# CS M152A

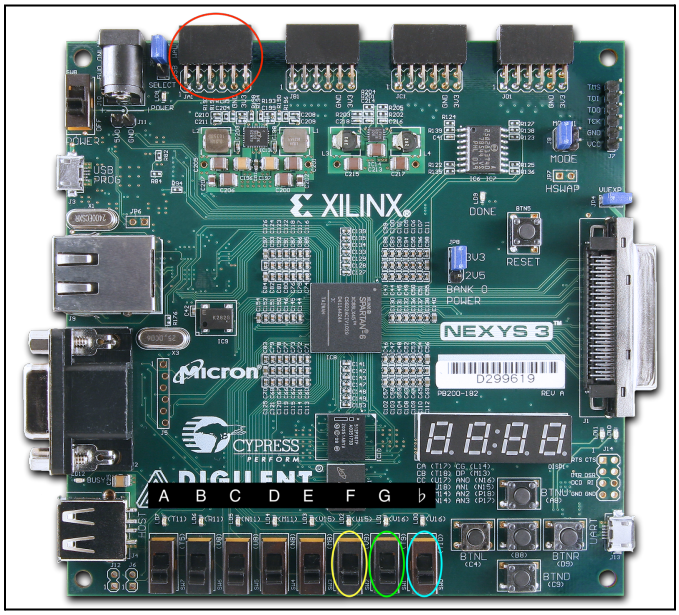
**Lab 4 Report**

**Puvali Chatterjee & Vivian Ha**

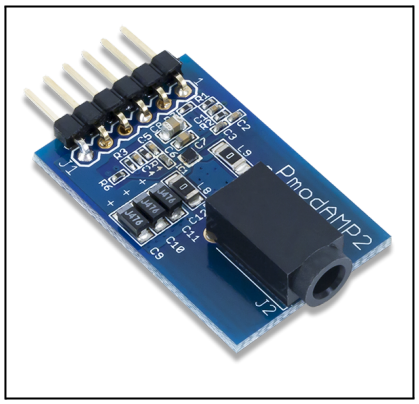
**TA: Samuel Lee**

## Introduction and Requirements

For our project, we created a music box on a Nexys 3 FPGA board that plays songs on speakers via a Digilent PMOD AMP2 amplifier. The music box can play 2 songs by reading in music notes from input ROMs, and the song to be played is selected with a switch on the board. As the music box plays a song, the song time (how long that particular song has been playing) is displayed on the seven-segment display. The 8 LEDs on the board light up corresponding to the music notes: LEDs LD7-LD1 light up for natural notes A-G and LD0 lights up if the note is flat (Ab-Gb). The music notes here are: G, Gb, F, E, Eb, D, Db, C, B, Bb, A and Ab. There is also a reset switch to reset the song to the beginning, and a pause switch which pauses the song and seven-segment display time when the switch is high (when the switch is low, the song plays and the seven-segment display marks time). Both songs keep looping once they finish.




Nexys 3 FPGA board




PMOD AMP2

The Nexys 3 FPGA board is shown above with the labeled music notes for LEDS 0 through 7 (LD0-LD7). Switches SW0, SW1 and SW2 (circled blue, green and yellow respectively) are the *RESET, song_sel* (to select the song) and *pause* switches respectively. The lower row of pinouts in the PMOD JA port (circled red) was used to connect the PMOD AMP2 amplifier (also shown above).

This music box can play any 2 songs when provided with the music notes, with only some minor tweaks to the code to accommodate for the tempo of the input songs and its length (number of notes).

To learn how to produce sound using the FPGA board, we referred to a sample project (1) for guidance. We referred to another reference (2) to figure out how we could play sounds using the PMOD AMP2 and speakers.

## Design Description

---

We implemented several Verilog modules to build the music box:

1. **module musicbox(input clk,**
   **input RESET,**
   **input pause,**
   **input song_sel,**
   **output AUDIO_IP,**
   **output GAIN_SEL,**
   **output reg ALS,**
   **output reg [3:0] anode,**
   **output reg [6:0] cathode,**
   **output reg ld7,**
   **output reg ld6,**
   **output reg ld5,**
   **output reg ld4,**
   **output reg ld3,**
   **output reg ld2,**
   **output reg ld1,**
   **output reg ld0);**

   The *musicbox* module is the top-level module which controls the overall logic of the program. *clk* is the internal 100 MHz clock of the board.

