182;end  21:begin x0=399;y0=301;x1=241;y1=179;end  22:begin x0=397;y0=303;x1=243;y1=177;end  23:begin x0=395;y0=306;x1=245;y1=174;end
            24:begin x0=392;y0=308;x1=248;y1=172;end  25:begin x0=390;y0=310;x1=250;y1=170;end  26:begin x0=388;y0=312;x1=252;y1=168;end  27:begin x0=386;y0=315;x1=254;y1=165;end
            28:begin x0=383;y0=317;x1=257;y1=163;end  29:begin x0=381;y0=319;x1=259;y1=161;end  30:begin x0=378;y0=320;x1=262;y1=160;end  31:begin x0=376;y0=322;x1=264;y1=158;end
            32:begin x0=373;y0=324;x1=267;y1=156;end  33:begin x0=370;y0=326;x1=270;y1=154;end  34:begin x0=368;y0=327;x1=272;y1=153;end  35:begin x0=365;y0=329;x1=275;y1=151;end
            36:begin x0=362;y0=330;x1=278;y1=150;end  37:begin x0=359;y0=331;x1=281;y1=149;end  38:begin x0=356;y0=332;x1=284;y1=148;end  39:begin x0=353;y0=334;x1=287;y1=146;end
            40:begin x0=350;y0=335;x1=290;y1=145;end  41:begin x0=347;y0=336;x1=293;y1=144;end  42:begin x0=344;y0=336;x1=296;y1=144;end  43:begin x0=341;y0=337;x1=299;y1=143;end
            44:begin x0=338;y0=338;x1=302;y1=142;end  45:begin x0=335;y0=338;x1=305;y1=142;end  46:begin x0=332;y0=339;x1=308;y1=141;end  47:begin x0=329;y0=339;x1=311;y1=141;end
            48:begin x0=326;y0=339;x1=314;y1=141;end  49:begin x0=323;y0=339;x1=317;y1=141;end  50:begin x0=320;y0=340;x1=320;y1=140;end  51:begin x0=317;y0=339;x1=323;y1=141;end
            52:begin x0=314;y0=339;x1=326;y1=141;end  53:begin x0=311;y0=339;x1=329;y1=141;end  54:begin x0=308;y0=339;x1=332;y1=141;end  55:begin x0=305;y0=338;x1=335;y1=142;end
            56:begin x0=302;y0=338;x1=338;y1=142;end  57:begin x0=299;y0=337;x1=341;y1=143;end  58:begin x0=296;y0=336;x1=344;y1=144;end  59:begin x0=293;y0=336;x1=347;y1=144;end
            60:begin x0=290;y0=335;x1=350;y1=145;end  61:begin x0=287;y0=334;x1=353;y1=146;end  62:begin x0=284;y0=332;x1=356;y1=148;end  63:begin x0=281;y0=331;x1=359;y1=149;end
            64:begin x0=278;y0=330;x1=362;y1=150;end  65:begin x0=275;y0=329;x1=365;y1=151;end  66:begin x0=272;y0=327;x1=368;y1=153;end  67:begin x0=270;y0=326;x1=370;y1=154;end
            68:begin x0=267;y0=324;x1=373;y1=156;end  69:begin x0=264;y0=322;x1=376;y1=158;end  70:begin x0=262;y0=320;x1=378;y1=160;end  71:begin x0=259;y0=319;x1=381;y1=161;end
            72:begin x0=257;y0=317;x1=383;y1=163;end  73:begin x0=254;y0=315;x1=386;y1=165;end  74:begin x0=252;y0=312;x1=388;y1=168;end  75:begin x0=250;y0=310;x1=390;y1=170;end
            76:begin x0=248;y0=308;x1=392;y1=172;end  77:begin x0=245;y0=306;x1=395;y1=174;end  78:begin x0=243;y0=303;x1=397;y1=177;end  79:begin x0=241;y0=301;x1=399;y1=179;end
            80:begin x0=240;y0=298;x1=400;y1=182;end  81:begin x0=238;y0=296;x1=402;y1=184;end  82:begin x0=236;y0=293;x1=404;y1=187;end  83:begin x0=234;y0=290;x1=406;y1=190;end
            84:begin x0=233;y0=288;x1=407;y1=192;end  85:begin x0=231;y0=285;x1=409;y1=195;end  86:begin x0=230;y0=282;x1=410;y1=198;end  87:begin x0=229;y0=279;x1=411;y1=201;end
            88:begin x0=228;y0=276;x1=412;y1=204;end  89:begin x0=226;y0=273;x1=414;y1=
                207;end  90:begin x0=225;y0=270;x1=415;y1=210;end  91:begin x0=224;y0=267;x1=
                416;y1=213;end
                92:begin x0=224;y0=264;x1=416;y1=216;end  93:begin x0=223;y0=261;x1=417;y1=
                219;end  94:begin x0=222;y0=258;x1=418;y1=222;end  95:begin x0=222;y0=255;x1=
                418;y1=225;end
                96:begin x0=221;y0=252;x1=419;y1=228;end  97:begin x0=221;y0=249;x1=419;y1=
                231;end  98:begin x0=221;y0=246;x1=419;y1=234;end  99:begin x0=221;y0=243;x1=
                419;y1=237;end
                default: begin x0=0;y0=0;x1=0;y1=0; end

