```
`timescale 1ns / 1ns

module animateSquare
    (
        CLOCK_50,                       //  On Board 50 MHz
        // Your inputs and outputs here
        KEY,                            // On Board Keys
        SW,
        // The ports below are for the VGA output.  Do not change.
        VGA_CLK,                        //  VGA Clock
        VGA_HS,                         //  VGA H_SYNC
        VGA_VS,                         //  VGA V_SYNC
        VGA_BLANK_N,                        //  VGA BLANK
        VGA_SYNC_N,                     //  VGA SYNC
        VGA_R,                          //  VGA Red[9:0]
        VGA_G,                          //  VGA Green[9:0]
        VGA_B                           //  VGA Blue[9:0]
    );

    input           CLOCK_50;               //  50 MHz
    input   [3:0]   KEY;
    // Declare your inputs and outputs here
    input [9:0] SW;
    // Do not change the following outputs
    output          VGA_CLK;                //  VGA Clock
    output          VGA_HS;                 //  VGA H_SYNC
    output          VGA_VS;                 //  VGA V_SYNC
    output          VGA_BLANK_N;                //  VGA BLANK
    output          VGA_SYNC_N;             //  VGA SYNC
    output  [7:0]   VGA_R;                  //  VGA Red[7:0] Changed from 10 to 8-bit DAC
    output  [7:0]   VGA_G;                  //  VGA Green[7:0]
    output  [7:0]   VGA_B;                  //  VGA Blue[7:0]

    wire resetn;
    assign resetn = KEY[0]; // Active low, so don't invert

    // Create the colour, x, y and writeEn wires that are inputs to the controller.

    wire [2:0] colour;
    wire [7:0] x,y;
    wire writeEn;

    // Create an Instance of a VGA controller - there can be only one!
    // Define the number of colours as well as the initial background
    // image file (.MIF) for the controller.
    vga_adapter VGA(
            .resetn(resetn),
            .clock(CLOCK_50),
            .colour(colour),
            .x(x),
            .y(y),
            .plot(writeEn),
            /* 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;
        defparam VGA.BACKGROUND_IMAGE = "black.mif";

    // Put your code here. Your code should produce signals x,y,colour and writeEn
    // for the VGA controller, in addition to any other functionality your design may
    require.
```

```
69
70      // Control wires
71      wire ld_xy,ld_colour,ld_pos;
72      wire set_black,draw_pixel;
73      wire[7:0] dx,dy;
74      wire[7:0] X_IN,Y_IN;
75      wire[2:0] C_DATA;
76
77      // Assign inputs
78      assign C_DATA   = SW[2:0];
79
80      // Control module
81      control c0(
82          .clock(CLOCK_50),.resetn(resetn),
83          .ld_xy(ld_xy),.ld_colour(ld_colour),.ld_pos(ld_pos),
84          .set_black(set_black),.draw_pixel(draw_pixel),
85          .dx(dx),.dy(dy),.x(X_IN),.y(Y_IN)
86      );
87
88      // Controls plotting on VGA
89      assign writeEn = draw_pixel;
90
91      // Datapath module
92      datapath d0(
93          .clock(CLOCK_50),.resetn(resetn),
94          .X_IN(X_IN),.Y_IN(Y_IN),.COLOUR_DATA(C_DATA),
95          .ld_xy(ld_xy),.ld_colour(ld_colour),
96          .ld_pos(ld_pos),.set_black(set_black),
97          .dx(dx),.dy(dy),
98          .xpos(x),.ypos(y),.colour(colour)
99      );
100 endmodule
101
102 // Tracks state and datapath control signals depending on state
103 module control(
104     input clock,resetn,
105     output reg ld_xy,ld_colour,ld_pos,
106     output reg set_black,draw_pixel,
107     output reg[7:0] dx,dy,x,y // Counts up to X and Y shape size
108     ); // x,y is top left coord of shape
109
110     // Parameters for counters
111     localparam  X_SHAPE     = 8'd4,
112                 Y_SHAPE = 8'd4,
113                 X_SCREEN = 8'd160,
114                 Y_SCREEN = 8'd120;
115
116     // Keeps track of state
117     reg[3:0] current_state,next_state;
118     // Used to draw shapes and clear screen
119     reg reset_dx,reset_dy,inc_dx,inc_dy;
120     reg inc_xy,update_dir;
121
122     reg[1:0] direction; // direction[x,y]: 1 -> right/up, 0 -> left/down
123
124     // Frame counter and control
125     wire[3:0] frame;
126     reg reset_frame,clear_shape,draw,erase;
127
128     // Instantiates frame counter
129     frameCounter fC0(
130         .clock(clock),.resetn(resetn),
131         .resetFrame(reset_frame),.frame(frame)