   When *RESET* is high, nothing is played and both the song and its corresponding timer are reset to 0. When *song_sel* is low, song 1 (from *ROM1*) is selected and when it is high, song 2 (from *ROM2)* is selected. When *pause* is high, whichever song was playing is paused and the amplifier turned off.

   *AUDIO_IP, GAIN_SEL* and *ALS* are the outputs to the PMOD AMP2. *AUDIO_IP* is the audio input (i.e. the signal to be played). *GAIN_SEL* is for the gain selection pin - there is a 6 dB or 12 dB gain applied to the incoming audio signals depending on whether this pin is driven high or low respectively. *ALS,* which stands for active low shutdown, is used to turn the amplifier on or off - when *ALS* is low, the amplifier is placed in a very low power state and effectively turned off; when *ALS* is high, the amplifier is brought back to normal operation mode with a maximum speaker output of 2.5 W.

   The registers *anode* and *cathode* are used for the seven-segment display and the registers ld7-ld0 correspond with the 8 LEDs on the board.

   ➔ First, the *clock* module is instantiated to get the divided 1 Hz and 500 Hz clocks. The 1 Hz clock is used in the counter that keeps track of how long a particular song has been playing and displays the elapsed time on the seven-segment display. The 500 Hz clock is used for the refresh rate of the seven-segment display.

➔ An always block triggered by posedge clk sets a counter *tone:*

```
reg [33:0] tone;
always @(posedge clk) begin
        if (RESET)
                tone = 0;
        else if (pause)
                tone = tone;
        else
                tone = tone + 1;
end
```

*tone* = 0 corresponds with the first note of the song that is playing. If neither *RESET* nor *pause* are high, tone is incremented normally so that a song can be played as a sequence of musical notes.

➔ Then, both *ROM1* and *ROM2* modules are instantiated as follows:

```
reg [10:0] fullnote;
wire [10:0] fullnote1;
wire [10:0] fullnote2;


ROM1 rom1(.clk(clk), .address(tone[31:23]), .note(fullnote1));
ROM2 rom2(.clk(clk), .address(tone[32:24]), .note(fullnote2));
```

*fullnote1* and *fullnote2* are wires that get the values of the currently playing full note (base note + octave) from their corresponding ROMs. We decided which 9 bits of *tone* must be sent to the ROMs by trial and error. For faster tempo songs, a less significant range of bits of *tone* must be selected. Here, ROM1 has a faster tempo so bits 31-23 are chosen. ROM2 has a slightly slower tempo so bits 32-24 are chosen.

➔ An always block triggered by *song_sel* sets *fullnote* to either *fullnote1* or *fullnote2* depending on which song is to be played:

```
always @(song_sel) begin
        //song 1
        if (~song_sel) begin
                fullnote = fullnote1;
                minutes = mins1;
                seconds = secs1;
        //song2
        end else begin
                fullnote = fullnote2;
                minutes = mins2;
                seconds = secs2;
        end
end
```

This always block also sets *seconds* and *minutes* to *secs1/mins1* or *secs2/mins2* depending on which song is playing.

➔ Another always block triggered by *pause* sets the *ALS* output.

```
always @(pause) begin
        if (RESET) begin
                ALS = 0;
        end else if (pause) begin
                ALS = 0;
        end else if (~pause) begin
                ALS = 1;
        end
end
```

➔ The *counter* module is instantiated and it counts the time elapsed while each song is playing and returns *secs1, mins1, secs2* and *mins2.* As shown above, *minutes* and *seconds* are set to the appropriate values. Then, they are separated into their tens' place and ones' place digits respectively. The *segments* module is then instantiated to obtain the cathode signal pattern for each signal to be displayed. An edge-triggered always block triggered by *RESET* and the 500 Hz clock (named *ssg_clk)* is used to display the elapsed time on the seven-segment display:

```
reg [1:0] digit_switch;
always @(posedge ssg_clk, posedge RESET) begin
        if (RESET) begin
                digit_switch = 0;
                anode = 4'b1111;

        end else if (digit_switch == 0) begin
                anode = 4'b0111;
                cathode = cathode3;
                digit_switch = digit_switch + 1;

        end else if (digit_switch == 1) begin
                anode = 4'b1011;
                cathode = cathode2;
                digit_switch = digit_switch + 1;

        end else if (digit_switch == 2) begin
                anode = 4'b1101;
                cathode = cathode1;
                digit_switch = digit_switch + 1;

        end else if (digit_switch == 3) begin
                anode = 4'b1110;
                cathode = cathode0;
                digit_switch = 0;
        end
end
```

