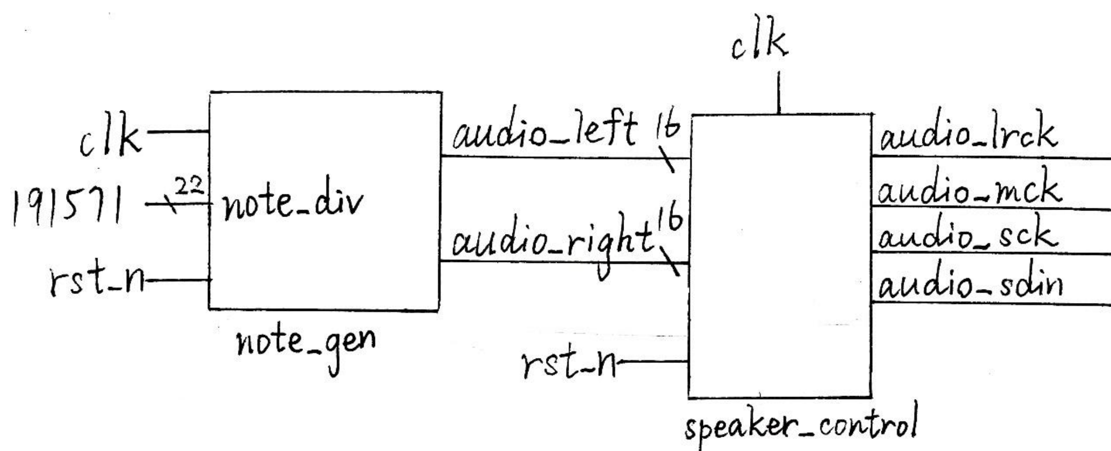
1 Please design an audio-data parallel-to-serial module to generate the speaker control signal with 100MHz system clock, 25 MHz master clock, (25/128) MHz Left-Right clock (Fs), and 6.25 MHz (32Fs) sampling clock.
1.1 Design a general frequency divider to generate the required frequencies for speaker clock.
1.2 Design a stereo signal parallel-to-serial processor to generate the speaker control signals. Please use verilog simulation waveform to verify your control signal.

1. Specification
內容: 寫下你的電路中的 **inputs, outputs** 以及其 **bit widths**，名稱必須跟你的 **verilog code** 中相同。

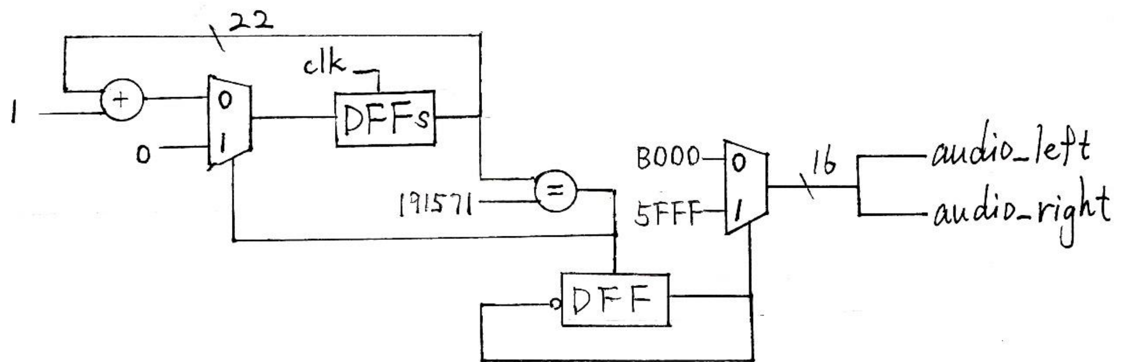
```
module speaker(
  output audio_mclk,     // master clock
  output audio_lrck,     // left-right clock
  output audio_sck,      // serial clock

  output audio_sdin,     // serial audio data input

  input  clk,            // clock from the crystal
  input  rst_n           // active low reset
);
```