            endcase

        end

        // Register the next frame signal to frame counter
        always_ff @(posedge clk) begin
            if (reset) begin
                next_frame_q <= 0;
            end else begin
                next_frame_q <= next_frame;
            end
        end

    endmodule  // animator

    // Module to test the functionality of the animator module
    module animator_testbench();

        logic [9:0] x0, x1;
        logic [8:0] y0, y1;
        logic clear;
        logic clk, reset;

        animator #(2) dut (.*);

        parameter CLOCK_PERIOD = 100;
        initial begin
            clk <= 0;
            forever #(CLOCK_PERIOD / 2) clk <= ~clk;
        end

        initial begin
            integer i;

            @(posedge clk) reset <= 1;
            @(posedge clk) reset <= 0;

            // Run through each of the frames in the animation
            for (i = 0; i < 500; i++) @(posedge clk);

            $stop;
        end


    endmodule  // animator_testbench

    /************************************

    counter_32b.sv

    ************************************/

    /*
     * Connor Aksama
     * 01/30/2023
     * CSE 371
     * Lab 3
    */

```

```
/**
 * A thirty two-bit increment/decrement counter. Overflows to 8'h0; underflows to
 8'hFFFF_FFFF.
 * reset takes precedence; if inc and dec are simultaneously raised, count is unchanged.
 * Inputs:
 *   inc [1 bit] - Increments the count by 1 when raised. Does nothing if signal is low.
 *   dec [1 bit] - Decrements the count by 1 when raised. Does nothing if signal is low.
 *   clk [1 bit] - The clock to use for this module.
 *   reset [1 bit] - Resets the counter to 0 when raised. Does nothing if signal is low.

 * Outputs:
 *      count [32 bit] - The count of this counter.
 */
module counter_32b(
    output logic [31:0] count
    ,input logic inc, dec, clk, reset
    );

    // Handles increment/decrement/reset operations
    always_ff @(posedge clk) begin

        if (reset)
            count <= 32'b0;
        else if (inc & ~dec)
            count <= count + 32'b1;
        else if (dec & ~inc)
            count <= count - 32'b1;

    end

endmodule  // counter_32b

// Module to test the functionality of the counter_32b module
module counter_32b_testbench();

    logic [31:0] count;
    logic inc, dec, clk, reset;

    counter_32b dut (.*);

    parameter CLOCK_PERIOD = 100;
    initial begin
        clk <= 0;
        forever #(CLOCK_PERIOD / 2) clk <= ~clk;
    end

    initial begin
        integer i;

        @(posedge clk) reset <= 1;
        @(posedge clk) reset <= 0; inc <= 1; dec <= 0;
        for (i = 0; i < 40; i++) begin
            @(posedge clk);
        end

        @(posedge clk) reset <= 0; inc <= 0; dec <= 1;
        for (i = 0; i < 50; i++) begin
            @(posedge clk);
        end