➔ The *divideby12* module is instantiated to separate *fullnote* into a base note and octave. The returned *note* is then used to set a divisor *dv* depending on the frequency of the musical note to be played. An always block is used:

```
//Note frequencies are 110 Hz, 117 Hz, ... , 208 Hz
//256 * 2 = 512 is for the 2nd (lowest) octave
always @ (note) begin
        case(note)
                0:              dv = clkf / 512 / 110 - 1;              //A
                1:              dv = clkf / 512 / 117 - 1;              //Bb
                2:              dv = clkf / 512 / 123 - 1;              //B
                3:              dv = clkf / 512 / 131 - 1;              //C
                4:              dv = clkf / 512 / 139 - 1;              //Db
                5:              dv = clkf / 512 / 147 - 1;              //D
                6:              dv = clkf / 512 / 156 - 1;              //Eb
                7:              dv = clkf / 512 / 165 - 1;              //E
                8:              dv = clkf / 512 / 175 - 1;              //F
                9:              dv = clkf / 512 / 185 - 1;              //Gb
                10:             dv = clkf / 512 / 196 - 1;              //G
                11:             dv = clkf / 512 / 208 - 1;              //Ab
                default: dv = 0;
        endcase
end
```

Here, *clkf* is 100,000,000 or 100 MHz, which is the frequency of the internal clock *clk.* Then, logic similar to the clock divider logic that we used in previous labs is used to assign values to a note counter *note_ctr* which starts at the decided divisor (*dv*) and continues decrementing until it reaches 0. Then it resets to *dv.* This generates a digital signal which is sent to the PMOD AMP2 and played on the speaker to produce a musical note.

```
always @(posedge clk) begin
        if (note_ctr == 0)
                note_ctr = dv;
        else
                note_ctr = note_ctr - 1;
end
```

The *octave* value that is returned by *divideby12* is used to adjust the octave of the music note that is played. There are 5 octaves and for every increase in octave, the frequency is doubled by halving the divisor. This is done as follows:

```
always @(posedge clk) begin
        if (note_ctr == 0) begin
                if (octave_ctr == 0) begin
                        case(octave)
                                0: octave_ctr = 255;
                                1: octave_ctr = 127;
                                2: octave_ctr = 63;
                                3: octave_ctr = 31;
                                4: octave_ctr = 15;
                                default: octave_ctr = 7;
                        endcase
                end else
                        octave_ctr = octave_ctr - 1;
        end
end
```

➔ The following block of code toggles a register *speaker* to generate the audio signal that is sent to the PMOD AMP2:

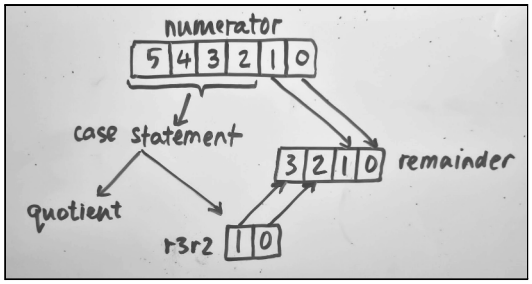
```
always @(posedge clk) begin
        if (note_ctr == 0 && octave_ctr == 0 && fullnote != 0)
                speaker = ~speaker;
end

assign AUDIO_IP = speaker & (tone[6:0] == 0);
assign GAIN_SEL = 1;                            //12 dB gain for low, 6 dB gain for high
```

(tone[6:0] == 0) is used to average out the analog signal to a lower volume since a 7 bit counter is 0 once every 128 cycles.

