

```

1 `default_nettype none
2
3
4 // module that represents our striped color for our VGA module
5 module VGA_colorOutput(
6     input logic [9:0] col,
7     input logic [8:0] row,
8     output logic [7:0] R,G,B);
9
10    always_comb begin
11        if (row >= 9'd240) begin
12            R = 9'h00;
13            G = 9'h00;
14            B = 9'h00;
15        end
16        else begin
17            if (col >= 10'd0 && col <= 10'd79) begin
18                R = 8'h00;
19                G = 8'h00;
20                B = 8'h00;
21            end
22            else if (col >= 10'd80 && col <= 10'd159) begin
23                R = 8'h00;
24                G = 8'h00;
25                B = 8'hFF;
26            end
27            else if (col >= 10'd160 && col <= 10'd239) begin
28                R = 8'h00;
29                G = 8'hFF;
30                B = 8'h00;
31            end
32            else if (col >= 10'd240 && col <= 10'd319) begin
33                R = 8'h00;
34                G = 8'hFF;
35                B = 8'hFF;
36            end
37            else if (col >= 10'd320 && col <= 10'd399) begin
38                R = 8'hFF;
39                G = 8'h00;
40                B = 8'h00;
41            end
42            else if (col >= 10'd400 && col <= 10'd479) begin
43                R = 8'hFF;
44                G = 8'h00;
45                B = 8'hFF;
46            end
47            else if (col >= 10'd480 && col <= 10'd559) begin
48                R = 8'hFF;
49                G = 8'hFF;
50                B = 8'h00;
51            end
52            else if (col >= 10'd560 && col <= 10'd639) begin
53                R = 8'hFF;
54                G = 8'hFF;
55                B = 8'hFF;
56            end
57            else begin
58                R = 8'h00;
59                G = 8'hFF;
60                B = 8'h00;
61            end
62        end
63    end

```

```

64 endmodule
65
66
67 // represents our color output for our Pong game
68 module pong_colorOutput (
69     input logic [9:0] col,
70     input logic [8:0] row,
71     input logic validPaddleLeft,
72     input logic validPaddleRight,
73     input logic validBall,
74     input logic right_scored,
75     input logic left_scored,
76     output logic [7:0] R,G,B
77 );
78 always_comb begin
79     if (validPaddleLeft) begin
80         // represents the left paddle
81         R = 8'd0;
82         G = 8'd255;
83         B = 8'd0;
84     end
85     else if (validPaddleRight) begin
Line contains tabs (each tab replaced by 2 spaces in this print)
86         // represents the left paddle
87         R = 8'd0;
88         G = 8'd255;
89         B = 8'd100;
90     end
91     else if (validBall) begin
92         // represents the left paddle
93         R = 8'd255;
94         G = 8'd255;
95         B = 8'd255;
96     end
97     else if (right_scored) begin
98         R = 8'd200;
99         G = 8'd200;
100        B = 8'd100;
101    end
102    else if (left_scored) begin
103        R = 8'd100;
104        G = 8'd0;
105        B = 8'd0;
106    end
107    else begin
108        R = 8'd0;
109        G = 8'd0;
110        B = 8'd0;
111    end
112 end
113 endmodule: pong_colorOutput
114
115
116 // ChipInterface for our VGA and pong module
117 module chipinterface
118     (input logic CLOCK_50,
119     input logic [3:0] KEY,
120     input logic [17:0] SW,
121     output logic [6:0] HEX0, HEX1, HEX2, HEX3,
122     HEX4, HEX5, HEX6, HEX7,
123     output logic [7:0] VGA_R, VGA_G, VGA_B,
124     output logic VGA_BLANK_N, VGA_CLK, VGA_SYNC_N,
125     output logic VGA_VS, VGA_HS);
126
127     logic [8:0] row;
128     logic [9:0] col;
129     logic BLANK0;
130     assign VGA_BLANK_N = ~BLANK0;
131     logic reset, serve;
Line contains tabs (each tab replaced by 2 spaces in this print)
132     assign reset = ~KEY[0];
Line contains tabs (each tab replaced by 2 spaces in this print)

