Lab 8

學號: 109000168 姓名: 許媄香

1. 實作過程

In this lab, we are going to turn a computer keyboard into instrumental keyboard. And I started this lab from the template given by the TAs.

Here are some requirements to do.

- **_mode**

  The piano has two modes:
  1. PLAY: In this mode, the user can play the piano using the keyboard.
     DEMONSTRATE: In this mode, the piano can demonstrate a song that has been written in it. So, the player can get to know how the song sounds like.

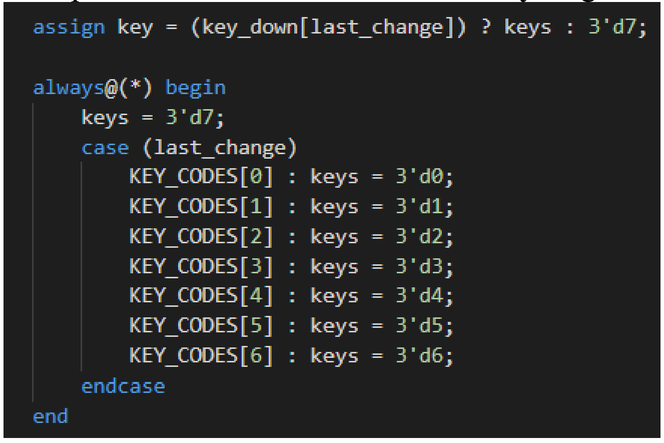
  It will go into PLAY mode when **_mode** signal is ‘0’, otherwise it will go into DEMOSTRATE mode.

- PLAY mode (**_mode** is ‘0’)

  In the PLAY mode, the piano should make no sound unless you press a key. The following table shows the key-note mapping.

Key	a	s	d	f	g	h	j
Note	C (or Do)	D (or Re)	E (or Mi)	F (or Fa)	G (or Sol)	A (or La)	B (or Ti)

Here, I assign the pressed keyboard signal in 3-bit wire, and I named it as **key**. Besides, I use a multiplexer to determine the value of **keys** reg which will be assigned to **key**.



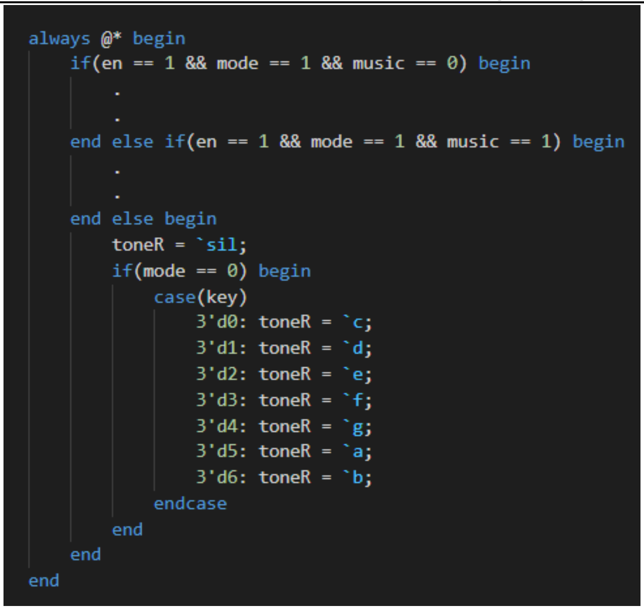
```
assign key = (key_down[last_change]) ? keys : 3'd7;

always@(*) begin
    keys = 3'd7;
    case (last_change)
        KEY_CODES[0] : keys = 3'd0;
        KEY_CODES[1] : keys = 3'd1;
        KEY_CODES[2] : keys = 3'd2;
        KEY_CODES[3] : keys = 3'd3;
        KEY_CODES[4] : keys = 3'd4;
        KEY_CODES[5] : keys = 3'd5;
        KEY_CODES[6] : keys = 3'd6;
    endcase
end
```

KEY_CODES[0] is ‘a’ button and KEY_CODES[6] is ‘j’ consecutively, and it will base on the changes of **last_change**. Then, if something is pressed, **key** value will be assigned to **keys** otherwise, it will be the default value, ‘7’ (means that nothing from a, s, d, f, g, h, h is pressed and will make no sound).

Next, I add **key** and **_music** as input in **music_example** module. And then, I change a bit the two combinational blocks inside the module where have been done by the TA. I change it to 3 if else case to select which sound to play (**toneR** and **toneL**).