        @(posedge clk) reset <= 0; inc <= 1; dec <= 0;
        for (i = 0; i < 15; i++) begin
            @(posedge clk);
        end

        $stop;
    end

endmodule  // counter_32b_testbench
```

```

/***********************************

DE1_SoC.sv

***********************************/

/*
 * Connor Aksama
 * 01/30/2023
 * CSE 371
 * Lab 3
*/

/**
 * Top-level module for Lab 3. Instantiates drawing logic modules and connections to VGA
 controller

 * Inputs:
 *  SW [10 bit] - The 10 onboard switches, respectively.
 *  KEY [4 bit] - The 4 onboard keys, respectively.
 *  CLOCK_50 [1 bit] - The system clock to use for this module.

 * Outputs:
 *  HEX0 [7 bit] - Data to show on the HEX0 display, formatted in standard 7 segment
 display format.
 *  HEX1 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
 display format.
 *  HEX2 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
 display format.
 *  HEX3 [7 bit] - Data to show on the HEX1 display, formatted in standard 7 segment
 display format.
 *  HEX4 [7 bit] - Data to show on the HEX4 display, formatted in standard 7 segment
 display format.
 *  HEX5 [7 bit] - Data to show on the HEX5 display, formatted in standard 7 segment
 display format.
 *  LEDR [10 bit] - Signal to output to 10 onboard LEDs, respectively.
 *  VGA_R [8 bit] - The red channel value of the pixel to write to the display
 *  VGA_G [8 bit] - The green channel value of the pixel to write to the display
 *  VGA_B [8 bit] - The blue channel value of the pixel to write to the display
 *  VGA_BLANK_N [1 bit] - Blank area signal
 *  VGA_CLK [1 bit] - Divided clock for VGA signal
 *  VGA_HS [1 bit] - VGA timing signal
 *  VGA_SYNC_N [1 bit] - Unused
 *  VGA_VS [1 bit] - VGA timing signal
*/
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 HEX1 = '1;
    assign HEX0 = '1;
    assign HEX2 = '1;
    assign HEX3 = '1;
    assign HEX4 = '1;
    assign HEX5 = '1;
    assign LEDR = SW;

    // Outputs from drawing modules
    logic [9:0] x0_l, x0_a, x1, x1_l, x1_a;
    logic [8:0] y0_l, y0_a, y1, y1_l, y1_a;
    // Inputs into frame buffer
    logic [9:0] x_l, x_a, x_c, x;
    logic [8:0] y_l, y_a, y_c, y;
    logic frame_start;
    logic pixel_color;
    logic reset, clear_screen;

    assign reset = SW[9];
    assign clear_screen = SW[8];


    //////// DOUBLE_FRAME_BUFFER ////////
    logic dfb_en;
    assign dfb_en = 1'b0;
    /////////////////////////////////////

    VGA_framebuffer fb(.clk(CLOCK_50), .rst(1'b0), .x, .y,
                .pixel_color, .pixel_write(1'b1), .dfb_en, .frame_start,
                .VGA_R, .VGA_G, .VGA_B, .VGA_CLK, .VGA_HS, .VGA_VS,
                .VGA_BLANK_N, .VGA_SYNC_N);

    // Line Drawing Module
    // Inputs -> system clock, reset signal, (x0_l, y0_l), (x1_l, y1_l) -> from local
    temp logic; defines endpoints of line
    // Outputs -> (x_l, y_l) -> pixel to draw to VGA - changes by at most one pixel each
    clock cycle, starting one
    //                          cycle after input points change
    line_drawer lines (.clk(CLOCK_50), .reset,
                .x0(x0_l), .y0(y0_l), .x1(x1_l), .y1(y1_l), .x(x_l), .y(y_l));

    // Animator Module
    // Inputs -> system clock, reset signal
    // Outputs -> (x0_a, y0_a), (x1_a, y1_a) -> Endpoints of line to draw
    //            clear -> signal is high when screen should be cleared
    logic clear;
    animator #(50000000) anim (.x0(x0_a), .x1(x1_a), .y0(y0_a), .y1(y1_a), .clear, .clk(
    CLOCK_50), .reset);

