Nicholas Marcopoli and Thomas Eckburg
Logic Design Group TU21
March 23, 2018

Midterm Project - Metronome

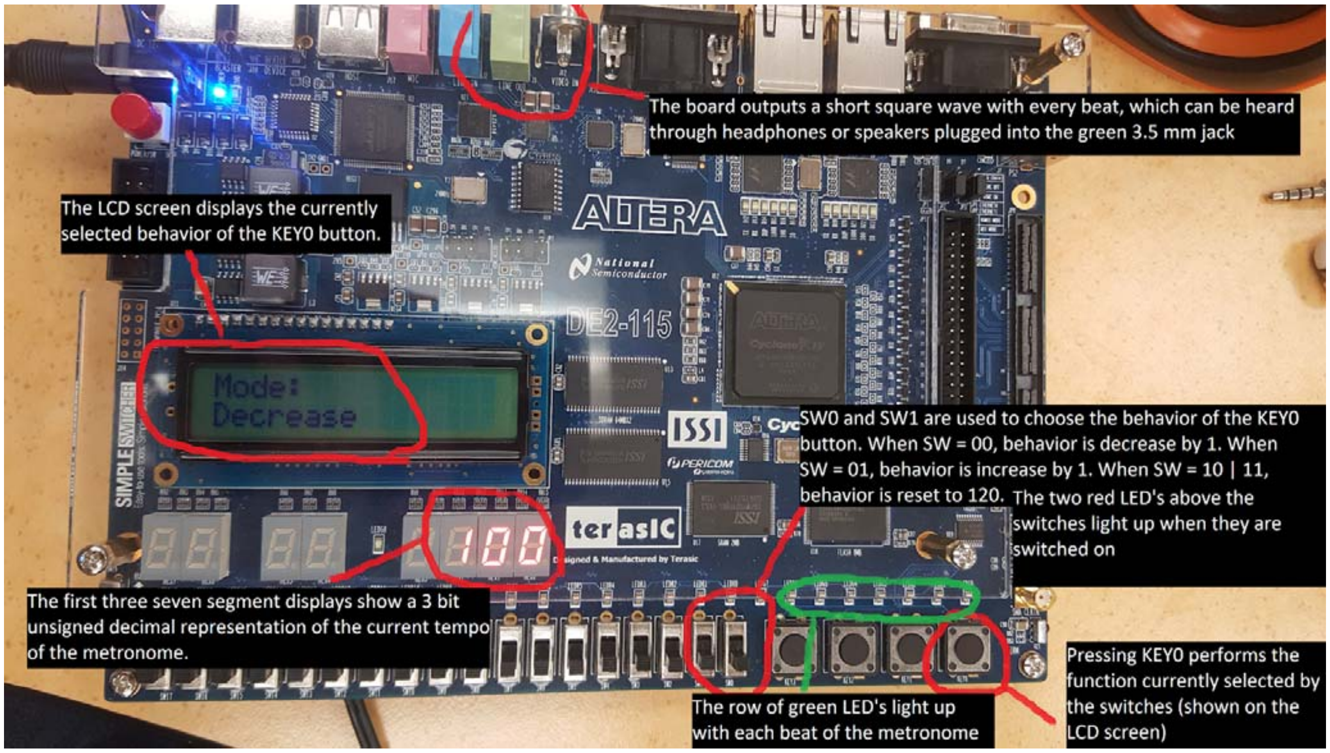
Member Contribution

5 points to each member - Nick and Thomas.