So, the first two cases are for DEMONSTRATE mode, where the only difference is the song. The first one is Lightly Row, and the other is Yoasobi’s song (will be explained more later). For the PLAY mode is at last if else case and the output is based on the **key** value.

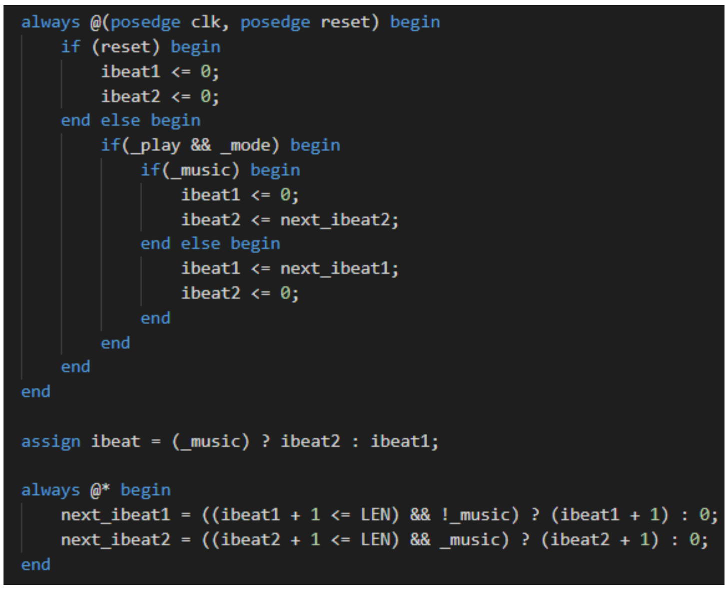


```
always @* begin
    if(en == 1 && mode == 1 && music == 0) begin
        .
        .
    end else if(en == 1 && mode == 1 && music == 1) begin
        .
        .
    end else begin
        toneR = `sil;
        if(mode == 0) begin
            case(key)
                3'd0: toneR = `c;
                3'd1: toneR = `d;
                3'd2: toneR = `e;
                3'd3: toneR = `f;
                3'd4: toneR = `g;
                3'd5: toneR = `a;
                3'd6: toneR = `b;
            endcase
        end
    end
end
```

- DEMONSTRATE mode (**_mode** is '1')

  (Optional bonus) You can provide two demonstrate songs for the DEMONSTRATE mode. The piano will play one of them when the **Music** switch (SW 3) is 0, and the other when the switch is 1. Every time the Music switch is changed, the piano start playing the corresponding song from the beginning.

  Since in this mode has two options of song to play, I make each beat count for each song, **ibeat1** and **ibeat2**. In order to the song play from beginning when the music is changed, I assign the beat count of the other song (the song that is not played) to '0'. It looks like the code below.



```
always @(posedge clk, posedge reset) begin
    if (reset) begin
        ibeat1 <= 0;
        ibeat2 <= 0;
    end else begin
        if(_play && _mode) begin
            if(_music) begin
                ibeat1 <= 0;
                ibeat2 <= next_ibeat2;
            end else begin
                ibeat1 <= next_ibeat1;
                ibeat2 <= 0;
            end
        end
    end
end

assign ibeat = (_music) ? ibeat2 : ibeat1;

always @* begin
    next_ibeat1 = ((ibeat1 + 1 <= LEN) && !_music) ? (ibeat1 + 1) : 0;
    next_ibeat2 = ((ibeat2 + 1 <= LEN) && _music) ? (ibeat2 + 1) : 0;
end
```

  And **ibeat** will choose the **ibeat** according to which song is played (**_music** = 0 for **ibeat1** and **_music** = 1 for **ibeat2**).

  In the DEMONSTRATE mode, the piano will **repeatedly** play a demonstration song (starting from the beginning again when reaching the end) and will NOT react to any key pressed on the keyboard.

