```
// stays when KEY[2] is not pressed

module reducedKeys(SW, KEY, CLOCK_50,HEX5, HEX3, HEX2, HEX1, HEX0, LEDR, VGA_R,
VGA_G, VGA_B, VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N, VGA_CLK,


    // Bidirectionals
    PS2_CLK,
    PS2_DAT,

    // Outputs

cIn, dIn, eIn, fIn, gIn, aIn, bIn,
csIn, dsIn, fsIn, gsIn, asIn,
cIn2, dIn2, eIn2, fIn2, gIn2, aIn2, bIn2,
csIn2, dsIn2, fsIn2, gsIn2, asIn2,


//for audio
AUD_ADCDAT, AUD_BCLK, AUD_ADCLRCK, AUD_DACLRCK, FPGA_I2C_SDAT, AUD_XCK,
AUD_DACDAT, FPGA_I2C_SCLK);


        input [3:0] KEY;

        input CLOCK_50;

        output [6:0] HEX3, HEX2, HEX1, HEX0;


        input [5:0] SW;

        output [7:0] VGA_R;

        output [7:0] VGA_G;

        output [7:0] VGA_B;

        output VGA_HS;

        output VGA_VS;

        output VGA_BLANK_N;
```

```
output VGA_SYNC_N;

output VGA_CLK;

reg [7:0] X;

reg [6:0] Y;

reg [7:0] X_count;

reg [6:0] Y_count;

wire [2:0] VGA_COLOR;

wire plot;

wire [1:0] key_type;

wire [5:0] key;

wire [2:0] color;
reg [2:0] color_output;

wire [2:0] color0;
wire [2:0] color1;
wire [2:0] color2;
wire [2:0] color3;
wire [2:0] color4;
wire [2:0] color5;
wire [2:0] color6;
wire [2:0] color7;
wire [2:0] color8;
wire [2:0] color9;
wire [2:0] color10;
wire [2:0] color11;
wire [2:0] color12;
wire [2:0] color13;
wire [2:0] color14;
wire [2:0] color15;

//reg [15:0] P;
wire load_enable;
wire key_load_enable;
```

```
	//reg [89:0] queue =
90'b000000000010000100010001010001110010010010110011000111000001000001000101001101010111;

	no_keys_pressed b0(160*Y + X, CLOCK_50, color0);
	Q_key b1(160*Y + X, CLOCK_50, color1);
	E_key b3(160*Y + X, CLOCK_50, color2);
	T_key b5(160*Y + X, CLOCK_50, color3);
	Y_key b6(160*Y + X, CLOCK_50, color4);
	I_key b8(160*Y + X, CLOCK_50, color5);
	P_key b10(160*Y + X, CLOCK_50, color6);
	S_key b12(160*Y + X, CLOCK_50, color7);
	D_key b13(160*Y + X, CLOCK_50, color8);
	G_key b15(160*Y + X, CLOCK_50, color9);
	J_key b17(160*Y + X, CLOCK_50, color10);
	K_key b18(160*Y + X, CLOCK_50, color11);
	Z_key b20(160*Y + X, CLOCK_50, color12);
	C_key b22(160*Y + X, CLOCK_50, color13);
	B_key b24(160*Y + X, CLOCK_50, color14);

  //regn UY (LEDR[5:0], KEY[0], ~KEY[1], CLOCK_50, key);

	//three_second T(KEY[0], CLOCK_50, load_enable);

	//key_prompt_shift_register S1(CLOCK_50, KEY[0], load_enable, queue, key);

	key_selection K1(KEY[0], LEDR[5:0], color0, color1, color2, color3, color4, color5,
color6, color7, color8, color9, color10, color11, color12, color13, color14, CLOCK_50, key_type,
color);


	assign plot = 1'b1;



	always @ (posedge CLOCK_50) begin

		if (plot) begin
			if (X_count == 8'd160 && ~(Y_count == 8'd120)) begin
				X_count <= 8'd0;
				Y_count <= Y_count + 1;
			end else if (X_count == 8'd159 && Y_count == 8'd119) begin
				X_count <= 8'd0;
				Y_count <= 8'd0;
```

```
                        X <= 0;
                        Y <= 0;
                    end else begin
                        X_count <= X_count + 1;
                        X <= X_count;
                        Y <= Y_count;
                        color_output <= color;
                    end
                end
            end

            // erase_draw U2(start_X, start_Y, CLOCK_50, KEY[0], load_enable, X, Y,
VGA_COLOR, plot);

    keyboard KEYBOARD (
    // Inputs
    .CLOCK_50(CLOCK_50),
    .KEY(KEY),

    // Bidirectionals
    .PS2_CLK(PS2_CLK),
    .PS2_DAT(PS2_DAT),

    // Outputs
    .LEDR(LEDR), .HEX5(HEX5),
.cln(cln), .dln(dln), .eln(eln), .fln(fln), .gln(gln), .aln(aln), .bln(bln),
.csln(csln), .dsln(dsln), .fsln(fsln), .gsln(gsln), .asln(asln),
.cln2(cln2), .dln2(dln2), .eln2(eln2), .fln2(fln2), .gln2(gln2), .aln2(aln2), .bln2(bln2),
.csln2(csln2), .dsln2(dsln2), .fsln2(fsln2), .gsln2(gsln2), .asln2(asln2),


//for audio
.AUD_ADCDAT(AUD_ADCDAT), .AUD_BCLK(AUD_BCLK),
.AUD_ADCLRCK(AUD_ADCLRCK), .AUD_DACLRCK(AUD_DACLRCK),
.FPGA_I2C_SDAT(FPGA_I2C_SDAT), .AUD_XCK(AUD_XCK),
.AUD_DACDAT(AUD_DACDAT), .FPGA_I2C_SCLK(FPGA_I2C_SCLK)
);




// Bidirectionals
inout           PS2_CLK;
inout           PS2_DAT;
```

```
// Outputs
output  [5:0]    LEDR;
output   [6:0]    HEX5;
output  cln, dln, eln, fln, gln, aln, bln;
output  csln, dsln, fsln, gsln, asln;
output  cln2, dln2, eln2, fln2, gln2, aln2, bln2;
output  csln2, dsln2, fsln2, gsln2, asln2;

/*****************************************************************************
 *                 Internal Wires and Registers Declarations                 *
 *****************************************************************************/

//for audio
input AUD_ADCDAT;
inout AUD_BCLK;
    inout AUD_ADCLRCK;
    inout AUD_DACLRCK;
    inout FPGA_I2C_SDAT;
    output AUD_XCK;
    output AUD_DACDAT;
    output FPGA_I2C_SCLK;

            vga_adapter VGA (

                                .resetn(KEY[0]),

                                .clock(CLOCK_50),

                                .colour(color_output),