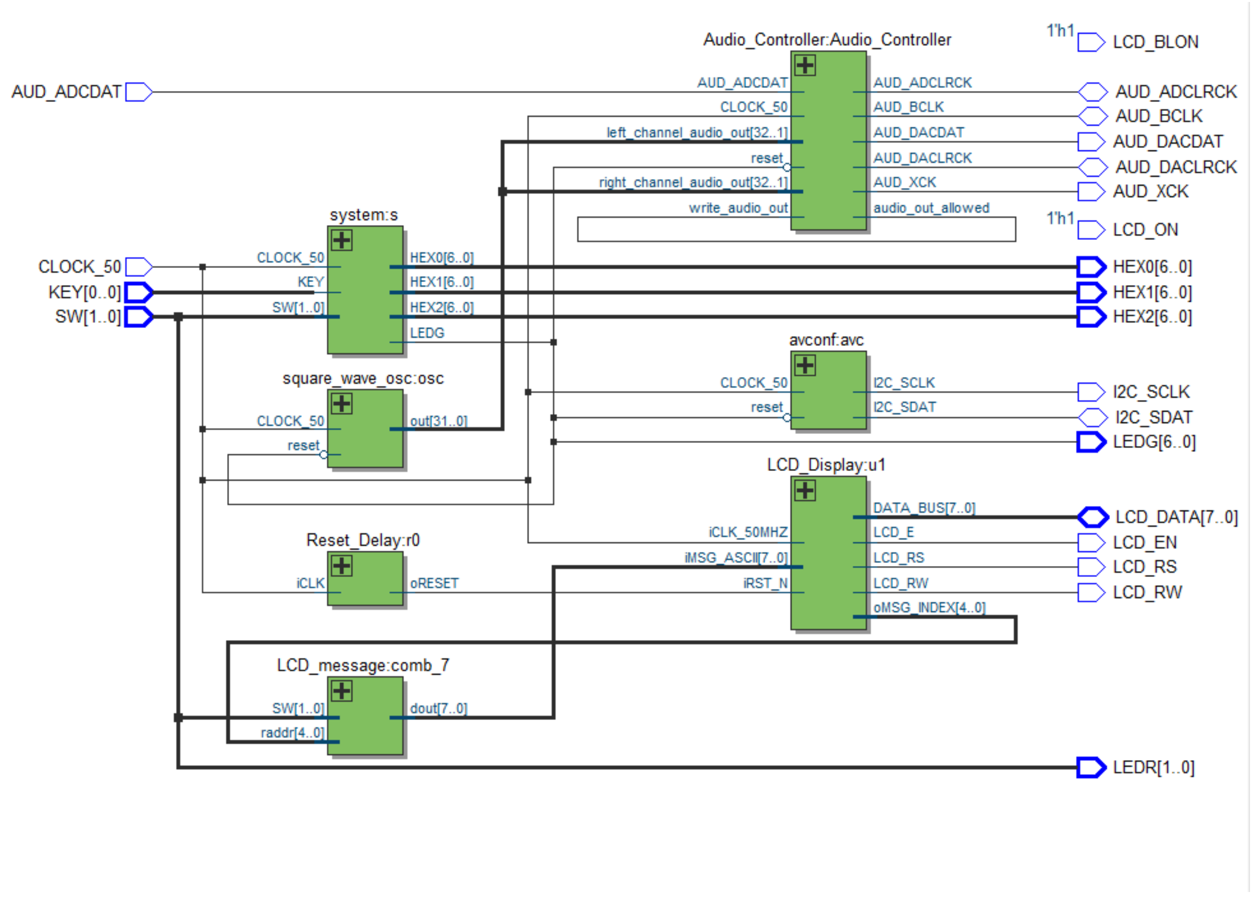
Executive Summary

The Metronome Project uses LEDs, switches, a debounced button, 7-segment displays, the LCD display, audio, and a memory register to create a metronome with both visual and audio beats. The user can select the number of Beats Per Minute (BPM) by using the debounced button, KEY0, and the switches, SW0 and SW1. By switching SW1 and SW0 to the down position, the user can use KEY0 to decrease the BPM, shown as three decimal digits on HEX0, HEX1, and HEX2. By switching SW1 to the down position and SW0 to the up position, the user can use KEY0 to increase the BPM. By switching SW1 to the up position (SW0 can be in either position), the user can use KEY0 to reset the BPM to 120. The red LEDs above the switches light up when the user has put them in the up position. The current mode - Increase, Decrease, or Reset - is indicated to the user on the LCD display. The green LEDs above the buttons flash and the audio beeps with the timing indicated by the 7-segment displays.

