Lab Code [10 points]
Filename: chipInterface.sv
AndrewID: kakinci

```
`default_nettype none


// module that represents our striped color for our VGA module
module VGA_colorOutput(
    input logic [9:0] col,
    input logic [8:0] row,
    output logic [7:0] R,G,B);

    always_comb begin
        if (row >= 9'd240) begin
                R = 9'h00;
                G = 9'h00;
                B = 9'h00;
        end
        else begin
            if(col >= 10'd0 && col <= 10'd79) begin
                R = 8'h00;
                G = 8'h00;
                B = 8'h00;
            end
            else if(col >= 10'd80 && col <= 10'd159) begin
                R = 8'h00;
                G = 8'h00;
                B = 8'hFF;
            end
            else if(col >= 10'd160 && col <= 10'd239) begin
                R = 8'h00;
                G = 8'hFF;
                B = 8'h00;
            end
            else if(col >= 10'd240 && col <= 10'd319) begin
                R = 8'h00;
                G = 8'hFF;
                B = 8'hFF;
            end
            else if(col >= 10'd320 && col <= 10'd399) begin
                R = 8'hFF;
                G = 8'h00;
                B = 8'h00;
            end
            else if(col >= 10'd400 && col <= 10'd479) begin
                R = 8'hFF;
                G = 8'h00;
                B = 8'hFF;
            end
            else if(col >= 10'd480 && col <= 10'd559) begin
                R = 8'hFF;
                G = 8'hFF;
                B = 8'h00;
            end
            else if(col >= 10'd560 && col <= 10'd639) begin
                R = 8'hFF;
                G = 8'hFF;
                B = 8'hFF;
            end
      else begin
Line contains tabs (each tab replaced by 2 spaces in this print)
        R = 8'h00;
Line contains tabs (each tab replaced by 2 spaces in this print)
        G = 8'hFF;
Line contains tabs (each tab replaced by 2 spaces in this print)
        B = 8'h00;
Line contains tabs (each tab replaced by 2 spaces in this print)
        end
Line contains tabs (each tab replaced by 2 spaces in this print)
      end
Line contains tabs (each tab replaced by 2 spaces in this print)
    end
```

```
endmodule


// represents our color output for our Pong game
module pong_colorOutput (
    input logic [9:0] col,
    input logic [8:0] row,
    input logic validPaddleLeft,
    input logic validPaddleRight,
    input logic validBall,
    input logic right_scored,
    input logic left_scored,
    output logic [7:0] R,G,B
    );
    always_comb begin
        if (validPaddleLeft) begin
            // represents the left paddle
            R = 8'd0;
            G = 8'd255;
            B = 8'd0;
        end
      else if (validPaddleRight) begin
Line contains tabs (each tab replaced by 2 spaces in this print)
            // represents the left paddle
            R = 8'd0;
            G = 8'd255;
            B = 8'd100;
        end
        else if (validBall) begin
            // represents the left paddle
            R = 8'd255;
            G = 8'd255;
            B = 8'd255;
        end
        else if (right_scored) begin
            R = 8'd200;
            G = 8'd200;
            B = 8'd100;
        end
        else if (left_scored) begin
            R = 8'd100;