                                .x(X),

                                .y(Y),

                                .plot(plot),

                                .VGA_R(VGA_R),

                                .VGA_G(VGA_G),

                                .VGA_B(VGA_B),

                                .VGA_HS(VGA_HS),
```

```
                        .VGA_VS(VGA_VS),
                        .VGA_BLANK_N(VGA_BLANK_N),
                        .VGA_SYNC_N(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 = "no_keys_pressed.mif";

          //counter part2(1'b1, KEY[0], KEY[2], Q);

          // hexadecimaldisplay part2A({load_enable, 3'b000}, HEX0);
          // hexadecimaldisplay part2B(Q[7:4], HEX1);
          //hexadecimaldisplay part2C(Q[11:8], HEX2);
          //hexadecimaldisplay part2D(Q[15:12], HEX3);



endmodule

module counter(enable, clear, clock, Q);
```

```
        input enable, clear, clock;

        output reg [15:0] Q;



        always @ (posedge clock)

          Q <= Q + 1'b1;


endmodule

module three_second(clear, clock, enable);  //enables every three seconds

        input clear, clock;

        output enable;

        // reg [1:0] Q;

        // wire oneSecEnable;

        reg [27:0] fastcount;
        wire enable_sec;
        reg [3:0] three_sec;

        // assign oneSecEnable = (fastcount == 26'd0) ? 1'b1: 1'b0;
        assign enable = (three_sec == 4'b1001) ? 1'b1: 1'b0;

        always @ (posedge clock) begin

      if (~clear || enable)

        fastcount <= 28'd150000000;

      else

        fastcount <= fastcount - 1'b1;

        end

        always @ (posedge clock) begin
```

```
        if (~clear) begin

            three_sec <= 4'b0000;

                    end else if (enable_sec)
                    begin
                                if ( three_sec == 4'b1001 )
                                        three_sec <= 4'b0000;
                                else  three_sec <= three_sec + 1'b1;
                    end
            end



            /*always @ (posedge clock)  begin
                    if (enable) begin
                            Q <= 2'd0;
                    end else if (oneSecEnable) begin
                            Q <= Q + 1'b1;
                    end
            end */

endmodule


module key_prompt_shift_register(

    input clk,              // Clock signal

    input reset,             // Reset signal

            input send_out,                                 // Signal to send out data from register

    input [143:0] in_data,      // 6-bit input data (parallel input)

    output reg [5:0] out_bit        // Output bit (serial-out)

);

            reg [143:0] data;

    always @(posedge clk) begin

        if (reset) begin
```

```
            data <= in_data;  // Reset to queue (start of song)

        end else if (send_out) begin

                                        // Shift out the MSB (most significant bit)

            data <= {data[138:0], 6'd0};  // Shift left by 1 bit

        end

                    out_bit <= data[143:138];

    end
endmodule


module hexadecimaldisplay(C, Display);

            input [3:0] C;

            output [6:0] Display;

            wire [6:0] h;

            wire [1:4] x;

            assign x = C;



            assign h[0] = (x[1] & ~x[3] & ~x[4]) | (x[1] & ~x[2] & ~x[3]) | (x[3] & ~x[4]) | (x[1] & x[2] &
x[3]) | ( ~x[2] & ~x[4]) | (~x[1] & x[3]) | (~x[1] & x[2] & x[4]);

            assign h[1] = (~x[2]& ~x[4]) | (~x[1] & ~x[2]) | (~x[1] & ~x[3] & ~x[4]) | (~x[1] & x[3] & x[4])
| (x[1] & ~x[3] & x[4]);

            assign h[2] = (~x[1] & x[2]) | (~x[3] & x[4]) | (~x[2] & x[3] & x[4]) | (x[1] & ~x[2] & x[3] &
~x[4]) | (x[1] & ~x[2]) | (~x[1] & x[2] & x[3] & ~x[4]) | (~x[1] & ~x[3] & ~x[4]);

            assign h[3] = (~x[1]&  ~x[2] & ~x[3] & ~x[4]) | (x[2] & ~x[3] & x[4]) | (~x[3] & ~x[4] & x[1])
|(x[4] & x[1] & ~x[2]) | (x[2] & x[3] & ~x[4]) | (~x[1] & ~x[2] & x[3]);
```

```
        assign h[4] = (~x[1]&  ~x[2] & ~x[3] & ~x[4]) | (~x[3] & ~x[4] & x[1]) | ( x[3] & x[4] & x[1]) |
(x[3] & ~x[4]) | (x[1] & x[2]);

        assign h[5] = (~x[3] & ~x[4]) | (~x[1] & x[2] & ~x[3]) | (x[1] & x[2] & x[3]) | (x[1] & ~x[2]) |
(x[2] & x[3] & ~x[4]);

        assign h[6] = (x[1] & ~x[2]) | (x[3] & ~x[4]) | (~x[1] & ~x[2] & x[3]) | (x[1] & x[2] & x[4]) |
(~x[1] & x[2] & ~x[3]);

        assign Display = ~h;



endmodule



module regn(R, Resetn, E, Clock, Q);

    input [5:0] R;

    input Resetn, E, Clock;

    output reg [5:0] Q;



    always @(posedge Clock)

        if (!Resetn)

            Q <= 0;

        else if (E)

            Q <= R;

endmodule



/*module letter_drawing(input [5:0] key, input enable, input resetn
```

```
// for A:

        x_coor =
120'b100000111000001010000001100000000111111101111110111110001111110111111100111111101111000000100000011000001100000101000000011;