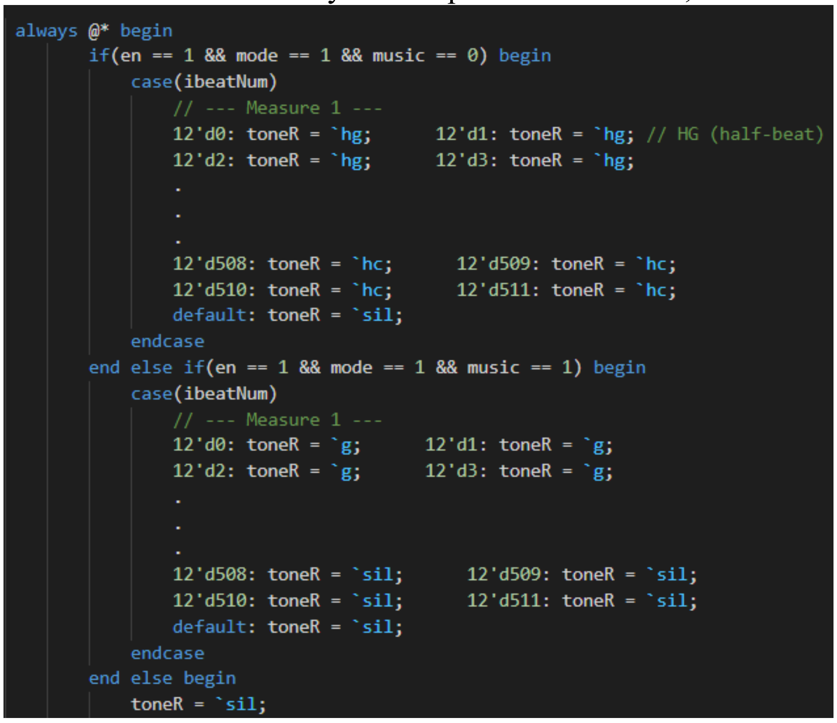
  To make the song played repeatedly, I make the **next_beat** of the corresponding song to '0' if it has reached the end of song.

  - Play / Pause

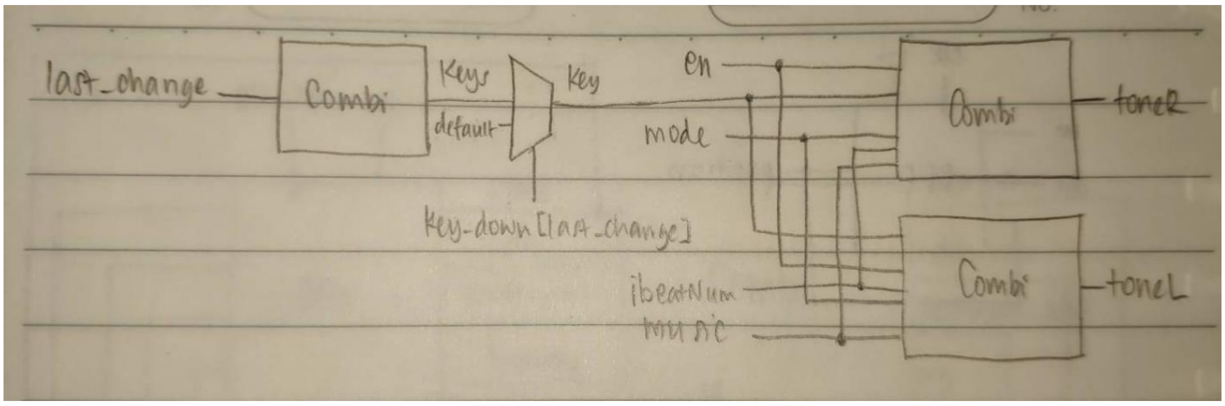
    By switching the **Play/Pause** switch (SW 0) to "pause," the piano should pause the demonstration song immediately and should play no tone. The piano should start/resume to play the song from where it paused by switching to "play."
    The switch takes effect only on the DEMONSTRATE mode.

    As the code above has shown, the corresponding **ibeat** will assigned to **next_beat** iff it's in demonstrate mode (**_mode** = ‘1’) and it’s not paused (**_play** = ‘1’)

After that, **ibeat** will be used in **music_example** module to choose which note to play (the reg name in **music_example** module is **ibeatNum**). The output of **toner** and **toneL** is based on the value of **ibeatNum** if only it’s not paused. Otherwise, it will be assigned to silence.



```
always @* begin
    if(en == 1 && mode == 1 && music == 0) begin
        case(ibeatNum)
            // --- Measure 1 ---
            12'd0: toneR = `hg;      12'd1: toneR = `hg; // HG (half-beat)
            12'd2: toneR = `hg;      12'd3: toneR = `hg;
            .
            .
            .
            12'd508: toneR = `hc;      12'd509: toneR = `hc;
            12'd510: toneR = `hc;      12'd511: toneR = `hc;
            default: toneR = `sil;
        endcase
    end else if(en == 1 && mode == 1 && music == 1) begin
        case(ibeatNum)
            // --- Measure 1 ---
            12'd0: toneR = `g;      12'd1: toneR = `g;
            12'd2: toneR = `g;      12'd3: toneR = `g;
            .
            .
            .
            12'd508: toneR = `sil;      12'd509: toneR = `sil;
            12'd510: toneR = `sil;      12'd511: toneR = `sil;
            default: toneR = `sil;
        endcase
    end else begin
        toneR = `sil;
```