```

```

133     assign serve = ~KEY[3];
134
135     // outputs of the paddles
136     logic validPaddleLeft;
137     logic validPaddleRight;
138
139     // outputs of the ball
140     logic validBall;
141
142     // booleans to represent score by either side
143     logic right_scored, left_scored;
144
145     vga    v1(.CLOCK_50(CLOCK_50), .reset(reset), .HS(VGA_HS), .VS(VGA_VS),
146              .blank(BLANK0), .row(row), .col(col));
147
148     // Lab 4 week 1:
149     // VGA_colorOutput f2(.col(col),.row(row),.R(VGA_R),.G(VGA_G),.B(VGA_B));
150
151     pong_colorOutput f3(.col(col), .row(row), .validPaddleLeft(validPaddleLeft),
152                        .validPaddleRight(validPaddleRight), .validBall(validBall),
153                        .right_scored(right_scored), .left_scored(left_scored),
154                        .R(VGA_R), .G(VGA_G), .B(VGA_B));
155
156     assign VGA_SYNC_N = 1'b0;
157
158     assign VGA_CLK = ~CLOCK_50;
159
160     logic [8:0] leftPaddleRow;
161     logic [8:0] rightPaddleRow;
162
163     logic [3:0] right_score, left_score; // right and left paddle scores
164
165
166
167     left_paddle lp(.row(row), .col(col), .do_move(SW[17]), .direction(SW[16]),
168                  .CLOCK_50(CLOCK_50), .reset(reset), .valid_paddle(validPaddleLeft),
169                  .left_paddle_row(leftPaddleRow));
170     right_paddle rp(.row(row), .col(col), .do_move(SW[1]), .direction(SW[0]),
171                   .CLOCK_50(CLOCK_50), .reset(reset), .valid_paddle(validPaddleRight),
172                   .right_paddle_row(rightPaddleRow));
173
174
175
176     ball b1(.row(row), .col(col), .clock(CLOCK_50), .reset(reset),
177            .serve(serve),
178            .paddle_left_row(leftPaddleRow),
179            .paddle_right_row(rightPaddleRow),
180            .valid_ball(validBall), .right_scored(right_scored),
181            .left_scored(left_scored),
182            .right_score(right_score),
183            .left_score(left_score));
184
185
186     BCDtoSevenSegment chex0(.bcd_digit(right_score), .segment(HEX0));
187     BCDtoSevenSegment chex7(.bcd_digit(left_score), .segment(HEX7));
188
189     BCDtoSevenSegment chex1(.bcd_digit(4'd0), .segment(HEX1));
190     BCDtoSevenSegment chex2(.bcd_digit(4'd0), .segment(HEX2));
191     BCDtoSevenSegment chex3(.bcd_digit(4'd0), .segment(HEX3));
192     BCDtoSevenSegment chex4(.bcd_digit(4'd0), .segment(HEX4));
193     BCDtoSevenSegment chex5(.bcd_digit(4'd0), .segment(HEX5));
194     BCDtoSevenSegment chex6(.bcd_digit(4'd0), .segment(HEX6));
195
196
197     endmodule: chipinterface
198
199
200
201
202
203

```

```
204
205 // This module converts a 4-bit BCD number to a 7-segment bit output.
206 module BCDtoSevenSegment
207 (
208     input logic [3:0] bcd_digit,
209     output logic [6:0] segment
210 );
211
212 always_comb begin
213     case (bcd_digit)
214         4'd0: segment = 7'b111_1111;
215         4'd1: segment = 7'b111_1001;
216         4'd2: segment = 7'b010_0100;
217         4'd3: segment = 7'b011_0000;
218         4'd4: segment = 7'b001_1001;
219         4'd5: segment = 7'b001_0010;
220         4'd6: segment = 7'b000_0010;
221         4'd7: segment = 7'b111_1000;
222         4'd8: segment = 7'b000_0000;
223         4'd9: segment = 7'b001_0000;
224         default: segment = 7'b111_1111;
225     endcase
226 end
227
228 endmodule: BCDtoSevenSegment
229
230
231
232
```

