ECE 3300 - Music Player

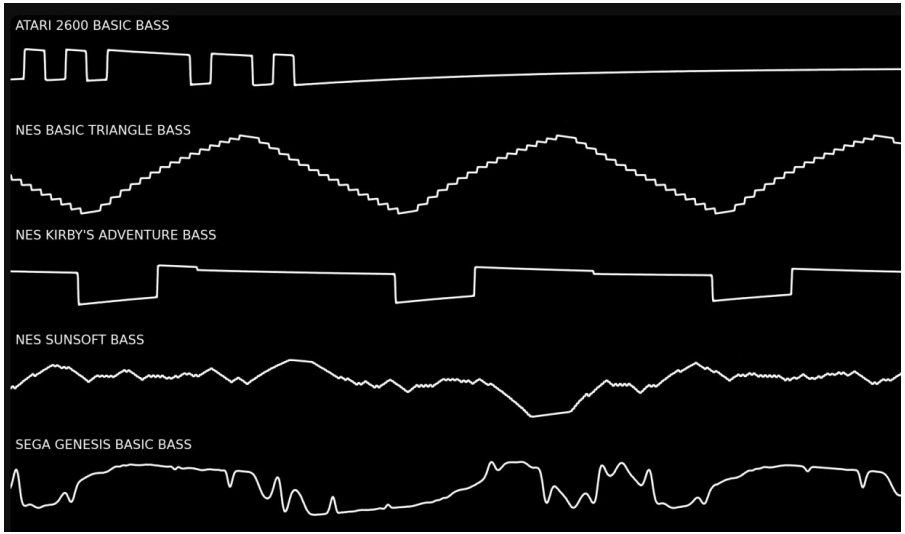
By: Ethan Song and Meredith Toledo

Functionality

- Plays a pre-made song when a specific switch is selected.
- Has a reset and enable switch to stop the music and restart.
- Possible to add more songs to the code.

- We are using the Audio PWM Amplifier, which generates a wave which is then used for a sound output.
- We can control the frequency by manipulating this pulse wave and control the sound we want.

PWM Example



- All these waves use a PWM to generate waves with different characteristics.
- The wave characteristics are manipulated to create a specific sound.
- Click here to see the video from where the screenshot was taken from.

Input and Outputs

Input

- Switches
  - Include a switch as an enable, and 3 more switches to control which song to play and what displays on the monitor through VGA
- Button (used to reset the song)

Output

- Audio Jack (used to output music)
- Seven Segment Display (used to display track number)
- LEDs (to flash whenever a note is playing)
- External Monitor (as an output for VGA, also displaying corresponding track number

Modules - ssd_driver and clk_div

```
module ssd_driver(
    input [3:0] ssd_driver_port_inp,
    input ssd_driver_port_idp,
    input ssd_clk,

    output [3:0] ssd_driver_port_led,
    output [6:0] ssd_driver_port_cc,
    output ssd_driver_port_odp,
    output [7:0] ssd_driver_port_an
    );

    assign ssd_driver_port_odp = ssd_driver_port_idp;
    assign ssd_driver_port_led = ssd_driver_port_inp;
    assign ssd_driver_port_an =  8'b01111111;
    assign ssd_driver_port_on = 1'b1;

    reg[6:0] ssd_driver_tmp_cc;
    wire[3:0] ssd_driver_digit;
    assign ssd_driver_digit=ssd_driver_port_inp;

    always@(posedge ssd_clk)
        begin:SEG_ENC
            case (ssd_driver_digit)
```

- The same old code we used for all the other labs.
- Allows us to display numbers using the seven segment displays given on the FPGA board.
- Used to show which song number is on.