        y_coor =
105'b011010101101100110111011100001110010011101001110111001111010111110011111100000010000011000010100001100000101000011;



{x, y} <= {8'd124, 7'd89}

);

*/


module key_selection(resetn, key, color0, color1, color2, color3, color4, color5, color6, color7,
color8, color9, color10, color11, color12, color13, color14, CLOCK_50, key_type, color_output);

        input [5:0] key;       // 6-bit key input

    input CLOCK_50;       // 50 MHz clock

    input resetn;


        input [2:0] color0, color1, color2, color3, color4, color5, color6, color7, color8, color9,
color10, color11, color12, color13, color14;

        output reg [2:0] color_output;

    output reg [1:0] key_type;



 always @(posedge CLOCK_50) begin

       if (resetn) begin

                                    case(key)

                                    //white edge tile - C
```

```
                6'b000000: begin

                        key_type <= 2'b00;
                        color_output <= color1;
                end


//white middle - D

                6'b000010: begin

                        key_type <= 2'b01;
                        color_output <= color2;
                end

//white edge - E

                6'b000100: begin

                        key_type <= 2'b00;
                        color_output <= color3;
                end

//white edge - F

                6'b000101: begin

                        key_type <= 2'b00;
                        color_output <= color4;
                end


//white middle - G

                6'b000111: begin

                        key_type <= 2'b01;
                        color_output <= color5;
                end


//white middle - A
```

```
                    6'b001001: begin

                                key_type <= 2'b01;
                                color_output <= color6;
                    end


//white edge - B

                    6'b001011: begin

                                key_type <= 2'b00;
                                color_output <= color7;
                    end

//white edge - C

                    6'b001100: begin

                                key_type <= 2'b00;
                                color_output <= color8;
                    end


//white middle - D

                    6'b001110: begin

                                key_type <= 2'b01;
                                color_output <= color9;
                    end

//white edge - E

                    6'b010000: begin

                                key_type <= 2'b00;
                                color_output <= color10;
                    end

//white edge - F

                    6'b010001: begin
```

```
                                key_type <= 2'b00;
                                color_output <= color11;
                        end


//white middle - G

                        6'b010011: begin

                                key_type <= 2'b01;
                                color_output <= color12;
                        end


//white middle - A

                        6'b010101: begin

                                key_type <= 2'b01;
                                color_output <= color13;
                        end


//white edge - B

                        6'b010111: begin

                                key_type <= 2'b00;
                                color_output <= color14;
                        end

                        default: begin
                                color_output <= color0;
                        end

                endcase

                end


                else begin

color_output <= color0;
```

```
                        end

                    end
endmodule


module erase_draw(
    input [7:0] x_coor,       // 6-bit key input
            input [6:0] y_coor,
    input CLOCK_50,        // 50 MHz clock
    input resetn,         // Reset signal
            input enable,  // signal to start drawing
           output reg [7:0] x_out,   // X coordinate of the dot
     // X coordinate of the dot
    output reg [6:0] y_out,


      // Y coordinate of the dot
           output reg [2:0] VGA_COLOR,
           output reg plot
);

    // State definitions
    reg [1:0] state, next_state;
    parameter IDLE = 2'b00, ERASE = 2'b01, DRAW = 2'b10;
```

```
        reg [7:0] prev_x;  // Previous X coordinate

reg [6:0] prev_y; // Previous Y coordinate

        reg [7:0] x;  //current coordinates
        reg [6:0] y;

        reg eraseDone;
        reg eraseEnable;
        reg drawDone;

        always @(posedge CLOCK_50) begin
                if (drawDone || eraseDone)
                        plot <= 1'b1;

                else
                        plot <= 1'b0;
        end


        always @(posedge CLOCK_50) begin
                if (enable) begin
                        eraseEnable <= enable;