132     );
133
134     // Assigning state variables
135     localparam  S_FULL_RESET    = 4'd0,
136                 S_DRAW_SHAPE    = 4'd1,
137                 S_DRAW_0        = 4'd2,
```

```
                    S_DRAW_1        = 4'd3,
                    S_DRAW_2        = 4'd4,
                    S_DRAW_3        = 4'd5,
                    S_DRAW_4        = 4'd6,
                    S_RESET_FRAME   = 4'd7,
                    S_FRAME_COUNT   = 4'd8,
                    S_CLEAR_SHAPE   = 4'd9,
                    S_MOVE          = 4'd10,
                    S_UPDATE_DIR    = 4'd11;

    // Register for whether it is drawing or erasing
    always @(posedge clock)
    begin
        if(!resetn) // Active low reset
            clear_shape <= 0;
        else if(draw) // Draw cycle
            clear_shape <= 0;
        else if(erase) // Erase cycle
            clear_shape <= 1;
    end

    // Register for direction
    always @(posedge clock)
    begin
        if(!resetn) // Active low reset
            direction <= 2'b00;
        else if(update_dir)
            begin // Check for bounds
                if((x==X_SCREEN-4 & direction[1]) | (x==0 & !direction[1]))
                    direction[1] <= !direction[1];
                if((y==Y_SCREEN-4 & direction[0]) | (y==0 & !direction[0]))
                    direction[0] <= !direction[0];
            end
    end

    // Counter registers for dx,dy and x,y
    always @(posedge clock)
    begin
        if(!resetn) // Active low reset
        begin
            dx <= 8'b0;
            dy <= 8'b0;
            x <= X_SCREEN-4;
            y <= 8'b0;
        end
        else
        begin
            // dx counter
            if(reset_dx)
                dx <= 8'b0;
            else if(inc_dx)
                dx <= dx+1;

            // dy counter
            if(reset_dy)
                dy <= 8'b0;
            else if(inc_dy)
                dy <= dy+1;

            // x and y counter
            // direction[x,y]: 1 -> right/up, 0 -> left/down
            if(inc_xy)
            begin
                x <= direction[1] ? (x+1):(x-1);
                y <= direction[0] ? (y-1):(y+1);
            end
        end
    end

```

```
207         // State table
208         always @(*)
209         begin
210             case(current_state)
211                 S_DRAW_SHAPE: // Prepares to draw shape (colour<-C, reset dx)
212                     next_state = S_DRAW_0;
213                 S_DRAW_0: // Reset dx
214                     next_state = S_DRAW_1;
215                 S_DRAW_1: // Set pixel fill position
216                     next_state = S_DRAW_2;
217                 S_DRAW_2: // Fill pixel
218                     next_state = S_DRAW_3;
219                 S_DRAW_3: // Increment dx
220                     next_state = (dx==X_SHAPE-1) ? S_DRAW_4:S_DRAW_1;
221                 S_DRAW_4: // Increment dy
222                 begin
223                     if(dy==Y_SHAPE-1) // See if this round drew or erased
224                         next_state = (clear_shape) ? S_MOVE:S_RESET_FRAME;
225                     else next_state = S_DRAW_0;
226                 end
227                 S_RESET_FRAME: // Resets frame count
228                     next_state = S_FRAME_COUNT;
229                 S_FRAME_COUNT: // Counts frame to 0
230                     next_state = (frame==0) ? S_CLEAR_SHAPE:S_FRAME_COUNT;
231                 S_CLEAR_SHAPE: // Prepares to erase shape (colour<-0, reset dx)
232                     next_state = S_DRAW_0;
233                 S_MOVE: // Move top-left corner of shape
234                     next_state = S_UPDATE_DIR;
235                 S_UPDATE_DIR: // Updates direction
236                     next_state = S_DRAW_SHAPE;
237                 default: next_state = S_DRAW_SHAPE;
238             endcase
239         end
240 
241         // Changing data control signals
242         always @(*)
243         begin
244             // Initializing signals to 0 to avoid latches
245             // Internal controls
246             reset_dx = 0; reset_dy = 0; inc_dx = 0; inc_dy = 0;
247             reset_frame = 0; draw = 0; erase = 0;
248             inc_xy = 0; update_dir = 0;
249             // External controls
250             ld_xy = 0; ld_colour = 0; ld_pos = 0;
251             set_black = 0; draw_pixel = 0;
252 
253             case(current_state)