2. Block Diagram
內容: 電路中的 **Block diagram(**可以用手畫拍照或電腦繪圖**)**。

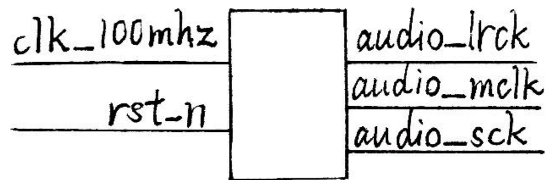
speaker.v : (top)



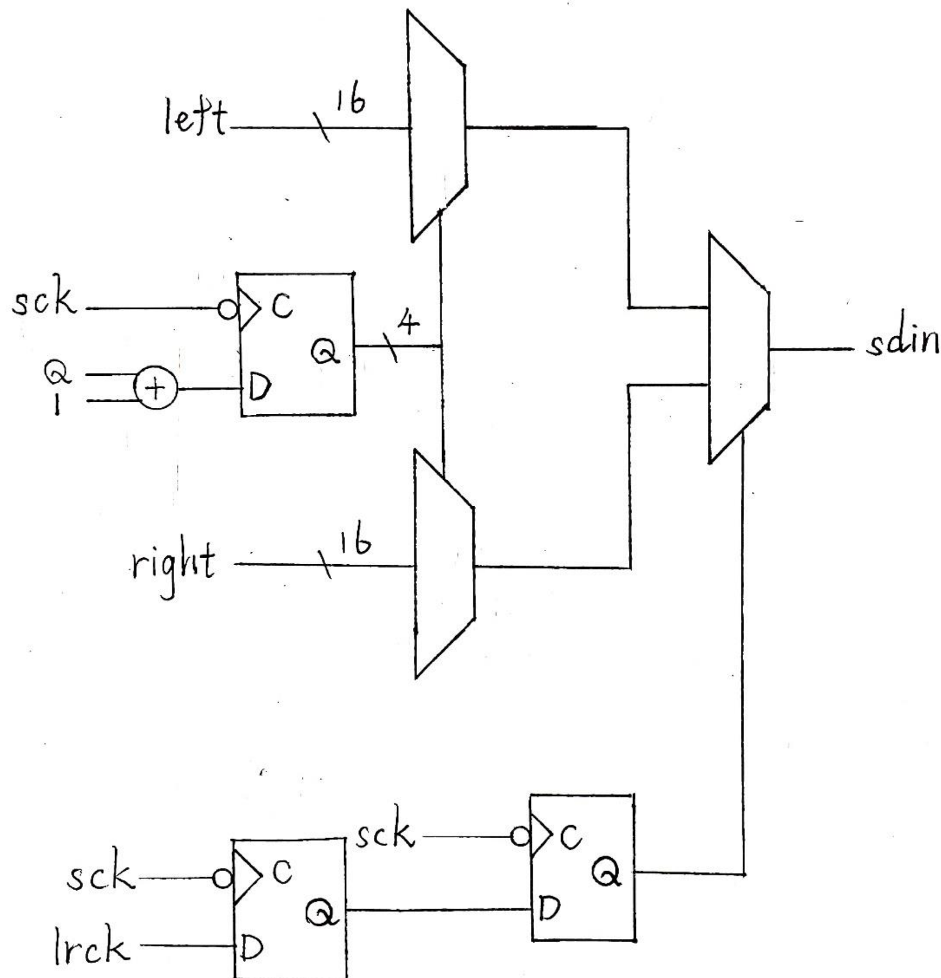
note_gen.v :



mclk_lrck_sck.v :



speaker_control.v :



3. Finite state machine

內容: 電路中的 Finite state machine，若無則寫無。

無

4. Implement

內容: 請列出相關的 logic function、詳細用文字解釋電路的運作方法、結果等等，可以貼 code 解釋或拍 FPGA 輔助解釋(但不能只貼 code 跟 FPGA 結果)。

note_gen.v 跟老師的投影片一樣，用一個二十二位元的計數器和一個 T Flip-flop 來產生所需的頻率。

```
20  // The buzzer frequency is obtained by dividing crystal 100 MHz by N
21  // The buzzer clock (note_div) is periodically inverted for every N/2 cloc
22  always @*
23    if (clk_cnt == note_div) begin
24      clk_cnt_next = 22'd0;
25      b_clk_next   = ~b_clk;
26    end else begin
27      clk_cnt_next = clk_cnt + 1'b1;
28      b_clk_next   = b_clk;
29    end
30
31  always @(posedge clk or negedge rst_n)
32    if (~rst_n) begin
33      clk_cnt <= 22'd0;
34      b_clk   <= 1'b0;
35    end else begin
36      clk_cnt <= clk_cnt_next;
37      b_clk   <= b_clk_next;
38    end
```