                end
                else if (drawDone) begin
                        eraseEnable <= enable;
                end
        end


 // FSM logic

 always @(posedge CLOCK_50) begin

    if (!resetn) begin

        state <= IDLE;

    end else begin

        state <= next_state;

    end
```

```
end



always @(*) begin
    case(state)
        IDLE: next_state = eraseEnable ? ERASE : IDLE;
        ERASE: next_state = eraseDone ? DRAW : ERASE;
        DRAW: next_state = drawDone ? IDLE : DRAW;


        default: next_state = IDLE;
    endcase
end

// Store the previous dot coordinates before drawing the new one
always @(posedge CLOCK_50) begin
    if (!resetn) begin
        prev_x <= 8'd0;
        prev_y <= 7'd0;
    end else if (state == DRAW) begin
        prev_x <= x_coor;
        prev_y <= y_coor;
    end
```

```
 end

always @(posedge CLOCK_50) begin

    case(state)

        ERASE: begin

                                        if (prev_y == 7'd114) begin

                                                VGA_COLOR <= 3'b111;

                                        end

          // Erase previous dot by setting color to background (black)

          else if (prev_y == 7'd89) begin

                                                VGA_COLOR <= 3'b000;

                                        end

                                        x_out <= prev_x;
                                        y_out <= prev_y;
                                        eraseDone <= 1'b1;

        end

        DRAW: begin



                                        if (y_coor == 7'd114) begin

                                                VGA_COLOR <= 3'b100;



                                        end


          else if (y_coor == 7'd89) begin
```

```
                                        VGA_COLOR <= 3'b110;

                                    end

                                    x_out <= x_coor;
                                    y_out <= y_coor;
                                drawDone <= 1'b1;

            end

        endcase

    end

endmodule

module keyboard (
    // Inputs
    CLOCK_50,
    KEY,

    // Bidirectionals
    PS2_CLK,
    PS2_DAT,

    // Outputs
    LEDR, HEX5,
cln, dln, eln, fln, gln, aln, bln,
csln, dsln, fsln, gsln, asln,
cln2, dln2, eln2, fln2, gln2, aln2, bln2,
csln2, dsln2, fsln2, gsln2, asln2,


//for audio
AUD_ADCDAT, AUD_BCLK, AUD_ADCLRCK, AUD_DACLRCK, FPGA_I2C_SDAT, AUD_XCK,
AUD_DACDAT, FPGA_I2C_SCLK
);

/*****************************************************************************
 *                      Parameter Declarations                               *
 *****************************************************************************/


/*****************************************************************************
```

```
 *                          Port Declarations                          *
 ******************************************************************************/

// Inputs
input              CLOCK_50;
input       [3:0]    KEY;

// Bidirectionals
inout              PS2_CLK;
inout              PS2_DAT;

// Outputs
output  reg      [5:0]    LEDR;
output reg       [6:0]    HEX5;
output reg cln, dln, eln, fln, gln, aln, bln;
output reg csln, dsln, fsln, gsln, asln;
output reg cln2, dln2, eln2, fln2, gln2, aln2, bln2;
output reg csln2, dsln2, fsln2, gsln2, asln2;

/*****************************************************************************
 *               Internal Wires and Registers Declarations                   *
 *****************************************************************************/

// Internal Wires
wire       [7:0]    ps2_key_data;
wire                ps2_key_pressed;
wire       [4:0]    ps2_count;  // Declare ps2_count as a wire
//for audio
input AUD_ADCDAT;
inout AUD_BCLK;
   inout AUD_ADCLRCK;
   inout AUD_DACLRCK;
   inout FPGA_I2C_SDAT;
   output AUD_XCK;
   output AUD_DACDAT;
   output FPGA_I2C_SCLK;

// Internal Registers
reg        [7:0]    last_data_received;

// State Machine Registers

/*****************************************************************************
 *                    Finite State Machine(s)                                *
```

```
 *****************************************************************************/


/*****************************************************************************
 *                             Sequential Logic                              *
 *****************************************************************************/

always @(posedge CLOCK_50) begin
    if (KEY[0] == 1'b0)
        last_data_received <= 8'h00;
    else if (ps2_key_pressed == 1'b1)
        last_data_received <= ps2_key_data;
end

/*****************************************************************************
 *                           Combinational Logic                             *
