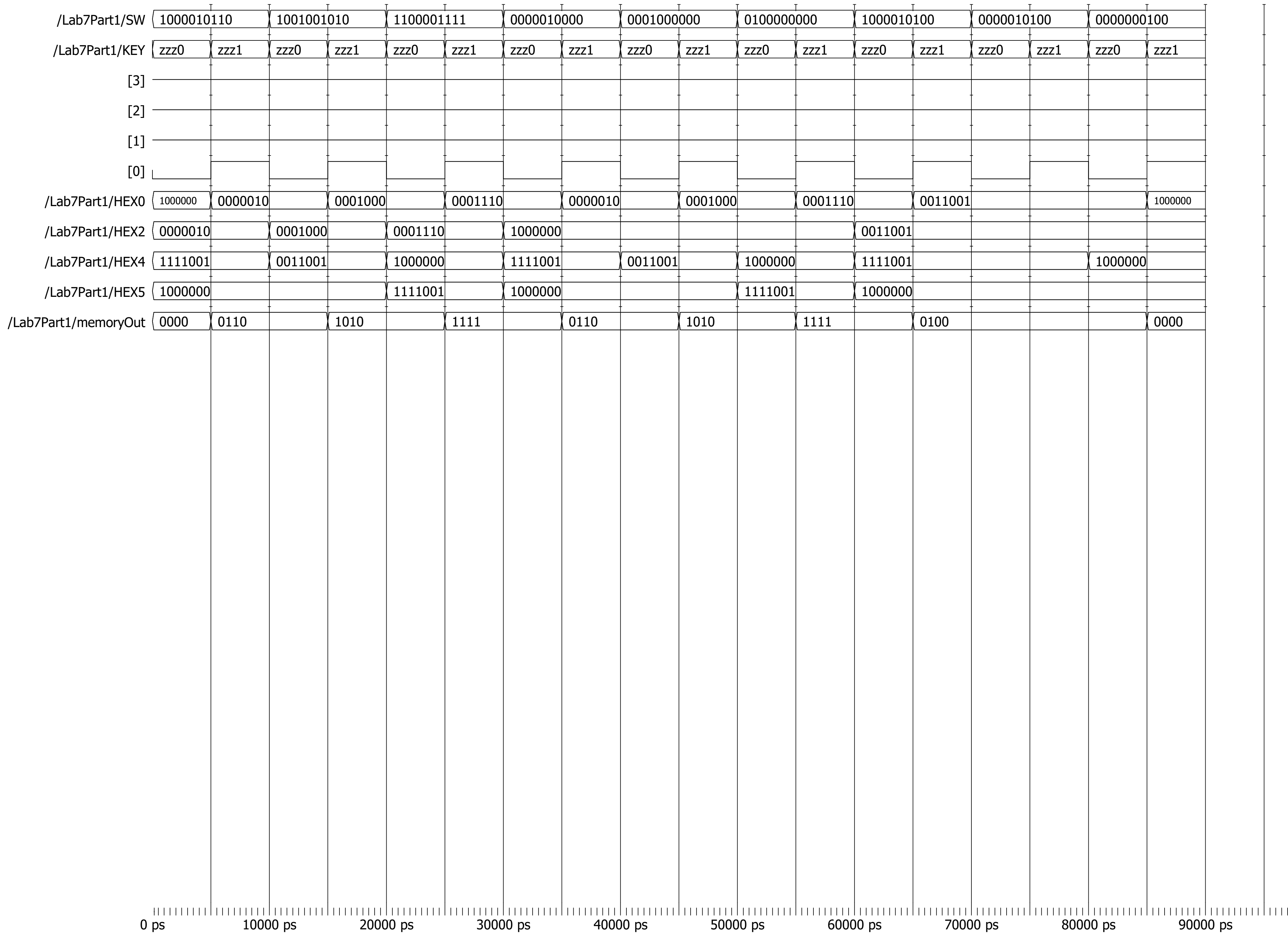
```verilog
`timescale 1ns / 1ns // `timescale time_unit/time_precision

/* top-level entity for 32 x 4 embedded memory */
module embedded_Memory (input [9:0] SW, input [0:0] KEY, output [6:0] HEX0, HEX2, HEX4,
                         HEX5, output [9:0] LEDR);

            assign LEDR[9:0] = 10'd0; // all LEDs on the board should be off

            /*
             * KEY[0] is the clock
             * SW[3:0] is the dataIn
             * HEX2 displays dataIn
             * SW[8:4] is the address
             * HEX4 and HEX5 display address
             * HEX0 displays dataOut
             * SW[9] is writeEnable
            */

            wire clock, writeEn;
            wire [3:0] dataIn, dataOut;
            wire [4:0] address;

            assign clock = KEY[0];
            assign writeEn = SW[9];
            assign dataIn = SW[3:0];
            assign address = SW[8:4];

            hex_decoder D1 (.hex_digit(dataIn), .segments(HEX2));
            hex_decoder D2 (.hex_digit(dataOut), .segments(HEX0));
            hex_decoder D3 (.hex_digit(address[3:0]), .segments(HEX4));
            hex_decoder D4 (.hex_digit({3'd0, address[4]}), .segments(HEX5));

            ram32x4 M1 (.address(address), .clock(clock), .data(dataIn), .wren(writeEn), .q(
dataOut));

endmodule // embedded_Memory