            G = 8'd0;
            B = 8'd0;
        end
        else begin
            R = 8'd0;
            G = 8'd0;
            B = 8'd0;
    end
    end
endmodule: pong_colorOutput


//  ChipInterface for our VGA and pong module
module chipinterface
    (input logic CLOCK_50,
    input logic [3:0] KEY,
    input logic [17:0] SW,
    output logic [6:0] HEX0, HEX1, HEX2, HEX3,
    HEX4, HEX5, HEX6, HEX7,
    output logic [7:0] VGA_R, VGA_G, VGA_B,
    output logic VGA_BLANK_N, VGA_CLK, VGA_SYNC_N,
    output logic VGA_VS, VGA_HS);

    logic [8:0] row;
    logic [9:0] col;
    logic BLANK0;
    assign VGA_BLANK_N = ~BLANK0;
  logic reset, serve;
Line contains tabs (each tab replaced by 2 spaces in this print)
  assign reset = ~KEY[0];
Line contains tabs (each tab replaced by 2 spaces in this print)
```

```
    assign serve = ~KEY[3];

    // outputs of the paddles
    logic validPaddleLeft;
    logic validPaddleRight;

    // outputs of the ball
    logic validBall;

    // booleans to represent score by either side
    logic right_scored, left_scored;

    vga    v1(.CLOCK_50(CLOCK_50), .reset(reset), .HS(VGA_HS), .VS(VGA_VS),
                                .blank(BLANK0), .row(row), .col(col));
    // Lab 4 week 1:
    // VGA_colorOutput f2(.col(col),.row(row),.R(VGA_R),.G(VGA_G),.B(VGA_B));


    pong_colorOutput f3(.col(col), .row(row), .validPaddleLeft(validPaddleLeft),
                    .validPaddleRight(validPaddleRight), .validBall(validBall),
                    .right_scored(right_scored), .left_scored(left_scored),
                             .R(VGA_R), .G(VGA_G), .B(VGA_B));

    assign VGA_SYNC_N = 1'b0;

    assign VGA_CLK = ~CLOCK_50;

    logic [8:0] leftPaddleRow;
    logic [8:0] rightPaddleRow;

    logic [3:0] right_score, left_score; // right and left paddle scores



    left_paddle lp(.row(row), .col(col), .do_move(SW[17]), .direction(SW[16]),
            .CLOCK_50(CLOCK_50), .reset(reset), .valid_paddle(validPaddleLeft),
            .left_paddle_row(leftPaddleRow));
    right_paddle rp(.row(row), .col(col), .do_move(SW[1]), .direction(SW[0]),
            .CLOCK_50(CLOCK_50), .reset(reset), .valid_paddle(validPaddleRight),
            .right_paddle_row(rightPaddleRow));



    ball b1(.row(row), .col(col), .clock(CLOCK_50), .reset(reset),
                .serve(serve),
                .paddle_left_row(leftPaddleRow),
                .paddle_right_row(rightPaddleRow),
                .valid_ball(validBall),.right_scored(right_scored),
                .left_scored(left_scored),
                .right_score(right_score),
                .left_score(left_score));


    BCDtoSevenSegment chex0(.bcd_digit(right_score), .segment(HEX0));
    BCDtoSevenSegment chex7(.bcd_digit(left_score), .segment(HEX7));

    BCDtoSevenSegment chex1(.bcd_digit(4'd0), .segment(HEX1));
    BCDtoSevenSegment chex2(.bcd_digit(4'd0), .segment(HEX2));
    BCDtoSevenSegment chex3(.bcd_digit(4'd0), .segment(HEX3));
    BCDtoSevenSegment chex4(.bcd_digit(4'd0), .segment(HEX4));
    BCDtoSevenSegment chex5(.bcd_digit(4'd0), .segment(HEX5));
    BCDtoSevenSegment chex6(.bcd_digit(4'd0), .segment(HEX6));


endmodule: chipinterface
```

```

// This module converts a 4-bit BCD number to a 7-segment bit output.
