Lab 7

學號: 110011138 姓名: 楊立慈

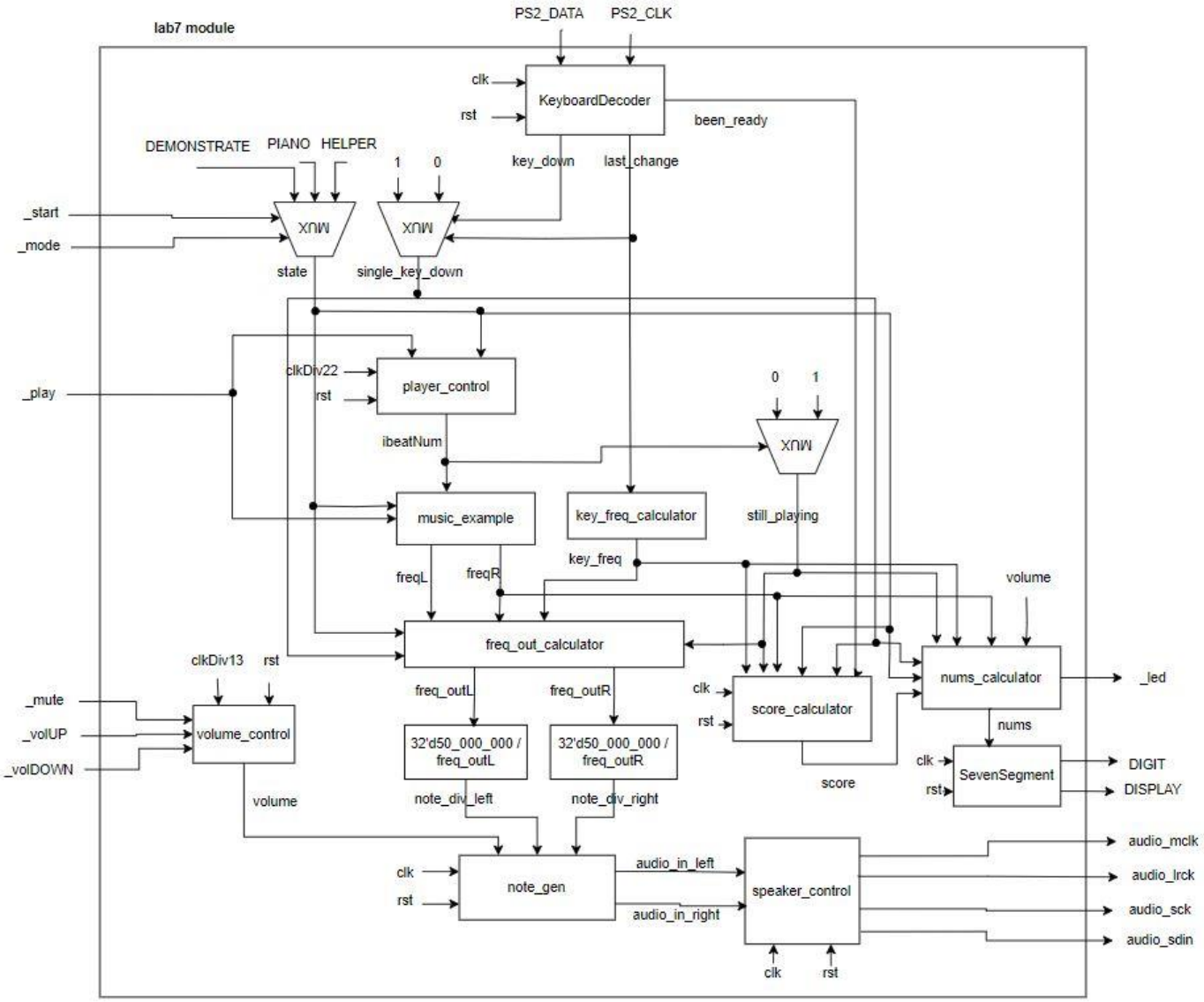
A. Lab Implementation

1. Block Diagram of the design with explanation (10%)

下圖為 lab7 的 top module block diagram。首先，圖中的 clkDiv22、clkDiv13 是用 clock_divider module output 出來的；state 是根據_start、_mode 訊號經過 MUX 判斷目前要在 DEMONSTRATE 或 PIANO 或 HELPER 模式；single_key_down 是根據 KeyboardDecoder output 的 key_down 和 last_change 判斷 key_down 裡是否只有 last_change 的單一按鍵按下。(圖中三個 MUX 的 code 在 block diagram 底下的圖)