/* bcd to hex for seven-segment display */
module hex_decoder (input [3:0] hex_digit, output reg [6:0] segments);

            always @(*)
            begin

               case (hex_digit)
                  4'h0: segments = 7'b100_0000;
                  4'h1: segments = 7'b111_1001;
                  4'h2: segments = 7'b010_0100;
                  4'h3: segments = 7'b011_0000;
                  4'h4: segments = 7'b001_1001;
                  4'h5: segments = 7'b001_0010;
                  4'h6: segments = 7'b000_0010;
                  4'h7: segments = 7'b111_1000;
                  4'h8: segments = 7'b000_0000;
                  4'h9: segments = 7'b001_1000;
                  4'hA: segments = 7'b000_1000;
                  4'hB: segments = 7'b000_0011;
                  4'hC: segments = 7'b100_0110;
                  4'hD: segments = 7'b010_0001;
                  4'hE: segments = 7'b000_0110;
                  4'hF: segments = 7'b000_1110;
                  default: segments = 7'h7f;
               endcase

            end

endmodule // hex_decoder
```



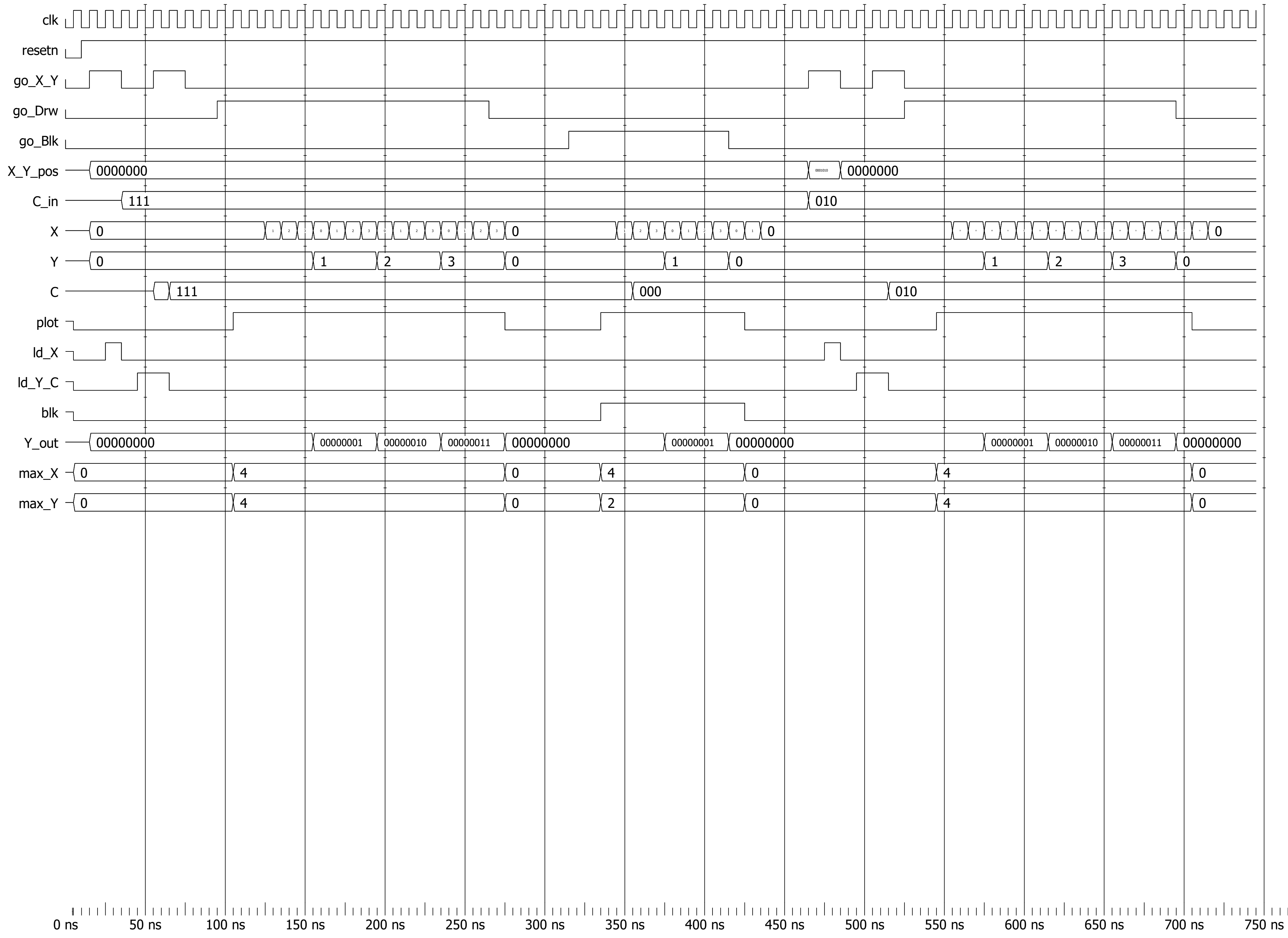
Entity:Lab7Part1 Architecture: Date: Sat Nov 03 17:40:36 EDT 2018 Row: 1 Page: 1

```verilog
module fill
	(
		CLOCK_50,						// On Board 50 MHz
		SW,
		KEY,							// On Board Keys
		// 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;