module BCDtoSevenSegment
(
    input logic [3:0] bcd_digit,
    output logic [6:0] segment
);

always_comb begin
    case (bcd_digit)
        4'd0: segment = 7'b111_1111;
        4'd1: segment = 7'b111_1001;
        4'd2: segment = 7'b010_0100;
        4'd3: segment = 7'b011_0000;
        4'd4: segment = 7'b001_1001;
        4'd5: segment = 7'b001_0010;
        4'd6: segment = 7'b000_0010;
        4'd7: segment = 7'b111_1000;
        4'd8: segment = 7'b000_0000;
        4'd9: segment = 7'b001_0000;
        default: segment = 7'b111_1111;
    endcase
end

endmodule: BCDtoSevenSegment
```

Lab Code [10 points]
Filename: library.sv
AndrewID: kakinci

```
`default_nettype none


// range check module
module RangeCheck
    #(parameter WIDTH = 6)
    (input logic [WIDTH-1:0] val,
    input logic [WIDTH-1:0] low,
    input logic [WIDTH-1:0] high,
    output logic is_between );

    assign is_between = (val >= low) && (val <= high);
endmodule: RangeCheck




// Typical offset check module
module OffsetCheck
#(parameter WIDTH = 6)
(input logic [WIDTH-1:0] delta,
input logic [WIDTH-1:0] low,
input logic [WIDTH-1:0] val,
output logic is_between);
    assign is_between = (val >= low) && (val <= low+delta);

endmodule



// represents our module for counting the score
module Score_Counter
  #(parameter WIDTH=8)
  (input  logic [WIDTH-1:0] D,
   input  logic             en, clear, load, clock, up,
   output logic [WIDTH-1:0] Q);

   always_ff @(posedge clock)
    if (clear)
      Q <= {WIDTH {1'b0}};
    else if (load)
      Q <= D;
    else if (en)
      if (up)
        Q <= Q + 1'b1;
      else
        Q <= Q;

endmodule : Score_Counter

// A binary up-down counter.
// Clear has priority over Load, which has priority over Enable
module Counter
  #(parameter WIDTH=8)
  (input  logic [WIDTH-1:0] D,
   input  logic             en, clear, load, clock, up,
   output logic [WIDTH-1:0] Q);

  always_ff @(posedge clock)
    if (clear)
      Q <= {WIDTH {1'b0}};
    else if (load)
      Q <= D;
    else if (en)
      if (up)
        Q <= Q + 1'b1;
      else
        Q <= Q - 1'b1;

```

```
endmodule : Counter

// A binary up-down counter.
// Clear has priority over Load, which has priority over Enable
module Counter_Row
  #(parameter WIDTH=8)
  (input  logic [WIDTH-1:0] D,
   input  logic             en, clear, load, clock, up,
   output logic [WIDTH-1:0] Q);

  always_ff @(posedge clock)
    if (clear)
      Q <= {WIDTH {1'b0}};
    else if (load)
      Q <= D;
    else if (en)
      if (up)
        Q <= Q + 1'b1;
      else
        Q <= Q - 1'b1;

endmodule : Counter_Row


// A binary up-down counter.
// Clear has priority over Load, which has priority over Enable
module Counter_Col
  #(parameter WIDTH=8)
  (input  logic [WIDTH-1:0] D,
   input  logic             en, clear, load, clock, up,
   output logic [WIDTH-1:0] Q);

  always_ff @(posedge clock)
    if (clear)
      Q <= {WIDTH {1'b0}};
    else if (load)
      Q <= D;
    else if (en)
      if (up)
        Q <= Q + 2'd2;
      else
        Q <= Q - 2'd2;

endmodule : Counter_Col

// A Magnitude Comparator does an unsigned comparison of two input values.
