| EECS 2070 02 Digital Design Labs 2020<br>Lab 8 |
| --- |
| 學號：108062281　　姓名：莊晴雯 |

## 1. 實作過程

In the last lab, which is lab8, we are required to design a music player than can play a song (switch between two songs for bonus). The music player should be able to play or pause the music, mute, repeat after the song ends, rewind the song, supports 5-level volume control and 3-level octave control, also display the current note with the 7-segment display.

I first figured that I need to generate the music notes, which looks like this:

```
hc,hc,sil,g,g,sil,e,e,sil,a,a,b,b,bb,a,sil
12'd0: toneR = `hc;      12'd1: toneR = `hc;
12'd2: toneR = `hc;      12'd3: toneR = `hc;

12'd4: toneR = `hc;      12'd5: toneR = `hc;
12'd6: toneR = `hc;      12'd7: toneR = `hc;

12'd8: toneR = `sil;      12'd9: toneR = `sil;
12'd10: toneR = `sil;      12'd11: toneR = `sil;

12'd12: toneR = `g;      12'd13: toneR = `g;
12'd14: toneR = `g;      12'd15: toneR = `g;

12'd16: toneR = `g;      12'd17: toneR = `g;
12'd18: toneR = `g;      12'd19: toneR = `g;

12'd20: toneR = `sil;      12'd21: toneR = `sil;
12'd22: toneR = `sil;      12'd23: toneR = `sil;

12'd24: toneR = `e;      12'd25: toneR = `e;
12'd26: toneR = `e;      12'd27: toneR = `e;

12'd28: toneR = `e;      12'd29: toneR = `e;
12'd30: toneR = `e;      12'd31: toneR = `e;

12'd32: toneR = `sil;      12'd33: toneR = `sil;
12'd34: toneR = `sil;      12'd35: toneR = `sil;
```

To be able to do this, I wrote a simple Python program that can generate the music notes output format like this when I input the music notes separated by ‘,’.

Simple Python program to generate music notes:

```python
note = input()
note_list = note.split(',')

count_even = 0
count_odd = 1
i = 0

bound = 4
while(bound <= 512):
  while(count_odd < bound):
    print("12'd" + str(count_even) + ": toneR = `" + note_list[i] + ';' + "      " + "12'd" + str(count_odd) + ": toneR = `" + note_list[i] + ';')
    count_even = count_even + 2
    count_odd = count_odd + 2
  bound = bound + 4
  i = i + 1
  print()
```

I need to define the music notes with its corresponding frequencies before putting the generated music notes on music_example module. Here are several music notes that I defined:

```verilog
`define la  32'd220
`define lb  32'd247
`define lc  32'd131
`define ld  32'd147
`define le  32'd165
`define lf  32'd175
`define lg  32'd196
`define c   32'd262   // C3
`define d   32'd294
`define e   32'd330
`define f   32'd349
`define g   32'd392   // G3
`define a   32'd440
`define b   32'd494   // B3
`define ha  32'd880
`define hb  32'd988
`define hc  32'd524   // C4 524
`define hd  32'd588   // D4 588
`define he  32'd660   // E4 660
`define hf  32'd698   // F4 698
`define hg  32'd784   // G4
```

```verilog
`define ab  32'd415
`define bb  32'd466
`define hfs 32'd740
`define hes 32'd698
`define hds 32'd622
`define gs  32'd415
`define as  32'd466
`define hcs 32'd554
`define db  32'd277
`define cb  32'd247
`define llg 32'd98
`define lab 32'd208
`define lbb 32'd233
`define hhc 32'd1048
`define heb 32'd622

`define sil   32'd50000000 // silence
`define silence   32'd50000000
```

I chose 你好不好 and Mario Theme Music to be demonstrated in this lab, so I first write down the music notes before inputting them into the Python program.
After generating the music, I put the generated music notes, toneR and toneL, on the music_example module. Because I did the bonus, I set 你好不好 as 1’b0 in _music register, and Mario Theme Song as 1’b1 in _music. So when switch4 is off, it will play 你好不好 and when it is on it will play the Mario Theme Song, if the music controller is in play mode.

To control the music controller, we can take a look at the player_control module. The player_control module takes clkDiv22, rst_op, _play, _repeat, _rewind, and _music as its inputs and outputs a 12 bits ibeatNum.

I set the parameter LEN to 512 because we have 512 beats in total (8 measures, 8*4*16), then declare prev_music to record the previous music being played, so that when we change the music it will play from the beginning. I first initiate it as the first song.

```
parameter LEN = 512;
reg prev_music = 1'b0;
```

Moving on, we have an always @(posedge clk, posedge reset) block. Inside, we first have 2 big if-else condition, which is if(reset) and else. If(reset), then both ibeat and prev_music will be set back to its initial value.

```
if (reset)
begin
    prev_music <= _music;
    ibeat <= 0;
end
```

Else, the music controller can operate based on user’s input. We first need to see whether _play is on or off.

