Oregon State University

OREGON STATE UNIVERSITY

ECE 271

Final Design Project

Benjamin Geyer
Christien Hotchkiss
Phi Luu

November 30th, 2018

Contents

```
    input logic[1:0] dir_x, dir_y,
    output logic vs, hs,
    output logic[3:0] r, g, b
);

logic[9:0] xpos, ypos;
logic slowclk;

logic[3:0] box_r, box_g, box_b;
logic[9:0] box_x, box_y, box_size;

half_clock hc (
    .oldclk(clk),
    .newclk(slowclk)
);

XYCounter c (
    .clk(slowclk),
    .vs(vs),
    .hs(hs),
    .x(xpos),
    .y(ypos)
);

box_controller box (
    .vs(vs),
    .dir_x(dir_x),
    .dir_y(dir_y),
    .x(xpos),
    .y(ypos),
    .box_r(box_r),
    .box_g(box_g),
    .box_b(box_b),
    .box_x(box_x),
    .box_y(box_y),
    .box_size(box_size)
);

Drawer d (
    .clk(slowclk),
    .box_r(box_r),
    .box_g(box_g),
    .box_b(box_b),
    .x(xpos),
    .y(ypos),
    .box_x(box_x),
    .box_y(box_y),
```

```
    .box_size(box_size),
    .r(r),
    .g(g),
    .b(b)
);

endmodule
```

7.1.2 PS/2 Top Level

```
module ps2_top_level(
    input logic clk, psClk, data,
    output logic vs, hs,
    output logic[3:0] r, g, b
);

// Track x and y and slow clock
logic[9:0] xpos, ypos;
logic slowclk;

// Get direction input
logic[1:0] dir_x, dir_y;

// Track box info
logic[3:0] box_r, box_g, box_b;
logic[9:0] box_x, box_y, box_size;

// Slow clock
half_clock hc (
    .oldclk(clk),
    .newclk(slowclk)
);

// Generate position and vs/hs signals
XYCounter c (
    .clk(slowclk),
    .vs(vs),
    .hs(hs),
    .x(xpos),
    .y(ypos)
);

// Get keyboard input for direction
ps2 P (
```

```
    .clk(psClk),
    .data(data),
    .dir_x(dir_x),
    .dir_y(dir_y)
);

// Move box
box_controller box (
    .vs(vs),
    .dir_x(dir_x),
    .dir_y(dir_y),
    .box_r(box_r),
    .box_g(box_g),
    .box_b(box_b),
    .box_x(box_x),
    .box_y(box_y),
    .box_size(box_size)
);

// Draw box
Drawer d (
    .clk(slowclk),
    .box_r(box_r),
    .box_g(box_g),
    .box_b(box_b),
    .x(xpos),
    .y(ypos),
    .box_x(box_x),
    .box_y(box_y),
    .box_size(box_size),
    .r(r),
    .g(g),
    .b(b)
);

endmodule
```