把產生的頻率連接到 MUX，以產生適當大小的震幅。

```
40  // Assign the amplitude of the note
41  // Dynamic Range 16'h5FFF ~ 16'hB000
42  assign audio_left  = (b_clk == 1'b0) ? 16'hB000 : 16'h5FFF;
43  assign audio_right = (b_clk == 1'b0) ? 16'hB000 : 16'h5FFF;
```

這樣就可以產生任意頻率的方波了。

我的 speaker_control.v 的 input 和 output 如下。

```
 1 module speaker_control(
 2   output        audio_mclk,  // master clock
 3   output        audio_lrck,  // left-right clock
 4   output        audio_sck,   // serial clock
 5
 6   output        audio_sdin,  // serial audio data input
 7
 8   input [15:0] audio_left,  // left  channel audio data input
 9   input [15:0] audio_right, // right channel audio data input
10
11   input         clk,         // clock from the crystal
12   input         rst_n        // active low reset
13 );
```

首先，先依據 100MHz 的時脈，用除頻器除以四，以創造出 25MHz 的 mclk。再除以四，則製造出 sck。再除以三十二，則可以製造出 lrck。為了使上述三個時脈同步，我先使用 free_run_counter.v 來製造出所有可能會用到的二次冪頻率。

```
 1 module free_run_counter#(
 2   parameter               WIDTH = 9
 3 ) (
 4   output reg [WIDTH-1:0] clk_free_run,
 5   input                  clk,
 6   input                  rst_n
 7 );
 8
 9   always @(posedge clk or negedge rst_n)
10     if (~rst_n) clk_free_run <= 'b0;
11     else        clk_free_run <= clk_free_run + 'b1;
12
13 endmodule
```

然後 mclk_lrck_sck.v 就可以直接把特定的 clk_free_run 連接到 mclk、sck、lrck。

```
 1 module mclk_lrck_sck(
 2   output audio_mclk,
 3   output audio_lrck,
 4   output audio_sck,
 5
 6   input  clk,
 7   input  rst_n
 8 );
 9   wire [8:0] clk_free_run;
10
11   free_run_counter free_run_counter(
12     .clk_free_run,
13     .clk,
14     .rst_n
15   );
16
17   assign audio_mclk = clk_free_run[1]; // 25MHz (~24.5760MHz)
18   assign audio_lrck = clk_free_run[8]; // 25MHz/128 (~192kHz)
19   assign audio_sck  = clk_free_run[3]; // 25MHz/128*32 (25MHz/4)
20
21 endmodule
```

這樣就可以直接於 speaker_control.v 裡面使用上述兩個模組所產生的 mclk、sck、lrck。

```
15   mclk_lrck_sck mclk_lrck_sck(
16     .audio_mclk,
17     .audio_lrck,
18     .audio_sck,
19     .clk,
20     .rst_n
21   );
```

至此，speaker_control.v 的 output 只剩 sdin 還沒製造出來，其餘 mclk、sck、lrck 都有了。接下來要想辦法製造出符合 DAC CS4344 的規定的 sdin。

如同老師的影音檔內所述，sdin 的 MSB 必須要比 lrck 的 riding edge 或 falling edge 晚一個 sck 週期。這個部分我的作法參考自一份 I²S bus specification 文件中所附的 block diagram，如下圖。（Philips Semiconductors; February 1986; Revised: June 5, 1996）留意下圖的左下角的那兩個 D Flip-flop，這是一個可以產生這種「sdin 比 lrck 慢一拍」的效果的絕妙方法。