This diagram highlights the features of our project:


The board outputs a short square wave with every beat, which can be heard through headphones or speakers plugged into the green 3.5 mm jack
The LCD screen displays the currently selected behavior of the KEY0 button.
Mode: Decrease
SW0 and SW1 are used to choose the behavior of the KEY0 button. When SW = 00, behavior is decrease by 1. When SW = 01, behavior is increase by 1. When SW = 10 | 11, behavior is reset to 120.
The two red LED's above the switches light up when they are switched on
The first three seven segment displays show a 3 bit unsigned decimal representation of the current tempo of the metronome.
100
The row of green LED's light up with each beat of the metronome
Pressing KEY0 performs the function currently selected by the switches (shown on the LCD screen)

The top level module in our project is, of course, *system_de2*. It takes inputs from the switches SW1 and SW0, the button KEY0, and the system clock. It sends outputs to the first 7 green LEDs (LEDG0-6), the first two red LEDs (LEDR0-1), and the first three seven segment hex displays (HEX0-2). It also includes all of the necessary inputs and outputs for the audio controllers, as well as those for the LCD display. It contains three modules related to the LCD screen, one for initialization, one to select a displayed message, and one for actually displaying the text. It also contains the module `system`, which contains most of the logic for our code. Finally, it contains the necessary modules for outputting audio as found on the Sakai Verilog archive. `system_de2` also assigns a few values, most notably the red and green LEDs. Here is an RTL view of `system_de2`, followed by the Verilog code itself:

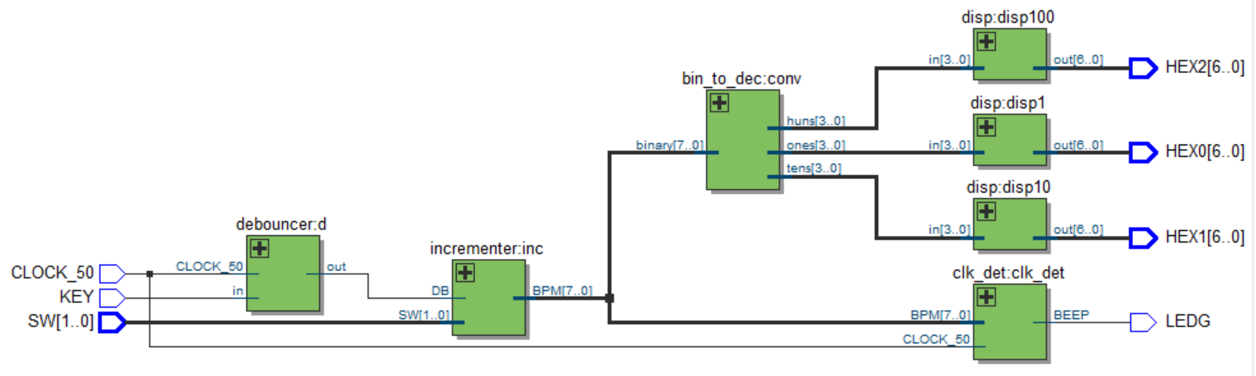


```
module system_de2(
    input [1:0] SW,
    input [0:0]     KEY,
    input CLOCK_50,

    // Audio Items
    input       AUD_ADCDAT,
    inout       AUD_BCLK,
    inout       AUD_ADCLRCK,
    inout       AUD_DACLRCK,
    inout       I2C_SDAT,
    output      AUD_XCK,
 output     AUD_DACDAT,
 output     I2C_SCLK,
 // End of Audio Items

 // LCD Items
     //LCD Module 16X2
 output                     LCD_ON,        // LCD Power ON/OFF
 output                     LCD_BLON,    // LCD Back Light ON/OFF
 output                     LCD_RW,        // LCD Read/Write Select, 0 = Write, 1 = Read
 output                     LCD_EN,        // LCD Enable
 output                     LCD_RS,        // LCD Command/Data Select, 0 = Command, 1 = Data
 inout [7:0]         LCD_DATA,      // LCD Data bus 8 bits

 // End of LCD items

 output [6:0] LEDG,
 output [1:0] LEDR,
 output [6:0] HEX0,
 output [6:0] HEX1,
 output [6:0] HEX2);

 assign LEDR = { SW };

 /******* Audio Items *******/
 wire                             audio_out_allowed;
wire        [31:0]   osc_out;



 /******* LCD Initialization *******/
 wire DLY_RST;

 // reset delay gives some time for peripherals to initialize
Reset_Delay r0 (
    .iCLK           (CLOCK_50),
    .oRESET    (DLY_RST)
 );

 assign    LCD_ON    =    1'b1;
 assign    LCD_BLON    =    1'b1;

 wire [4:0] disp_addr;
 wire [7:0] disp_data;

 LCD_message (
     .SW     (SW),
     .raddr    (disp_addr),
     .dout          (disp_data)
 );

 LCD_Display u1(
 // Host Side
     .iCLK_50MHZ    (CLOCK_50),
     .iRST_N        (DLY_RST),
     .oMSG_INDEX    (disp_addr),
     .iMSG_ASCII    (disp_data),
 // LCD Side
     .DATA_BUS    (LCD_DATA),
     .LCD_RW        (LCD_RW),
     .LCD_E         (LCD_EN),
     .LCD_RS        (LCD_RS)
 );



 /******* System Initialization *******/
```

```
    system s(
         .SW            (SW),
         .CLOCK_50    (CLOCK_50),
         .KEY                   (KEY),
         .LEDG                  (LEDG[0]),
         .HEX0                  (HEX0),
         .HEX1                  (HEX1),
         .HEX2                  (HEX2));

    assign    LEDG[1] = LEDG[0];
    assign     LEDG[2] = LEDG[0];
    assign     LEDG[3] = LEDG[0];
    assign     LEDG[4] = LEDG[0];
    assign     LEDG[5] = LEDG[0];
    assign     LEDG[6] = LEDG[0];



    /******* Audio Module Initialization *******/
    square_wave_osc osc (
        .CLOCK_50                                    (CLOCK_50),
        .reset                                              (~LEDG[0]),
        .out                          (osc_out)
   );

    Audio_Controller Audio_Controller (
        // Inputs
        .CLOCK_50                                    (CLOCK_50),
        .reset                                              (~LEDG[0]),
        .left_channel_audio_out        (osc_out),
        .right_channel_audio_out    (osc_out),
        .write_audio_out                    (audio_out_allowed),
        .AUD_ADCDAT                                 (AUD_ADCDAT),
        // Bidirectionals
        .AUD_BCLK                                   (AUD_BCLK),
        .AUD_ADCLRCK                        (AUD_ADCLRCK),
        .AUD_DACLRCK                        (AUD_DACLRCK),
        // Outputs
        .audio_out_allowed             (audio_out_allowed),
        .AUD_XCK                                    (AUD_XCK),
        .AUD_DACDAT                                 (AUD_DACDAT),
   );

   avconf avc (
        .I2C_SCLK                            (I2C_SCLK),
        .I2C_SDAT                            (I2C_SDAT),
        .CLOCK_50                            (CLOCK_50),
        .reset                                       (~LEDG[0])
   );

Endmodule
```

The Audio modules are unchanged and will not be included in this report. The only LCD module which was changed is the LCD_message module, which will be detailed later.