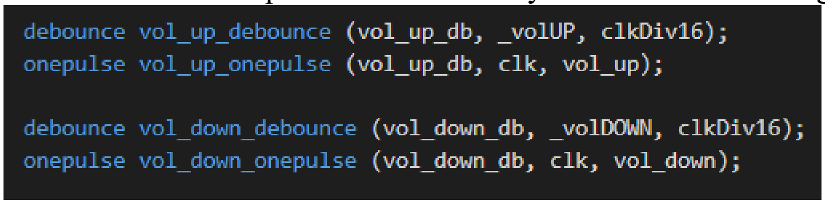
- Volume

  The piano should support 5 distinguishable levels of volume, controlled by **Volume Up** (BtnU) and **Volume Down** (BtnD). The default level of volume is 3. At the lowest level, the sound should still be able to heard. Pressing **Volume Up** at level 5 or pressing **Volume Down** at level 1 takes no effect.

  LED 0~4 indicates the current volume level:

Volume Level	LED 4	LED 3	LED 2	LED 1	LED 0
Muted	○	○	○	○	○
Level 1	○	○	○	○	●
Level 2	○	○	○	●	●
Level 3	○	○	●	●	●
Level 4	○	●	●	●	●
Level 5 (the loudest)	●	●	●	●	●

To prevent bouncing to happen, I instantiate a 100MHz / 2^{16} clock divider to debounce the button. And use **clk** for one pulse because the keyboard decoder is using **clk** clock.

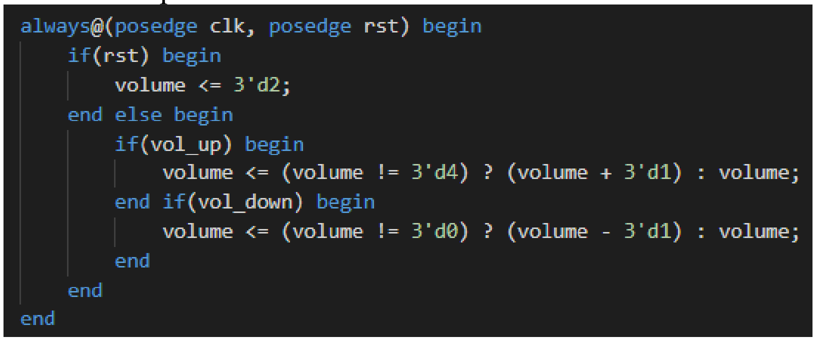


```
debounce vol_up_debounce (vol_up_db, _volUP, clkDiv16);
onepulse vol_up_onepulse (vol_up_db, clk, vol_up);

debounce vol_down_debounce (vol_down_db, _volDOWN, clkDiv16);
onepulse vol_down_onepulse (vol_down_db, clk, vol_down);
```

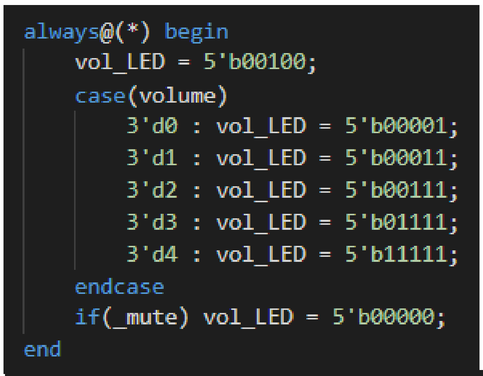
Here, I make the corresponding level of volume as below.
Level 1 ➔ volume = 0
Level 2 ➔ volume = 1
Level 3 ➔ volume = 2
Level 4 ➔ volume = 3
Level 5 ➔ volume = 4

To keep in track with the volume level, I make a new 3-bit reg. Because the initial volume level is level 3, therefore the volume is equal to 2 when reset button is pressed. And when the volume up button is pressed, volume will increase by one if only if the current volume is not equal to 4 otherwise it will stay the same. For the volume down its similar but it won't decrease if the current volume is equal to 0.