再來是最主要產生音樂的部分。首先是自動播音樂的部分，由 player_control 產生數目前音樂進行到哪的 ibeat 訊號後，會送入 music_example 並根據 ibeat、目前 state、是否有 pause 來決定輸出的 freqL、freqR 訊號，並送入 freq_out_calculator。另一邊，由鍵盤決定音符的部分，將 last_change 經過 key_freq_calculator 計算 last_change 對應的 key_freq 是多少，並同樣送入 freq_out_calculator。在 freq_out_calculator 則會根據 state 決定目前的左右聲道要用 music_example 還是鍵盤的 key_freq，並將最終的 freq_outL、freq_outR 經過一個被 50M 除的運算後，送到 note_gen。在 note_gen 則會根據 volume 值(0~5)決定最終的聲波的振幅，並將結果輸入到 speaker_control 計算最後要給 audio 的輸出。

最後則是一些顯示、在 HELPER state 計算分數的 block。首先我們有個 MUX 會根據 ibeatNum 的值判斷是否還在玩遊戲(still_playing 訊號)。在 score_calculator 當還在玩的時候，會根據目前鍵盤 key_freq、主旋律 freqR、一些跟鍵盤有關的訊號等，判斷分數是否要增加。而計算出的 score 會跟 freqR、key_freq、volume 等訊號一起送進 nums_calculator 來計算 SevenSegment module 要顯示用的 nums，並順便輸出 led 的 pattern。




```
assign still_playing = (ibeatNum < LEN) ? 1'b1 : 1'b0;
assign single_key_down = (key_down == 512'b1 << last_change) ? 1'b1 : 1'b0;
assign state = (_mode == 1'b1) ? DEMONSTRATE :
                    (_start == 1'b1) ? HELPER : PIANO;
```

2. Partial code screenshot with the explanation (35%)

a. Volume control

下圖為 volume_control module。圖中的 volUP_p、volDOWN_p 等訊號為 button 的訊號經過 debounce、one_pulse 處理後的。首先我的 volume 決定用 DFF 儲存，並在 reset 時設為 3。在計算 next_volume 時則會根據 volUP_p、volDOWN_p 是否有拉起來決定增加 or 減少 volume。最終輸出的 volume_out 會依據是否要 mute 決定輸出是 0 會是正常的 volume。

```
assign volume_out = (mute == 1'b1) ? 3'd0 : volume;

always @(posedge clk or posedge rst) begin
    if (rst == 1'b1) begin
        volume <= 3'd3; // default volume
    end
    else begin
        volume <= next_volume;
    end
end

always @(*) begin
    next_volume = volume;

    if (volUP_p == 1'b1)
        next_volume = (volume == 5) ? 5 : volume + 1;
    else if (volDOWN_p == 1'b1)
        next_volume = (volume == 1) ? 1 : volume - 1;
end
```

在 note_gen 裡計算最終要輸出的聲波 amplitude 時則會根據 volume 的值最為倍數計算(如下圖)。當 volume 是 0 時，則要將振幅設為 0，不然就是正常輸出 amplitude。

```
always @(*) begin
    amplitude = 16'd1000 + 16'd3000 * volume;

    if (note_div_left == 22'd1 || volume == 3'd0) begin
        audio_left = 16'h0000;
        audio_right = 16'h0000;
    end
    else begin
        // note: in 2's complement, -N = ~N + 1'b1
        audio_left = (b_clk == 1'b0) ? ~amplitude + 1'b1 : amplitude;
        audio_right = (c_clk == 1'b0) ? ~amplitude + 1'b1 : amplitude;
    end
end
```

最後是顯示的部分，也就是 nums_calculator 裡計算 led 的部分。這裡我把 led 拆成左半邊跟右半邊，左半邊是給 HELPER 用的，右半邊則是顯示 volume 的。計算的 Comb. Logic 如下圖，根據不同的 volume 用 case 給予不同的顯示 pattern。

```
always @(*) begin
    led_right = 5'b00_000;
    case (volume)
        3'd1: led_right = 5'b00_001;
        3'd2: led_right = 5'b00_011;
        3'd3: led_right = 5'b00_111;
        3'd4: led_right = 5'b01_111;
        3'd5: led_right = 5'b11_111;
    endcase
end
```