	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];

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

	wire [2:0] colour;
	wire [7:0] x;
	wire [6:0] y;
	wire writeEn;

	// Defining 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";

	// Outputs x,y,colour and writeEn for the VGA controller
	drawRectangle DR1 (.clk(CLOCK_50), .resetn(resetn), .go_X_Y(~KEY[3]), .go_Drw(~KEY[1]),
						.go_Blk(~KEY[2]), .X_Y_pos(SW[6:0]), .C_in(SW[9:7]), .X(x), .Y(y),
						.C(colour), .plot(writeEn));

endmodule // fill
```

```
`timescale 1ns / 1ns // `timescale time_unit/time_precision


/* module that handles the action of drawing rectangles */
module drawRectangle (input clk, resetn, go_X_Y, go_Drw, go_Blk, input [6:0] X_Y_pos, input
                     [2:0] C_in, output [7:0] X, output [6:0] Y, output [2:0] C, output plot
);

            wire ld_X, ld_Y_C, blk;
            wire [7:0] Y_out, max_X, max_Y;

            assign Y = Y_out[6:0];

            control C1 (.clk(clk), .resetn(resetn), .go_X_Y(go_X_Y), .go_Drw(go_Drw), .
go_Blk(go_Blk),
                        .ld_X(ld_X), .ld_Y_C(ld_Y_C), .plot(plot), .blk(blk), .ii(max_X), .
jj(max_Y));

            datapath D1 (.clk(clk), .resetn(resetn), .ld_X(ld_X), .ld_Y_C(ld_Y_C), .blk(blk
), .ii(max_X),
                        .jj(max_Y), .X_Y_in(X_Y_pos), .C_in(C_in), .X_r(X), .Y_r(Y_out), .