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

```
1 `default_nettype none
2
3
4 // range check module
5 module RangeCheck
6     #(parameter WIDTH = 6)
7     (input logic [WIDTH-1:0] val,
8      input logic [WIDTH-1:0] low,
9      input logic [WIDTH-1:0] high,
10     output logic is_between );
11
12     assign is_between = (val >= low) && (val <= high);
13 endmodule: RangeCheck
14
15
16
17
18 // Typical offset check module
19 module OffsetCheck
20     #(parameter WIDTH = 6)
21     (input logic [WIDTH-1:0] delta,
22     input logic [WIDTH-1:0] low,
23     input logic [WIDTH-1:0] val,
24     output logic is_between);
25     assign is_between = (val >= low) && (val <= low+delta);
26
27 endmodule
28
29
30
31 // represents our module for counting the score
32 module Score_Counter
33     #(parameter WIDTH=8)
34     (input logic [WIDTH-1:0] D,
35      input logic          en, clear, load, clock, up,
36      output logic [WIDTH-1:0] Q);
37
38     always_ff @(posedge clock)
39         if (clear)
40             Q <= {WIDTH {1'b0}};
41         else if (load)
42             Q <= D;
43         else if (en)
44             if (up)
45                 Q <= Q + 1'b1;
46             else
47                 Q <= Q;
48
49 endmodule : Score_Counter
50
51 // A binary up-down counter.
52 // Clear has priority over Load, which has priority over Enable
53 module Counter
54     #(parameter WIDTH=8)
55     (input logic [WIDTH-1:0] D,
56      input logic          en, clear, load, clock, up,
57      output logic [WIDTH-1:0] Q);
58
59     always_ff @(posedge clock)
60         if (clear)
61             Q <= {WIDTH {1'b0}};
62         else if (load)
63             Q <= D;
64         else if (en)
65             if (up)
66                 Q <= Q + 1'b1;
67             else
68                 Q <= Q - 1'b1;
69
```

```

70 endmodule : Counter
71
72 // A binary up-down counter.
73 // Clear has priority over Load, which has priority over Enable
74 module Counter_Row
75     #(parameter WIDTH=8)
76     (input logic [WIDTH-1:0] D,
77      input logic          en, clear, load, clock, up,
78      output logic [WIDTH-1:0] Q);
79
80     always_ff @(posedge clock)
81         if (clear)
82             Q <= {WIDTH {1'b0}};
83         else if (load)
84             Q <= D;
85         else if (en)
86             if (up)
87                 Q <= Q + 1'b1;
88             else
89                 Q <= Q - 1'b1;
90
91 endmodule : Counter_Row
92
93
94 // A binary up-down counter.
95 // Clear has priority over Load, which has priority over Enable
96 module Counter_Col
97     #(parameter WIDTH=8)
98     (input logic [WIDTH-1:0] D,
99      input logic          en, clear, load, clock, up,
100      output logic [WIDTH-1:0] Q);
101
102     always_ff @(posedge clock)
103         if (clear)
104             Q <= {WIDTH {1'b0}};
105         else if (load)
106             Q <= D;
107         else if (en)
108             if (up)
109                 Q <= Q + 2'd2;
110             else
111                 Q <= Q - 2'd2;
112
113 endmodule : Counter_Col
114
115 // A Magnitude Comparator does an unsigned comparison of two input values.
116 module MagComp
117     #(parameter WIDTH = 8)
118     (output logic      AltB, AeqB, AgtB,
119      input logic [WIDTH-1:0] A, B);
120
121     assign AeqB = (A == B);
122     assign AltB  = (A <  B);
123     assign AgtB  = (A >  B);
124
125 endmodule: MagComp
126
127
128 // 2 to 1 multiplexer module
129 module Mux2to1
130     #(parameter WIDTH = 8)
131     (input logic [WIDTH-1:0] I0, I1,
132      input logic          S,
133      output logic [WIDTH-1:0] Y);
134
135     assign Y = (S) ? I1 : I0;
136
137 endmodule : Mux2to1
138
139
140 // Register module

```

```
141 module Register
142     #(parameter WIDTH=8)
143     (input logic [WIDTH-1:0] D,
144      input logic      en, clear, clock,
145      output logic [WIDTH-1:0] Q);
146
147     always_ff @(posedge clock)
148         if (en)
149             Q <= D;
150         else if (clear)
151             Q <= '0;
152
153 endmodule : Register
154
155
```

Lab Code [10 points]