module MagComp
  #(parameter   WIDTH = 8)
  (output logic             AltB, AeqB, AgtB,
   input  logic [WIDTH-1:0] A, B);

  assign AeqB = (A == B);
  assign AltB = (A <  B);
  assign AgtB = (A >  B);

endmodule: MagComp


// 2 to 1 multiplexer module
module Mux2to1
  #(parameter WIDTH = 8)
  (input  logic [WIDTH-1:0] I0, I1,
   input  logic             S,
   output logic [WIDTH-1:0] Y);

  assign Y = (S) ? I1 : I0;

endmodule : Mux2to1


// Register module
```

```
module Register
  #(parameter WIDTH=8)
  (input  logic [WIDTH-1:0] D,
   input  logic             en, clear, clock,
   output logic [WIDTH-1:0] Q);

  always_ff @(posedge clock)
    if (en)
      Q <= D;
    else if (clear)
      Q <= '0;

endmodule : Register


```

Lab Code [10 points]
Filename: library_tests.sv
AndrewID: kakinci

```
`default_nettype none




// module RangeCheck_test #(parameter WIDTH = 8)();


//     logic [WIDTH-1:0] low;
//     logic [WIDTH-1:0] high;
//     logic [WIDTH-1:0] val;
//     logic is_between;
//     RangeCheck #(WIDTH) rc (.*);
//     initial begin
//         $monitor($time,, "low= %d, high = %d, val= %d, in_between = %d", l...
Line length of 81 (max is 80)
//                                                  high, val, is_between);
//             low = 0;
//             high = 2;
//             val = 3;
//         #10 low= 0;
//          high = 4;
//         #10 low = 123;
//              high = 0;
//              val = 12;
//         #10 low = 12;
//             high = 20;
//             val = 1;


//         #10 $finish;
//     end
// endmodule: RangeCheck_test





module OffsetCheck_test #(parameter WIDTH = 8)();
    logic [WIDTH-1:0] low;
    logic [WIDTH-1:0] delta;
    logic [WIDTH-1:0] val;
    logic is_between;
    OffsetCheck #(WIDTH) rc (.*);
    initial begin
        $monitor($time,, "low= %d, delta = %d, val= %d, in_between = %d", low,
                                              delta, val, is_between);
            low = 0;
            delta = 2;
            val = 3;
        #10 low= 0;
         delta = 4;
        #10 low = 123;
             delta = 0;
             val = 12;
        #10 low = 12;
            delta = 20;
            val = 1;
        #10 low = 12;
            delta = 2;
            val = 13;
        #10 $finish;
    end
endmodule: OffsetCheck_test
```

Lab Code [10 points]
Filename: pong.sv
AndrewID: kakinci

```
`default_nettype none


// Module to handle the movement of our left paddle
module left_paddle
    (input logic [8:0] row,
    input logic [9:0] col,
    input logic do_move,
    input logic direction,
    input logic CLOCK_50,
    input logic reset,
    output logic valid_paddle,
    output logic [8:0] left_paddle_row);

    logic valid_col, valid_row; // represents if row and col are valid
    logic reachedtop, reachedbottom; // represents if row
        //has reached the top or bottom of the paddle
    logic [8:0] row_high; // represents where the current
    logic [8:0] row_high_add, row_high_sub; // represents the output of the
                                            // two mux2to1's
    //whether row_high or row_high+5 or row_high-5 is chosen
    logic [8:0] row_reset_value;
    logic [8:0] row_direction; //out of mux2to1 whether we move up or down

    logic [8:0] row_high_plus5, row_high_minus5;

    logic register_enable, refresh;

    RangeCheck #(10) inCol(.val(col), .low(10'd60), .high(10'd63),
                                                    .is_between(valid_col));
    OffsetCheck #(9) inRow(.delta(9'd48), .low(row_high), .val(row),
                                                    .is_between(valid_row));
    assign valid_paddle = valid_col & valid_row;


    // Check to see if the high row of our paddle has reached a bounday