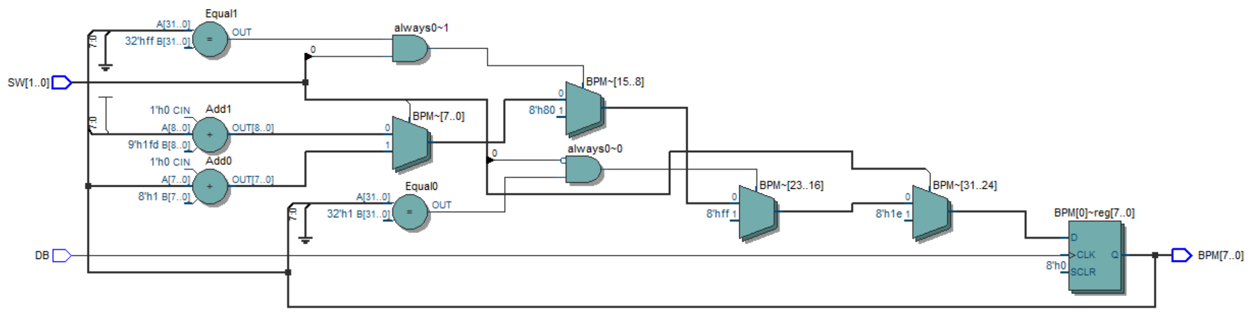
The next module is `system`, which performs most of the metronome logic. The inputs and outputs share names with their counterparts in `system_de2`. `System` first debounces (the debouncer module is unchanged from the Verilog Companion and will not be included in this report) the KEY0 button, and then sends the debounced button output and the switch inputs to the `incrementer` module. The incrementer module contains the logic to increase, decrease, or reset the tempo. The incrementer outputs a tempo value to both `bin_to_dec` and `clk_det`. These two modules convert the 8 bit binary number to 4 bit binary representation of the three equivalent decimal digits and determine the amount of time between beats, respectively. From there, `bin_to_dec` sends the 4 bit binary representations to three different `disp` modules (the display modules are unchanged from the Verilog Companion and will not be included in this report), which output to the seven segment displays. `clk_det` outputs the beats at the appropriate frequency to LEDG. Here is an RTL view of `system,` followed by the Verilog code itself:



```
module system(
        input [1:0] SW,
        input KEY,
        input CLOCK_50,

        output LEDG,
        output [6:0] HEX0,
        output [6:0] HEX1,
        output [6:0] HEX2);

        // Main system module to connect everything together.
        wire debouncer_to_inc;
        wire debouncer_to_res;
        wire [7:0]inc_to_clk_det;
        wire [3:0]inc_to_disp100;
        wire [3:0]inc_to_disp10;
        wire [3:0]inc_to_disp1;

        // debounces button and sends button input to incrementer
```

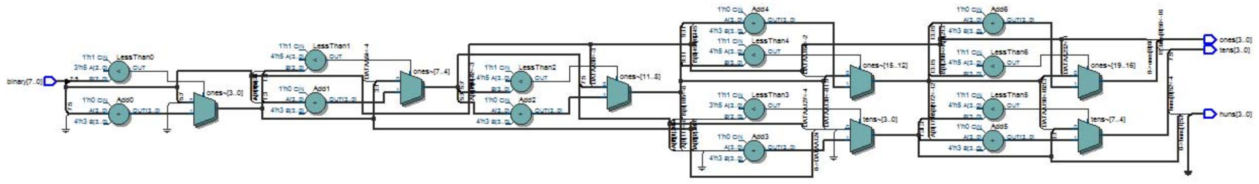
```
	debouncer d(
		.CLOCK_50		(CLOCK_50),
		.in				(KEY),
		.out			(debouncer_to_inc)
	);

	// increments BPM value and sends data to clk_det and the three displays
	incrementer inc(
		.SW				(SW),
		.DB				(debouncer_to_inc),
		.BPM			(inc_to_clk_det)
	);

	bin_to_dec conv(
		.binary			(inc_to_clk_det),
		.huns			(inc_to_disp100),
		.tens			(inc_to_disp10),
		.ones			(inc_to_disp1)
	);

	// determines when the light shall shine
	clk_det clk_det(
		.BPM			(inc_to_clk_det),
		.CLOCK_50		(CLOCK_50),
		.BEEP			(LEDG)
	);

	//display modules
	disp disp100(
		.in				(inc_to_disp100),
		//check to see if this is actually the 100's place on the board
		.out			(HEX2)
	);

	disp disp10(
		.in		(inc_to_disp10),
		.out	(HEX1)
	);

	disp disp1(
		.in		(inc_to_disp1),
		.out	(HEX0)
	);

endmodule
```