C_r(C),
                        .plot(plot));

endmodule // drawRectangle


/* module for controlling registers and state transitons */
module control (input clk, resetn, go_X_Y, go_Drw, go_Blk, output reg ld_X, ld_Y_C, plot,
               blk, output reg [7:0] ii, jj);

            reg [3:0] current_state, next_state;
            // state_FF assignments
            localparam  KEY_WAIT       = 4'd0,
                        LOAD_X         = 4'd1,
                        LOAD_X_WAIT    = 4'd2,
                        LOAD_Y_C       = 4'd3,
                        LOAD_Y_C_WAIT  = 4'd4,
                        PRINT_BLACK    = 4'd5,
                        BLACK_WAIT     = 4'd6,
                        DRW_SHP        = 4'd7,
                        DRW_SHP_WAIT   = 4'd8;
            // shape and window sizes
            localparam  SHAPE_SZ_X     = 8'd4,
                        SHAPE_SZ_Y     = 8'd4,
                        WINDOW_SZ_X    = 8'd160,
                        WINDOW_SZ_Y    = 8'd120;

            // state table
            always @(*)
            begin

               case (current_state)
                  // remain at KEY_WAIT state until a valid key is pressed
                  KEY_WAIT:   begin
                                 // if only go_Blk is pressed
                                 if ((go_Blk == 1'b1) & (go_X_Y == 1'b0))
                                    next_state = PRINT_BLACK;
                                 // if only go_X is pressed
                                 else if ((go_Blk == 1'b0) & (go_X_Y == 1'b1))
                                    next_state = LOAD_X;
                                 // if another invalid combination is pressed
                                 else
                                    next_state = KEY_WAIT;
                              end
                  // load x position
                  LOAD_X:     begin
                                 // if only go_X is pressed
                                 if ((go_X_Y == 1'b1) & (go_Blk == 1'b0))
                                    next_state = LOAD_X_WAIT;
                                 // if go_Blk is pressed
                                 else if (go_Blk == 1'b1)
                                    next_state = PRINT_BLACK;
                                 else
                                    next_state = LOAD_X;
                              end
                  // wait until go_X goes low
                  LOAD_X_WAIT:   next_state = go_X_Y ? LOAD_X_WAIT : LOAD_Y_C;
                  // load y position
                  LOAD_Y_C:   begin
                                 // if only go_Y is pressed
                                 if ((go_X_Y == 1'b1) & (go_Blk == 1'b0))
                                    next_state = LOAD_Y_C_WAIT;
                                 // if go_Blk is pressed
                                 else if (go_Blk == 1'b1)
                                    next_state = PRINT_BLACK;
                                 else
                                    next_state = LOAD_Y_C;
                              end
                  // wait until go_Y goes low
                  LOAD_Y_C_WAIT: next_state = go_X_Y ? LOAD_Y_C_WAIT : DRW_SHP;
                  // draw to frame buffer
                  DRW_SHP:    begin
                                 // dont goto draw state w8 unless work is done
                                 // if only go_Drw is pressed
                                 if ((go_Drw == 1'b1) & (go_Blk == 1'b0))
                                    next_state = DRW_SHP_WAIT;
                                 // if go_Blk is pressed
                                 else if (go_Blk == 1'b1)
                                    next_state = PRINT_BLACK;
                                 else
                                    next_state = DRW_SHP;
                              end
                  // start over after drawing
                  DRW_SHP_WAIT: next_state = go_Drw ? DRW_SHP_WAIT : KEY_WAIT;
                  // load black for all pixels
                  PRINT_BLACK:   next_state = go_Blk ? BLACK_WAIT : PRINT_BLACK;
                  // wait untile go_Blk goes low
                  BLACK_WAIT:    next_state = go_Blk ? BLACK_WAIT : KEY_WAIT;
                  // stay at KEY_WAIT by default
                  default:       next_state = KEY_WAIT;
               endcase

            end // state_table

            // output logic for datapath control signals
            always @(*)
            begin

               // all signals are 0 by default, to avoid latches
               ld_X   = 1'b0;
               ld_Y_C= 1'b0;
               blk    = 1'b0;