7.1.3 PS/2 Controller

```
module ps2 #(parameter
    UP = 8'h1D, DOWN = 8'h1B, LEFT = 8'h1C, RIGHT = 8'h23  // PS2 key code of W,
    ↪  S, A, and D, repsectively
)(
    input logic clk, data,
```

```
5       output logic[1:0] dir_x, dir_y
6   );
7
8   // Track data input
9   logic[7:0] bits = 8'b0;
10
11  // Track count of bits coming in and number of ones
12  logic[3:0] count = 0;
13  logic[3:0] even_odd = 0;
14
15  // Hold output values
16  // 0 = up/left
17  // 1 = down/right
18  // 2 = stop
19  logic[1:0] x = 2;
20  logic[1:0] y = 2;
21
22  // Track valid input and whether past input was F0 (indicating key release)
23  logic valid = 0;
24  logic fo = 0;
25
26  always_ff @(negedge clk)
27      begin
28          if (count == 10) // Ending bit
29              begin
30                  if (valid)
31                      // Translate directions to movement
32                      begin
33                          // not released && is not moving && to move up -> move up
34                          if (~fo && y == 2 && bits == UP) y <= 0;
35                          // released && is moving up && to move up -> stop
36                          else if (fo && y == 0 && bits == UP) y <= 2;
37                          // not released && is not moving && to move down -> move
                            ↪  down
38                          else if (~fo && y == 2 && bits == DOWN) y <= 1;
39                          // released && is moving down && to move down -> stop
40                          else if (fo && y == 1 && bits == DOWN) y <= 2;
41
42                          // not released && is not moving && to move left -> move
                            ↪  left
43                          if (~fo && x == 2 && bits == LEFT) x <= 0;
44                          // released && is moving left && to move left -> stop
45                          else if (fo && x == 0 && bits == LEFT) x <= 2;
46                          // not released && is not moving && to move right -> move
                            ↪  right
47                          else if (~fo && x == 2 && bits == RIGHT) x <= 1;
48                          // released && is moving right && to move right -> stop
```

```
                        else if (fo && x == 1 && bits == RIGHT) x <= 2;

                        // Get code for key release
                        if (bits == 8'hF0) fo <= 1;
                        else fo <= 0;
                    end

                even_odd <= 0;
                count <= 0;
                bits <= 8'd0;
                valid <= 0;
            end
        else if (count == 9)  // Check parity bit
            begin
                count <= count + 1;

                // Check validity based on number of ones and parity bit (odd
                ↪  total)
                valid <= even_odd[0] == ~data;
            end
        else if (count == 0)  // Start bit -- do nothing but advance count
            count <= count + 1;
        else
            begin
                // Track count and number of odd
                even_odd <= even_odd + data;
                count <= count + 1;

                // Load data
                bits <= (bits >> 1) + (data * 128);
            end
    end

// Set output
assign dir_x = x;
assign dir_y = y;

endmodule
```

7.1.4 RC5 IR Top Level

```
module ir_top_level(
    input logic clk, data,
    output logic vs, hs,
```

```
    output logic[3:0] r, g, b
);

// Track x and y and slow clock
logic[9:0] xpos, ypos;
logic slowclk;

// Get direction input
logic[1:0] dir_x, dir_y;

// Track box info
logic[3:0] box_r, box_g, box_b;
logic[9:0] box_x, box_y, box_size;

// Slow clock
half_clock hc (
    .oldclk(clk),
    .newclk(slowclk)
);

// Generate position and vs/hs signals
XYCounter c (
    .clk(slowclk),
    .vs(vs),
    .hs(hs),
    .x(xpos),
    .y(ypos)
);

// Get IR input for direction
ir I (
    .clk(clk),
    .data(data),
    .dir_x(dir_x),
    .dir_y(dir_y)
);

// Move box
box_controller box (
    .vs(vs),
    .dir_x(dir_x),
    .dir_y(dir_y),
    .box_r(box_r),
    .box_g(box_g),
    .box_b(box_b),
    .box_x(box_x),
    .box_y(box_y),
```

```
    .box_size(box_size)
);

// Draw box
Drawer d (
    .clk(slowclk),
    .box_r(box_r),
    .box_g(box_g),
    .box_b(box_b),
    .x(xpos),
    .y(ypos),
    .box_x(box_x),
    .box_y(box_y),
    .box_size(box_size),
    .r(r),
    .g(g),
    .b(b)
);

endmodule
```