b. DEMONSTATE mode

下圖為 player_control module 裡有關輸出給 music_example 的 ibeat 的 next_ibeat 的計算。首先在 DEMONSTRATE state 時，如果正常播放的話 next_ibeat 就照常從 0 數到 LEN-1 再從頭循環的數，否則就維持 ibeat 的值，代表暫停歌曲。在其他 state 時則維持 ibeat 的值，這樣之後回到 DEMONSTRATE state 就可以繼續由上一次 ibeat 的值播放。

```
always @* begin
    next_ibeat = ibeat;
    next_ibeat2 = ibeat2;

    case (state)
        DEMONSTRATE: begin
            if (_play == 'b1) next_ibeat = (ibeat + 1 < LEN) ? ibeat + 1 : 0; // repeat
            else next_ibeat = ibeat;
            next_ibeat2 = 0; // prepare for helper state
        end
        PIANO: begin
            next_ibeat2 = 0; // prepare for helper state
        end
        HELPER: begin
            // stops after last note (i.e., stops at LEN, so that we can detect song end)
            next_ibeat2 = (ibeat2 < LEN) ? ibeat2 + 1 : LEN;
        end
    endcase
end
```

而在 player_control 輸出 ibeat_out 時則會用以下的 Comb. Logic 根據 state 決定輸出 ibeat。

```
always @(*) begin
    ibeat_out = 0;

    case (state)
        DEMONSTRATE: ibeat_out = ibeat;
        HELPER: ibeat_out = ibeat2;
    endcase
end
```

下圖為在 music_example 裡決定輸出的 freqR、freqL 的 Comb. Logic。其中 toneL、toneR 是根據 ibeat 決定的目前音符的 freq。然而我們要考慮暫停的情形，因此在 DEMONSTRATE state 時要看是否要 play 來決定輸出是 tone 或者 sil。

```
always @(*) begin
    // silence by default
    freqL = sil;
    freqR = sil;

    case (state)
        DEMONSTRATE: begin
            if (_play == 1'b1) begin
                freqL = toneL;
                freqR = toneR;
            end
        end
        HELPER: begin
            freqL = toneL;
            freqR = toneR;
        end
    endcase
end
```

下圖為 freq_out_calculator 裡計算 freq_outL、freq_outR 的 Comb. Logic。在此 state 就是單純把 music_example 給我們的 freqL、freqR 輸出就好。

```
DEMONSTRATE: begin
    freq_outL = freqL;
    freq_outR = freqR;
end
```

最後則是顯示的部分，也就是 nums_calculator 會計算的 nums、led 輸出。首先我會用 Comb. Logic 計算當前要顯示的 freq 要以 music_example 的 freqR 或鍵盤的 key_freq 為準，如下圖。由於在此 state 顯示的音符以右手旋律為準，因此 freq 是 freqR。

```
DEMONSTRATE: begin
    freq = freqR;
end
```

再來則依據當前的 freq 決定它對應的 freq_char、freq_num，舉例來說如果 freq 是 C5 的 freq 值，則 freq_char 是 C_NUM (C 在 nums 裡要給的值)、freq_num 是 5。Comb. Logic 如下圖，基本上 freq_num 可以用 freq 的範圍判斷，freq_char 則只能一個一個音符檢查。

```
always @(*) begin
    // display -- by default
    freq_char = DASH_NUM;
    freq_num = DASH_NUM;

    // freq_num
    if (freq >= c3 && freq <= b3) freq_num = 5'd3;
    else if (freq >= (c3 << 1) && freq <= (b3 << 1)) freq_num = 5'd4;
    else if (freq >= (c3 << 2) && freq <= (b3 << 2)) freq_num = 5'd5;

    // freq_char
    if (freq == c3 || freq == (c3 << 1) || freq == (c3 << 2)) freq_char = C_NUM;
    else if (freq == d3 || freq == (d3 << 1) || freq == (d3 << 2)) freq_char = D_NUM;
    else if (freq == e3 || freq == (e3 << 1) || freq == (e3 << 2)) freq_char = E_NUM;
    else if (freq == f3 || freq == (f3 << 1) || freq == (f3 << 2)) freq_char = F_NUM;
    else if (freq == g3 || freq == (g3 << 1) || freq == (g3 << 2)) freq_char = G_NUM;
    else if (freq == a3 || freq == (a3 << 1) || freq == (a3 << 2)) freq_char = A_NUM;
    else if (freq == b3 || freq == (b3 << 1) || freq == (b3 << 2)) freq_char = B_NUM;
end
```