    MagComp #(9) reach_top(.AltB(), .AeqB(reachedtop), .AgtB(), .A(row_high),
                                                            .B(9));
    MagComp #(9) reach_bottom(.AltB(), .AeqB(), .AgtB(reachedbottom),
                                        .A(row_high), .B(9'd425));
    logic real_refresh = 0;
    assign refresh = (col == 10'd639) & (row == 9'd479) ;
    assign register_enable = do_move & (refresh); // enables every refresh and
                                        // when the player is moving
    Register #(9) high_row_value(.D(row_reset_value), .en(register_enable),
                                .clear(1'd0), .clock(CLOCK_50),
                                            .Q(row_high));

    assign row_high_plus5 = row_high + 5;
    assign row_high_minus5 = row_high - 5;


    Mux2to1 #(9) choose_too_big (.I0(row_high_plus5), .I1(row_high),
                                        .S(reachedbottom),
                                                .Y(row_high_add));
    Mux2to1 #(9) choose_too_small (.I0(row_high_minus5), .I1(row_high),
                                    .S(reachedtop),.Y(row_high_sub));
    Mux2to1 #(9) choose_direction (.I0(row_high_add), .I1(row_high_sub),
                                    .S(direction), .Y(row_direction));
    Mux2to1 #(9) choose_reset (.I0(row_direction), .I1(9'd192),
                                    .S(reset),.Y(row_reset_value));

    assign left_paddle_row = row_high;

endmodule: left_paddle


// module to handle the movement of our right paddle
module right_paddle
```

```
    (input logic [8:0] row,
    input logic [9:0] col,
    input logic do_move,
    input logic direction,
    input logic CLOCK_50,
    input logic reset,
    output logic valid_paddle,
    output logic [8:0] right_paddle_row);

    logic valid_col, valid_row; // represents if row and col are valid
    logic reachedtop, reachedbottom; // represents if row
        //has reached the top or bottom of the paddle
    logic [8:0] row_high; // represents where the current
    logic [8:0] row_high_add, row_high_sub; // represents the output of the
                                                          // two mux2to1's
    //whether row_high or row_high+5 or row_high-5 is chosen
    logic [8:0] row_reset_value;
    logic [8:0] row_direction; //out of mux2to1 whether we move up or down

    logic [8:0] row_high_plus5, row_high_minus5;

    logic register_enable, refresh;

    RangeCheck #(10) inCol(.val(col), .low(10'd577), .high(10'd580),
                                                   .is_between(valid_col));
    OffsetCheck #(9) inRow(.delta(9'd48), .low(row_high), .val(row),
                                                   .is_between(valid_row));
    assign valid_paddle = valid_col & valid_row;


    // Check to see if the high row of our paddle has reached a bounday
    MagComp #(9) reach_top(.AltB(), .AeqB(reachedtop), .AgtB(), .A(row_high),
                                                           .B(9'd0));
    MagComp #(9) reach_bottom(.AltB(), .AeqB(), .AgtB(reachedbottom),
                                           .A(row_high), .B(9'd432));

    assign refresh = (col == 10'd639) & (row == 9'd479);
    assign register_enable = do_move & (refresh); // enables every refresh and
                                           // when the player is moving
    Register #(9) high_row_value(.D(row_reset_value), .en(register_enable),
                                 .clear(1'd0), .clock(CLOCK_50),
                                                           .Q(row_high));

    assign row_high_plus5 = row_high + 5;
    assign row_high_minus5 = row_high - 5;


    Mux2to1 #(9) choose_too_big (.I0(row_high_plus5), .I1(row_high),
