Final Project: Talking Calculator

Author: Ruyi Zhou 49581911

Lab Section: L2C

1. *My SOF file is located at:*

Final_project\rtl\
Named: simple_ipod_Solution.sof

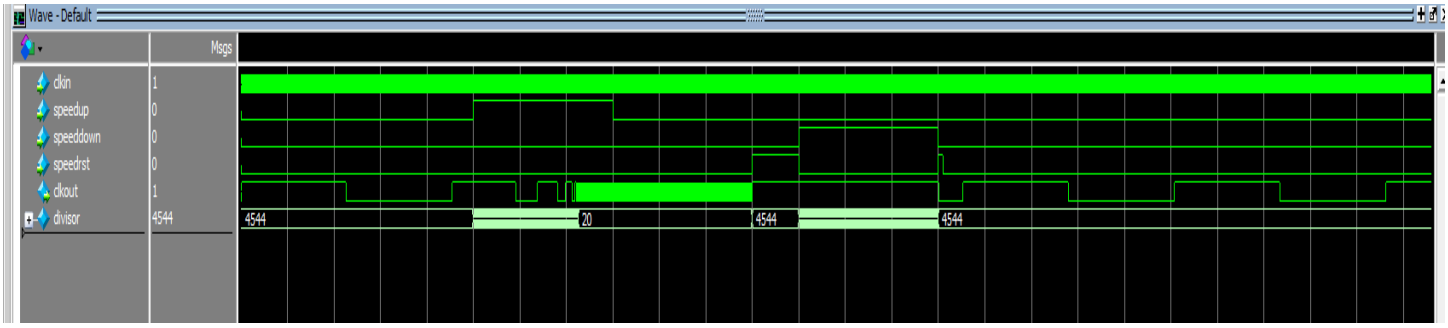
2. *The status of the lab:*

My design can work perfectly and achieves all requirements in the lab handouts. The simulations of other FSMs are working well. The speech synthesizer, calculator, LED, speed control and 8b10 all work well as the solution.

3. *Simulation screenshots:*

I wrote the testbenches for these FSMs: *clk_divider_spd_controller*, *Read_Data*, *Flash_handle*, *keyboard* and *led_control*. And I simulated them in ModelSim 10.3c.
Among them, the *clk_divider_spd_controller* and *Read_Data* remains unchanged from my own lab 2 so the simulation is the same as before.

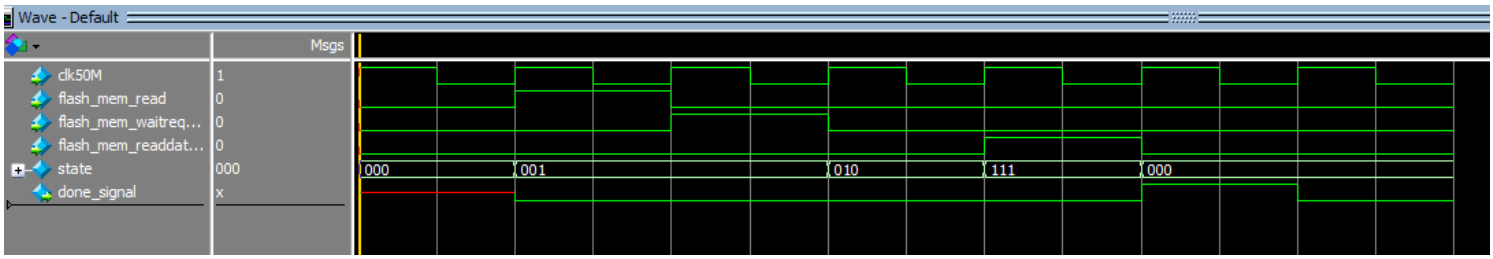
(a) clk_divider_spd_controller:



The smaller divisor, the higher speed of the song, and vice versa. The clkdout is the frequency that control the speed of the song. As shown above, when KEY[0] is pressed (speedup = 1), the divisor is decreasing until the lower limit (I set it to 20), and the corresponding clkout frequency is increasing. Then, we press KEY [2] (speedrst =1), the divisor is turned to the default divisor

and the output frequency is the default frequency. Then we slow the speed of the song by press KEY [1] (speeddown), the divisor is increased, and the corresponding output frequency is lower. Then we press the KEY [2] again. The divisor and the output frequency become the default value again. This simulation matches my expectation which indicates the design is fine.

(b) Read_Data:



Initially, the state is ‘idle’. When the input ‘flash_mem_read’ commend is 1, the state goes to ‘reading’, and checking if the waitrequest if from 1 to 0. Once the waitrequest falls to 0, the state goes to ‘check_data’. If the data is valid (flash_mem_readdatavalid = 1), the state goes to ‘done’ and output 1 at ‘done’ state. This simulation matches my expectation which indicates the design is fine.