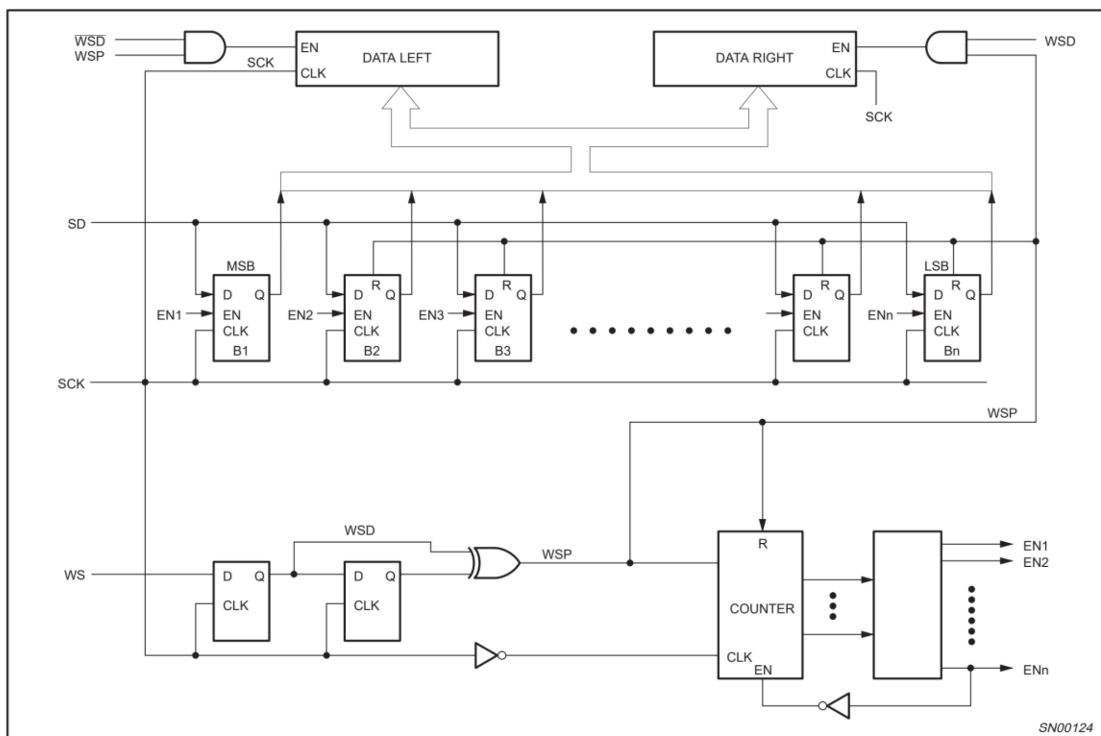


Figure 6. Possible receiver configuration. The latches and the counter use synchronous set,

我先使用 D Flip-flop 來製造出一個比正常的 lrck 晚一個 sck 週期的訊號。這裡要使用兩個 D Flip-flop 的原因是因為，如果只使用一個的話，此 D Flip-flop 會因為連接到 D 的 lrck 與連接到 CLK 的 sck 本來就是同步訊號的關係，而使其產生出來的 lrck_delayed 沒有被延遲一個 sck 週期。但是，我們並非徒勞無功，因經過這第一個 D Flip-flop 處理之後所產生的 lrck_delayed 已經晚於 sck 的 falling edge，所以 lrck_delayed 不可能在同一個 sck 週期內再穿過任何其他一樣由 sck 的 falling edge 作為 CLK 的 D Flip-flop。因此，如果在此時另外多使用一個 D Flip-flop 來串連，其輸出 lrck_delayed_delayed 就一定會剛好比正常的 lrck 晚一個 sck 週期。綜上所述，我可以用 verilog 描述如下。

```
23  reg audio_lrck_delayed;
24
25  always @(negedge audio_sck or negedge rst_n)
26    if (~rst_n) audio_lrck_delayed <= 'b0;
27    else        audio_lrck_delayed <= audio_lrck;
28
29  reg audio_lrck_delayed_delayed;
30
31  always @(negedge audio_sck or negedge rst_n)
32    if (~rst_n) audio_lrck_delayed_delayed <= 'b0;
33    else        audio_lrck_delayed_delayed <= audio_lrck_delayed;
```