最後就是計算 led_left、nums 的部分，Comb. Logic 如下圖。由於在此不用顯示左半邊的 led 因此就是預設值 7'b0_000_000。而 nums 則是左邊兩個 dash，右邊則是 freq 對應的符號。

```
DEMONSTRATE: begin
    nums = {DASH_NUM, DASH_NUM, freq_char, freq_num};
end
```

c. PLAY mode – Piano

下圖為 freq_out_calculator 裡計算 freq_outL、freq_outR 的 Comb. Logic。由於此 state 就是單純按按鍵就產生聲音，且我一次只會允許一個按鍵產生聲音，因此左右聲道都設為先前由 key_freq_calculator 計算的 key_freq，並且只有當 single_key_down 時才會是 key_freq，否則就是預設值 SILENCE。

```
PIANO: begin
    if (single_key_down) begin
        freq_outL = key_freq;
        freq_outR = key_freq;
    end
end
```

再來就是顯示的 nums_calculator 部分。下圖為計算要用的 freq 的 Comb. Logic。由於我們只允許一個按鍵按下會產生聲音，因此當 single_key_down 則 freq 是 key_freq，否則就是 SILENCE 代表沒聲音。

```
PIANO: begin
    if (single_key_down) freq = key_freq;
    else freq = SILENCE;
end
```

在此 state 同樣的 led_left 不用管，因此就是預設值 7'b0_000_000。至於 nums 則同樣左邊

是--，右邊是 freq 對應的符號。

```
PIANO: begin
    nums = {DASH_NUM, DASH_NUM, freq_char, freq_num};
end
```

d. PLAY mode – Helper

下圖為在 freq_out_calculator 裡計算 freq_outL、freq_outR 的 Comb. Logic。當遊戲還在進行且只有單一按鍵按下時，我們才會輸出 key_freq 的聲音，否則就是預設值 SILENCE。

```
HELPER: begin
    if (still_playing && single_key_down) begin
        freq_outL = key_freq;
        freq_outR = key_freq;
    end
end
```

由於我們希望在背景可以繼續讓 music_example 產生歌曲目前對應的音符(freqR、freqL)，但我們又不能用原本 DEMONSTRATE 的 ibeat，因此我在 player_control 用了另一個 DFF 存 ibeat2 作為此 state 的 music_example 要用的 beat。決定 next_ibeat2 的 Comb. Logic 如下圖，可看到我們在其他 state 都把 next_ibeat2 設為 0，這樣每次進入 HELPER 都會從頭開始，並且在 HELPER 時，我們只讓 ibeat2 數到 LEN 就停，不會從頭開始。(要數到 LEN 是因為我的 still_playing 是利用 ibeat < LEN 判斷是否還在玩)

```
always @* begin
    next_ibeat = ibeat;
    next_ibeat2 = ibeat2;

    case (state)
        DEMONSTRATE: begin
            if (_play == 'b1) next_ibeat = (ibeat + 1 < LEN) ? ibeat + 1 : 0; // repeat
            else next_ibeat = ibeat;
            next_ibeat2 = 0; // prepare for helper state
        end
        PIANO: begin
            next_ibeat2 = 0; // prepare for helper state
        end
        HELPER: begin
            // stops after last note (i.e., stops at LEN, so that we can detect song end)
            next_ibeat2 = (ibeat2 < LEN) ? ibeat2 + 1 : LEN;
        end
    endcase
end
```

而在 player_control 輸出 ibeat_out 時則會用以下的 Comb. Logic 根據 state 決定輸出 ibeat2。

```
always @(*) begin
    ibeat_out = 0;

    case (state)
        DEMONSTRATE: ibeat_out = ibeat;
        HELPER: ibeat_out = ibeat2;
    endcase
end
```

這樣一來，我們 freqL、freqR 就會在 HELPER state 也能正常輸出歌曲的音符，我們可以將其用在 score_calculator、顯示用的 nums_calculator。

在 music_example 決定 freqR、freqL 的 Comb. Logic 也要記得設為 toneL、toneR (雖然嚴格來講 toneL 在這裡不會用到)，並且因為不會暫停，因此不用考慮_play==1'b1，code 如下。

```
HELPER: begin
    freqL = toneL;
    freqR = toneR;
end
```

再來就是顯示的 nums_calculator 部分。首先是計算要用的 freq 的 Comb. Logic，如下圖，當還在進行遊戲則用 music_example 產生的主旋律 freqR，否則當遊戲結束，就使用歌曲的最後一個音符，也就是 C5。

```
HELPER: begin
    if (still_playing) freq = freqR;
    else freq = (c3 << 2); // Hardcoded last note: C5
end
```