(c) Flash_handle:

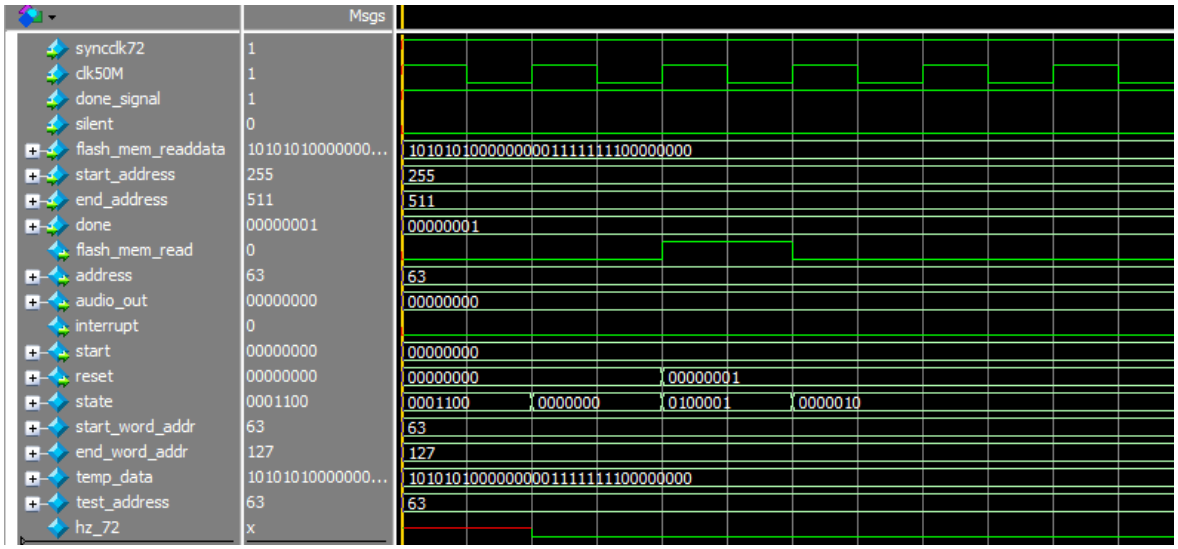


Figure a. The zoomed version of the Flash_handle simulation

As shown in **Figure a**, the initial state is 'init (0001100), then when the 'done' (from picoblaze, indicate the picoblaze has finished choosing a phoneme_sel) is 1, we have the start and end byte address (and we convert them to the word address by divided it by 4) and go to the 'idle (0000000)' state. At the 'idle' state we set the 'reset' to 1 to pause the picoblaze and do the algorithm. Then we go to the 'rdflash (0100001)' and when the done_signal (from Read_Data module) is 1, we go to 'gtdata (0000010)' state.

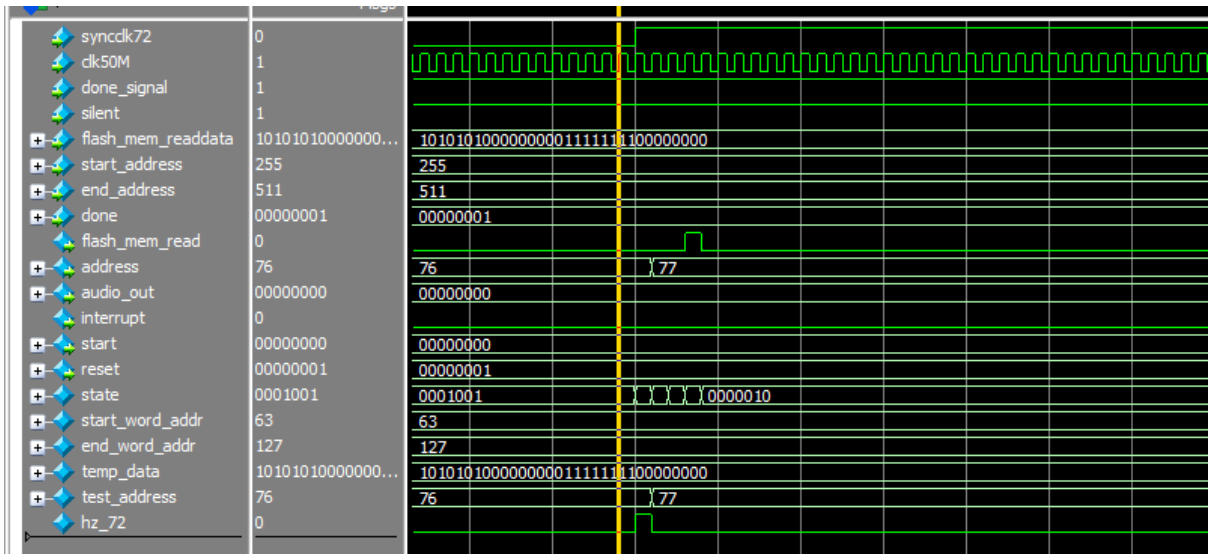


Figure b. The second part of the zoomed version of the Flash_handle simulation

Since the 7200Hz and 50MHz have big different scale so I put the simulation in two pictures. **Figure b** is the continued of **Figure a**. When we at the 'gtdata (0000010)' state, when the posedge 7200Hz, we start to read the data. (As shown in Figure b, I use an edge_trap module to simulate the *posedge 7200Hz* in an if statement. The hz_72 is 1 at the posedge of 50M and 7200Hz, and then return to 0 and the negedge of 50M.) We use four states to read the data since each flash_mem_data contains 32 bits and the audio_out only has 8 bits. After read one data,

the word address starts to be increased by 1. Then check if test_address (77 right now) equals the end_word_address (127). If not, the state goes back to ‘gtdata’ and read the data at the new address.