因 audio_left 和 audio_right 都是 parallel-loading 的，所以需要建構 parallel-to-serial 的功能。我仿照 I²S bus specification 文件中所附的 block diagram 中右下角的方法，使用一個四位元的計數器，以便在每一個 lrck 中的不同的 sck 中，分別讓 audio_left 和 audio_right 的不同位元連接到 sdin。因為是由 MSB 到 LSB 倒序傳送，所以計數器要下數。

```
35  reg [3:0] counter;
36
37  always @(negedge audio_sck or negedge rst_n)
38    if (~rst_n) counter <= 4'h0;
39    else        counter <= counter - 'b1;
```

雖然 I²S bus specification 文件圖中畫的是數個 latch，但其實不需要在 verilog 中描述出來。我們可以直接用 audio_left[counter] 和 audio_right[counter] 達到相同的效果。

```
41  assign audio_sdin = audio_lrck_delayed_delayed
42                    ? audio_left [counter]
43                    : audio_right[counter];
44
45 endmodule
```

至此，我們已將所有需要使用到的模組建構完畢。

speaker.v (top) 的寫法跟著老師的 PPT 一步一步照樣做就好，把該接的線接一接即可。

```
12  // Declare internal nodes
13  // Note generation
14  wire [15:0] audio_left;
15  wire [15:0] audio_right;
```

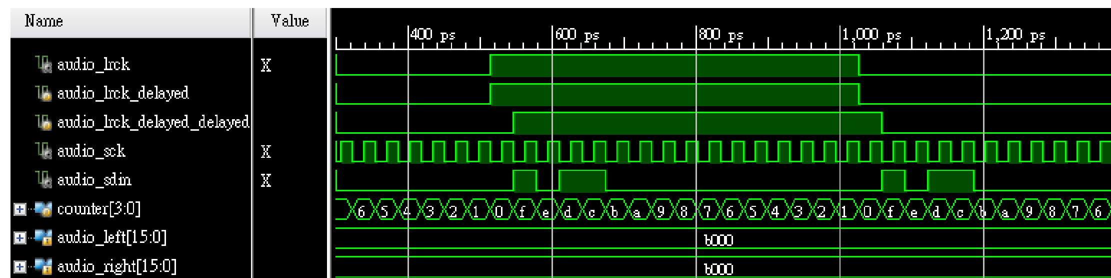
先用 note_gen.v 製造特定頻率、特定震幅的聲音資料。

```
17  // a general frequency divider
18  // to generate the required frequencies for speaker clock
19  note_gen Ung(
20    .audio_left,            // left sound audio
21    .audio_right,           // right sound audio
22    .note_div(22'd191571),  // div for note generation
23    .clk,                   // clock from crystal
24    .rst_n                  // active low reset
25  );
```

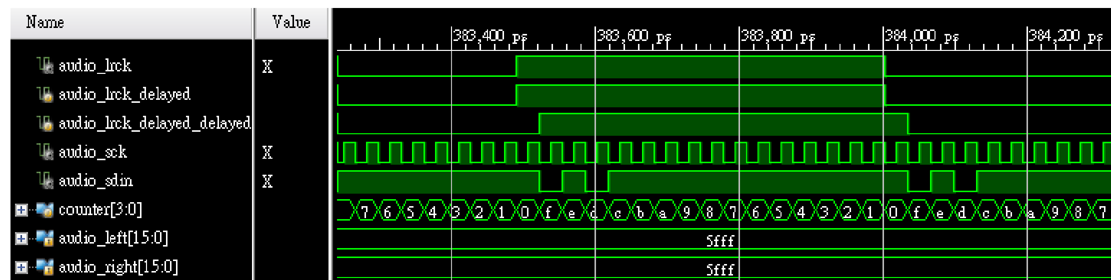
再連接到 parallel-to-serial，並產生 output 所需的 mclk、sck、lrck 等訊號，這樣就大功告成了。

```
27  // Speaker controllor
28  // with a stereo signal parallel-to-serial processor
29  // to generate the speaker control signals
30  speaker_control Usc(
31    .audio_mclk,  // master clock
32    .audio_lrck,  // left-right clock
33    .audio_sck,   // serial clock
34    .audio_sdin,  // serial audio data input
35    .audio_left,  // left channel audio data input
36    .audio_right, // right channel audio data input
37    .clk,         // clock from the crystal
38    .rst_n        // active low reset
39  );
```