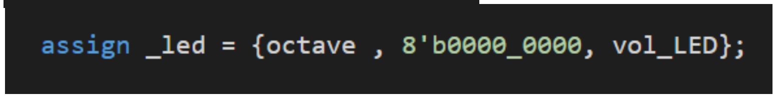


```
always@(posedge clk, posedge rst) begin
    if(rst) begin
        volume <= 3'd2;
    end else begin
        if(vol_up) begin
            volume <= (volume != 3'd4) ? (volume + 3'd1) : volume;
        end if(vol_down) begin
            volume <= (volume != 3'd0) ? (volume - 3'd1) : volume;
        end
    end
end
```

And to use it in LED light, I make a new 5-bit reg (**vol_LED**) to assign it in **_led** output. However, if **_mute** is on, then **vol_LED** all will be turned off. As a result, **vol_LED** value is corresponding to volume value and the code looks like below.



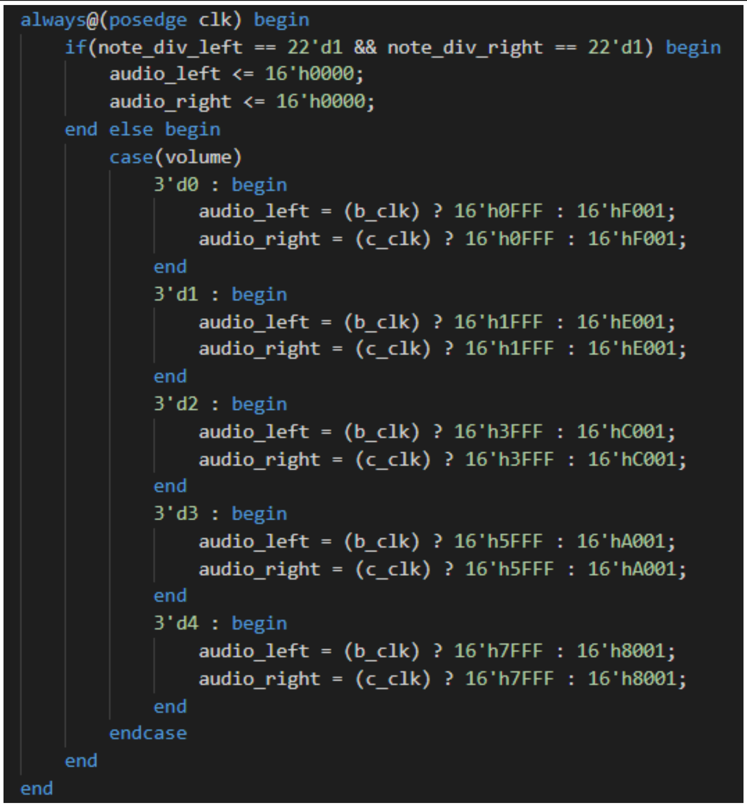
```
always@(*) begin
    vol_LED = 5'b00100;
    case(volume)
        3'd0 : vol_LED = 5'b00001;
        3'd1 : vol_LED = 5'b00011;
        3'd2 : vol_LED = 5'b00111;
        3'd3 : vol_LED = 5'b01111;
        3'd4 : vol_LED = 5'b11111;
    endcase
    if(_mute) vol_LED = 5'b00000;
end
```



```
assign _led = {octave , 8'b0000_0000, vol_LED};
```

Since **_led[12:5]** is not used, I just make it off all of the time. Also, the **octave** will be explained more later.

Besides, to assign the volume amplitude, I make the loudest level be **7FFF** and the lowest level (not mute) to **0FFF**. And also, for the lowest peak, the value will be 2's complement of the highest peak. However, if it's muted, then the value of **audio_left** and **audio_right** will be zero.



```
always@(posedge clk) begin
    if(note_div_left == 22'd1 && note_div_right == 22'd1) begin
        audio_left <= 16'h0000;
        audio_right <= 16'h0000;
    end else begin
        case(volume)
            3'd0 : begin
                audio_left = (b_clk) ? 16'h0FFF : 16'hF001;
                audio_right = (c_clk) ? 16'h0FFF : 16'hF001;
            end
            3'd1 : begin
                audio_left = (b_clk) ? 16'h1FFF : 16'hE001;
                audio_right = (c_clk) ? 16'h1FFF : 16'hE001;
            end
            3'd2 : begin
                audio_left = (b_clk) ? 16'h3FFF : 16'hC001;
                audio_right = (c_clk) ? 16'h3FFF : 16'hC001;
            end
            3'd3 : begin
                audio_left = (b_clk) ? 16'h5FFF : 16'hA001;
                audio_right = (c_clk) ? 16'h5FFF : 16'hA001;
            end
            3'd4 : begin
                audio_left = (b_clk) ? 16'h7FFF : 16'h8001;
                audio_right = (c_clk) ? 16'h7FFF : 16'h8001;
            end
        endcase
    end
end
```