Modules - MusicSheet

```
module MusicSheet(
input [9:0] number,
input [2:0] song_sel,
output reg [19:0] note, //what is the max frequency
output reg [4:0] duration
);
    /*
    Quarter has been changed from 5'b00010 to 00100 and Two and Four has been commented out
    because we will never use Two or Four but we will use 1/6
    */

    parameter   QUARTER = 5'b00100;
    parameter HALF = 5'b01000;
    parameter ONE = 2* HALF;
   // parameter TWO = 2* ONE;
    //parameter FOUR = 2* TWO;

    //Notes for Sad Machine
    parameter Ef = 160706, E = 151686, C = 95556.6, Bf = 107258, G = 127552, F = 143173, SP = 1;

    //Notes for Corridors of Time
    parameter Cs5 = 90192, Gs5 = 60197, Fs5 = 67568, A5 = 56818, B5 = 50619, E5 = 75844;

    //Notes for Tetris
    parameter C5=95556.62, D5 = 85131.02, F5=72520.5233;
    parameter A4b = 60196.72, G5 = 63776.32, C6 = 1046.50, A6 = 28409.09;
    parameter C4=382262.99/2, D4=340524.06/2, E4 = 303372.28/2, F4=286344.24/2, G4 = 255102.04/2;
    parameter  A4 = 113636.36, B4 = 101238.5525;
```

- Number controls which sequential note plays and song_sel refers to which song will play.
- First three parameters set up values for timing.
- Last couple parameters are the frequency for associated notes.

Modules - MusicSheet (Cont.)

```
always @ (number) begin
    case(song_sel)
        //SHELTER
        1:
        begin
            case(number)
                //One
                0: begin note = Ef; duration = QUARTER / 2; end
                1: begin note = SP; duration = QUARTER / 4; end
                2: begin note = Ef; duration = QUARTER / 2; end
                3: begin note = SP; duration = QUARTER / 4; end
                4: begin note = Ef; duration = QUARTER / 2; end
                5: begin note = SP; duration = QUARTER / 4; end

                6: begin note = Bf; duration = QUARTER / 4; end
                7: begin note = G; duration = QUARTER / 4; end
                8: begin note = Bf; duration = QUARTER / 2; end
                9: begin note = G; duration = QUARTER / 4; end
                10: begin note = F; duration = QUARTER / 4; end
```

- Songs are in a nested case statement.
- Outer case uses song_sel to pick which song will play.
- Inner case statement uses number to play through the notes sequentially.
- Depending on song_sel, a series of sequential notes will play

Modules - MusicSheet (Cont.)

```
//Corridors of Time
2:
begin
    case(number)
        0: begin note = Cs5; duration = QUARTER / 2; end
        //Measure 1
        1: begin note = Gs5; duration = QUARTER / 2; end
        2: begin note = SP; duration = QUARTER / 4; end
        3: begin note = Gs5; duration = QUARTER / 2; end
        4: begin note = SP; duration = QUARTER / 4; end
        5: begin note = Fs5; duration = QUARTER / 2; end
        6: begin note = SP; duration = QUARTER / 4; end
        7: begin note = Fs5; duration = QUARTER; end
        //Measure 2
        8: begin note = SP; duration = HALF; end
        9: begin note = Fs5; duration = QUARTER / 4; end
        10: begin note = Gs5; duration = QUARTER / 4; end
        11: begin note = A5; duration = QUARTER / 2; end
        12: begin note = B5; duration = QUARTER / 2; end
        13: begin note = A5; duration = QUARTER / 4; end
        14: begin note = Gs5; duration = QUARTER / 4; end
```

```
//Tetris
3:
begin
    case(number)
        0: begin note = E5; duration = QUARTER / 2; end
        1: begin note = B4; duration = QUARTER / 4; end
        2: begin note = C5; duration = QUARTER / 4; end
        3: begin note = D5; duration = QUARTER / 2; end
        4: begin note = C5; duration = QUARTER / 4; end
        5: begin note = B4; duration = QUARTER / 4; end
        6: begin note = A4; duration = QUARTER / 2; end
        7: begin note = SP; duration = QUARTER/4; end
        8: begin note = A4; duration = QUARTER / 4; end
        9: begin note = C5; duration = QUARTER / 2; end
        10: begin note = E5; duration = QUARTER / 2; end
        11: begin note = D5; duration = QUARTER / 4; end
        12: begin note = C5; duration = QUARTER / 4; end
        13: begin note = B4; duration = QUARTER / 2; end
        14: begin note = SP; duration = QUARTER / 4; end
        15: begin note = B4; duration = QUARTER / 4; end
        16: begin note = C5; duration = QUARTER / 4; end
```