7.1.5 RC5 IR Controller

```
// IR module for RC5 code
module ir #(parameter
    UP = 2, DOWN = 8, LEFT = 4, RIGHT = 6
)(
    input logic data, clk,
    output logic[1:0] dir_x, dir_y
);

// Count for slowing clock and new clock
logic[10:0] clkCount = 0;
logic newclk = 0;

// Complete data word and its update status
logic[5:0] dataWord = 0;
logic newDataWord = 0;

// Internal x/y movement
logic[1:0] x = 2;
logic[1:0] y = 2;

// Used for handling pulses
```

```
22  logic pusleVal = 0;
23  logic newPulseVal = 0;
24  logic waitForRepeat = 0;
25
26  // Slow clock from 50MHz to 36KHz as is used for RC5
27  always_ff @(posedge clk)
28      if (clkCount < 1388)
29              clkCount <= clkCount + 1;
30      else
31          begin
32              clkCount <= 0;
33              newclk = ~newclk;
34          end
35
36  // Converts from encoded pulse to binary
37  irConvertPulse cp(
38      .data(data),
39      .clk(newclk),
40      .val(pusleVal),
41      .newval(newPulseVal),
42      .waitForRepeat(waitForRepeat)
43  );
44
45  // Converts from encoded binary to data word
46  irReadWord rw(
47      .val(pulseVal),
48      .clk(newPulseVal),
49      .waitForRepeat(waitForRepeat),
50      .data(dataWord),
51      .finished(newDataWord)
52  );
53
54  // Translate from data word to movement value
55  always_ff @(posedge newDataWord)
56      begin
57          if (dataWord == UP) y <= 0;
58          else if (dataWord == DOWN) y <= 1;
59          else y <= 2;
60
61          if (dataWord == LEFT) x <= 0;
62          else if (dataWord == RIGHT) x <= 1;
63          else x <= 2;
64      end
65
66  assign dir_x = x;
67  assign dir_y = y;
68
```

```
endmodule


module irReadWord(
    input logic val, clk, waitForRepeat,
    output logic[5:0] data,
    output logic finished
);

// Stores output and counts bits
logic[5:0] out = 0;
logic[4:0] count = 0;

// Tracks previous toggle -- not used in this implementation
// but necessary for certain controls so it will be tracked
// anyway
logic prevToggle = 1;

always_ff @(posedge clk, negedge waitForRepeat)
    begin
        if (~waitForRepeat)  // Reset if time limit for repeating signal is up
            begin
                out <= 0;
                count <= 0;
                finished <= 1;
            end
        else if (count == 13)  // Reset count, store final value, and signal
        ↪ finished transmission
            begin
                out <= (out << 1) + val;
                finished <= 1;
                count <= 0;
            end
        else if (count == 2 && val != prevToggle)  // Set toggle and reset value
        ↪ if different button press
            begin
                prevToggle <= val;
                out <= 0;
                count <= count + 1;
                finished <= 0;
            end
        else if (count > 7)  // If in data word, add to output
            begin
                out <= (out << 1) + val;
                count <= count + 1;
                finished <= 0;
            end
```

```
114             else
115                 begin
116                     count <= count + 1;
117                     finished <= 0;
118                 end
119         end
120 
121     assign data = out;
122 
123     endmodule
124 
125 
126     module irConvertPulse(
127         input logic data, clk,
128         output logic val, newval, waitForRepeat
129     );
130 
131     // Tracks two sections with potential to have pulses
132     logic[5:0] dataCounterOne = 0;
133     logic[5:0] dataCounterTwo = 0;
134 
135     // Counts time
136     logic[5:0] timeCounter = 0;
137 
138     // Counts time from
139     logic[6:0] waitCounter = 78;
140     logic idle = 1;
141     logic out = 0;
142 
143     logic tcReset = 0;
144     logic dcReset = 0;
145     logic dcResetSuccess = 0;
146 
147     always_ff @(posedge data)
148         begin
149             if (idle) // Reset time counter if start of a transmission
150                 begin
151                     dataCounterOne <= 0;
152                     dataCounterTwo <= 1;
153                     tcReset <= 1;
154                 end
155             else if (dcReset)
156                 begin
157                     if (timeCounter > 31 && dataCounterTwo < 32)
158                         begin
159                             dataCounterOne <= 0;
160                             dataCounterTwo <= 1;
```

```
161                     end
162                 else if (dataCounterOne < 32)
163                     begin
164                         dataCounterOne <= 1;
165                         dataCounterTwo <= 0;
166                     end
167 
168                 dcResetSuccess <= 1;
169             end
170         else  // Increment data
171             begin
172                 if (timeCounter > 31 && dataCounterTwo < 32) dataCounterTwo <=
                    ↪  dataCounterTwo + 1;
173                 else if (dataCounterOne < 32) dataCounterOne <= dataCounterOne +
                    ↪  1;
174 
175                 tcReset <= 0;
176                 dcResetSuccess <= 0;
177             end
178     end
179 
180 always_ff @(posedge clk)
181     begin
182         if (tcReset)
183             begin
184                 timeCounter <= 33;
185                 waitCounter <= 0;
186                 idle <= 0;
187             end
188         else if (timeCounter == 63)  // Set output and reset
189             begin
190                 // Increment counter if still in window for additional
191                 if (idle && dataCounterOne == 0 && dataCounterTwo == 0 &&
                    ↪  waitCounter < 78)
192                     waitCounter <= waitCounter + 1;
193 
194                 out <= dataCounterOne == 0 && dataCounterTwo == 32 && ~dcReset;
195                 newval <= (dataCounterOne == 32 || dataCounterTwo == 32) &&
                    ↪  ~dcReset;
196                 idle <= (dataCounterOne == 0 && dataCounterTwo == 0) || ~dcReset;
197                 dcReset <= 1;
198                 timeCounter <= 0;
199             end
200         else  // Increment data
201             begin
202                 timeCounter <= timeCounter + 1;
203                 newval <= 0;
```

```
                if (dcResetSuccess)
                    dcReset <= 0;
            end
    end

assign val = out;

// Do not wait if outside of continued transmission window
assign waitForRepeat = waitCounter < 78;

endmodule
```