再來是計算 nums 跟 led_left 的 Comb. Logic，如下圖。在這裡 nums 最左邊兩個 digit 會是 score 的十位、個位，右邊則是 freq 對應的符號。Led_left 則根據 freq_char 的值來讓不同位置的 led 亮起來，當遊戲結束後，則將 led_left 設為全亮。

```
HELPER: begin
    nums[19:15] = score / 10;
    nums[14:10] = score % 10;
    nums[9:0] = {freq_char, freq_num};

    case (freq_char)
        C_NUM: led_left[6] = 1'b1;
        D_NUM: led_left[5] = 1'b1;
        E_NUM: led_left[4] = 1'b1;
        F_NUM: led_left[3] = 1'b1;
        G_NUM: led_left[2] = 1'b1;
        A_NUM: led_left[1] = 1'b1;
        B_NUM: led_left[0] = 1'b1;
    endcase
    if (!still_playing) led_left = 7'b1_111_111;
end
```

e. Score mechanism of the Helper

首先，我的 score 是用 DFF 存的，下圖為 score_calculator 計算 next_score 的 Comb. Logic。在其他 state 時將 score 歸零；在 HELPER state 時，如果只有一個按鍵按下且按下的按鍵是有聲音的(i.e., 合法的按鍵)且遊戲還在進行且按鍵的 freq 等於 music_example 主旋律的 freq，則增加一分(注意不要超過 99 分)。當連續按壓按鍵時，由於鍵盤會傳來多個 been_ready，因此如果按對按鍵的話，分數會短時間內不斷增加，不過由於我選的歌的音符都很短，因此連續按壓頂多一個音符增加 10 分左右而已。

```
always @(*) begin
    next_score = score;

    case (state)
        DEMONSTRATE: begin
            next_score = 0;
        end
        PIANO: begin
            next_score = 0;
        end
        HELPER: begin
            if ((been_ready && single_key_down)
                && (key_freq != SILENCE && still_playing)
                && (key_freq == freqR))
                next_score = (score == 8'd99) ? 8'd99 : score + 1'b1;
        end
    endcase
end
```

B. Questions and Discussions (40%)

(a) 可以仿照目前的 music_example module，並寫另一個 music_example2 module 給另一首歌，並把以上兩個 module 產生的 freqR、freqL 各自經過一個 MUX 並根據 switch 狀態決定目前要用哪一首歌的 freqR、freqL，再將送進 freq_out_calculator 的 freqR、freqL 訊號改為上述 MUX 輸出的訊號。

此外，如果 music_example2 跟 music_example 用一樣的 ibeat 會產生用 switch 切歌後，新的歌不會從頭播，而是從同個時間點繼續播新的歌的現象。如果希望新的歌可以從頭播的話，要在 player_control 再加上一個 ibeat3 訊號作為第二首歌要用的 ibeatNum，如此一來當第一首歌正在播，第二首歌的 ibeat3 訊號就可以歸零，這樣切到第二首歌時就會從頭開始播，反之亦然。要注意的是在 player_control 裡在 DEMONSTRATE mode 就要再根據 switch 決定目前要輸出 ibeat 還是 ibeat3。

(b) 這相當於原本的 2 個 beat 變成現在的 1 個 beat，會造成音樂以 2 倍速撥放的效果。此外，由於 ibeat 初始值是 0，因此 ibeat 只會是偶數，所以如果有重複音符中間的 sil 是在奇數的 ibeat 上的話，該 sil 不會有效果，造成該重複音符會聽不出來。

C. Problem Encountered (10%)

在寫 music_example 時，我原本想說直接慢慢手刻就好，但大概寫到 200 多個 ibeat 時就覺得實在太慢了，而且看到眼睛很花。因此我決定另外用 python 寫自動生成 code 的程式，而在這

過程中比較難的部分是如何判斷重複音符 sil 的條件、如何印出排版好看的 comment、決定怎麼存一首歌、處理 index out of bounds 的邊界條件等。雖然最後花了比預期更多的時間，但好處是 final project 如果需要音樂的話可以很快速的用這個程式直接生成 music_example 的 code，python code 的連結: link

D. Suggestions (5%)

希望每次 lab 都可以提供像是這次 lab 的 appendix 裡關於音色的額外資訊，雖然不一定每個人都會做，但看過去還是蠻有趣的，才發現原來這次 lab 使用的零件還有其他部分可以玩。