If _play is on, then we check if the current music that is playing is not equal to previous music (1), then we have another condition, whether the controller is on _rewind mode or not. If the controller is on _rewind mode (2), then we need to set the value of prev_music to the current music, and set ibeat to LEN+12, which will make the music controller silent for a spare second and make the transition from one song to the other song more smoots, and play the switched song from the beginning, because if we directly set ibeat to 0, it will play the first note for a split second first and the transition is not smooth because of it. If it is not on _rewind mode (3), we also need to set prev_music to the current music, but ibeat can directly be set to 0 for it to immediately play the new song from the beginning.

Else, if the current music is the same as prev_music (4), which means that the user did not switch between one song to another, then again we have the if else condition for _rewind. If the user rewinds (5), then if the condition ((0 < ibeat) && (ibeat != LEN+12)), which is the boundaries for ibeat is satisfied, then ibeat will decrease its value by 1 every clock cycle. Otherwise, if the condition is not satisfied, then we make the music controller go silent by assigning ibeat to LEN+12. If the controller is not on _rewind mode (6), then we need to see if the controller is on _repeat mode or not. If the controller is on _repeat mode (7), when ibeat is equal to LEN+12, which is the end of the song, we need to set ibeat back to 0 which is the first note (beginning of the song) for the music to keep playing. When it has not reached LEN, if the boundaries for ibeat is still satisfied ((0 <= ibeat) && (ibeat != LEN)), then ibeat will keep increasing its value by one every clock cycle. Otherwise when it is out of boundary then the value of ibeat will be set back to 0 for the song to keep playing. If it is not in _repeat mode (8), when ibeat is equal to LEN+12, ibeat will be set back to 0 first, because LEN+12 will only be obtained _rewind mode is on and there has been change in the two songs, or when the music rewinds until the beginning of the song and we need to stop the song, and when the _play switch is turned on, the song must start playing from the beginning. Otherwise, if the boundaries for ibeat satisfies, which is ((0 <= ibeat) && (ibeat != LEN)), then ibeat will keep increasing its value by one every clock cycle and by the time it reaches the boundary, ibeat will be set to LEN to stop the music. Otherwise if the controller is not in _play mode (9), then ibeat will be set to its initial value itself. Note that we do not need to put _repeat condition inside _rewind condition since the spec says that _repeat has no effect on _rewind.

```
if(_play)
begin
    if(_music != prev_music)   (1)
    begin
        if(_rewind)   (2)
        begin
            prev_music <= _music;
            ibeat <= LEN+12;
        end

        else   (3)
        begin
            prev_music <= _music;
            ibeat <= 0;
        end
    end

    else   (4)
    begin
        if(_rewind)   (5)
        begin
            if((0 < ibeat) && (ibeat != LEN+12))
                ibeat <= ibeat - 1;
            else
                ibeat <= LEN + 12;
        end

        else   (6)
        begin
            if(_repeat) //play again if repeat   (7)
            begin
                if(ibeat == LEN+12)
                    ibeat <= 0;
                else
                begin
                    if((0 <= ibeat) && (ibeat != LEN))
                        ibeat <= ibeat + 1;
                    else
                        ibeat <= 0;
                end
            end

            else //stop otherwise   (8)
            begin
                if(ibeat == LEN+12)
                    ibeat <= 0;
                else   output reg [11:0]
                begin
                    if((0 <= ibeat) && (ibeat != LEN))
                        ibeat <= ibeat + 1;
                    else
                        ibeat <= LEN;
                end
            end
        end

else   (9)
    ibeat <= ibeat;
```

Volume is controlled in the note_gen module that takes clk, rst_one_pulse, note_div_left, note_div_right, volume, and outputs audio_left and audio_right. To control the volume, we need to assign the amplitude of the note to both audio_left and audio_right, and both are handled in different always@(*) block. We need to have 5 different volume levels so I implemented it as follows:

For audio_left:

```
always@(*)
begin
    if(note_div_left == 22'd1)
        audio_left = 16'h0000;
    else
    begin
        if(volume == 1)
            audio_left = (b_clk == 1'b0) ? 16'hFF40 : 16'hC0; //192
        else if(volume == 2)
            audio_left = (b_clk == 1'b0) ? 16'hF388 : 16'hC78; //3192
        else if(volume == 3)
            audio_left = (b_clk == 1'b0) ? 16'hE7D0 : 16'h1830; //6192
        else if(volume == 4)
            audio_left = (b_clk == 1'b0) ? 16'hDC18 : 16'h23E8; //9192
        else if(volume == 5)
            audio_left = (b_clk == 1'b0) ? 16'hD060 : 16'h2FA0; //12192
        else
            audio_left = 16'h0000;
    end
end
```