7.1.6 NES Top Level

```
module nes_top_level(
    input logic clk, data,
    output logic vs, hs, latch_out, clk_out,
    output logic[3:0] r, g, b
);

// Track x and y and slow clock
logic[9:0] xpos, ypos;
logic slowclk;

// Get direction input
logic[1:0] dir_x, dir_y;

// Track box info
logic[3:0] box_r, box_g, box_b;
logic[9:0] box_x, box_y, box_size;

// Slow clock
half_clock hc (
    .oldclk(clk),
    .newclk(slowclk)
);

// Generate position and vs/hs signals
XYCounter c (
    .clk(slowclk),
    .vs(vs),
    .hs(hs),
    .x(xpos),
```

```
    .y(ypos)
);

// Get nes input for direction
nes N (
    .clk(clk),
    .data(data),
    .dir_x(dir_x),
    .dir_y(dir_y),
    .latch_out(latch_out),
    .clk_out(clk_out)
);

// Move box
box_controller box (
    .vs(vs),
    .dir_x(dir_x),
    .dir_y(dir_y),
    .box_r(box_r),
    .box_g(box_g),
    .box_b(box_b),
    .box_x(box_x),
    .box_y(box_y),
    .box_size(box_size)
);

// Draw box
Drawer d (
    .clk(slowclk),
    .box_r(box_r),
    .box_g(box_g),
    .box_b(box_b),
    .x(xpos),
    .y(ypos),
    .box_x(box_x),
    .box_y(box_y),
    .box_size(box_size),
    .r(r),
    .g(g),
    .b(b)
);

endmodule
```