    // Screen Clearning Module; cycles through each pixel on screen
    // Inputs -> system clock, reset signal
    // Output -> (x_c, y_c) -> Coordinate of pixel to clear on display
    fill_space #(640,480) clearer (.x(x_c), .y(y_c), .clk(CLOCK_50), .reset);

    // Multiplex endpoints into line drawing module
    // Multiplex coordinates into frame buffer
    always_comb begin
        if (SW[7]) begin
            // Display arbitrary lines based on SW input
            case (SW[2:0])
                0: begin
                    // vertical
                    x0_l = 50;
                    y0_l = 50;
                    x1_l = 50;
                    y1_l = 400;
                    pixel_color = 1'b1;
                end
                1: begin
                    // horizontal
                    x0_l = 90;
                    y0_l = 90;
                    x1_l = 450;
                    y1_l = 90;
```

```
                        pixel_color = 1'b1;
                    end
                    2: begin
                        // negative diagonal
                        x0_l = 240;
                        y0_l = 240;
                        x1_l = 0;
                        y1_l = 0;
                        pixel_color = 1'b1;
                    end
                    3: begin
                        // positive diagonal
                        x0_l = 400;
                        y0_l = 40;
                        x1_l = 40;
                        y1_l = 400;
                        pixel_color = 1'b1;
                    end
                    4: begin
                        // positive shallow
                        x0_l = 400;
                        y0_l = 40;
                        x1_l = 40;
                        y1_l = 100;
                        pixel_color = 1'b1;
                    end
                    5: begin
                        // negative shallow
                        x0_l = 300;
                        y0_l = 200;
                        x1_l = 40;
                        y1_l = 100;
                        pixel_color = 1'b1;
                    end
                    6: begin
                        // positive steep
                        x0_l = 40;
                        y0_l = 400;
                        x1_l = 80;
                        y1_l = 40;
                        pixel_color = 1'b1;
                    end
                    7: begin
                        // negative steep
                        x0_l = 40;
                        y0_l = 40;
                        x1_l = 80;
                        y1_l = 400;
                        pixel_color = 1'b1;
                    end
                    default: begin
                        x0_l = 0;
                        y0_l = 0;
                        x1_l = 0;
                        y1_l = 0;
                        pixel_color = 1'b1;
                    end

                endcase
                x = x_l;
                y = y_l;
            end
            else if (clear | clear_screen) begin
                // Clear Screen
                x0_l = 0;
                y0_l = 0;
                x1_l = 0;
                y1_l = 0;
                pixel_color = 1'b0;
                x = x_c;
                y = y_c;
            end
            else begin
                // Display animation
                x0_l = x0_a;
                y0_l = y0_a;
                x1_l = x1_a;
                y1_l = y1_a;
                pixel_color = 1'b1;
                x = x_l;
                y = y_l;
            end
        end


    endmodule  // DE1_SoC

    // Module to test the functionality of the DE1_SoC module
    module DE1_SoC_testbench();
        logic [6:0] HEX0, HEX1, HEX2, HEX3, HEX4, HEX5;
        logic [9:0] LEDR;
        logic [3:0] KEY;
        logic [9:0] SW;

        logic clk;
        logic [7:0] VGA_R;
        logic [7:0] VGA_G;
        logic [7:0] VGA_B;
        logic VGA_BLANK_N;
        logic VGA_CLK;
        logic VGA_HS;
        logic VGA_SYNC_N;
        logic VGA_VS;

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

        parameter CLOCK_PERIOD = 100;
        initial begin
            clk <= 0;
            forever #(CLOCK_PERIOD / 2) clk <= ~clk;
        end

        initial begin
            integer i;

            @(posedge clk) SW[9] <= 1;
            @(posedge clk) SW[9] <= 0;
            @(posedge clk) SW[7] <= 0;

            for (i = 0; i < 50; i++) begin
                @(posedge clk);
            end

            // Run through each of the arbitrary lines
            @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b000;
            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end
            @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b001;
            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end
            @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b010;
            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end
            @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b011;
            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end
            @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b100;
            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end
            @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b101;
            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end
            @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b110;
            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end
            @(posedge clk) SW[7] <= 1; SW[2:0] = 3'b111;
            for (i = 0; i < 10; i++) begin
                @(posedge clk);
            end
            // Clear the screen
            @(posedge clk) SW[7] <= 0; SW[8] <= 1;
            for (i = 0; i < 20; i++) begin
                @(posedge clk);
            end
            // Run through the animation
            @(posedge clk) SW[8] <= 0;
            for (i = 0; i < 100; i++) begin
                @(posedge clk);
            end