Filename: library_tests.sv

AndrewID: kakinci

```
1 `default_nettype none
2
3
4
5
6 // module RangeCheck_test #(parameter WIDTH = 8)();
7
8
9 //     logic [WIDTH-1:0] low;
10 //     logic [WIDTH-1:0] high;
11 //     logic [WIDTH-1:0] val;
12 //     logic is_between;
13 //     RangeCheck #(WIDTH) rc (.*);
14 //     initial begin
15 //         $monitor($time,, "low= %d, high = %d, val= %d, in_between = %d", l...
Line length of 81 (max is 80)
16 //                                     high, val, is_between);
17 //
18 //         low = 0;
19 //         high = 2;
20 //         val = 3;
21 //         #10 low= 0;
22 //         high = 4;
23 //         #10 low = 123;
24 //         high = 0;
25 //         val = 12;
26 //         #10 low = 12;
27 //         high = 20;
28 //         val = 1;
29
30 //         #10 $finish;
31 //     end
32 // endmodule: RangeCheck_test
33
34
35
36
37
38 module OffsetCheck_test #(parameter WIDTH = 8)();
39     logic [WIDTH-1:0] low;
40     logic [WIDTH-1:0] delta;
41     logic [WIDTH-1:0] val;
42     logic is_between;
43     OffsetCheck #(WIDTH) rc (.*);
44     initial begin
45         $monitor($time,, "low= %d, delta = %d, val= %d, in_between = %d", low,
46                                     delta, val, is_between);
47         low = 0;
48         delta = 2;
49         val = 3;
50         #10 low= 0;
51         delta = 4;
52         #10 low = 123;
53         delta = 0;
54         val = 12;
55         #10 low = 12;
56         delta = 20;
57         val = 1;
58         #10 low = 12;
59         delta = 2;
60         val = 13;
61         #10 $finish;
62     end
63 endmodule: OffsetCheck_test
```