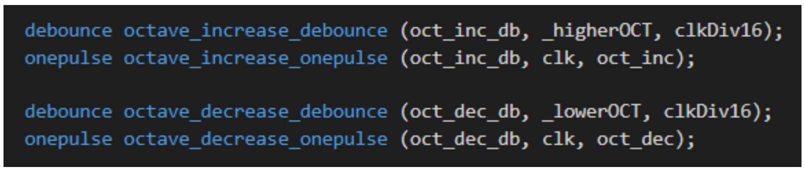
- Octave

The piano should support 3 levels of octaves, controlled by **Higher Octave** (BtnR) and **Lower Octave** (BtnL). The default level of the octave is 2. Pressing **Higher Octave** at level 3 or pressing **Lower Octave** at level 1 takes no effect.
Raising a note an octave higher means to double the note's frequency, and vise versa.

LED 13~15 indicates the current octave level:

Octave Level		LED 15	LED 14	LED 13
Level 1	Lower	●	○	○
Level 2	Normal	○	●	○
Level 3	Higher	○	○	●

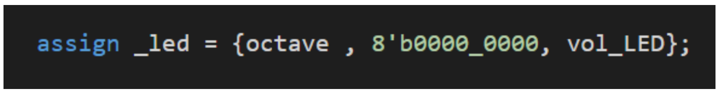
Same as volume, I instantiate the debounce and one pulse in similar way and also control it in similar method.



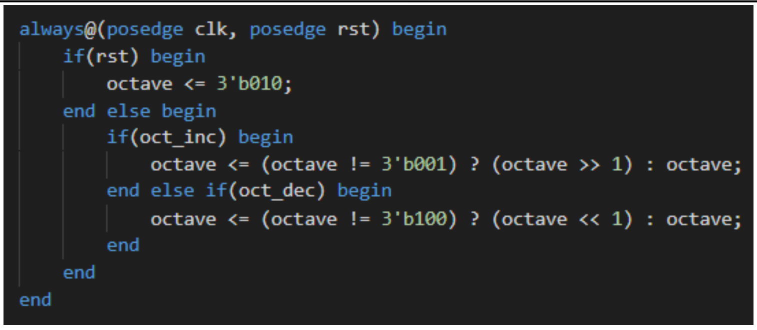
```
debounce octave_increase_debounce (oct_inc_db, _higherOCT, clkDiv16);
onepulse octave_increase_onepulse (oct_inc_db, clk, oct_inc);

debounce octave_decrease_debounce (oct_dec_db, _lowerOCT, clkDiv16);
onepulse octave_decrease_onepulse (oct_dec_db, clk, oct_dec);
```

To keep in track the level of octave, I make a 3-bit reg (**octave**) as one hot encoding. The initial level is 2, therefore, I just make **octave** equal to 010 (in binary) when the reset button is pressed. After that, if increase octave button is pressed, it will shift to right by one if the current level is not the highest otherwise it will stay the same. Similar if want to decrease the octave level, it will shift to left by one if the current level is not the lowest. After that, as the previous code has shown, I use **octave** value to assign it in **_led.**

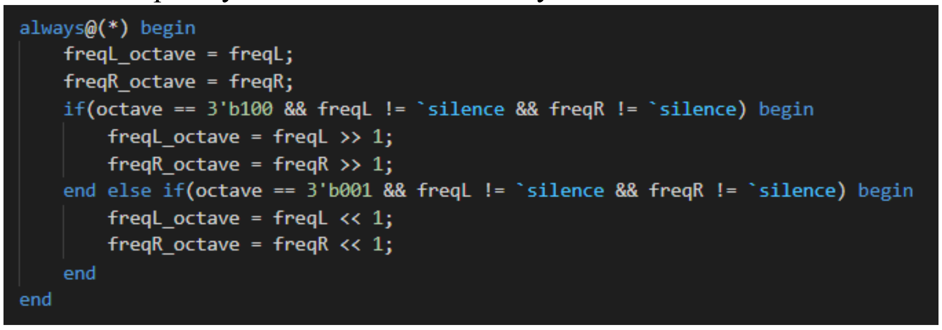


```
assign _led = {octave , 8'b0000_0000, vol_LED};
```



```
always@(posedge clk, posedge rst) begin
    if(rst) begin
        octave <= 3'b010;
    end else begin
        if(oct_inc) begin
            octave <= (octave != 3'b001) ? (octave >> 1) : octave;
        end else if(oct_dec) begin
            octave <= (octave != 3'b100) ? (octave << 1) : octave;
        end
    end
end
```

To increase/decrease the frequency, I make a new reg to store the frequency according to the octave level. If the level is 1 and it's not silence, then the frequency will be shifted to right by one because we need to make it two times lower. And if the level is 3 and the previous frequency is not silence, then the frequency will be shifted to left by one.