254                 S_DRAW_SHAPE: // Prepares to draw shape (colour<-C, reset dx)
255                 begin
256                     ld_xy = 1;
257                     ld_colour = 1;
258                     draw = 1;
259                     reset_dy = 1;
260                 end
261                 S_DRAW_0: // Reset dx
262                 begin
263                     reset_dx = 1;
264                 end
265                 S_DRAW_1: // Set pixel fill position
266                 begin
267                     ld_pos = 1;
268                 end
269                 S_DRAW_2: // Fill pixel
270                 begin
271                     draw_pixel = 1;
272                 end
273                 S_DRAW_3: // Increment dx
274                 begin
275                     inc_dx = 1;
```

```
276                 end
277                 S_DRAW_4: // Increment dy
278                 begin
279                     inc_dy = 1;
280                 end
281                 S_RESET_FRAME: // Resets frame count
282                 begin
283                     reset_frame = 1;
284                 end
285                 S_CLEAR_SHAPE: // Prepares to erase shape (colour<-0, reset dx)
286                 begin
287                     set_black = 1;
288                     erase = 1;
289                     reset_dy = 1;
290                 end
291                 S_MOVE: // Move top-left corner of shape
292                 begin
293                     inc_xy = 1;
294                 end
295                 S_UPDATE_DIR: // Updates direction
296                 begin
297                     update_dir = 1;
298                 end
299             endcase
300         end
301 
302         // Register for current state
303         always @(posedge clock)
304         begin
305             if(!resetn) // Reset to value input, active low
306                 current_state <= S_DRAW_SHAPE;
307             else // Load next state
308                 current_state <= next_state;
309         end
310     endmodule
311 
312     // Modifies data and outputs depending on control signals
313     module datapath(
314         input clock,resetn,
315         input[7:0] X_IN,Y_IN,
316         input[2:0] COLOUR_DATA,
317         input ld_xy,ld_colour,ld_pos,
318         input set_black,
319         input[7:0] dx,dy,
320         output reg[7:0] xpos,
321         output reg[6:0] ypos,
322         output reg[2:0] colour
323         );
324 
325         // Internal registers
326         reg[7:0] x,y;
327 
328         // Registers x,y,xpos,ypos and colour with input logic
329         always @(posedge clock)
330         begin
331             if(!resetn) // Active low reset
332             begin
333                 x <= 8'b0;
334                 y <= 8'b0;
335                 xpos <= 8'b0;
336                 ypos <= 8'b0;
337                 colour <= 3'b0;
338             end
339             else
340             begin
341                 // x and register
342                 if(ld_xy)
343                 begin
344                     x <= X_IN;
```

```
                y <= Y_IN;
            end
            // xpos and ypos registers (for drawing shape)
            if(ld_pos)
            begin
                xpos <= x+dx;
                ypos <= y+dy;
            end
            // colour register
            if(ld_colour)
                colour <= COLOUR_DATA;
            else if(set_black)
                colour <= 3'b000;
        end
    end
endmodule

module frameCounter(input clock,resetn,resetFrame, output reg[3:0] frame);
    wire enable;

    reg[19:0] rateDivider;
    // Change to 100000 for simulation
    localparam divisor = 100000;

    // Rate divider
    always @(posedge clock)
    begin
        if(!resetn) // Active low reset
            rateDivider <= 833333/divisor-1;
        else if(resetFrame | rateDivider==0) // Reset at command and 1 cycle
            rateDivider <= 833333/divisor-1;
        else // Count down
            rateDivider <= rateDivider-1;
    end

    assign enable = (rateDivider==0);

    // Frame counter
    always @(posedge clock)
    begin
        if(!resetn) // Active low reset
            frame <= 15-1;
        else if(resetFrame | frame==0) // Reset at command and 1 cycle
            frame <= 15-1;
        else // Count down
            frame <= frame-1;
    end
endmodule

```