2. **module divideby12(input [5:0] numerator,**
   **output reg [2:0] quotient,**
   **output [3:0] remainder);**

This module contains the logic for separating a full note into the base note and its corresponding octave. *numerator* gets *fullnote* and the outputs *quotient* and *remainder* are stored in *octave* and *note* in the *musicbox* module respectively. The following diagram explains the logic used:




The case statement used to determine quotient and the 2 most significant bits of remainder (i.e. *r2r2*) is shown below:

```
always @(numerator[5:2]) begin
        case (numerator[5:2])
                0:  begin quotient = 0; r3r2 = 0; end
                1:  begin quotient = 0; r3r2 = 1; end
                2:  begin quotient = 0; r3r2 = 2; end
                3:  begin quotient = 1; r3r2 = 0; end
                4:  begin quotient = 1; r3r2 = 1; end
                5:  begin quotient = 1; r3r2 = 2; end
                6:  begin quotient = 2; r3r2 = 0; end
                7:  begin quotient = 2; r3r2 = 1; end
                8:  begin quotient = 2; r3r2 = 2; end
                9:  begin quotient = 3; r3r2 = 0; end
                10: begin quotient = 3; r3r2 = 1; end
                11: begin quotient = 3; r3r2 = 2; end
                12: begin quotient = 4; r3r2 = 0; end
                13: begin quotient = 4; r3r2 = 1; end
                14: begin quotient = 4; r3r2 = 2; end
                15: begin quotient = 5; r3r2 = 0; end
        endcase
end
```

3. **module counter(input RESET,**
   **input song_sel,**
   **input pause,**
   **input clk_1hz,**
   **output reg [5:0] mins1,**
   **output reg [5:0] secs1,**
   **output reg [5:0] mins2,**
   **output reg [5:0] secs2);**

   This module increments the minutes and seconds that are displayed on the seven-segment display in the same way as lab 3. If either song is paused, the time for that song is kept constant. If RESET is high, then all minutes and seconds variables are set to 0. Otherwise, if song 1 is playing, *secs1* is incremented at 1 Hz and *mins1* increments when *secs1* reaches 59. Both reset to 0 once the time hits 59:59. Likewise, *secs2* and *mins2* are incremented if song 2 is playing.

4. **module segments(input [3:0] digit,**
   **output reg [6:0] cathode);**

   This module takes in a digit from 0-9 as argument and returns the cathode signals pattern for it to be displayed on the seven-segment display. A 7-bit cathode pattern encodes the seven circuit nodes (segments) A-G for that digit. When a bit in the cathode pattern is low, the LED on the display for that segment is lit up. Hence, the cathode patterns for digits 0-9 are decided as follows:

```
always @* begin
        case (digit)              //ABCDEFG
                4'd0: cathode = 7'b0000001;
                4'd1: cathode = 7'b1001111;
                4'd2: cathode = 7'b0010010;
                4'd3: cathode = 7'b0000110;
                4'd4: cathode = 7'b1001100;
                4'd5: cathode = 7'b0100100;
                4'd6: cathode = 7'b0100000;
                4'd7: cathode = 7'b0001111;
                4'd8: cathode = 7'b0000000;
                4'd9: cathode = 7'b0000100;
                default: cathode = 7'b1111111;
        endcase
        end
```

5. **module clock(input RESET,**
   **input clk,**
   **output reg clk_1hz,**
   **output reg ssg_clk);**

   This module uses clock divider logic with counters and dividers to divide the internal 100 MHz clock *clk* into *clk_1hz* and *ssg_clk,* which are of frequencies 1 Hz and 500 Hz respectively.

6. **module ROM1(input clk,**
   **input [10:0] address,**
   **output reg [10:0] note);**

This module was a giant switch case that corresponded to different music notes in “Hedwig’s Theme”, a song from Harry Potter. A section of the case statement is shown below:

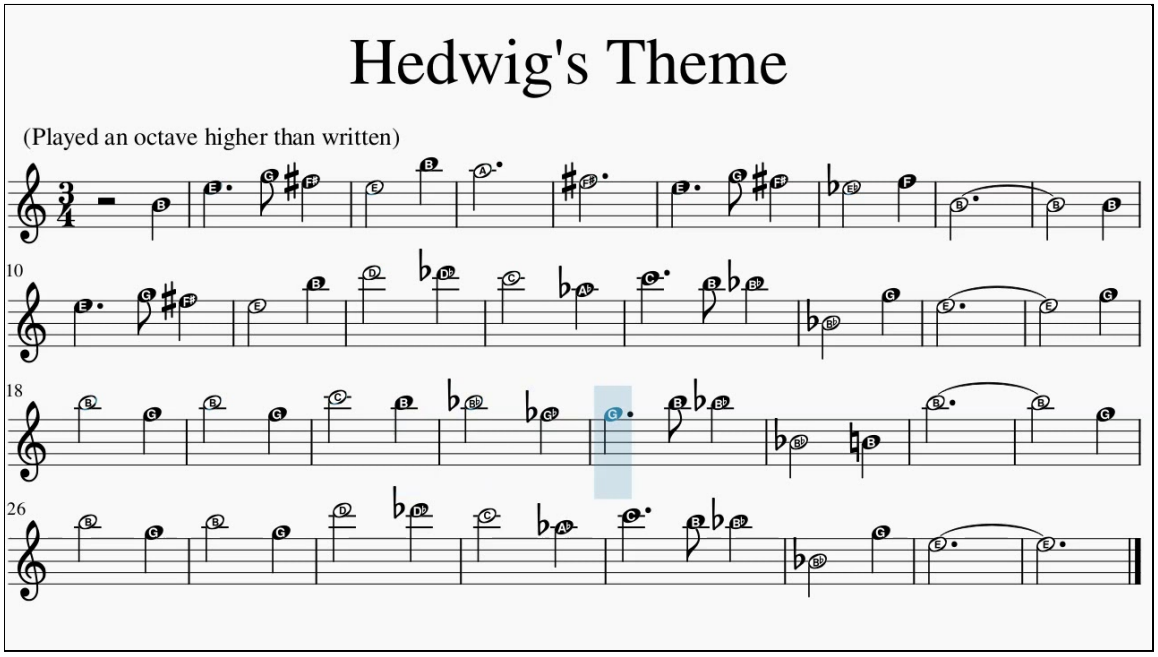
```
always @(posedge clk)
case(address)
    //B
    0:note <= 34;
    1:note <= 34;
    2:note <= 34;
    3:note <= 34;
    //E
    4:note <= 39;
    5:note <= 39;
    6:note <= 39;
    7:note <= 39;
    8:note <= 39;
    9:note <= 39;
    //G
    10:note <= 42;
    11:note <= 42;
    //F#
    12:note <= 41;
    13:note <= 41;
    14:note <= 41;
    15:note <= 41;
    //E
    16:note <= 39;
    17:note <= 39;
    18:note <= 39;
    19:note <= 39;
    20:note <= 39;
    21:note <= 39;
    22:note <= 39;
    23:note <= 39;
```

We referred to sheet music (3) for the musical notes of the song, then used Google Sheets to transpose the musical notes into integer values that corresponded to that musical note:

| Note | Number | Binary | numer binary | numer bin [5:2] | numer[5:2] dec | quotient decimal | remain_bit3_bit2 [3:0] | remain [3:0] | remain dec | Note | Number | Octave |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 22 | 10110 | 010110 | 0101 | 5 | 1 | 0010 | 001010 | 10 | G | 22 | |
| | 23 | 10111 | 010111 | 0101 | 5 | 1 | 0010 | 001011 | 11 | Ab | 23 | |
| | 24 | 11000 | 011000 | 0110 | 6 | 2 | 0000 | 000000 | 0 | A | 24 | 2 |
| | 25 | 11001 | 011001 | 0110 | 6 | 2 | 0000 | 000001 | 1 | Bb | 25 | 2 |
| | 26 | 11010 | 011010 | 0110 | 6 | 2 | 0000 | 000010 | 2 | B | 26 | 2 |
| | 27 | 11011 | 011011 | 0110 | 6 | 2 | 0000 | 000011 | 3 | C | 27 | 2 |
| | 28 | 11100 | 011100 | 0111 | 7 | 2 | 0001 | 000100 | 4 | Db | 28 | 2 |
| | 29 | 11101 | 011101 | 0111 | 7 | 2 | 0001 | 000101 | 5 | D | 29 | 2 |
| | 30 | 11110 | 011110 | 0111 | 7 | 2 | 0001 | 000110 | 6 | Eb | 30 | 2 |
| | 31 | 11111 | 011111 | 0111 | 7 | 2 | 0001 | 000111 | 7 | E | 31 | 2 |
| | 32 | 100000 | 000000 | 0000 | 0 | 0 | 0000 | 000000 | 0 | A | 32 | 0 |
| | 33 | 100001 | 000001 | 0000 | 0 | 0 | 0000 | 000001 | 1 | Bb | 33 | 0 |
| | 34 | 100010 | 000010 | 0000 | 0 | 0 | 0000 | 000010 | 2 | B | 34 | 0 |
| | 35 | 100011 | 000011 | 0000 | 0 | 0 | 0000 | 000011 | 3 | C | 35 | 0 |
| | 36 | 100100 | 000100 | 0001 | 1 | 0 | 0001 | 000100 | 4 | Db | 36 | 0 |
| | 37 | 100101 | 000101 | 0001 | 1 | 0 | 0001 | 000101 | 5 | D | 37 | 0 |
| | 38 | 100110 | 000110 | 0001 | 1 | 0 | 0001 | 000110 | 6 | Eb | 38 | 0 |
| | 39 | 100111 | 000111 | 0001 | 1 | 0 | 0001 | 000111 | 7 | E | 39 | 0 |
| | 40 | 101000 | 001000 | 0010 | 2 | 0 | 0010 | 001000 | 8 | F | 40 | 0 |
| | 41 | 101001 | 001001 | 0010 | 2 | 0 | 0010 | 001001 | 9 | Gb | 41 | 0 |
| | 42 | 101010 | 001010 | 0010 | 2 | 0 | 0010 | 001010 | 10 | G | 42 | 0 |
| | 43 | 101011 | 001011 | 0010 | 2 | 0 | 0010 | 001011 | 11 | Ab | 43 | 0 |
| | 44 | 101100 | 001100 | 0011 | 3 | 1 | 0000 | 000000 | 0 | A | 44 | 1 |
| | 45 | 101101 | 001101 | 0011 | 3 | 1 | 0000 | 000001 | 1 | Bb | 45 | 1 |
| | 46 | 101110 | 001110 | 0011 | 3 | 1 | 0000 | 000010 | 2 | B | 46 | 1 |
| | 47 | 101111 | 001111 | 0011 | 3 | 1 | 0000 | 000011 | 3 | C | 47 | 1 |
| | 48 | 110000 | 010000 | 0100 | 4 | 1 | 0001 | 000100 | 4 | Db | 48 | 1 |
| | 49 | 110001 | 010001 | 0100 | 4 | 1 | 0001 | 000101 | 5 | D | 49 | 1 |
| | 50 | 110010 | 010010 | 0100 | 4 | 1 | 0001 | 000110 | 6 | Eb | 50 | 1 |

These integer values are stored in *note* here and in *fullnote1* inside the *musicbox* module. ROM1 is composed of 364 switch cases. It's worth noting that a value of 0 corresponds to a musical rest where no sound is played. The ROM receives *address,* which is incremented in order to determine which notes to play at any given time, and this corresponds to *tone[31:23]* in the *musicbox* module.

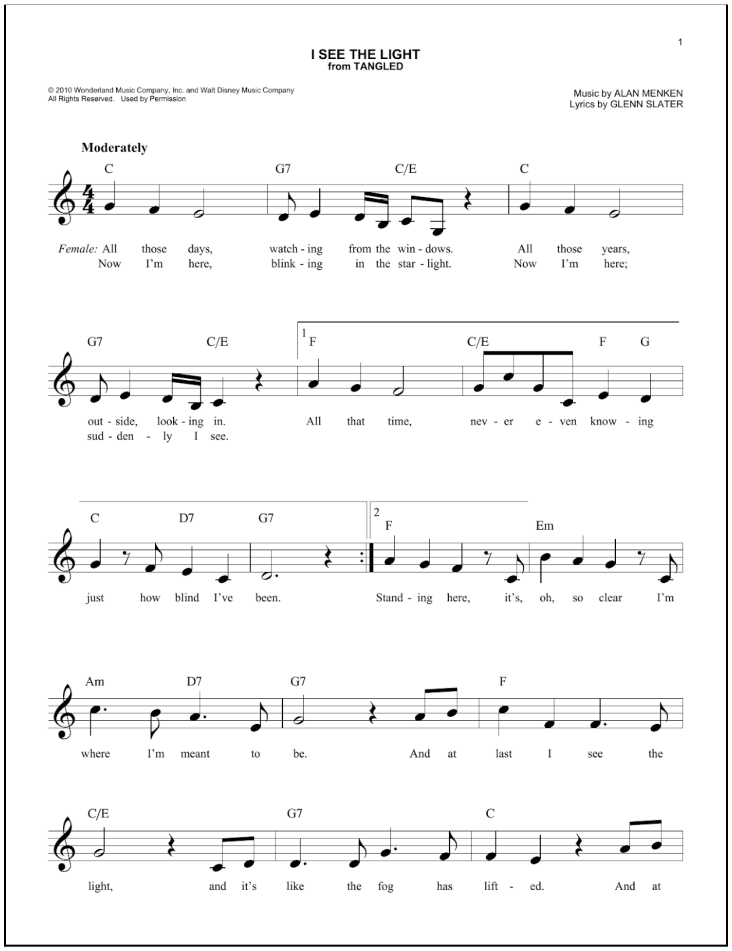
The original sheet music for Hedwig's Theme is shown below:




7. **module ROM2(input clk,**
   **input [10:0] address,**
   **output reg [10:0] note);**

This module was a giant switch case that corresponded to different music notes in "I See the Light", a song on the soundtrack of the Disney movie "Tangled". ROM2 is composed of 378 switch cases. The logic for this ROM is the same as ROM2. Here, as in ROM1, 0 corresponds to a musical rest where no sound is played. A section of the case statement and the original sheet music (4) for "I See The Light" are shown below:

```
always @(posedge clk)
case(address)
    //A 44
    0:note <= 44;
    1:note <= 44;
    2:note <= 44;
    3:note <= 44;
    //G 42
    4:note <= 42;
    5:note <= 42;
    6:note <= 42;
    7:note <= 42;
    //F# 41
    8:note <= 41;
    9:note <= 41;
    10:note <= 41;
    11:note <= 41;
    12:note <= 41;
    13:note <= 41;
    14:note <= 41;
    15:note <= 41;
    //E 39
    16:note <= 39;
    17:note <= 39;
    //F# 41
    18:note <= 41;
    19:note <= 41;
    20:note <= 41;
    21:note <= 41;
    //E 39
    22:note <= 39;
    //C# 36
    23:note <= 36;
    //D 37
    24:note <= 37;
    25:note <= 37;
```



I SEE THE LIGHT
from TANGLED



Music by ALAN MENKEN
Lyrics by GLENN SLATER

Moderately

C G7 C/E C

*Female:* All those days, watch - ing from the win - dows. All those years,
Now I'm here, blink - ing in the star - light. Now I'm here;

G7 C/E 1 F C/E F G

out - side, look - ing in. All that time, nev - er e - ven know - ing
sud - den - ly I see.

C D7 G7 2 F Em

just how blind I've been. Stand - ing here, it's, oh, so clear I'm

Am D7 G7 F

where I'm meant to be. And at last I see the

C/E G7 C

light, and it's like the fog has lift - ed. And at

## Conclusion

---

We built an FPGA music box that plays two songs on a speaker connected to the board via a PMOD AMP2 module. To play music, the code uses the transcribed music notes to generate an audio signal by separating each note into a base musical note and its octave. These notes and octaves are then implemented using counters and divisors to generate an analog audio signal of the appropriate frequency, which is then fed to the AMP2. The songs can be reset, changed or paused, and the seven-segment display on the board shows how long a particular song has been playing.

We faced numerous challenges while working on this project. We had purchased a PMOD AMP2 module but it stopped working after 2-3 uses so we had to buy another one and wait for it to ship. We referred to a sample project to learn how to turn music notes into audio signals, but there was not much explanation provided so we had difficulty understanding the logic of the signal generation. Furthermore, when transcribing sheet music to ROMs containing music notes, we faced issues with wrong tempo, some sections being skipped, music that sounded wrong, etc. so we performed much trial and error to make the songs sound as perfect and close to the originals as possible.

Both team members contributed to the project, with Puvali working on the code for signal generation and other functionality and Vivian working on the ROMs and notes transcription. We both worked together on debugging in lab.

## References

---


1. https://www.fpga4fun.com/MusicBox.html
2. https://my.ece.utah.edu/~kalla/ECE3700/pmod_ref.pdf
3. https://www.youtube.com/watch?v=Wef5lrSpiw8
4. https://www.sheetmusicdirect.com/en-US/se/ID_No/253962/Product.aspx