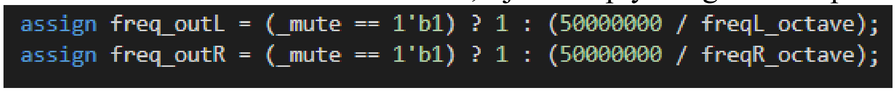


```
always@(*) begin
    freqL_octave = freqL;
    freqR_octave = freqR;
    if(octave == 3'b100 && freqL != `silence && freqR != `silence) begin
        freqL_octave = freqL >> 1;
        freqR_octave = freqR >> 1;
    end else if(octave == 3'b001 && freqL != `silence && freqR != `silence) begin
        freqL_octave = freqL << 1;
        freqR_octave = freqR << 1;
    end
end
```

- Mute

  By switching the **Muted** switch (SW 1) to 1, player can mute the piano. When muted, the key in the PLAY mode can still be pressed and the demonstration song in the DEMONSTRATE mode will keep playing. But you will hear no sound.
  The switch takes effect on both modes.

  Because **mute** works on both modes, I just simply assign the output as below.



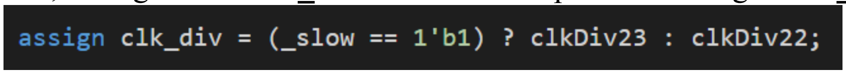
```
assign freq_outL = (_mute == 1'b1) ? 1 : (50000000 / freqL_octave);
assign freq_outR = (_mute == 1'b1) ? 1 : (50000000 / freqR_octave);
```

  Since '1' is silence, I just declare both frequencies to '1' when **mute** is on.

- Slow down

  By switching the **Slow Down** switch (SW 2) to 1, the piano should play the demonstration song two times slower than the normal speed. (Each note plays two times longer.) The piano should play the song at normal speed when the switch is switched to 0.
  The switch takes effect only on the DEMONSTRATE mode.

  Firstly, I instantiate 100MHz / 2^{22} (normal speed) and 100MHz / 2^{23} (slower) clock divider. After that, I assign a new **clk_div** to choose the speed according to the **_slow** signal.



```
assign clk_div = (_slow == 1'b1) ? clkDiv23 : clkDiv22;
```

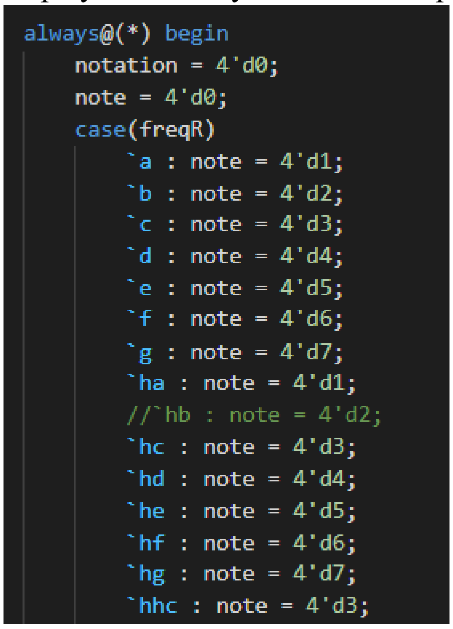
  Then, I passed the **clk_div** signal to **player_control** module so that it will increase the beat count base on **clk_div** speed.