            $stop;
        end

    endmodule  // DE1_SoC_testbench

    /***********************************

    fill_space.sv

    ***********************************/

    /*
     * Connor Aksama
     * 01/30/2023
     * CSE 371
     * Lab 3
    */

    /**
     * Cycles through each pixel on a screen with the size of the parameterized dimensions.
     * Ex. (0,0) -> (1,0) -> (2,0) -> ... -> (638,479) -> (639, 479) -> (0,0)
     * Outputs a new pixel on each clock cycle.

     * Inputs:
     *  clk [1 bit] - the clock to use for this module
    *   reset [1 bit] - the reset signal for this module; resets to point (0,0)

     * Outputs:
     *  x [10 bit] - The x coordinate of the pixel to draw to
     *  y [9 bit] - The y coordinate of the pixel to draw to
    */
    module fill_space #(
        parameter width = 640, height = 480
        )
        (
        output logic [9:0] x
        ,output logic [8:0] y
        ,input logic clk, reset
        );

        logic [9:0] x_d, x_q;
        logic [8:0] y_d, y_q;

        assign x = x_q;
        assign y = y_q;

        // Determine next coordinate
        always_comb begin
            if (x_q >= width - 1 && y_q >= height - 1) begin
                // Reached x max and y max
                x_d = 0;
                y_d = 0;
            end else if (x_q >= width - 1) begin
                // Reached x max
                x_d = 0;
                y_d = y_q + 1;
            end else begin
                // Reached y max
                x_d = x_q + 1;
                y_d = y_q;
            end
        end

        // Register outputs
        always_ff @(posedge clk) begin
            if (reset) begin
                x_q <= 0;
                y_q <= 0;
            end else begin
                x_q <= x_d;
                y_q <= y_d;
            end
        end

    endmodule  // fill_space

    // Module to test the functionality of the fill_space module
    module fill_space_testbench();

        logic [9:0] x;
        logic [8:0] y;
        logic clk, reset;

        fill_space #(4,5) dut (.*);

        parameter CLOCK_PERIOD = 100;
        initial begin
            clk <= 0;
            forever #(CLOCK_PERIOD / 2) clk <= ~clk;
        end

        initial begin
            integer i;

            @(posedge clk) reset <= 1;
            @(posedge clk) reset <= 0;

            // Run through each pixel in the screen
            for (i = 0; i < 50; i++) begin
                @(posedge clk);
            end

            $stop;
        end

    endmodule  // fill_space_testbench
```

```

/**********************************

line_drawer.sv

**********************************/

/*
 * Connor Aksama
 * 01/30/2023
 * CSE 371
 * Lab 3
*/

/**
 * Given two endpoints (x0, y0), (x1, y1), outputs a sequence of points (x,y) that connect the two points.
 * Outputs at most one new point (x,y) every clock cycle starting on the cycle after the input endpoints change.

 * Inputs:
 *  clk [1 bit] - the clock to use for this module
 *  reset [1 bit] - the reset signal for this module; resets algorithm to first iteration given inputs
 *  x0 [10 bit] - The x coordinate of the first point of the line to draw
 *  y0 [9 bit] - The y coordinate of the first point of the line to draw
 *  x1 [10 bit] - The x coordinate of the second point of the line to draw
 *  y1 [9 bit] - The y coordinate of the second point of the line to draw

 * Outputs:
 *  x [10 bit] - The x coordinate of the current pixel to draw
 *  y [9 bit] - The y coordinate of the current pixel to draw
*/
module line_drawer(
    input logic clk, reset,

    // x and y coordinates for the start and end points of the line
    input logic [9:0] x0, x1,
    input logic [8:0] y0, y1,