如下圖，第一個訊號 lrck 是輸出到 DAC 的，第二個 lrck_delayed 是經過一次 D Flip-flop 處理後的訊號，而 lrck_delayed_delayed 則是經過兩次 D Flip-flop 的訊號。圖中可見，lrck_delayed_delayed 比輸出到 DAC 的 lrck 慢了一個 sck 週期。且由於 counter 是在重置之後是從 4'h0 開始下數的，因此在 lrck 的 rising 或 falling edge 之後的第一個 sck 內的數值都會是 4'h0，然後到了第二個 sck 的時候才會下數減一，變成 4'hF。



上圖要傳送的資料 sdin 是 16'hB000，可觀察到其 MSB 比 lrck 還要慢了一個 sck 的週期才被傳送。



上圖要傳送的資料 sdin 是 16'h5FFF，可觀察到其 LSB 被傳送的時機在於 lrck 的 riding 或 falling edge 之後的第一個 clk 週期內。由以上 simulation 的波形圖可以驗證我寫的 module 所製造出來的 lrck 與 sdin 兩者都符合 DAC CS4344 的規定。

5. Conclusion

內容: 可以寫下你的這個 lab 的想法、遇到的問題、解決方法、心得等等，請自由發揮。

這題比較讓人感到有樂趣，尤其是在經過不斷的調整和測試之後，終究還是順利的製造出了符合 DAC CS4344 的規定的 lrck 與 sdin 的波形圖。這也是我第一次寫 parallel-to-serial 的 converter，一開始我還有點不知道應該要如何下手，就很直接的寫了 32 個 D Flip-flop，想說可以在一個 lrck 的週期內用一個五位元的計數器從 31 數到 0，並拿計數器的第五的位元（即 MSB）用來判斷要傳送 left 還是 right 的資訊。結果看了波型圖才發現這種做法的瓶頸在於，sck 的相位剛好相反，要切換到下一個 bit 的資料的時機一不小心就會將 rising 和 falling edge 弄混，且修正之後的 code 有點複雜，難以用簡潔且讓人易懂的方法描述出該硬體的配置。所以在搜尋相關文獻之後，索性參考那份 I^2S bus specification 文件所附的 block-diagram，才改成了現在的做法。

2 Speaker control

2.1 Please produce the buzzer sounds of Do, Re, and Mi by pressing buttons (Left, Center, Right) respectively. When you press down the button, the speaker produces corresponding frequency sound. When you release the switch, the speaker stops the sound.

2.2 Please control the volumn of the sound by pressing button (Up) as increase and (Down) and decrease the volumn. Please also quantize the audio dynamic range as 16 levels and show the current sound level in the 7-segment display.