Module - vga_out

```
module vga_out(

    input clk,          // 50 MHz
    input [3:0] sw,
    output o_hsync,      // horizontal sync
    output o_vsync,      // vertical sync
    output [3:0] o_red,
    output [3:0] o_blue,
    output [3:0] o_green
);

    reg [9:0] counter_x = 0;  // horizontal counter
    reg [9:0] counter_y = 0;  // vertical counter
    reg [3:0] r_red = 0;
    reg [3:0] r_blue = 0;
    reg [3:0] r_green = 0;

    reg reset = 0;  // for PLL
```

- We have inputs for a clock, and 3 switches that determine what displays on the board (made a 4-bit bus for later purposes, synced with top file values)
- For outputs we have what will be used as horizontal and vertical signals that will be sent to the monitor and 4-bit output buses per color
- We have registers that will serve as temp variables to be assigned values that will fluctuate based on counters, and colors we decide to use

Module - vga_out (Cont.)

```
always @(posedge clk)  // horizontal counter
    begin
        if (counter_x < 799)
            counter_x <= counter_x + 1;  // horizontal cour
        else
            counter_x <= 0;
    end  // always

always @ (posedge clk)  // vertical counter
    begin
        if (counter_x == 799)  // only counts up 1 count a
            begin
                if (c      < 525)  // vertical counter
                    counter_y <= counter_y + 1;
                else
                    counter_y <= 0;
            end  // if (counter_x...
    end  // always
// end counter and sync generation

////////////////////////////////////////////////////////////////
// hsync and vsync output assignments
assign o_hsync = (counter_x >= 0 && counter_x < 96) ? 1:0;
assign o_vsync = (counter_y >= 0 && counter_y < 2) ? 1:0;
```

reg [9:0]

- This part of the VGA code addresses the parameters in which a visible image will be produced on the monitor
- There will be a counter that ensures once the hsync signal hits a certain number of pixels, it will activate the vertical signal to create an image on the screen

Module vga_out (Cont.)

```
always @ (posedge clk)
begin
    ////////////////////////////////////////////////////////////////////
    case(sw)

    4'b0000: // SONG
        if(counter_y < 135)
            begin
            end
         else if(counter_y >= 135 && counter_y < 152)
            begin
                if(counter_x < 254)
                    begin
                       r_red <= 4'hA;    // other
                       r_blue <= 4'hA;
                       r_green <= 4'h0;
                    end
                else if(counter_x >= 315 && counter_x < 335)
                    begin
                       r_red <= 4'hA;    // other
                       r_blue <= 4'hA;
                       r_green <= 4'h0;
                    end
                else if(counter_x >= 431 && counter_x < 451)
                    begin
                       r_red <= 4'hA;    // other
                       r_blue <= 4'hA;
```

- The body of the code precedes to actually draw out the image “SONG” and the number corresponding to the track number
- The case statement makes sure that by switch input, it will activate and display the correct image that corresponds to the song

Module vga_out (Cont.)

```
assign o_red = (counter_x > 144 && counter_x <= 783 && counter_y > 35 && counter_y <= 514) ? r_red : 4'h0;
assign o_blue = (counter_x > 144 && counter_x <= 783 && counter_y > 35 && counter_y <= 514) ? r_blue : 4'h0;
assign o_green = (counter_x > 144 && counter_x <= 783 && counter_y > 35 && counter_y <= 514) ? r_green : 4'h0;
```