7.1.7 NES Controller

```
module nes #(parameter
    UP = 5, DOWN = 6, LEFT = 7, RIGHT = 8
)(
    input logic data, clk,
    output logic latch_out, clk_out,
    output logic[1:0] dir_x, dir_y
);

// Track status of x and y direction and if a directional button was pressed
logic[1:0] x = 2;
logic[1:0] y = 2;
logic x_btn_press = 0;
logic y_btn_press = 0;

// Track when to poll controller for button presses
logic[19:0] pollClkCount = 0;
logic pollClk = 0;

// Track when to pulse clock
logic[9:0] pulseClkCount = 0;
logic pulseClk = 0;

// Track info about latch and clock pulse
logic latch_inner = 0;
logic new_pulse = 0;
logic start_pulse_out = 0;
logic[3:0] pulseCounter = 0;

// Convert clk from 50MHz to 60Hz for polling controller
// and to 83kHz to produce clock pulses
always_ff @(posedge clk)
    begin
        if (pollClkCount == 416666)
            begin
                // Start polling controller by resetting pulse data, starting
                ↪  latch,
                // and signaling start of new pulse
                if (~pollClk)
                    begin
                        latch_inner <= 1;
                        new_pulse <= 1;
                        pulseClkCount <= 0;
                        pulseClk <= 0;
                        pulseCounter <= 0;
```

```
                    end

                pollClkCount <= 0;
                pollClk <= ~pollClk;
            end
        else
            pollClkCount <= pollClkCount + 1;

        if (pulseClkCount == 301)
            begin
                if (new_pulse && ~pulseClk)  // Only update if controller poll
                ↪  has started
                    begin
                        // Track beginning of clock pulse to controller (after
                        ↪  latch)
                        if (pulseCounter > 0 && latch_inner == 0)
                            start_pulse_out <= 1;

                        // Count pulse
                        pulseCounter <= pulseCounter + 1;
                    end

                if (new_pulse && pulseClk)  // Only update if controller poll has
                ↪  started
                    begin
                        // Reset latch once its 12us have passed
                        if (pulseCounter > 0 && latch_inner)
                            latch_inner <= 0;

                        // Reset pulse data if all buttons have been accounted
                        if (pulseCounter == 9)
                            begin
                                new_pulse <= 0;
                                start_pulse_out <= 0;
                                y_btn_press <= 0;
                                x_btn_press <= 0;
                            end
                        else
                            begin
                                // Check for UP or DOWN buttons being pressed and
                                ↪  track that
                                // a button in y direction has been pressed
                                if (pulseCounter == UP && data == 0)
                                    begin
                                        y <= 0;
                                        y_btn_press <= 1;
                                    end
```

```
87                                  else if (pulseCounter == DOWN && data == 0)
88                                      begin
89                                          y <= 1;
90                                          y_btn_press <= 1;
91                                      end
92                                  else if (pulseCounter >= UP && pulseCounter >=
                                    ↪  DOWN && y_btn_press == 0)  // Reset if no y
                                    ↪  button press
93                                      y <= 2;
94
95                                  // Check for RIGHT or LEFT buttons being pressed
                                    ↪  and track that
96                                  // a button in x direction has been pressed
97                                  if (pulseCounter == LEFT && data == 0)
98                                      begin
99                                          x <= 0;
100                                         x_btn_press <= 1;
101                                     end
102                                 else if (pulseCounter == RIGHT && data == 0)
103                                     begin
104                                         x <= 1;
105                                         x_btn_press <= 1;
106                                     end
107                                 else if (pulseCounter >= LEFT && pulseCounter >=
                                    ↪  RIGHT && y_btn_press == 0)  // Reset if no x
                                    ↪  button press
108                                     x <= 2;
109                             end
110                     end
111
112                 pulseClkCount <= 0;
113                 pulseClk <= ~pulseClk;
114             end
115         else
116             pulseClkCount <= pulseClkCount + 1;
117     end
118
119 // Assign outputs, and only set clock if part of pulse output
120 assign latch_out = latch_inner;
121 assign clk_out = start_pulse_out && pulseClk;
122 assign dir_x = x;
123 assign dir_y = y;
124
125 endmodule
```