1. Specification

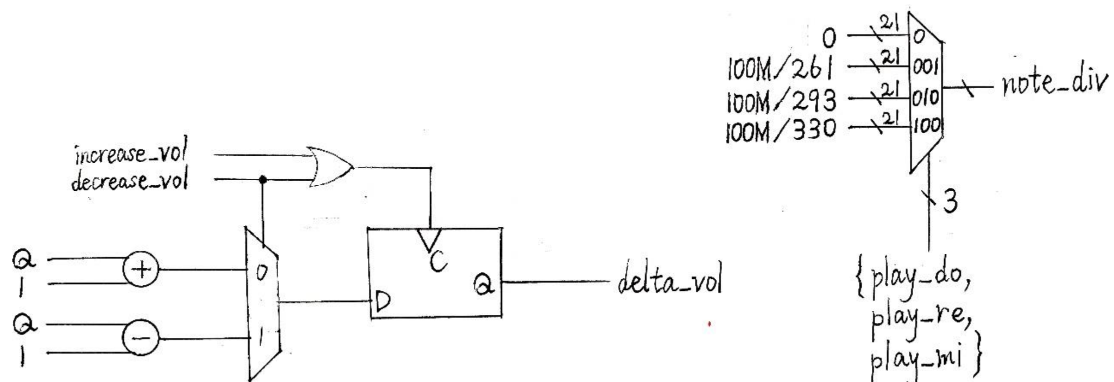
內容: 寫下你的電路中的 inputs, outputs 以及其 bit widths，名稱必須跟你的 verilog code 中相同。

```
module speaker(
  output [7:0] cathode,     // 七段顯示器

  output audio_mclk,        // master clock
  output audio_lrck,        // left-right clock
  output audio_sck,         // serial clock

  output audio_sdin,        // serial audio data input

  input  increase_volume,   // 按一下會增加音量
  input  decrease_volume,   // 按一下會減少音量

  input  play_do,           // 按下時會撥放 Do
  input  play_re,           // 按下時會撥放 Re
  input  play_mi,           // 按下時會撥放 Mi

  input  clk,               // clock from the crystal
  input  rst_n              // active low reset
);
```

2. Block Diagram

內容: 電路中的 Block diagram(可以用手畫拍照或電腦繪圖)。

跟上一題比較起來，這一題只要加上兩個新的模組就好。首先，因為要有改變音高的能力，所以使一個 MUX 來依照當下按下的按鈕來產生適當的除數，再把除數傳送到上一題的除頻器以便依照被按下的按鈕來產生對應的聲音資訊。由於還要有調整音高的能力，所以左和右的聲音資訊在傳送到 DAC 之前，要先跟音量的係數相乘。而音量係數由一個新的模組來產生。此外，還需要用到前幾次 LAB 中用來控制七段顯示器的模組。



3. Finite state machine

內容: 電路中的 Finite state machine，若無則寫無。

無

4. Implement

內容: 請列出相關的 logic function、詳細用文字解釋電路的運作方法、結果等等，可以貼 code 解釋或拍 FPGA 輔助解釋(但不能只貼 code 跟 FPGA 結果)。

control_volumn.v 產生控制音量的係數 delta_volume，到時候在 top 可以直接跟 note_gen.v 所產生的聲音資訊的震幅相乘，就可以改變音量大小。

```
12  debounce_onepulse debounce_onepulse_increase_volume(
13    .onepulse(onepulse_increase_volume),
14    .debounced(),
15    .undebounced(increase_volume),
16    .clk,
17    .rst_n
18  );
19
20  debounce_onepulse debounce_onepulse_decrease_volume(
21    .onepulse(onepulse_decrease_volume),
22    .debounced(),
23    .undebounced(decrease_volume),
24    .clk,
25    .rst_n
26  );
```

如上圖，所有除了時脈以外的 input 都要先 debounce 或 onrpulse 之後再使用。

```
28 assign delta_volume_next =
29   (increase_volume & delta_volume < 4'hf) ? (delta_volume + 'h1) :
30   (decrease_volume & delta_volume > 4'h0) ? (delta_volume - 'h1) :
31                                             (delta_volume)       ;
32
33 always @(posedge onepulse_increase_volume | onepulse_decrease_volume or negedge rst_n)
34   if (~rst_n) delta_volume <= 4'h0;
35   else        delta_volume <= delta_volume_next;
```

如上圖，產生音量控制係數的方法很單純。如果增加音量的按鈕被按下，且當下的音量不是最大值，就加一；如果降低音量的按鈕被按下，且當下的音量不是最小值，就減一。

note_div_lookup.v 產生對應 Do、Re、Mi 的頻率的除數，供 note_gen.v 參照。

```
 9 assign note_div = (play_do) ? (100_000_000 / 261) :
10                   (play_re) ? (100_000_000 / 293) :
11                   (play_mi) ? (100_000_000 / 330) : (0);
```

