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.

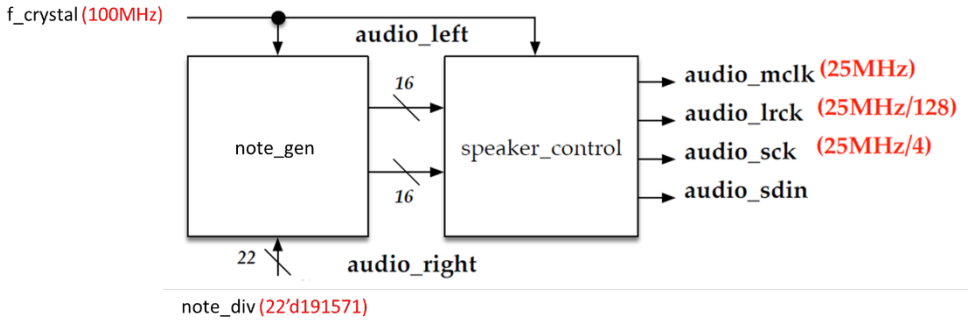
1.2 Design a stereo signal parallel-to-serial processor to generate the speaker control signals. Please use Verilog simulation waveform to verify your control signal.

**Design Specification:**

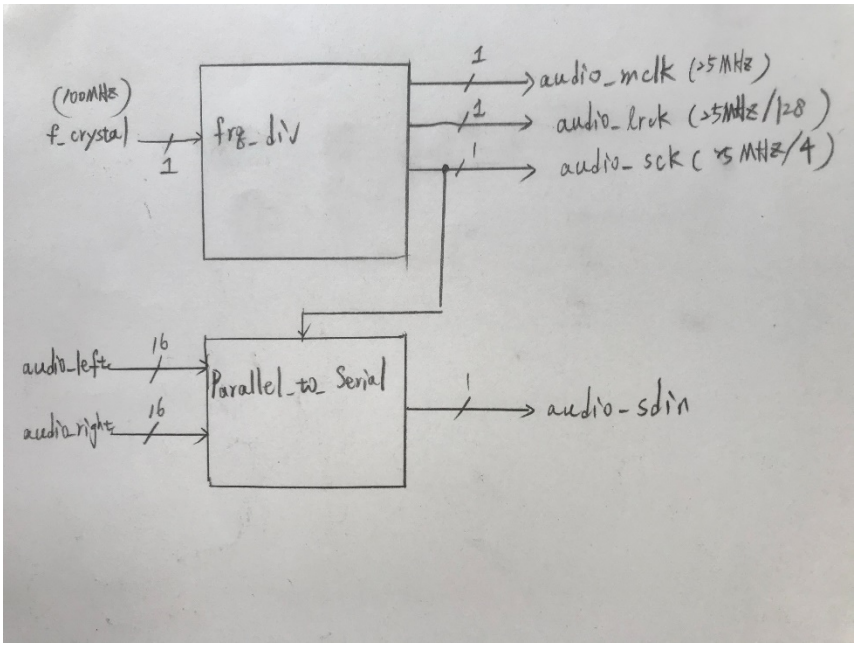
Input: f_crystal, rst_n

Output: audio_mclk, audio_lrck, audio_sck, audio_sdin

Block diagram:




Inside speaker_control module,




**Design Implementation:**

I/O pin assignment:

f_crystal—W5

rst_n—V17

audio_mclk—A14

audio_lrck—A16

audio_sck—B15

audio_sdin—B16

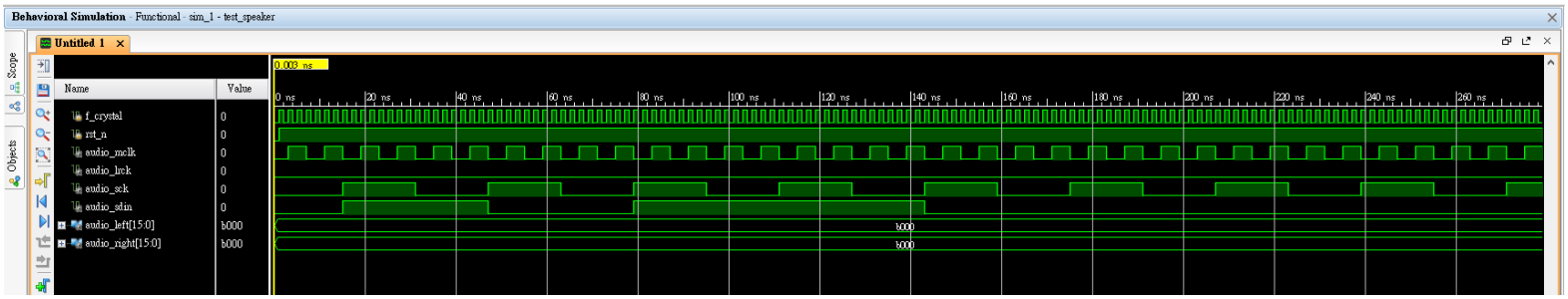
About note_gen module, it decides what note (Do, Re, Mi…) to output according note_div (in block diagram) value. For example, in this experiment, I chose to output Do. At the same time, it also decides the amplitude (i.e. volume) of the sound through audio_left (左聲道) and audio_right (右聲道) in the block diagram.

About speaker_control module, its function can be decomposed into two main parts. One is frequency divider and the other one is parallel to serial converter. Frequency divider outputs appropriate clock frequencies for Pmod I2S so that our ears are able to hear 'Do' from FPGA.

And the most important function of speaker_control is parallel to serial converter. As both audio_left and audio_right are 16-bit parallel signals coming from note_gen, we need convert it to serial signal before output as audio_sdin. Also, we need to a clock delay (a audio_sck clock delay) and start from MSB of audio_left to LSB of audio_left, then continue with MSB of audio_right to LSB of audio_right. (So the serial sequence would like, audio_right[0]->audio_left[15]->audio_left[14] …)

I used **case** in Verilog to implement this function. By using counter (5 bits in total->32 different values) with audio_sck, which is 32X faster than audio_lrck, I can output each bit from audio_left and audio_right serially.

Simulation Result:




f_crystal serves as a clock from FPGA board. According to the waveform, audio_mclk is 4X slower than f_crystal and audio_sck is 4X slower than audio_mclk. However, since audio_lrck is too slow compared to other divided clock frequencies, I didn't provide the whole period of the audio_lrck. I checked it in the waveform and it's 32X slower than audio_sck. Hence, the divided frequencies are correct.

Finally, through the control of audio_sck, audio_sdin could output audio_left and audio_right serially (both are 16'hb000).

**Discussion:**

In this experiment, the most challenging part is to implement parallel to serial converter, which I haven't done before. Through the use of **case** in Verilog, it becomes easy and fast to implement.