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

```
1 `default_nettype none
2
3
4 // Module to handle the movement of our left paddle
5 module left_paddle
6     (input logic [8:0] row,
7      input logic [9:0] col,
8      input logic do_move,
9      input logic direction,
10     input logic CLOCK_50,
11     input logic reset,
12     output logic valid_paddle,
13     output logic [8:0] left_paddle_row);
14
15     logic valid_col, valid_row; // represents if row and col are valid
16     logic reachedtop, reachedbottom; // represents if row
17         //has reached the top or bottom of the paddle
18     logic [8:0] row_high; // represents where the current
19     logic [8:0] row_high_add, row_high_sub; // represents the output of the
20         // two mux2to1's
21     //whether row_high or row_high+5 or row_high-5 is chosen
22     logic [8:0] row_reset_value;
23     logic [8:0] row_direction; //out of mux2to1 whether we move up or down
24
25     logic [8:0] row_high_plus5, row_high_minus5;
26
27     logic register_enable, refresh;
28
29     RangeCheck #(10) inCol(.val(col), .low(10'd60), .high(10'd63),
30         .is_between(valid_col));
31     OffsetCheck #(9) inRow(.delta(9'd48), .low(row_high), .val(row),
32         .is_between(valid_row));
33     assign valid_paddle = valid_col & valid_row;
34
35
36     // Check to see if the high row of our paddle has reached a boundary
37     MagComp #(9) reach_top(.AltB(), .AeqB(reachedtop), .AgtB(), .A(row_high),
38         .B(9));
39     MagComp #(9) reach_bottom(.AltB(), .AeqB(), .AgtB(reachedbottom),
40         .A(row_high), .B(9'd425));
41     logic real_refresh = 0;
42     assign refresh = (col == 10'd639) & (row == 9'd479) ;
43     assign register_enable = do_move & (refresh); // enables every refresh and
44         // when the player is moving
45     Register #(9) high_row_value(.D(row_reset_value), .en(register_enable),
46         .clear(1'd0), .clock(CLOCK_50),
47         .Q(row_high));
48
49     assign row_high_plus5 = row_high + 5;
50     assign row_high_minus5 = row_high - 5;
51
52
53     Mux2to1 #(9) choose_too_big (.I0(row_high_plus5), .I1(row_high),
54         .S(reachedbottom),
55         .Y(row_high_add));
56     Mux2to1 #(9) choose_too_small (.I0(row_high_minus5), .I1(row_high),
57         .S(reachedtop), .Y(row_high_sub));
58     Mux2to1 #(9) choose_direction (.I0(row_high_add), .I1(row_high_sub),
59         .S(direction), .Y(row_direction));
60     Mux2to1 #(9) choose_reset (.I0(row_direction), .I1(9'd192),
61         .S(reset), .Y(row_reset_value));
62
63     assign left_paddle_row = row_high;
64
65 endmodule: left_paddle
66
67
68 // module to handle the movement of our right paddle
69 module right_paddle
```

```

70     (input logic [8:0] row,
71     input logic [9:0] col,
72     input logic do_move,
73     input logic direction,
74     input logic CLOCK_50,
75     input logic reset,
76     output logic valid_paddle,
77     output logic [8:0] right_paddle_row);
78
79     logic valid_col, valid_row; // represents if row and col are valid
80     logic reachedtop, reachedbottom; // represents if row
81     //has reached the top or bottom of the paddle
82     logic [8:0] row_high; // represents where the current
83     logic [8:0] row_high_add, row_high_sub; // represents the output of the
84     // two mux2to1's
85     //whether row_high or row_high+5 or row_high-5 is chosen
86     logic [8:0] row_reset_value;
87     logic [8:0] row_direction; //out of mux2to1 whether we move up or down
88
89     logic [8:0] row_high_plus5, row_high_minus5;
90
91     logic register_enable, refresh;
92
93     RangeCheck #(10) inCol(.val(col), .low(10'd577), .high(10'd580),
94     .is_between(valid_col));
95     OffsetCheck #(9) inRow(.delta(9'd48), .low(row_high), .val(row),
96     .is_between(valid_row));
97     assign valid_paddle = valid_col & valid_row;
98
99
100    // Check to see if the high row of our paddle has reached a boundary
101    MagComp #(9) reach_top(.AltB(), .AeqB(reachedtop), .AgtB(), .A(row_high),
102    .B(9'd0));
103    MagComp #(9) reach_bottom(.AltB(), .AeqB(), .AgtB(reachedbottom),
104    .A(row_high), .B(9'd432));
105
106    assign refresh = (col == 10'd639) & (row == 9'd479);
107    assign register_enable = do_move & (refresh); // enables every refresh and
108    // when the player is moving
109    Register #(9) high_row_value(.D(row_reset_value), .en(register_enable),
110    .clear(1'd0), .clock(CLOCK_50),
111    .Q(row_high));
112
113    assign row_high_plus5 = row_high + 5;
114    assign row_high_minus5 = row_high - 5;
115
116
117    Mux2to1 #(9) choose_too_big (.IO(row_high_plus5), .I1(row_high),
118    .S(reachedbottom),
119    .Y(row_high_add));
120    Mux2to1 #(9) choose_too_small (.IO(row_high_minus5), .I1(row_high),
121    .S(reachedtop), .Y(row_high_sub));
122    Mux2to1 #(9) choose_direction (.IO(row_high_add), .I1(row_high_sub),
123    .S(direction), .Y(row_direction));
124    Mux2to1 #(9) choose_reset (.IO(row_direction), .I1(9'd192),
125    .S(reset), .Y(row_reset_value));
126    assign right_paddle_row = row_high;
127
128    endmodule: right_paddle
129
130
131    // Module that represents the movement of our ball
132
133    module ball
134        (input logic [8:0] row,
135        input logic [9:0] col,
136        input logic clock,
137        input logic reset,
138        input logic serve,
139        input logic [8:0] paddle_left_row,
140        input logic [8:0] paddle_right_row,