The next module is `incrementer`, which, as the name implies, increments the tempo of the metronome. It takes inputs from both the `debouncer` and directly from the board. The switches SW1 and SW0 are used to determine which operation the button will perform. If SW1 is active, pressing the button will always reset the tempo of the metronome to 120, regardless of whether or not SW0 is on. If SW1 is not on, then SW0 determines whether `incrementer` will add or subtract 1 to the current tempo. SW0 on adds, while SW0 off subtracts. The module then outputs the new value of the tempo, which is utilized by both `clk_det` and `bin_to_dec.` Here is an RTL view of the module followed by the Verilog code itself:



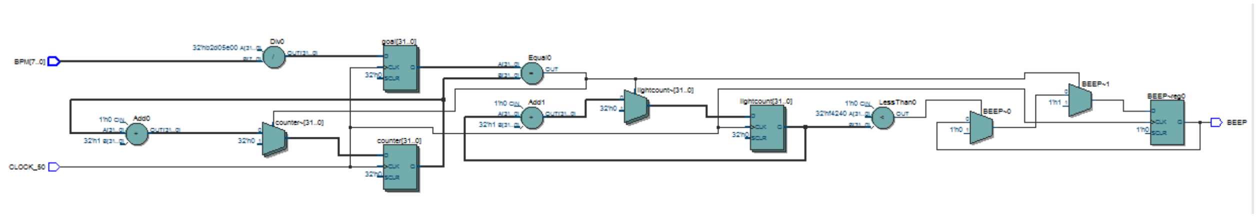
```
module incrementer(

    input [1:0] SW,
    input DB,

    output reg [7:0] BPM
    );

         initial BPM = 8'd121;

        always @(posedge DB) begin

                if(SW[1])
                        BPM <= 8'd120;
                // account for divide by 0
                else if(BPM == 1 & SW[0] == 0)
                        BPM <= 8'd255;
                else if(BPM == 255 & SW[0] == 1)
                        BPM <= 8'd1;
                else if(SW[0])
                        BPM <= BPM + 1;
                else
                        BPM <= BPM - 1;

        end

endmodule
```

The next module is `bin_to_dec`, which takes the 8 bit binary tempo value from `incrementer` and converts it into binary-coded decimal form. This is so that each digit of the tempo can be displayed by a seven segment display in unsigned decimal form, which is easier for users to understand than binary or hexadecimal. The concept of a binary to BCD converter was taken from the engineering department of the University of Utah, who explained the process and how to implement it in a hardware description language at this web address: http://www.eng.utah.edu/~nmcdonal/Tutorials/BCDTutorial/BCDConversion.html
The outputs of this module are sent to three different `disp` modules. Here is an RTL view of the module, followed by the Verilog code itself:



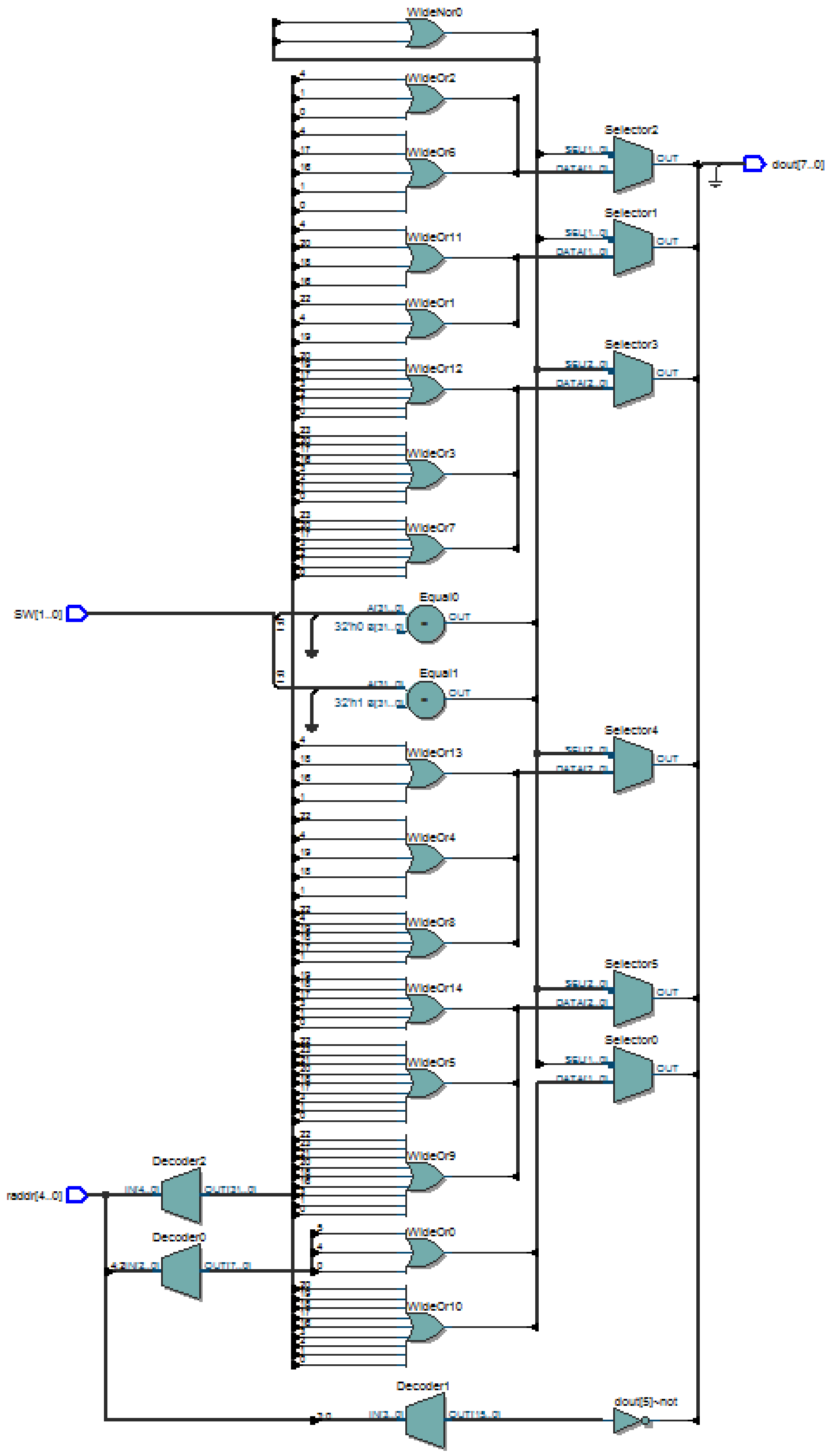
```
module bin_to_dec (
        input [7:0] binary,
        output reg [3:0] huns,
        output reg [3:0] tens,
        output reg [3:0] ones
        );
        // Concept of decimal to BCD converter taken from:
        // http://www.eng.utah.edu/~nmcdonal/Tutorials/BCDTutorial/BCDConversion.html

        integer count;

        always @(binary)
                begin
                        // zero out each binary representation of the decimals
                        huns = 4'd0;
                        tens = 4'd0;
                        ones = 4'd0;

                for ( count = 7; count >=0; count = count-1)
                        begin   // first always check to see if three needs to be added to any of
the columns
                                if (huns >= 5)
                                        huns = huns + 3;
                                if (tens >= 5)
                                        tens = tens + 3;
                                if (ones >= 5)
                                        ones = ones + 3;

                                // now shift everything to the left
                                huns = huns << 1;
                                huns[0] = tens[3];
                                tens = tens << 1;
                                tens[0] = ones[3];
                        ones = ones << 1;
                        ones[0] = binary[count];
                    end
                end
        endmodule
```