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

**Design Specification:**

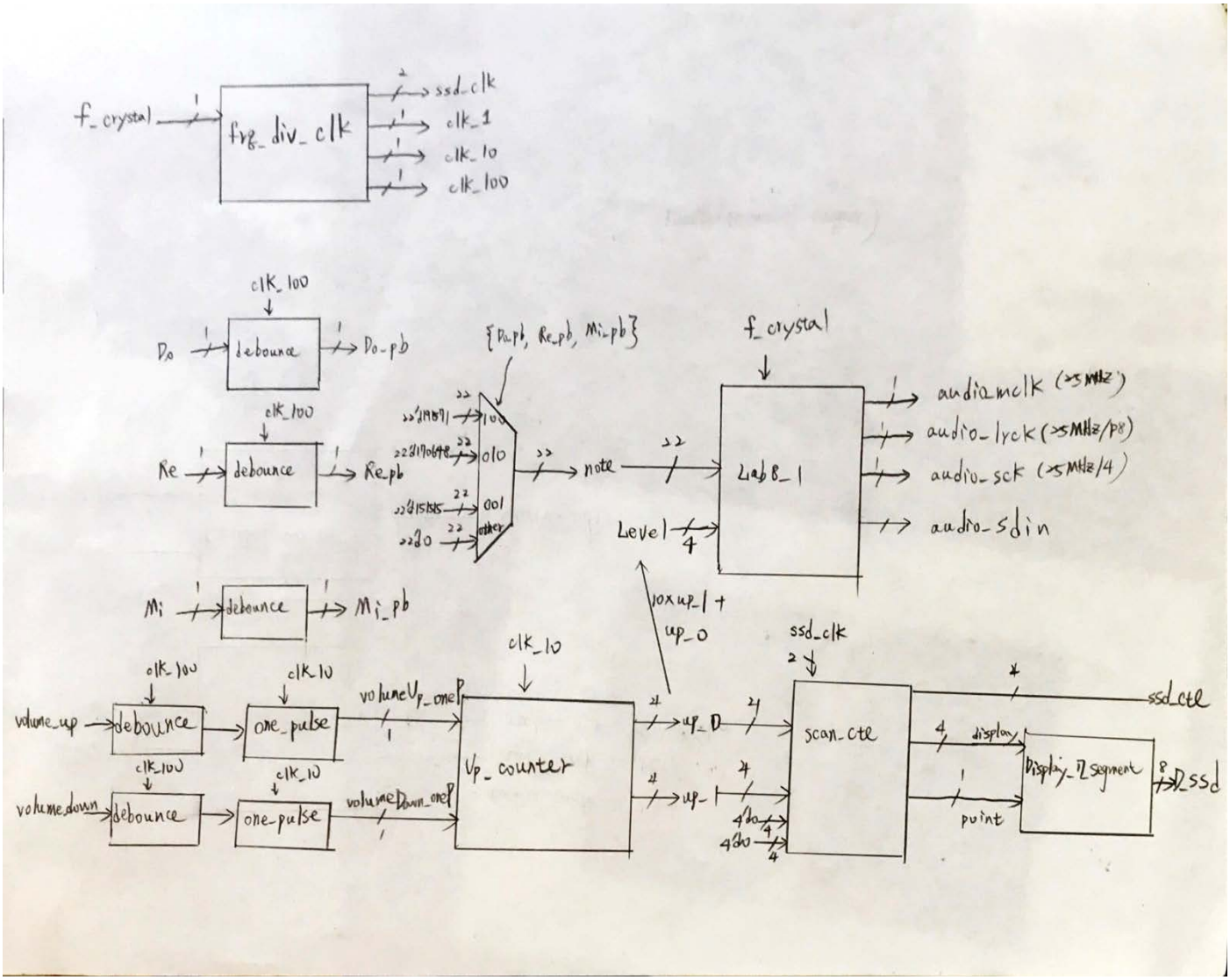
Input: Do, Re, Mi, volume_up, volume_down, f_crystal, rst

Output: audio_mclk, audio_lrck, audio_sck, audio_sdin

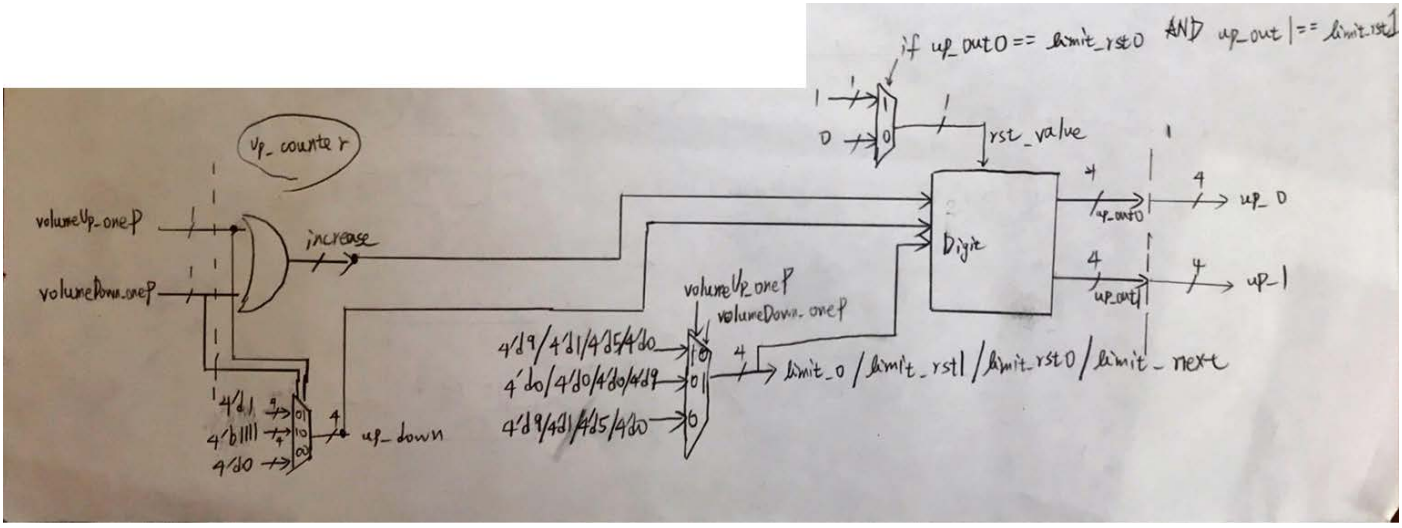
Output: [3:0]ssd_ctl

Output: [7:0]D_ssd

Block diagram:



Inside Up_counter module:




**Design Implementation:**

I/O pin assignment:

ssd_ctl[3]—W4

ssd_ctl[2]—V4

ssd_ctl[1]—U4

ssd_ctl[0]—U2

D_ssd[7]—W7

D_ssd[6] — W6

D_ssd[5] — U8

D_ssd[4] — V8

D_ssd[3] — U5

D_ssd[2] — V5

D_ssd[1] — U7

D_ssd[0] — V7

f_crystal — W5

rst—V17

audio_mclk—A14

audio_lrck—A16

audio_sck—B15

audio_sdin—B16

Do—W19

Re—U18

Mi—T17

volume_up—T17

volume_down—U17

In this experiment, we are required to design to produce the buzzer sounds **Do, Re and Mi** by pushing the button. So, I used debounce module to pre-process the signal from push button, then designed a MUX to choose the frequency input for note_gen module in Lab8_1 according to which button is pushed. If no

button is pushed, I just sent 22'd0 signal into note_gen module.

The other part of this experiment is to control the volume of the sound, which should be quantized into 16 levels. Therefore, I designed a counter with up-counting and down-counting function at the same time. As I use BCD to represent the counter value, I need two 4-bit counter to implement it. So, there are two identical Digit module in the block diagram of Up_counter. But MSB only has two possible values 4'd0, 4'd1, I didn't draw it in the block diagram. The most difficult part is LSB of the counter.

First of all, when the counter needs to change its value, I checked if it's a increasing signal or decreasing signal by one pulse signal. If the signal is coming from volume_up (button), one pulse module for volume up would output 1 and of course, one pulse module for volume down would output 0. Similar situation for signal coming from volume_down.

No matter it's increasing or decreasing, I used **adding** to do it. I would add 4'd1 on current value if it's increasing and 4'b1111 on current value if it's decreasing. That's because it's 2's complement representation.

When it's increasing, I set limit_0 = 4'd9, limit_next = 4'd0. As when LSB reaches 4'd9, it would pass a carry to MSB and becomes 4'd0 when next clock edge arrives. Also, I set limit_rst0 = 4'd5, limit_rst1 = 4'd1. Because there are only 16 levels of volume, it should become 0 when next clock edge arrives when reaching 15 if it's still increasing signal. Opposite setting for decreasing signal, which is limit_0 = 4'd0, limit_next = 4'd9, limit_rst0 =4'd0, limit_rst1 = 4'd0. Finally, I don't need to care about the setting if it's neither increasing nor decreasing signal.

As long as I got each value of BCD, I converted it into 4-bit decimal number from 0~15. Then, I sent this value to note_gen to adjust the volume accordingly. As audio_left and audio_right in Lab8_1 are 16-bit 2's complement number. The most positive number is 16'h7FFF (about 32767.9in decimal). Hence, level difference is about 32767.9/16 = 2047.9 With this value, I can produce different amplitude of the sound.

**Discussion:**

The most challenging part of this experiment is undoubtedly the counter with up counting and down counting function at the same time. It's becomes more difficult as I used BCD to represent the value. I need to discuss the different cases for increasing and decreasing separately. Many details need to be considered. I believe it would have been a lot easier if I just a 4-bit counter and only need to deal with the value when showing the result on 7-segment display.

**Conclusion for Lab8:**

Through the lab, I know the speaker function on FPGA board. And I know I could control the frequency for note_gen module so that FPGA board would produce different kinds of note. Also, I have noticed the communication protocol applied in the lab, which is how to implement parallel to serial converter. Now, I know how to design a counter with up-counting and down-counting function at the same time. I think this is very important as I have seen this function in our daily life a lot, ex: volume control in the smartphone.

**Reference:**

**08_Speaker.pdf** given by professor. Through this handout, I have learned how to control the sound with Verilog and implement it on FPGA.