 *****************************************************************************/



/*****************************************************************************
 *                            Internal Modules                               *
 *****************************************************************************/

PS2_Controller PS2 (
    // Inputs
    .CLOCK_50               (CLOCK_50),
    .reset                  (~KEY[0]),

    // Bidirectionals
    .PS2_CLK                (PS2_CLK),
    .PS2_DAT                (PS2_DAT),

    // Outputs
    .received_data          (ps2_key_data),
    .received_data_en       (ps2_key_pressed),
 // Pass the ps2_count output to the top module
);


/*****************************************************************************
 *                  LED Control Logic based on Key Presses                   *
 *****************************************************************************/
```

```
always @(posedge ps2_key_pressed) begin

//press Q
    if ( ps2_key_data == 8'h15) begin
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;



end

//press W
    else if ( ps2_key_data == 8'h1D) begin
        LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
//E
else if ( ps2_key_data == 8'h24) begin
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
//R
else if ( ps2_key_data == 8'h2D) begin
        LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
//T
else if ( ps2_key_data == 8'h2C) begin
```

```
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
//Y
else if ( ps2_key_data == 8'h35) begin
        LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
//U
else if ( ps2_key_data == 8'h3C) begin
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
//I
else if ( ps2_key_data == 8'h43) begin
        LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h44) begin
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b1;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h4D) begin
        LEDR[0] <= 1'b1;
```

```
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b1;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h1C) begin
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b1;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h1B) begin
        LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b1;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h23) begin
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b1;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h2B) begin
        LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b1;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h34) begin
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b1;
 LEDR[4] <= 1'b0;
```

```
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h33) begin
        LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b1;
 LEDR[4] <= 1'b0;
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h3B) begin
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b1;
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h42) begin
        LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b1;
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h4B) begin
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b1;
 LEDR[5] <= 1'b0;
end
else if ( ps2_key_data == 8'h1A) begin
        LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b0;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b1;
 LEDR[5] <= 1'b0;
end

else if ( ps2_key_data == 8'h22) begin
```

```
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b1;
 LEDR[5] <= 1'b0;
end

else if ( ps2_key_data == 8'h21) begin
        LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b0;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b1;
 LEDR[5] <= 1'b0;
end

else if ( ps2_key_data == 8'h2A) begin
        LEDR[0] <= 1'b0;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b1;
 LEDR[5] <= 1'b0;
end

else if ( ps2_key_data == 8'h32) begin
        LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b0;
 LEDR[4] <= 1'b1;
 LEDR[5] <= 1'b0;
end


//when no key is pressed all LEDR is turned on
else  begin
 LEDR[0] <= 1'b1;
 LEDR[1] <= 1'b1;
 LEDR[2] <= 1'b1;
 LEDR[3] <= 1'b1;
 LEDR[4] <= 1'b1;
 LEDR[5] <= 1'b1;
```

```
end
end


always @(posedge ps2_key_pressed) begin
   if ( ps2_key_data == 8'h16)
      HEX5 <= 7'b1111001;
else if ( ps2_key_data == 8'h1E)
      HEX5 <= 7'b0100100;
else if ( ps2_key_data == 8'h26)
      HEX5 <= 7'b0110000;
   else
      HEX5 <= 7'b1000000;
end


always @(posedge ps2_key_pressed) begin
//cIn Q
   if ( ps2_key_data == 8'h15) begin
      cIn <= 1'b1;

end

//csIn W
   else if ( ps2_key_data == 8'h1D) begin
      csIn <= 1'b1;
end
//dIn E
else if ( ps2_key_data == 8'h24) begin
      dIn <= 1'b1;
end
//dsIn R
else if ( ps2_key_data == 8'h2D) begin
      dsIn <= 1'b1;
end
//eIn T
else if ( ps2_key_data == 8'h2C) begin
      eIn <= 1'b1;
end
//fIn Y
else if ( ps2_key_data == 8'h35) begin