               plot   = 1'b0;
               ii     = 8'd0;
               jj     = 8'd0;

               case (current_state)
                  LOAD_X:     begin
                                 ld_X = 1'b1; // enable load for register X
                              end
                  LOAD_Y_C:   begin
                                 ld_Y_C = 1'b1; // enable load for register Y and C
                              end
                  DRW_SHP_WAIT:begin // draw square
                                 // enable plot to store values into frame buffer
                                 plot = 1'b1;
                                 // max cycles for traversing along row (x) axis
                                 ii = SHAPE_SZ_X;
                                 // max cycles for traversing along column (y) axis
                                 jj = SHAPE_SZ_Y;
                              end
                  BLACK_WAIT: begin
                                 // load black for all pixels in the window
                                 blk = 1'b1;
                                 // enable plot to store values into frame buffer
                                 plot = 1'b1;
                                 // max cycles for traversing along row (x) axis
                                 ii = WINDOW_SZ_X;
                                 // max cycles for traversing along column (y) axis
                                 jj = WINDOW_SZ_Y;
                              end
```

```
                endcase // no default needed; all of our outputs were assigned a value

            end // enable_signals

            // current_state registers
            always @(posedge clk)
            begin
                if(!resetn)
                    current_state <= KEY_WAIT;
                else
                    current_state <= next_state;
            end // state_FFs transition

endmodule // control


/* datapath module */
module datapath (input clk, resetn, ld_X, ld_Y_C, blk, plot, input [7:0] ii, jj, input [6:0]
                X_Y_in, input [2:0] C_in, output reg [7:0] X_r, Y_r, output reg [2:0] C_r);

            // input registers
            reg [7:0] X, Y, cnt_ii, cnt_jj;
            reg [2:0] C;

            // registers X, Y, C with respective input logic
            always @(posedge clk)
            begin
              if(!resetn)
              begin
                    X      <= 8'd0;
                    Y      <= 8'd0;
                    cnt_ii<= 8'd0;
                    cnt_jj<= 8'd0;
                    C      <= 3'd0;
                    X_r    <= X;
                    Y_r    <= Y;
                    C_r    <= C;
              end

              else
              begin
                    // load X and X_r
                    if (ld_X == 1'b1)
                    begin
                        X     <= {1'b0, X_Y_in};
                        X_r   <= X;
                    end
                    // load Y, C, Y_r, C_r
                    if (ld_Y_C == 1'b1)
                    begin
                        Y     <= {1'b0, X_Y_in};
                        C     <= C_in;
                        Y_r   <= Y;
                        C_r   <= C;
                    end
                    //set (X,Y) to (0,0) and load C with black
                    if (blk == 1'b1)
                    begin
                       X      <= 8'd0;
                       Y      <= 8'd0;
                       C      <= 3'd0;
                       X_r    <= X;
                       Y_r    <= Y;
                       C_r    <= C;
                    end
                    // increment X and Y by ii and jj respectively
                    if ((cnt_ii == (ii - 1'b1)) & (ii > 8'd0) & (plot == 1'b1))
                    begin
                       cnt_ii <= 8'd0;
                       cnt_jj <= cnt_jj + 1'b1;
                    end

                    else if ((ii > 8'd0) & (plot == 1'b1))
                    begin
                       cnt_ii <= cnt_ii + 1'b1;
                    end
```

```verilog

                  if ((cnt_jj == (jj - 1'b1)) & (cnt_ii == (ii - 1'b1))
                     & (jj > 8'd0) & (plot == 1'b1))
                  begin
                     cnt_jj <= 8'd0;
                  end

                  if (plot == 1'b1)
                  begin
                     X_r <= X + cnt_ii;
                     Y_r <= Y + cnt_jj;
                  end
                  else
                  begin
                     X_r <= 8'd0;
                     Y_r <= 8'd0;
                  end
               end
            end

endmodule // datapath
```



Entity:drawRectangle Architecture: Date: Mon Nov 05 07:33:28 EST 2018   Row: 1 Page: 1