The next module is `clk_det`, which causes the light to flash and the audio to beep. On each rising edge of the clock, the *goal* variable is evaluated based on the user-selected tempo using the equation $goal = (50,000,000 * 60) / BPM$. Goal represents the number of clock cycles that the system should wait between beats. The module achieves this delay by incrementing a variable *counter* at the rising edge of each clock cycle. When the *counter* variable is equal to the *goal* variable, the module sets the *BEEP* variable, which represents a beat of the metronome, to 1. It also sets a third variable, *lightcount,* to zero. In order for the beat of the metronome to be visible/audible to humans, we needed the *BEEP* variable to stay 1 for longer than a single clock cycle. As such, *BEEP* remains one for as long as *lightcount* remains less than 1,000,000. *Lightcount* increments at the same speed as the *counter* variable. In this way, the metronome can alter the delay between beats based on the input from the user and display these beats in audible bursts of a square wave and visible flashes of the green LEDs. Here is an RTL view of the module followed by the Verilog code itself.



```
module clk_det(
        input [7:0]BPM,
        input    CLOCK_50,

        output reg BEEP
        );

        reg [31:0] goal = 0;
        reg [31:0] counter = 0;
        reg [31:0] lightcount = 0;

        always@ (posedge(CLOCK_50)) begin
                goal <= 50000000 * 60 / BPM;

                counter <= counter + 1;
                lightcount <= lightcount + 1;

                if(counter == goal) begin
                        BEEP <= 1;
                        counter <= 0;
                        lightcount <= 0;

                end
                else if(lightcount > 1000000)
                        BEEP <= 0;

        // use clock_50 to blink light in time with BPM
        end
endmodule
```

The last module we will explain in depth is the `LCD_message` module, which selects what mode to display on the LCD Display based on the positioning of the switches SW0 and SW1. The module uses nested case statements inside an always block, where the always block's conditions are raddr and SW. Immediately inside the always block is a case statement, determined by the position of the switches. Inside each case is a second case statement, determined by the value of raddr, which is determined by a separate module which we took from the Verilog Companion. When the switches' combined value is 0 (i.e. both switches are down), the raddr case statement causes the LCD to display the message "Mode: decrease." When the switches' combined value is 1 (i.e. SW1 is down and SW0 is up), the raddr case statement causes the LCD to display the message "Mode: increase." In any other case (i.e. when SW1 is up), the raddr case statement causes the LCD to display the message "Mode: reset."



WideNor0
WideOr2
WideOr6
WideOr11
WideOr1
WideOr12
WideOr3
WideOr7
Selector2
Selector1
Selector3
dout[7..0]
SW[1..0]
Equal0
32'h0
Equal1
32'h1
Selector4
WideOr13
WideOr4
WideOr8
WideOr14
Selector5
Selector0
WideOr5
WideOr9
Decoder2
raddr[4..0]
Decoder0
WideOr0
WideOr10
Decoder1
dout[5]~not

```
module LCD_message (
        input              [1:0] SW,
        input                   [4:0] raddr,
        output reg     [7:0]   dout
        );

   always @(raddr, SW)
                case (SW[1:0])

                        0: begin
                                case(raddr)
                                        0: dout = "M";
                                        1: dout = "o";
                                        2: dout = "d";
                                        3: dout = "e";
                                        4: dout = ":";
                                        16: dout = "D";
                                        17: dout = "e";
                                        18: dout = "c";
                                        19: dout = "r";
                                        20: dout = "e";
                                        21: dout = "a";
                                        22: dout = "s";
                                        23: dout = "e";
                                        default: dout = " ";
                                endcase
                        end

                        1: begin
                                case(raddr)
                                        0: dout = "M";
                                        1: dout = "o";
                                        2: dout = "d";
                                        3: dout = "e";
                                        4: dout = ":";
                                        16: dout = "I";
                                        17: dout = "n";
                                        18: dout = "c";
                                        19: dout = "r";
                                        20: dout = "e";
                                        21: dout = "a";
                                        22: dout = "s";
                                        23: dout = "e";
                                        default: dout = " ";
                                endcase
                        end

                        default: begin
                                case(raddr)
                                        0: dout = "M";
                                        1: dout = "o";
                                        2: dout = "d";
                                        3: dout = "e";
                                        4: dout = ":";
                                        16: dout = "R";
                                        17: dout = "e";
                                        18: dout = "s";
                                        19: dout = "e";
                                        20: dout = "t";
                                        default: dout = " ";
                                endcase
                        end
                endcase

endmodule
```