如下圖，我把 note_gen.v 簡化，因為到時候在 top 要改變音量，所以其實不用像上一題一樣產生特定震幅的左和右聲音資訊。

```
20 always @*
21   if (clk_cnt == note_div) begin
22     clk_cnt_next = 22'd0;
23     b_clk_next   = ~b_clk;
24   end else begin
25     clk_cnt_next = clk_cnt + 1'b1;
26     b_clk_next   = b_clk;
27 end
```

```
29 always @(posedge clk or negedge rst_n)
30   if (~rst_n) begin
31     clk_cnt <= 22'd0;
32     b_clk   <= 1'b0;
33   end else begin
34     clk_cnt <= clk_cnt_next;
35     b_clk   <= b_clk_next;
36   end
```

如上圖，因為要產生的聲音是方波，所以只要保留能依照頻率除數產生特定頻率的 b_clk 即可。

下圖是二進位轉七段顯示器樣式的解碼器，以顯示當下音量大小。

```
 6 always @*
 7   case (binary)
 8     0:       ssd = 8'b0000001_1;  // 0
 9     1:       ssd = 8'b1001111_1;  // 1
10     2:       ssd = 8'b0010010_1;  // 2
11     3:       ssd = 8'b0000110_1;  // 3
12     4:       ssd = 8'b1001100_1;  // 4
13     5:       ssd = 8'b0100100_1;  // 5
14     6:       ssd = 8'b0100000_1;  // 6
15     7:       ssd = 8'b0001111_1;  // 7
16     8:       ssd = 8'b0000000_1;  // 8
17     9:       ssd = 8'b0000100_1;  // 9
18     10:      ssd = 8'b0001000_1;  // a
19     11:      ssd = 8'b1100000_1;  // b
20     12:      ssd = 8'b0110001_1;  // c
21     13:      ssd = 8'b1000010_1;  // d
22     14:      ssd = 8'b0110000_1;  // e
23     15:      ssd = 8'b0111000_1;  // f
24     default: ssd = 8'b0000000_0;
25   endcase
```

只要新增或修改上述的模組，其餘的模組直接沿用上一題的即可。接下來要建構 top。

```
20 wire [3:0] delta_volume;
21
22 control_volumn control_volumn(
23   .delta_volume,
24   .increase_volume,
25   .decrease_volume,
26   .clk,
27   .rst_n
28 );
```

```
30 decoder decoder(
31   .ssd    (cathode),
32   .binary(delta_volume)
33 );
```

如上圖，先產生用來控制音量的係數 delta_volume，並把它轉換成七段顯示器的樣式，以便顯示音量大小。

```
35  wire [21:0] note_div;
36
37  note_div_lookup note_div_lookup(
38    .note_div,
39    .play_do,
40    .play_re,
41    .play_mi
42  );
```

```
44  // Note generation
45  wire [15:0] b_clk;
46
47  // a general frequency divider
48  // to generate the required frequencies for spe
49  note_gen Ung(
50    .b_clk(b_clk[11]),
51    .note_div,      // div for note generation
52    .clk,           // clock from crystal
53    .rst_n          // active low reset
54  );
```

如上，產生用來改變音高的除數 note_div，並連接到除頻器以產生方波 b_clk。

```
56  // Speaker controllor
57  // with a stereo signal parallel-to-serial processor
58  // to generate the speaker control signals
59  speaker_control Usc(
60    .audio_mclk,  // master clock
61    .audio_lrck,  // left-right clock
62    .audio_sck,   // serial clock
63    .audio_sdin,  // serial audio data input
64    .audio_left (b_clk * delta_volume), // left  channel audio data input
65    .audio_right(b_clk * delta_volume), // right channel audio data input
66    .clk,         // clock from the crystal
67    .rst_n        // active low reset
68  );
```

最後，把方波連接到用來控制 DAC 的模組即可。須注意 b_clk 此時要跟音量係數相乘。

5. Conclusion

內容: 可以寫下你的這個 lab 的想法、遇到的問題、解決方法、心得等等，請自由發揮。

這題比較沒什麼問題，因為都是一些前幾次 LAB 就有寫過的東西，複製貼上再稍加修改就好。比較難的只有上一題，因為對 DAC 不熟，對它的運作原理也很陌生。