      fIn <= 1'b1;
end
//fsIn U
```

```
else if ( ps2_key_data == 8'h3C) begin
	fsIn <= 1'b1;
end
//gIn I
else if ( ps2_key_data == 8'h43) begin
	gIn <= 1'b1;
end
//gsIn O
else if ( ps2_key_data == 8'h44) begin
	gsIn <= 1'b1;
end
//aIn P
else if ( ps2_key_data == 8'h4D) begin
	aIn <= 1'b1;
end
//asIn A
else if ( ps2_key_data == 8'h1C) begin
	asIn <= 1'b1;
end
//bIn S
else if ( ps2_key_data == 8'h1B) begin
	bIn <= 1'b1;
end
//cIn2 D
else if ( ps2_key_data == 8'h23) begin
	cIn2 <= 1'b1;
end
//csIn2 F
else if ( ps2_key_data == 8'h2B) begin
	csIn2 <= 1'b1;
end
//dIn2 G
else if ( ps2_key_data == 8'h34) begin
	dIn2 <= 1'b1;

end
//dsIn2 H
else if ( ps2_key_data == 8'h33) begin
	dsIn2 <= 1'b1;
end
//eIn2 J
else if ( ps2_key_data == 8'h3B) begin
	eIn2 <= 1'b1;
end
```

```
//fIn2 K
else if ( ps2_key_data == 8'h42) begin
	fIn2 <= 1'b1;
end
//fsIn2 L
else if ( ps2_key_data == 8'h4B) begin
	fsIn2 <= 1'b1;
end
//gIn2 Z
else if ( ps2_key_data == 8'h1A) begin
	gIn2 <= 1'b1;
end
//gsIn2 X
else if ( ps2_key_data == 8'h22) begin
	gsIn2 <= 1'b1;
end
//aIn2 C
else if ( ps2_key_data == 8'h21) begin
	aIn2 <= 1'b1;
end
//asIn2 V
else if ( ps2_key_data == 8'h2A) begin
	asIn2 <= 1'b1;
end
//bIn2 B
else if ( ps2_key_data == 8'h32) begin
	bIn2 <= 1'b1;
end


//when no key is pressed all LEDR is turned on
else  begin
//first octave
//white keys
 cIn <= 1'b0;
 dIn <= 1'b0;
 eIn <= 1'b0;
 fIn <= 1'b0;
 gIn <= 1'b0;
 aIn <= 1'b0;
 bIn <= 1'b0;
//black keys
csIn <= 1'b0;
dsIn <= 1'b0;
```

```
fsIn <= 1'b0;
gsIn <= 1'b0;
asIn <= 1'b0;
//second octave
//white keys
cIn2 <= 1'b0;
 dIn2 <= 1'b0;
 eIn2 <= 1'b0;
 fIn2 <= 1'b0;
 gIn2 <= 1'b0;
 aIn2 <= 1'b0;
 bIn2 <= 1'b0;

//black keys
csIn2 <= 1'b0;
dsIn2 <= 1'b0;
fsIn2 <= 1'b0;
gsIn2 <= 1'b0;
asIn2 <= 1'b0;

end
end

    audio Audio (
        .CLOCK_50(CLOCK_50),
        .KEY(KEY[0]),
        .AUD_ADCDAT(AUD_ADCDAT),
        .AUD_BCLK(AUD_BCLK),
        .AUD_ADCLRCK(AUD_ADCLRCK),
        .AUD_DACLRCK(AUD_DACLRCK),
        .FPGA_I2C_SDAT(FPGA_I2C_SDAT),
        .AUD_XCK(AUD_XCK),
        .AUD_DACDAT(AUD_DACDAT),
        .FPGA_I2C_SCLK(FPGA_I2C_SCLK),
        .cIn(cIn), .dIn(dIn), .eIn(eIn),
        .fIn(fIn), .gIn(gIn), .aIn(aIn),
        .bIn(bIn),
        .csIn(csIn),
        .dsIn(dsIn),
        .fsIn(fsIn),
        .gsIn(gsIn),
        .asIn(asIn),
 .cIn2(cIn2), .dIn2(dIn2), .eIn2(eIn2),
        .fIn2(fIn2), .gIn2(gIn2), .aIn2(aIn2),
```

```
        .bIn2(bIn2),
        .csIn2(csIn2),
        .dsIn2(dsIn2),
        .fsIn2(fsIn2),
        .gsIn2(gsIn2),
        .asIn2(asIn2)


    );

endmodule


module audio (
// Inputs
CLOCK_50,
KEY,

AUD_ADCDAT,

// Bidirectionals
AUD_BCLK,
AUD_ADCLRCK,
AUD_DACLRCK,

FPGA_I2C_SDAT,

// Outputs
AUD_XCK,
AUD_DACDAT,

FPGA_I2C_SCLK,
cIn, dIn, eIn, fIn, gIn, aIn, bIn,
csIn, dsIn, fsIn, gsIn, asIn,
cIn2, dIn2, eIn2, fIn2, gIn2, aIn2, bIn2,
csIn2, dsIn2, fsIn2, gsIn2, asIn2
);

/*****************************************************************************
 *                           Parameter Declarations                          *
 *****************************************************************************/


/*****************************************************************************
```

```
 *                            Port Declarations                                *
 *********************************************************************************/
// Inputs
input CLOCK_50;
input [3:0] KEY;

input AUD_ADCDAT;

// Bidirectionals
inout AUD_BCLK;
inout AUD_ADCLRCK;
inout AUD_DACLRCK;

inout FPGA_I2C_SDAT;

// Outputs
output AUD_XCK;
output AUD_DACDAT;

output FPGA_I2C_SCLK;

//first octave
input cIn, dIn, eIn, fIn, gIn, aIn, bIn;
input csIn, dsIn, fsIn, gsIn, asIn;

//second octave
input cIn2, dIn2, eIn2, fIn2, gIn2, aIn2, bIn2;
input csIn2, dsIn2, fsIn2, gsIn2, asIn2;

/*********************************************************************************
 *                Internal Wires and Registers Declarations                      *
 *********************************************************************************/
// Internal Wires
wire audio_in_available;
wire [31:0] left_channel_audio_in;
wire [31:0] right_channel_audio_in;
wire read_audio_in;

wire audio_out_allowed;
wire [31:0] left_channel_audio_out;
wire [31:0] right_channel_audio_out;
wire write_audio_out;

// Internal Registers
```

```
reg [18:0] cDelay, dDelay, eDelay, fDelay, gDelay, aDelay, bDelay;
wire [18:0] C3, D3, E3, F3, G3, A3, B3;

reg[18:0] csDelay, dsDelay, fsDelay, gsDelay, asDelay;
wire [18:0] C3s, D3s, F3s, G3s, A3s;

reg cSnd, dSnd, eSnd, fSnd, gSnd, aSnd, bSnd;
reg csSnd, dsSnd, fsSnd, gsSnd, asSnd;

//second octave

reg [18:0] cDelay2, dDelay2, eDelay2, fDelay2, gDelay2, aDelay2, bDelay2;
wire [18:0] C4, D4, E4, F4, G4, A4, B4;

reg[18:0] csDelay2, dsDelay2, fsDelay2, gsDelay2, asDelay2;
wire [18:0] C4s, D4s, F4s, G4s, A4s;

reg cSnd2, dSnd2, eSnd2, fSnd2, gSnd2, aSnd2, bSnd2;