- The ending assignments of the code just makes sure that all 4-bit output buses for the colors fall within range of “addressable video time”
- If the statements fall false, they will not display properly
- This confirms that the image and colors will all display properly on the external monitor that we are using

Module - SongPlayer (Top File)

```
module SongPlayer(
        input clock,
        input reset,
        input playSound,

        //Song selector input
        input [3:0] song_sel,

        output reg audioOut,     //Audio Enable
        output wire aud_sd,      //Audio Shutdown

        //OUTPUTS FOR THE SONG NUMBER
        //output [3:0] ssd_driver_port_led,
        output [6:0] ssd_driver_output_cc,
        output ssd_driver_port_odp,
        output [6:0] ssd_driver_port_an,

        output reg led_out = 0
    );
```

- We have the input for clock, reset (button), playSound (switch), and song_sel (switch).
- For outputs, we have audioOut (enables audio), aud_sd (audio shutdown), seven segment output, and an led_out.

Module - SongPlayer (Cont.)

```
reg [19:0] counter;
reg [31:0] time1, noteTime;
reg [9:0] msec, number; //millisecond counter, and sequence number of musical note.
wire [4:0] note, duration;
wire [19:0] notePeriod;
parameter clockFrequency = 100_000_000;
assign aud_sd = 1'b1;


MusicSheet mysong(
    number,
    s_num,

    notePeriod,
    duration
);

wire ssd_clk;
clk_div ssd_seg(
    .top_clk(clock),
    .clock_out(ssd_clk)
);

ssd_driver ssd(
    .ssd_driver_port_inp(s_num),
    .ssd_driver_port_idp(1'b0),
    .ssd_clk(ssd_clk),

    .ssd_driver_port_led(ssd_driver_port_led),
    .ssd_driver_port_cc(ssd_driver_output_cc),
    .ssd_driver_port_odp(ssd_driver_port_odp),
    .ssd_driver_port_an(ssd_driver_port_an)
);
```

- Creating all the parameters, registers, and wires for the sound system to work.
- Instantiation of MusicSheet, clk_div, and ssd_driver.

Module - SongPlayer (Cont.)

```
//USED TO SELECT THE SONG
integer s_num = 0;
integer s_length = 0;

always @ (posedge clock)
begin
    //Selects song and length
    //SAD MACHINE
    if(song_sel == 4'b0001)
    begin
        s_num <= 1;
        s_length <= 45;
    end
    if(song_sel == 4'b0010)
    begin
        s_num <= 2;
        s_length <= 32;
    end
    if(song_sel == 4'b0100)
    begin
        s_num <= 3;
        s_length <= 103;
    end
```

- Beginning of the clock loop.
- Creation of integers to keep track of what song is going to play and its length.
- If statements to correlate input with the song number and its length.

Module - SongPlayer (Cont.)

```
    //Stop playing sound
    if(reset | ~playSound)
        begin
            counter <=0;
            time1<=0;
            number <=0;
            audioOut <=1;
            s_num <= 0;
        end
    //Play Song
    else
    begin
        //Count and
        counter <= counter + 1;
        time1<= time1+1;

        if( counter >= notePeriod)
        begin
            counter <= 0;
            audioOut <= ~audioOut ;
        end //toggle audio output (audio is toggled via audioOut, counter is the freq)
        if( time1 >= noteTime)
        begin
            time1 <=0;
            number <= number + 1;
            led_out = ~led_out;
        end  //play next note (if duration is met, then we go to next note which is indicated by number)
        if(number == s_length) number <=0; // Make the number reset at the end of the song
    end
end

always @(duration) noteTime = duration * clockFrequency/4;
   //number of   FPGA clock periods in one note.
```

- The main loop where the sounds are played.
- First if statement stops the song and resets all the counters.
- Second if statement contains two counters (counter and time1).
    - Counter counts up to the note frequency.
    - Time1 counts up to the note duration.
- An led output is triggered for every time the note finishes its duration.
- Lastly, we have an always statement to have a consistent time.