For audio_right:

```
always@(*)
begin
    if(note_div_right == 22'd1)
        audio_right = 16'h0000;
    else
    begin
        if(volume == 1)
            audio_right = (c_clk == 1'b0) ? 16'hFF40 : 16'hC0;
        else if(volume == 2)
            audio_right = (c_clk == 1'b0) ? 16'hF388 : 16'hC78;
        else if(volume == 3)
            audio_right = (c_clk == 1'b0) ? 16'hE7D0 : 16'h1830;
        else if(volume == 4)
            audio_right = (c_clk == 1'b0) ? 16'hDC18 : 16'h23E8;
        else if(volume == 5)
            audio_right = (c_clk == 1'b0) ? 16'hD060 : 16'h2FA0;
        else
            audio_right = 16'h0000;
    end
end
```

We need to match both amplitude of audio_left and audio_right for the volume to be balanced, and audio_left's value depends on b_clk while audio_right's value depends on c_clk. When b_clk (or c_clk) is equal to 1'b0, then audio_left (or audio_right) will be set to the two's complement hexadecimal value of when b_clk (or c_clk) is 1'b1. Here I have the if-else if-else statements to handle 5 different volume levels.

For the 3 octave levels, we know that for higher octave level (level 3), we need to double the frequency of octave level 2, and to make it to the lowest level (level 1), we need to halve the frequency of octave level 2. Hence, I handled them all here, all together with _mute:

```
assign freq_outL = (octave == 3) ? (50000000 / (_mute ? `silence : freqL*2)) : ((octave == 1) ? (50000000 / (_mute ? `silence : freqL/2)) : (50000000 / (_mute ? `silence : freqL)));
assign freq_outR = (octave == 3) ? (50000000 / (_mute ? `silence : freqR*2)) : ((octave == 1) ? (50000000 / (_mute ? `silence : freqR/2)) : (50000000 / (_mute ? `silence : freqR)));
```

To switch between volume and octave levels which depends on the buttons pressed, we first need to instantiate debounce and one pulse modules to handle volume, octave, also reset button. We first set volume as 3'd3 and octave as 3'd2 because that is their initial value after reset or after first being programmed, as described in the spec.

```
reg[2:0] volume = 3'd3;
reg[2:0] octave = 3'd2;

wire _volUP_debounced, _volDOWN_debounced, _higherOCT_debounced, _lowerOCT_debounced;
wire _volUP_one_pulse, _volDOWN_one_pulse, _higherOCT_one_pulse, _lowerOCT_one_pulse;

//rst button
debounce debounce_rst(.pb_debounced(rst_debounced), .pb(rst) , .clk(clkDiv16));
onepulse onepulse_rst(.signal(rst_debounced), .clk(clkDiv16), .op(rst_one_pulse));

//volume up button
debounce debounce__volUP(.pb_debounced(_volUP_debounced), .pb(_volUP) , .clk(clkDiv16));
onepulse onepulse__volUP(.signal(_volUP_debounced), .clk(clkDiv16), .op(_volUP_one_pulse));

//volume down button
debounce debounce__volDOWN(.pb_debounced(_volDOWN_debounced), .pb(_volDOWN) , .clk(clkDiv16));
onepulse onepulse__volDOWN(.signal(_volDOWN_debounced), .clk(clkDiv16), .op(_volDOWN_one_pulse));

//higher octave button
debounce debounce__higherOCT(.pb_debounced(_higherOCT_debounced), .pb(_higherOCT) , .clk(clkDiv16));
onepulse onepulse__higherOCT(.signal(_higherOCT_debounced), .clk(clkDiv16), .op(_higherOCT_one_pulse));

//lower octave button
debounce debounce__lowerOCT(.pb_debounced(_lowerOCT_debounced), .pb(_lowerOCT) , .clk(clkDiv16));
onepulse onepulse__lowerOCT(.signal(_lowerOCT_debounced), .clk(clkDiv16), .op(_lowerOCT_one_pulse));
```