```

```

141     output logic valid_ball,
142     output logic right_scored,
143     output logic left_scored,
144     output logic [3:0] right_score,
145     output logic [3:0] left_score);
146
147     logic [9:0] ball_col;
148     logic [8:0] ball_row;
149     logic in_ball_col, in_ball_row;
150
151     logic hvelocity;
152     logic vvelocity;
153     logic enable;
154
155     logic refresh, served;
156
157
158     logic score_left, score_right;
159
160     logic top, bottom; //true if the ball is on the top or bottom of screen
161     logic dVV, dHV;
162
163
164     // score for left and right module
165     always_comb begin
166
167         if (ball_col < 10'd5) begin
168             score_right = 1;
169             score_left = 0;
170         end
171         else if (ball_col > 10'd635) begin
172             score_left = 1;
173             score_right = 0;
174         end
175         else begin
176             score_left = 0;
177             score_right = 0;
178         end
179     end
180
181     ballFsm d(.serve(serve), .reset(reset), .clock(clock),
182             .score_left(score_left), .score_right(score_right),
183             .served(served), .right_scored(right_scored),
184             .left_scored(left_scored));
185
186     // final answer returns
187     OffsetCheck #(10) ball_col_check (.low(ball_col),.delta(10'd4),.val(col),
188             .is_between(in_ball_col));
189     OffsetCheck #(9) ball_row_check (.low(ball_row),.delta(9'd4),.val(row),
190             .is_between(in_ball_row));
191     assign valid_ball = in_ball_col & in_ball_row;
192
193     // Counters for score
194
195     Score_Counter #(4) left_scoreC(.D(4'd0), .en(1'd1), .clear(1'd0),
196             .load(reset), .clock(clock), .up(score_left), .Q(left_score));
197     Score_Counter #(4) right_scoreC(.D(4'd0), .en(1'd1), .clear(1'd0),
198             .load(reset), .clock(clock), .up(score_right), .Q(right_score));
199
200
201     assign refresh = (col == 10'd639) & (row == 9'd479);
202     // boundary checks
203
204     MagComp #(9) compare_row_top (.A(ball_row),.B(9'd2),.AeqB(),.AgtB(),
205             .AltB(top));
206     MagComp #(9) compare_row_bottom (.A(ball_row),.B(9'd477),.AeqB(),
207             .AgtB(bottom),.AltB());
208
209     // // Left Paddle Hit
210     logic left_paddle_hit, left1, left2;
211     OffsetCheck #(9) compareRowLeft(.val(ball_row), .low(paddle_left_row),

```

[illegible]

```
282 endmodule: ball
283
284
285 // represents the finite state machine of our ball movement
286 module ballFsm
287     (input logic serve, reset, clock, score_left, score_right,
288      output logic served, right_scored, left_scored);
289
290     enum logic [1:0] {INIT, SERVED, RIGHT_SCORED, LEFT_SCORED} currState,
291                      nextState;
292
293     always_comb begin
294         case(currState)
295             // initial state
296             INIT: begin
297                 if(serve)begin
298                     served = 1'd1;
299                     nextState = SERVED;
300                     left_scored = 1'b0;
301                     right_scored = 1'b0;
302                 end else begin
303                     served = 1'd0;
304                     nextState = INIT;
305                     left_scored = 1'b0;
306                     right_scored = 1'b0;
307                 end
308             end
309             // served state
310             SERVED: begin
311                 served = 1'd1;
312                 nextState = SERVED;
313                 if (score_right) begin
314                     nextState = RIGHT_SCORED;
315                     served = 1'b0;
316                     right_scored = 1'b1;
317                     left_scored = 1'b0;
318                 end
319                 else if (score_left) begin
320                     nextState = LEFT_SCORED;
321                     served = 1'b0;
322                     left_scored = 1'b1;
323                     right_scored = 1'b0;
324                 end
325                 else begin
326                     nextState = SERVED;
327                     served = 1'b1;
328                     left_scored = 1'b0;
329                     right_scored = 1'b0;
330                 end
331             end
332             // right scored state
333             RIGHT_SCORED: begin
334                 if (serve) begin
335                     served = 1'd1;
336                     nextState = SERVED;
337                     left_scored = 1'b0;
338                     right_scored = 1'b0;
339                 end else begin
340                     served = 1'd0;
341                     nextState = RIGHT_SCORED;
342                     left_scored = 1'b0;
343                     right_scored = 1'b1;
344                 end
345             end
346             // left scored state
347             LEFT_SCORED: begin
348                 if (serve) begin
349                     served = 1'd1;
350                     nextState = SERVED;
351                     left_scored = 1'b0;
352                     right_scored = 1'b0;
```

```
353         end else begin
354             served = 1'd0;
355             nextState = LEFT_SCORED;
356             left_scored = 1'b1;
357             right_scored = 1'b0;
358         end
359     end
360
361 endcase
362 end
363
364 always_ff @(posedge clock) begin
365     if (reset)
366         currState <= INIT;
367     else
368         currState <= nextState;
369     end
370
371
372 endmodule
373
374
```