7.1.8 Box Controller

```
module box_controller #(parameter
    SIZE = 40
)(
    input logic vs,
    input logic[1:0] dir_x, dir_y,
    output logic[3:0] box_r, box_g, box_b,
    output logic[9:0] box_x, box_y, box_size
);

// Track x and y
logic[9:0] xpos = 300;
logic[9:0] ypos = 220;

// Track box color
logic[3:0] r = 4'b1111;
logic[3:0] g = 4'b0;
logic[3:0] b = 4'b0;

always_ff @(posedge vs)
    begin
        if(~(dir_x == 0 && xpos == 0) && ~(dir_x == 1 && xpos == 640 - SIZE))  // Check that box will remain in screen bounds
            begin
                if((xpos - 1 == 0 && dir_x == 0) || (xpos + 1 == 640 - SIZE && dir_x == 1))  // Check for collision
                    begin
                        // Swap colors if collision occurs
                        r <= g;
                        g <= b;
                        b <= r;
                    end

                // Increment position
                case(dir_x)
                    0: xpos <= xpos - 1;
                    1: xpos <= xpos + 1;
                endcase
            end

        if(~(dir_y == 0 && ypos == 0) && ~(dir_y == 1 && ypos == 480 - SIZE))  // Check that box will remain in screen bounds
            begin
                if((ypos - 1 == 0 && dir_y == 0) || (ypos + 1 == 480 - SIZE && dir_y == 1))  // Check for collision
```

```
                    begin
                        // Swap colors if collision occurs
                        r <= g;
                        g <= b;
                        b <= r;
                    end

                // Increment position
                case(dir_y)
                    0: ypos <= ypos - 1;
                    1: ypos <= ypos + 1;
                endcase
            end
    end

assign box_x = xpos;
assign box_y = ypos;
assign box_size = SIZE;
assign box_r = r;
assign box_g = g;
assign box_b = b;

endmodule
```

7.1.9 Half Clock

```
module half_clock(
    input logic oldclk,
    output logic newclk
);

logic count = 0;

always_ff @(posedge oldclk)
    if (count == 1)
        begin
            newclk <= 1;
            count <= 0;
        end
    else
        begin
            newclk <= 0;
            count <= count + 1;
        end
```

```
19
20  endmodule
```

7.1.10 XY Counter

```
1   module XYCounter(
2       input logic clk,
3       output logic vs, hs,
4       output logic[9:0] x, y
5   );
6
7   // Track current x and y
8   logic[9:0] xpos = 0;
9   logic[9:0] ypos = 0;
10
11  always_ff @(posedge clk)
12      begin
13          // Generate HSync and VSync signals when in sync section
14          hs <= ~((xpos >= 640 + 16) && (xpos < 640 + 16 + 96));
15          vs <= ~((ypos >= 480 + 10) && (ypos < 480 + 10 + 2));
16
17          if(xpos == 800 && ypos == 525)  // Reset when at end of screen
18              begin
19                  ypos <= 0;
20                  xpos <= 0;
21              end
22          else if(xpos == 800)  // Increment y and reset x at end of line
23              begin
24                  ypos <= ypos + 1;
25                  xpos <= 0;
26              end
27          else  // Otherwise increment x
28              xpos <= xpos + 1;
29      end
30
31  assign x = xpos;
32  assign y = ypos;
33
34  endmodule
```

7.1.11 Drawer

```
module Drawer(
    input logic clk,
    input logic[3:0] box_r, box_g, box_b,
    input logic[9:0] x, y, box_x, box_y, box_size,
    output logic[3:0] r, g, b
);

always_ff @(posedge clk)
    // Draw black if not in box pixels
    if (x > 640 || y > 480 || x < box_x || y < box_y || x > box_x + box_size || y
    ↪  > box_y + box_size)
        begin
            r <= 4'b0;
            g <= 4'b0;
            b <= 4'b0;
        end
    else  // Draw box
        begin
            r <= box_r;
            g <= box_g;
            b <= box_b;
        end

endmodule
```