Then to handle their values after a specific button was pressed:

```
always@(posedge clkDiv16 or posedge rst_one_pulse)
begin
    if(rst_one_pulse)
    begin
        volume <= 3'd3;
        octave <= 3'd2;
    end

    else
    begin
        if(_volUP_one_pulse == 1'b1 && volume == 3'd5)
            volume <= volume;
        else if(_volUP_one_pulse == 1'b1 && volume < 3'd5)
            volume <= volume + 3'd1;
        else if(_volDOWN_one_pulse == 1'b1 && volume > 3'd1)
            volume <= volume - 3'd1;
        else if(_volDOWN_one_pulse == 1'b1 && volume == 3'd1)
            volume <= volume;
        else
            volume <= volume;

        if(_higherOCT_one_pulse == 1'b1 && octave == 3'd3)
            octave <= octave;
        else if(_higherOCT_one_pulse == 1'b1 && octave < 3'd3)
            octave <= octave + 3'd1;
        else if(_lowerOCT_one_pulse == 1'b1 && octave > 3'd1)
            octave <= octave - 3'd1;
        else if(_lowerOCT_one_pulse == 1'b1 && octave == 3'd1)
            octave <= octave;
        else
            octave <= octave;
    end
end
```

When user pressed volume up when the current volume is at 5, it would not increase the volume level again, the same thing when user pressed volume down at level 1. Hence, volume's level will be set to volume again.

When user pressed higher oct when the current octave is at 3, it would not increase the volume level again, the same thing when user pressed lower oct at level 1. Hence, octave's level will be set to octave again.

This part is to represent the LED lights of volume and octave.

```
always@(*)
begin
    if(octave == 1)
        _led[15] = 1'b1;
    else
        _led[15] = 1'b0;

    if(octave == 2)
        _led[14] = 1'b1;
    else
        _led[14] = 1'b0;

    if(octave == 3)
        _led[13] = 1'b1;
    else
        _led[13] = 1'b0;
end
```

```
always@(*)
begin
    if(_mute)
        _led[4:0] = 5'b00000;

    else
    begin
        if(volume == 1)
            _led[4:0] = 5'b00001;
        else if(volume == 2)
            _led[4:0] = 5'b00011;
        else if(volume == 3)
            _led[4:0] = 5'b00111;
        else if(volume == 4)
            _led[4:0] = 5'b01111;
        else if(volume == 5)
            _led[4:0] = 5'b11111;
        else
            _led[4:0] = 5'b00000;
    end
end
```

To display the music notes on the seven-segment display, I instantiated the module SevenSegment which takes nums, rst_one_pulse, and clk as the inputs and generates DISPLAY and DIGIT as the outputs. I have new registers num1, num2, num3, and num4 which are 4 bits each to handle the four different digits on the seven-segment display, but I directly set the value of num4, num3, and num2 as 4'd11 which will not show anything, since the one we will be using is only num1.

```
assign nums = {4'd11, 4'd11, 4'd11, num1};
SevenSegment(.display(DISPLAY), .digit(DIGIT), .nums(nums), .rst(rst_op), .clk(clk));
```

num1's value depends on freqR, which is the audio right frequency that plays the

```
always@(*)
begin
    if(freqR == `la || freqR == `a || freqR == `ha || freqR == `ab || freqR == `as || freqR == `lab)
        num1 = 4'd7;
    else if(freqR == `lb || freqR == `b || freqR == `hb || freqR == `bb || freqR == `lbb)
        num1 = 4'd8;
    else if(freqR == `lc || freqR == `c || freqR == `hc || freqR == `hcs || freqR == `cb || freqR == `hhc)
        num1 = 4'd2;
    else if(freqR == `ld || freqR == `d || freqR == `hd || freqR == `hds || freqR == `db)
        num1 = 4'd3;
    else if(freqR == `le || freqR == `e || freqR == `he || freqR == `hes || freqR == `heb)
        num1 = 4'd4;
    else if(freqR == `lf || freqR == `f || freqR == `hf || freqR == `hfs)
        num1 = 4'd5;
    else if(freqR == `lg || freqR == `g || freqR == `hg || freqR == `gs || freqR == `llg)
        num1 = 4'd6;
    else if(`sil)
        num1 = 4'd10;
    else
        num1 = 4'd10;
end
```

melody of the song. With the defined music notes, we can directly assign values to num1 based on freqR, as follows:

If the music controller is not playing any notes or is silent, then num1 will take the value 4'd10 which displays '-'. Otherwise, the music controller will display either C, D, E, F, G, A, B, depending on the current music notes.

2. 學到的東西與遇到的困難

I encountered a problem when reading the music notes, as I was not really used to it. Other than that, I might be a little bit clueless when first trying to do this lab, but I can quickly understand what was going on and what I needed to do. I learned a lot about how different frequencies can produce different sounds, also how to control different volume and octave levels.

3. 想對老師或助教說的話

I really enjoyed doing this lab, and other labs were actually pretty fun too. I would like to thank the Professor and all the TAs who helped us during the labs and guided us step by step from having the tiniest knowledge about Verilog, Vivado, and the FPGA board itself, to knowing how to implement lots of cool stuffs and finally making our own final project. I am happy to be able to learn about a lot of things and learning from lots of mistakes by participating in this course.