reg csSnd2, dsSnd2, fsSnd2, gsSnd2, asSnd2;


// State Machine Registers

/*****************************************************************************
 *                      Finite State Machine(s)                              *
 *****************************************************************************/


/*****************************************************************************
 *                            Sequential Logic                               *
 *****************************************************************************/
//second octave
 //white tiles
always @(posedge CLOCK_50)
if(cDelay2 == C4) begin
cDelay2 <= 0;
cSnd2 <= !cSnd2;
end else cDelay2 <= cDelay2 + 1;

always @(posedge CLOCK_50)
if(dDelay2 == D4) begin
dDelay2 <= 0;
```

```
dSnd2 <= !dSnd2;
end else dDelay2 <= dDelay2 + 1;

always @(posedge CLOCK_50)
if(eDelay2 == E4) begin
eDelay2 <= 0;
eSnd2 <= !eSnd2;
end else eDelay2 <= eDelay2 + 1;

always @(posedge CLOCK_50)
if(fDelay2 == F4) begin
fDelay2 <= 0;
fSnd2 <= !fSnd2;
end else fDelay2 <= fDelay2 + 1;

always @(posedge CLOCK_50)
if(gDelay2 == G4) begin
gDelay2 <= 0;
gSnd2 <= !gSnd2;
end else gDelay2 <= gDelay2 + 1;

always @(posedge CLOCK_50)
if(aDelay2 == A4) begin
aDelay2 <= 0;
aSnd2 <= !aSnd2;
end else aDelay2 <= aDelay2 + 1;

always @(posedge CLOCK_50)
if(bDelay2 == B4) begin
bDelay2 <= 0;
bSnd2 <= !bSnd2;
end else bDelay2 <= bDelay2 + 1;

//black tiles

always @(posedge CLOCK_50)
if(csDelay2 == C4s) begin
csDelay2 <= 0;
csSnd2 <= !csSnd2;
end else csDelay2 <= csDelay2 + 1;

always @(posedge CLOCK_50)
if(dsDelay2 == D4s) begin
dsDelay2 <= 0;
```

```
dsSnd2 <= !dsSnd2;
end else dsDelay2 <= dsDelay2 + 1;

always @(posedge CLOCK_50)
if(fsDelay2 == F4s) begin
fsDelay2 <= 0;
fsSnd2 <= !fsSnd2;
end else fsDelay2 <= fsDelay2 + 1;

always @(posedge CLOCK_50)
if(gsDelay2 == G4s) begin
gsDelay2 <= 0;
gsSnd2 <= !gsSnd2;
end else gsDelay2 <= gsDelay2 + 1;

always @(posedge CLOCK_50)
if(asDelay2 == A4s) begin
asDelay2 <= 0;
asSnd2 <= !asSnd2;
end else asDelay2 <= asDelay2 + 1;


//first octave
 //white tiles
always @(posedge CLOCK_50)
if(cDelay == C3) begin
cDelay <= 0;
cSnd <= !cSnd;
end else cDelay <= cDelay + 1;

always @(posedge CLOCK_50)
if(dDelay == D3) begin
dDelay <= 0;
dSnd <= !dSnd;
end else dDelay <= dDelay + 1;

always @(posedge CLOCK_50)
if(eDelay == E3) begin
eDelay <= 0;
eSnd <= !eSnd;
end else eDelay <= eDelay + 1;

always @(posedge CLOCK_50)
if(fDelay == F3) begin
```

```
fDelay <= 0;
fSnd <= !fSnd;
end else fDelay <= fDelay + 1;

always @(posedge CLOCK_50)
if(gDelay == G3) begin
gDelay <= 0;
gSnd <= !gSnd;
end else gDelay <= gDelay + 1;

always @(posedge CLOCK_50)
if(aDelay == A3) begin
aDelay <= 0;
aSnd <= !aSnd;
end else aDelay <= aDelay + 1;

always @(posedge CLOCK_50)
if(bDelay == B3) begin
bDelay <= 0;
bSnd <= !bSnd;
end else bDelay <= bDelay + 1;

//black tiles

always @(posedge CLOCK_50)
if(csDelay == C3s) begin
csDelay <= 0;
csSnd <= !csSnd;
end else csDelay <= csDelay + 1;

always @(posedge CLOCK_50)
if(dsDelay == D3s) begin
dsDelay <= 0;
dsSnd <= !dsSnd;
end else dsDelay <= dsDelay + 1;

always @(posedge CLOCK_50)
if(fsDelay == F3s) begin
fsDelay <= 0;
fsSnd <= !fsSnd;
end else fsDelay <= fsDelay + 1;

always @(posedge CLOCK_50)
if(gsDelay == G3s) begin
```

```
gsDelay <= 0;
gsSnd <= !gsSnd;
end else gsDelay <= gsDelay + 1;

always @(posedge CLOCK_50)
if(asDelay == A3s) begin
asDelay <= 0;
asSnd <= !asSnd;
end else asDelay <= asDelay + 1;
/*****************************************************************************
 *                          Combinational Logic                              *
 *****************************************************************************/

 //first octave
assign C3 = 18'd190080;  // 262 Hz
assign D3 = 18'd170068;  // 294 Hz
assign E3 = 18'd151515;  // 330 Hz
assign F3 = 18'd143003;  // 349 Hz
assign G3 = 18'd127551;  // 392 Hz
assign A3 = 18'd113636;  // 440 Hz
assign B3 = 18'd101214;  // 494 Hz

assign C3s = 18'd180129;  // 277 Hz
assign D3s = 18'd160458;  // 311 Hz
assign F3s = 18'd135135;  // 370 Hz
assign G3s = 18'd120481;  // 415 Hz
assign A3s = 18'd107290;  // 466 Hz

//second octave
assign C4 = 18'd95238;    // 524 Hz
assign D4 = 18'd85470;    // 588 Hz
assign E4 = 18'd75758;    // 660 Hz
assign F4 = 18'd71644;    // 698 Hz
assign G4 = 18'd63723;    // 784 Hz
assign A4 = 18'd56818;    // 880 Hz
assign B4 = 18'd50584;    // 988 Hz

// Frequencies for C4s, D4s, E4s, etc. (sharp notes of the 2nd octave)
assign C4s = 18'd90177;   // 554 Hz
assign D4s = 18'd80290;   // 622 Hz
assign F4s = 18'd67567;   // 740 Hz
assign G4s = 18'd60240;   // 830 Hz
assign A4s = 18'd53598;   // 932 Hz
```

```
wire [31:0] sound = ((cIn == 0) ? 0 : cSnd ? 32'd100000000 : -32'd100000000)+((dIn == 0) ? 0 :
dSnd ? 32'd100000000 : -32'd100000000)+
((eIn == 0) ? 0 : eSnd ? 32'd100000000 : -32'd100000000)+((fIn == 0) ? 0 : fSnd ?