                                           .S(reachedbottom),
                                                   .Y(row_high_add));
    Mux2to1 #(9) choose_too_small (.I0(row_high_minus5), .I1(row_high),
                                       .S(reachedtop),.Y(row_high_sub));
    Mux2to1 #(9) choose_direction (.I0(row_high_add), .I1(row_high_sub),
                                        .S(direction), .Y(row_direction));
    Mux2to1 #(9) choose_reset (.I0(row_direction), .I1(9'd192),
                                       .S(reset),.Y(row_reset_value));
    assign right_paddle_row = row_high;

endmodule: right_paddle


// Module that represents the movement of our ball

module ball
    (input logic [8:0] row,
    input logic [9:0] col,
    input logic clock,
    input logic reset,
    input logic serve,
    input logic [8:0] paddle_left_row,
    input logic [8:0] paddle_right_row,
    output logic valid_ball,
    output logic right_scored,
    output logic left_scored,
    output logic [3:0] right_score,
    output logic [3:0] left_score);

    logic [9:0] ball_col;
    logic [8:0] ball_row;
    logic in_ball_col, in_ball_row;

    logic hvelocity;
    logic vvelocity;
    logic enable;

    logic refresh, served;


    logic score_left, score_right;

    logic top, bottom; //true if the ball is on the top or bottom of screen
    logic dVV, dHV;


    // score for left and right module
    always_comb begin

        if (ball_col < 10'd5) begin
            score_right = 1;
            score_left = 0;
        end
        else if (ball_col > 10'd635) begin
            score_left = 1;
            score_right = 0;
        end
        else begin
            score_left = 0;
            score_right = 0;
        end
    end

    ballFsm d(.serve(serve), .reset(reset), .clock(clock),
                    .score_left(score_left), .score_right(score_right),
                    .served(served), .right_scored(right_scored),
                    .left_scored(left_scored));

    // final answer returns
    OffsetCheck #(10) ball_col_check (.low(ball_col),.delta(10'd4),.val(col),
         .is_between(in_ball_col));
    OffsetCheck #(9) ball_row_check (.low(ball_row),.delta(9'd4),.val(row),
         .is_between(in_ball_row));
    assign valid_ball = in_ball_col & in_ball_row;

    // Counters for score

    Score_Counter #(4) left_scoreC(.D(4'd0), .en(1'd1), .clear(1'd0),
         .load(reset), .clock(clock), .up(score_left), .Q(left_score));
    Score_Counter #(4) right_scoreC(.D(4'd0), .en(1'd1), .clear(1'd0),
         .load(reset), .clock(clock), .up(score_right), .Q(right_score));


    assign refresh = (col == 10'd639) & (row == 9'd479);
    // boundary checks

    MagComp #(9) compare_row_top (.A(ball_row),.B(9'd2),.AeqB(),.AgtB(),
                                                           .AltB(top));
    MagComp #(9) compare_row_bottom (.A(ball_row),.B(9'd477),.AeqB(),
                                                  .AgtB(bottom),.AltB());

    // // Left Paddle Hit
    logic left_paddle_hit, left1, left2;
    OffsetCheck #(9) compareRowLeft(.val(ball_row), .low(paddle_left_row),
                                  .delta(9'd48),  .is_between(left1));
    RangeCheck #(10) compareColLeft(.val(ball_col), .low(10'd60), .high(10'd63),
                                                    .is_between(left2));
    assign left_paddle_hit = left1 & left2;




    // // Right Paddle Hit
    logic right1, right2, right_paddle_hit;
    OffsetCheck #(9) compareRowRight(.val(ball_row), .low(paddle_right_row),
                            .delta(9'd48), .is_between(right1));
    RangeCheck #(10) compareColRight(.val(ball_col), .low(10'd570), .high(10'...