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

```
1 `default_nettype none
2
3
4 // Module that represents our VGA output row, col, HS, VS and blank
5 module vga
6     (input logic CLOCK_50, reset,
7      output logic HS, VS, blank,
8      output logic [8:0] row,
9      output logic [9:0] col);
10
11     logic [19:0] T_VS; // Represents the period T for VS
12     logic [19:0] T_HS; // Represents the period T for HS
13
14     logic T_bp_VS, T_disp_VS, T_fp_VS; // different states for the VS
15     logic T_bp_HS, T_disp_HS, T_fp_HS;
16
17     logic clock_clear_HS, clock_clear_VS; // clears the HS and VS
18
19
20     logic column_enable, column_load;
21
22     logic column_end, row_load;
23
24     logic row_enable;
25
26
27     // assign clock_row_
28     // Clocks for the period of VS and HS and for the row and column
29     Counter #(20) clock_counter_hs(.D(20'd0), .en(1'd1), .clear(reset),
30                                     .load(clock_clear_HS), .clock(CLOCK_50), .up(1'b1), .Q(T_HS));
31                                     // Clock for the period T
32
33     Counter #(20) clock_counter_vs(.D(20'd0), .en(1'd1), .clear(reset),
34                                     .load(clock_clear_VS), .clock(CLOCK_50), .up(1'b1), .Q(T_VS));
35     assign column_enable = T_disp_HS & ~T_HS[0];
36     assign column_load = col > 10'd640;
37     Counter #(10) clock_counter_col(.D(10'd0), .en(column_enable),
38                                     .clear(reset), .load(column_load), .clock(CLOCK_50), .up(1'b1), .Q(col));
39
40
41     assign row_load = (row > 9'd479);
42     assign row_enable = column_load & T_disp_VS;
43     Counter #(9) clock_counter_row(.D(9'd0), .en(row_enable), .clear(reset),
44                                     .load(row_load), .clock(CLOCK_50), .up(1'b1), .Q(row));
45
46
47     //Offset Checkers for VS and HS States
48     OffsetCheck #(20) BP_Checker_HS(.low(20'd192), .delta(20'd96), .val(T_HS),
49                                     .is_between(T_bp_HS));
50     OffsetCheck #(20) Display_Checker_HS(.low(20'd288), .delta(20'd1280),
51                                     .val(T_HS), .is_between(T_disp_HS));
52     OffsetCheck #(20) FP_Checker_HS(.low(20'd1569), .delta(20'd32),
53                                     .val(T_HS), .is_between(T_fp_HS));
54
55
56     OffsetCheck #(20) BP_Checker_VS(.low(20'd3200), .delta(20'd46400),
57                                     .val(T_VS), .is_between(T_bp_VS));
58     OffsetCheck #(20) Display_Checker_VS(.low(20'd49600),
59                                     .delta(20'd768000), .val(T_VS), .is_between(T_disp_VS));
60     OffsetCheck #(20) FP_Checker_VS(.low(20'd817601), .delta(20'd16000),
61                                     .val(T_VS), .is_between(T_fp_VS));
62
63     assign blank = ~(T_disp_VS & T_disp_HS);
64
65     // Outputs for VS and HS
66     MagComp #(32) HSOutput_MagComp (.AltB(), .AeqB(), .AgtB(HS), .A(T_HS),
67                                     .B(32'd192));
68     MagComp #(32) VSOutput_MagComp (.AltB(), .AeqB(), .AgtB(VS), .A(T_VS),
69                                     .B(32'd3200));
```

```
70
71 // Enable for clear
72
73 // Outputs for the clock_clear
74
75 MagComp #(32) HSClockClear (.AltB(), .AeqB(clock_clear_HS), .AgtB(),
76                                     .A(T_HS), .B(32'd1599));
77 MagComp #(32) VSClockClear(.AltB(), .AeqB(clock_clear_VS), .AgtB(),
78                                     .A(T_VS), .B(32'd833599));
79
80 endmodule: vga
```


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

```
1 `default_nettype none
2
3
4
5
6
7 module VGACheck_test ();
8
9     logic CLOCK_50, reset;
10    logic HS, VS, blank;
11    logic [8:0] row;
12    logic [9:0] col;
13    vga test (.*) ;
14    initial begin
15        CLOCK_50 = 0;
16        forever #5 CLOCK_50 = ~CLOCK_50;
17
18    end
19
20    initial begin
21
22        $monitor($time,, "Time= %d, reset = %d, HS= %d, VS = %d, blank = %d, \
23        row = %d, col = %d",
24        CLOCK_50, reset, HS, VS, blank, row, col);
25
26        reset = 0;
27        @(posedge CLOCK_50)
28        reset = 1;
29        @(posedge CLOCK_50)
30        reset = 0;
31        @(posedge CLOCK_50)
32
33        #200000000 $finish;
34    end
35 endmodule: VGACheck_test
36
```