32'd100000000 : -32'd100000000)+
((gIn == 0) ? 0 : gSnd ? 32'd100000000 : -32'd100000000)+((aIn == 0) ? 0 : aSnd ?
32'd100000000 : -32'd100000000)+
((bIn == 0) ? 0 : bSnd ? 32'd100000000 : -32'd100000000)+((csIn == 0) ? 0 : csSnd ?
32'd100000000 : -32'd100000000)+
((dsIn == 0) ? 0 : dsSnd ? 32'd100000000 : -32'd100000000)+((fsIn == 0) ? 0 : fsSnd ?
32'd100000000 : -32'd100000000)+
((gsIn == 0) ? 0 : gsSnd ? 32'd100000000 : -32'd100000000)+((asIn == 0) ? 0 : asSnd ?
32'd100000000 : -32'd100000000)+
((cIn2 == 0) ? 0 : cSnd2 ? 32'd100000000 : -32'd100000000)+((dIn2 == 0) ? 0 : dSnd2 ?
32'd100000000 : -32'd100000000)+
((eIn2 == 0) ? 0 : eSnd2 ? 32'd100000000 : -32'd100000000)+((fIn2 == 0) ? 0 : fSnd2 ?
32'd100000000 : -32'd100000000)+
((gIn2 == 0) ? 0 : gSnd2 ? 32'd100000000 : -32'd100000000)+((aIn2 == 0) ? 0 : aSnd2 ?
32'd100000000 : -32'd100000000)+
((bIn2 == 0) ? 0 : bSnd2 ? 32'd100000000 : -32'd100000000)+((csIn2 == 0) ? 0 : csSnd2 ?
32'd100000000 : -32'd100000000)+
((dsIn2 == 0) ? 0 : dsSnd2 ? 32'd100000000 : -32'd100000000)+((fsIn2 == 0) ? 0 : fsSnd2 ?
32'd100000000 : -32'd100000000)+
((gsIn2 == 0) ? 0 : gsSnd2 ? 32'd100000000 : -32'd100000000)+((asIn2 == 0) ? 0 : asSnd2 ?
32'd100000000 : -32'd100000000);


assign read_audio_in = audio_in_available & audio_out_allowed;

assign left_channel_audio_out = left_channel_audio_in+sound;
assign right_channel_audio_out = right_channel_audio_in+sound;
assign write_audio_out = audio_in_available & audio_out_allowed;

/*****************************************************************************
 *                             Internal Modules                              *
 *****************************************************************************/

Audio_Controller Audio_Controller (
// Inputs
.CLOCK_50 (CLOCK_50),
.reset (~KEY[0]),

.clear_audio_in_memory (),
.read_audio_in (read_audio_in),
```

```
.clear_audio_out_memory (),
.left_channel_audio_out (left_channel_audio_out),
.right_channel_audio_out (right_channel_audio_out),
.write_audio_out (write_audio_out),

.AUD_ADCDAT (AUD_ADCDAT),

// Bidirectionals
.AUD_BCLK (AUD_BCLK),
.AUD_ADCLRCK (AUD_ADCLRCK),
.AUD_DACLRCK (AUD_DACLRCK),


// Outputs
.audio_in_available (audio_in_available),
.left_channel_audio_in (left_channel_audio_in),
.right_channel_audio_in (right_channel_audio_in),

.audio_out_allowed (audio_out_allowed),

.AUD_XCK (AUD_XCK),
.AUD_DACDAT (AUD_DACDAT)

);

avconf #(.USE_MIC_INPUT(1)) avc (
.FPGA_I2C_SCLK (FPGA_I2C_SCLK),
.FPGA_I2C_SDAT (FPGA_I2C_SDAT),
.CLOCK_50 (CLOCK_50),
.reset (~KEY[0])
);

endmodule
```