Line length of 83 (max is 80)
                                .is_between(right2));
    assign right_paddle_hit = right1 & right2;

    // Row and column of the ball
    logic load;
    assign enable = (refresh & served);
    assign load = reset | score_left | score_right;
    Counter_Row #(9) CounterRow(.D(9'd220), .en(enable), .clear(1'd0),
                                                    .load(load), .clock(clock),
                            .up(~vvelocity), .Q(ball_row));
    Counter_Col #(10) CounterCol(.D(10'd330), .en(enable), .clear(1'd0),
                                                .load(load), .clock(clock),
                                             .up(~hvelocity), .Q(ball_col));

    // handles movement of horiztonal velocity
    logic nextdVV, nextdHV;
    always_comb begin
        nextdVV = 1;
        if (top) begin
            nextdVV = 0;
        end
        else if (bottom) begin
            nextdVV = 1;
        end
        else begin
            nextdVV = vvelocity;
        end
    end

    // handles movement for horiztonal velocity

    always_comb begin
        nextdHV = 1;
        if (right_paddle_hit) begin
            nextdHV = 1;
        end
        else if (left_paddle_hit) begin
            nextdHV = 0;
        end
        else begin
            nextdHV = hvelocity;
        end
    end

    // Registers for the velocity

    Register #(1) VerticalVelocity(.D(nextdVV), .en(enable), .clear(~served),
                                                  .clock(clock), .Q(vvelocity));
    logic result12,result13; // mid results for our horizontal velocity

    Mux2to1 #(1) function21(.I0(nextdHV), .I1(1'b0), .S(score_left),
                                                                .Y(result12));
    Mux2to1 #(1) function23(.I0(result12), .I1(1'b1), .S(score_right),
                                                                 .Y(result13));
    Register #(1) HorizontalVelocity(.D(result13), .en(enable), .clear(0),
                                               .clock(clock), .Q(hvelocity));

```

```
endmodule: ball


// represents the finite state machine of our ball maovement
module ballFsm
    (input logic serve, reset, clock, score_left, score_right,
    output logic served, right_scored, left_scored);

    enum logic [1:0] {INIT, SERVED, RIGHT_SCORED, LEFT_SCORED} currState,
                                                                nextState;

    always_comb begin
        case(currState)
            // inital state
            INIT: begin
                if(serve)begin
                    served = 1'd1;
                    nextState = SERVED;
                    left_scored = 1'b0;
                    right_scored = 1'b0;
                end else begin
                    served = 1'd0;
                    nextState = INIT;
                    left_scored = 1'b0;
                    right_scored = 1'b0;
                end
            end
            // served state
            SERVED: begin
                served = 1'd1;
                nextState = SERVED;
                if (score_right) begin
                    nextState = RIGHT_SCORED;
                    served = 1'b0;
                    right_scored = 1'b1;
                    left_scored = 1'b0;
                end
                else if (score_left) begin
                    nextState = LEFT_SCORED;
                    served = 1'b0;
                    left_scored = 1'b1;
                    right_scored = 1'b0;
                end
                else begin
                    nextState = SERVED;
                    served = 1'b1;
                    left_scored = 1'b0;
                    right_scored = 1'b0;
                end
            end
            // right scored state
            RIGHT_SCORED: begin
                if (serve) begin
                    served = 1'd1;
                    nextState = SERVED;
                    left_scored = 1'b0;
                    right_scored = 1'b0;
                end else begin
                    served = 1'd0;
                    nextState = RIGHT_SCORED;
                    left_scored = 1'b0;
                    right_scored = 1'b1;
                end
            end
            // left scored state
            LEFT_SCORED: begin
                if (serve) begin
                    served = 1'd1;
                    nextState = SERVED;
                    left_scored = 1'b0;