- Seven Segment Display

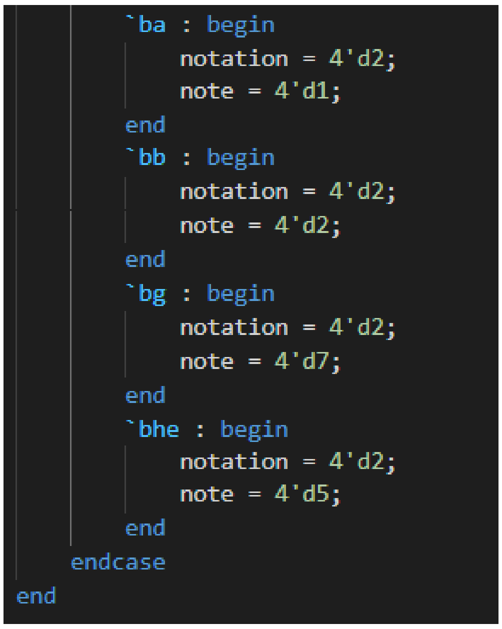
  Display the melody pitch (C, D, E, F, G, A, and B) on the 7-segment display for both PLAY and DEMONSTRATE modes. Note that Sharp/flat notation need to be displayed also.
  For example:
  You should display '- - - G' when the first note of *Lightly row* is being played.
  You should display '- - bB' when the third note of *Havana* is being played.
  You should display '- - - -' when no key is pressed in the PLAY mode and when the demonstration song is paused in the DEMONSTRATE mode.
  You can use to represent the sharp notation (#) and to represent the flat notation (b).
  Note that when muted, the 7-segment display should still display the melody pitch.

Here, I make 2 4-bit reg, **notation** and **note**. **note** is for the right most digit in seven segment display and **notation** is the second right most digit.

Therefore, I just change **note** and **notation** based on the changes of **freqR** because we need to display the melody note in the display. And the code will be like below.

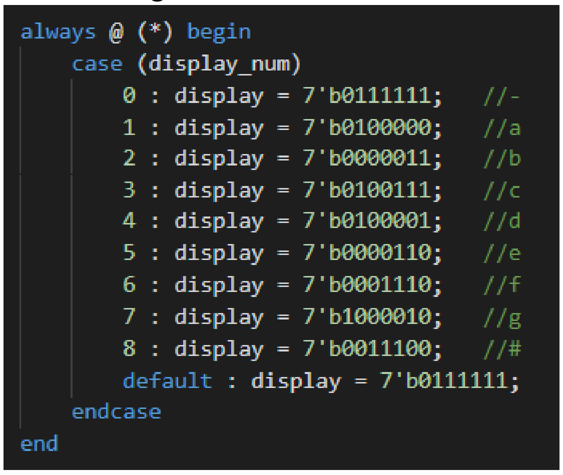


```
always@(*) begin
    notation = 4'd0;
    note = 4'd0;
    case(freqR)
        `a : note = 4'd1;
        `b : note = 4'd2;
        `c : note = 4'd3;
        `d : note = 4'd4;
        `e : note = 4'd5;
        `f : note = 4'd6;
        `g : note = 4'd7;
        `ha : note = 4'd1;
        //`hb : note = 4'd2;
        `hc : note = 4'd3;
        `hd : note = 4'd4;
        `he : note = 4'd5;
        `hf : note = 4'd6;
        `hg : note = 4'd7;
        `hhc : note = 4'd3;
```



```
        `ba : begin
            notation = 4'd2;
            note = 4'd1;
        end
        `bb : begin
            notation = 4'd2;
            note = 4'd2;
        end
        `bg : begin
            notation = 4'd2;
            note = 4'd7;
        end
        `bhe : begin
            notation = 4'd2;
            note = 4'd5;
        end
    endcase
end
```

After having both **note** and **notation** value, then the seven-segment display will change as below.



```
always @ (*) begin
    case (display_num)
        0 : display = 7'b0111111;   //-
        1 : display = 7'b0100000;   //a
        2 : display = 7'b0000011;   //b
        3 : display = 7'b0100111;   //c
        4 : display = 7'b0100001;   //d
        5 : display = 7'b0000110;   //e
        6 : display = 7'b0001110;   //f
        7 : display = 7'b1000010;   //g
        8 : display = 7'b0011100;   //#
        default : display = 7'b0111111;
    endcase
end
```

2. 學到的東西與遇到的困難

First, I was having trouble in understanding the sample code from the TA. But after reading it one by one, I finally understand how it works. However, I didn't know how to control the volume. But with the help of the comment (because the TA put a comment in note_gen module), a comment to let us know where to control the volume. Also, there is a simple template to assign the audio frequency. Therefore, I just try to change the hexadecimal there and somehow it works. For that reason, I just use the same format as TA and added some volume multiplexer.

3. 想對老師或助教說的話

life after college
I NEED A JOB
YOU DON'T HAVE EXPERIENCE
BUT I NEED TO WORK TO GET EXPERIENCE
THEN WORK
THAT'S WHY I'M HERE
EXPERIENCE
HOW AM GONNA GET EXPERIENCE WITHOUT A WORK
WORK