    //outputs cooresponding to the coordinate pair (x, y)
    output logic [9:0] x,
    output logic [8:0] y
    );

    // Bresenham error
    logic signed [11:0] error, p_error, n_error;

    // Temp swap logic
    logic signed [9:0] x_min, x_max, y_start,  y_end;
    logic signed [9:0] x0_r, x1_r, y0_r, y1_r;
    // Bresenham temp calculations
    logic is_steep, is_steep_r;
    logic signed [10:0] delta_x;
    logic signed [10:0] delta_y;
    logic signed [9:0] step;

    // Registered output
    logic signed [9:0] px, nx;
    logic signed [8:0] py, ny;
    logic [9:0] prev_x0, prev_x1;
    logic [8:0] prev_y0, prev_y1;

    // Compute initial values before entering main loop
    always_comb begin

        delta_x = x1 - x0;
        delta_y = y1 - y0;

```

```
            is_steep = (delta_y < 0 ? -delta_y : delta_y) > (delta_x < 0 ? -delta_x : delta_x
            );

            if (is_steep) begin
                // Swap x, y
                x0_r = y0;
                y0_r = x0;
                x1_r = y1;
                y1_r = x1;
            end else begin
                x0_r = x0;
                x1_r = x1;
                y0_r = y0;
                y1_r = y1;
            end

            if (x0_r > x1_r) begin
                // swap p0, p1
                x_min = x1_r;
                x_max = x0_r;
                y_start = y1_r;
                y_end = y0_r;
            end else begin
                x_min = x0_r;
                x_max = x1_r;
                y_start = y0_r;
                y_end = y1_r;
            end

            delta_x = x_max - x_min;
            delta_y = (y_end - y_start < 0 ? -(y_end - y_start) : y_end - y_start);
            error = -(delta_x / 2);

            if (y_start < y_end) step = 1;
            else                 step = -1;

            if (p_error + delta_y >= 0 && px < x_max) begin
                // Step x and y, compute next error
                ny = py + step;
                nx = px + 1;
                n_error = p_error + delta_y - delta_x;
            end else if (px < x_max) begin
                // Step x, compute next error
                ny = py;
                nx = px + 1;
                n_error = p_error + delta_y;
            end else begin
                // Reached end of line, keep outputting same point
                nx = px;
                ny = py;
                n_error = p_error;
            end

            if (is_steep_r) begin
                // Swap x,y
                x = py;
                y = px;
            end else begin
                x = px;
                y = py;
            end

        end

        // Update new values of current x,y in main loop
        always_ff @(posedge clk) begin
            prev_x0 <= x0;
            prev_x1 <= x1;
            prev_y0 <= y0;
            prev_y1 <= y1;
            is_steep_r <= is_steep;

            if (reset || prev_x0 != x0 || prev_y0 != y0 || prev_x1 != x1 || prev_y1 != y1)
            begin
                // Reset initial values to input if input changes or reset
                px <= x_min;
                py <= y_start;
                p_error <= error;
            end else begin
                // Output computed next values
                px <= nx;
                py <= ny;
                p_error <= n_error;
            end

        end

    endmodule  // line_drawer

    // Module to test the functionality of the line_drawer module
    module line_drawer_testbench();

        logic clk, reset;

        logic [9:0] x0, x1;
        logic [8:0] y0, y1;

        logic [9:0] x;
        logic [8:0] y;

        line_drawer dut (.*);

        parameter CLOCK_PERIOD = 100;
        initial begin
            clk <= 0;
            forever #(CLOCK_PERIOD  / 2) clk <= ~clk;
        end

        initial begin
            integer i;

            @(posedge clk) reset <= 1; x0 <= 0; y0 <= 0; x1 <= 0; y1 <= 0;

            // Horizontal
            @(posedge clk) reset <= 0; x0 <= 0; y0 <= 0; x1 <= 10; y1 <= 0;
            for (i = 0; i < 20; i++) begin
                @(posedge clk);
            end

            // Vertical
            @(posedge clk) x0 <= 0; y0 <= 0; x1 <= 0; y1 <= 10;
            for (i = 0; i < 20; i++) begin
                @(posedge clk);
            end