                    right_scored = 1'b0;
                end else begin
                    served = 1'd0;
                    nextState = LEFT_SCORED;
                    left_scored = 1'b1;
                    right_scored = 1'b0;
                end
            end

        endcase
    end

    always_ff @(posedge clock) begin
    if (reset)
      currState <= INIT;
    else
      currState <= nextState;
    end


endmodule
```

Lab Code [10 points]
Filename: vga.sv
AndrewID: kakinci

```systemverilog
`default_nettype none


// Module that represents our VGA output row, col, HS, VS and blank
module vga
    (input logic CLOCK_50, reset,
    output logic HS, VS, blank,
    output logic [8:0] row,
    output logic [9:0] col);

    logic [19:0] T_VS; // Represents the period T for VS
    logic [19:0] T_HS; // Represents the period T for HS

    logic T_bp_VS, T_disp_VS, T_fp_VS; // different states for the VS
    logic T_bp_HS, T_disp_HS, T_fp_HS;

    logic clock_clear_HS, clock_clear_VS; // clears the HS and VS


    logic column_enable, column_load;

    logic column_end, row_load;

    logic row_enable;


    // assign clock_row_
    // Clocks for the period of VS and HS and for the row and column
    Counter #(20) clock_counter_hs(.D(20'd0), .en(1'd1), .clear(reset),
                .load(clock_clear_HS), .clock(CLOCK_50), .up(1'b1), .Q(T_HS));
                                 // Clock for the period T

    Counter #(20) clock_counter_vs(.D(20'd0), .en(1'd1), .clear(reset),
                .load(clock_clear_VS), .clock(CLOCK_50), .up(1'b1), .Q(T_VS));
    assign column_enable = T_disp_HS & ~T_HS[0];
    assign column_load = col > 10'd640;
    Counter #(10) clock_counter_col(.D(10'd0), .en(column_enable),
        .clear(reset),.load(column_load), .clock(CLOCK_50), .up(1'b1), .Q(col));


    assign row_load = (row > 9'd479);
    assign row_enable = column_load & T_disp_VS;
    Counter #(9) clock_counter_row(.D(9'd0), .en(row_enable), .clear(reset),
                        .load(row_load), .clock(CLOCK_50), .up(1'b1), .Q(row));


    //Offset Checkers for VS and HS States
    OffsetCheck #(20) BP_Checker_HS(.low(20'd192),.delta(20'd96),.val(T_HS),
                                                 .is_between(T_bp_HS));
    OffsetCheck #(20) Display_Checker_HS(.low(20'd288),.delta(20'd1280),
                                         .val(T_HS), .is_between(T_disp_HS));
    OffsetCheck #(20) FP_Checker_HS(.low(20'd1569),.delta(20'd32),
                                         .val(T_HS), .is_between(T_fp_HS));


    OffsetCheck #(20) BP_Checker_VS(.low(20'd3200), .delta(20'd46400),
                                          .val(T_VS), .is_between(T_bp_VS));
    OffsetCheck #(20) Display_Checker_VS(.low(20'd49600),
                        .delta(20'd768000), .val(T_VS), .is_between(T_disp_VS));
    OffsetCheck #(20) FP_Checker_VS(.low(20'd817601), .delta(20'd16000),
                                          .val(T_VS), .is_between(T_fp_VS));

    assign blank = ~(T_disp_VS & T_disp_HS);

    // Outputs for VS and HS
    MagComp #(32) HSOutput_MagComp (.AltB(), .AeqB(), .AgtB(HS), .A(T_HS),
                                                             .B(32'd192));
    MagComp #(32) VSOutput_MagComp (.AltB(), .AeqB(), .AgtB(VS), .A(T_VS),
                                                            .B(32'd3200));

    // Enable for clear

    // Outputs for the clock_clear

    MagComp #(32) HSClockClear (.AltB(), .AeqB(clock_clear_HS), .AgtB(),
                                                    .A(T_HS), .B(32'd1599));
    MagComp #(32) VSClockClear(.AltB(), .AeqB(clock_clear_VS), .AgtB(),
                                                 .A(T_VS), .B(32'd833599));

endmodule: vga
```

Lab Code [10 points]
Filename: vga_test.sv
AndrewID: kakinci

```
`default_nettype none





module VGACheck_test ();

    logic CLOCK_50, reset;
    logic HS, VS, blank;
    logic [8:0] row;
    logic [9:0] col;
    vga test (.*);
    initial begin
        CLOCK_50 = 0;
        forever #5 CLOCK_50 = ~CLOCK_50;

    end

    initial begin

        $monitor($time,, "Time= %d, reset = %d, HS= %d, VS = %d, blank = %d, \
        row = %d, col = %d",
        CLOCK_50, reset, HS, VS, blank, row, col);

        reset = 0;
        @(posedge CLOCK_50)
        reset = 1;
        @(posedge CLOCK_50)
        reset = 0;
        @(posedge CLOCK_50)

        #20000000 $finish;
    end
endmodule: VGACheck_test

```