            // Diagonal
            @(posedge clk) x0 <= 0; y0 <= 0; x1 <= 5; y1 <= 5;
            for (i = 0; i < 20; i++) begin
                @(posedge clk);
            end

            // Shallow
            @(posedge clk) x0 <= 0; y0 <= 0; x1 <= 10; y1 <= 2;
            for (i = 0; i < 20; i++) begin
                @(posedge clk);
            end

            // Steep
            @(posedge clk) x0 <= 0; y0 <= 0; x1 <= 2; y1 <= 10;
            for (i = 0; i < 20; i++) begin
                @(posedge clk);
            end

            $stop;
        end

    endmodule  // line_drawer_testbench

    /************************************

    VGA_framebuffer.sv

    ************************************/

    // VGA driver: provides I/O timing and double-buffering for the VGA port.

    module VGA_framebuffer(
        input logic clk, rst,
        input logic [9:0] x, // The x coordinate to write to the buffer.
        input logic [8:0] y, // The y coordinate to write to the buffer.
        input logic pixel_color, pixel_write, // The data to write (color) and write-enable.

        input logic dfb_en, // Double-Frame Buffer Enable

        output logic frame_start,   // Pulse is fired at the start of a frame.

        // Outputs to the VGA port.
        output logic [7:0] VGA_R, VGA_G, VGA_B,
        output logic VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N
    );

        /*
        *
        * HCOUNT 1599 0             1279       1599 0
        *            _______________              ________
        * __________|    Video      |____________|  Video
        *
        *
        * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
        *       _______________________      _____________
        * |____|       VGA_HS          |____|
        *
        */

        // Constants for VGA timing.
        localparam HPX = 11'd640*2, HFP = 11'd16*2, HSP = 11'd96*2, HBP = 11'd48*2;
        localparam VLN = 11'd480,   VFP = 10'd11,   VSP = 10'd2,    VBP = 10'd31;
        localparam HTOTAL = HPX + HFP + HSP + HBP; // 800*2=1600
        localparam VTOTAL = VLN + VFP + VSP + VBP; // 524

        // Horizontal counter.
        logic [10:0] h_count;
        logic end_of_line;

        assign end_of_line = h_count == HTOTAL - 1;

        always_ff @(posedge clk)
            if (rst) h_count <= 0;
            else if (end_of_line) h_count <= 0;
            else h_count <= h_count + 11'd1;

        // Vertical counter & buffer swapping.
        logic [9:0] v_count;
        logic end_of_field;
        logic front_odd; // whether odd address is the front buffer.

        assign end_of_field = v_count == VTOTAL - 1;
        assign frame_start = !h_count && !v_count;

        always_ff @(posedge clk)
            if (rst) begin
                v_count <= 0;
                front_odd <= 0;
            end else if (end_of_line)
                if (end_of_field) begin
                    v_count <= 0;
                    front_odd <= !front_odd;
                end else
                    v_count <= v_count + 10'd1;

        // Sync signals.
        assign VGA_CLK = h_count[0]; // 25 MHz clock: pixel latched on rising edge.
        assign VGA_HS = !(h_count - (HPX + HFP) < HSP);
        assign VGA_VS = !(v_count - (VLN + VFP) < VSP);
        assign VGA_SYNC_N = 1; // Unused by VGA

        // Blank area signal.
        logic blank;
        assign blank = h_count >= HPX || v_count >= VLN;

        // Double-buffering.
        logic buffer[640*480*2-1:0];
        logic [19:0] wr_addr, rd_addr;
        logic rd_data;

        assign wr_addr = {y * 19'd640 + x, (!front_odd & dfb_en)};
        assign rd_addr = {v_count * 19'd640 + (h_count / 19'd2), (front_odd & dfb_en)};

        always_ff @(posedge clk) begin
            if (pixel_write) buffer[wr_addr] <= pixel_color;
            if (VGA_CLK) begin
                rd_data <= buffer[rd_addr];
                VGA_BLANK_N <= ~blank;
            end
        end

        // Color output.
        assign {VGA_R, VGA_G, VGA_B} = rd_data ? 24'hFFFFFF : 24'h000000;
    endmodule
```