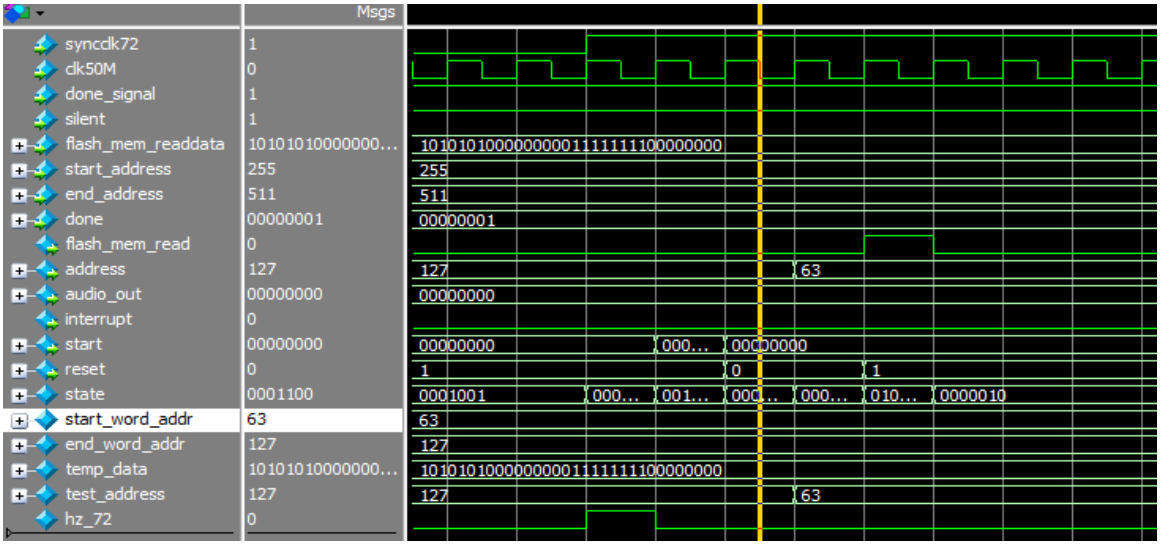


Figure c. The third part of the zoomed version of the Flash_handle simulation

Figure c is the continued of **Figure b**. As shown above, when the test_address reaches the end_word_address, the state goes back to the ‘init (1100)’, and the reset is 0 to tell the picoblaze it can start and get a new phoneme_sel. The new phoneme_sel will give the FSM a new start and end address. Then the reset becomes 1 to pause the picoblaze. The FSM then repeats the steps to read the data at the new address.

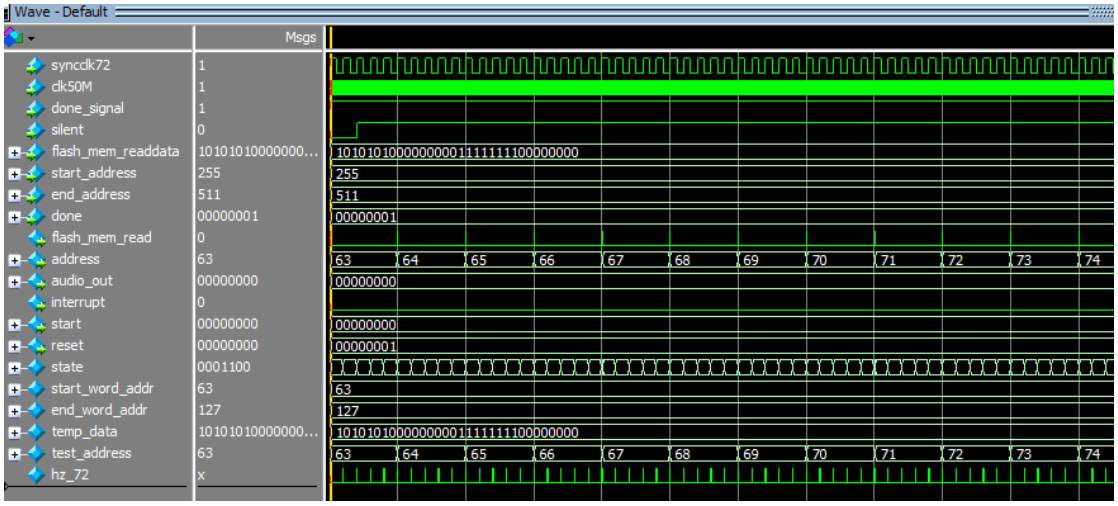
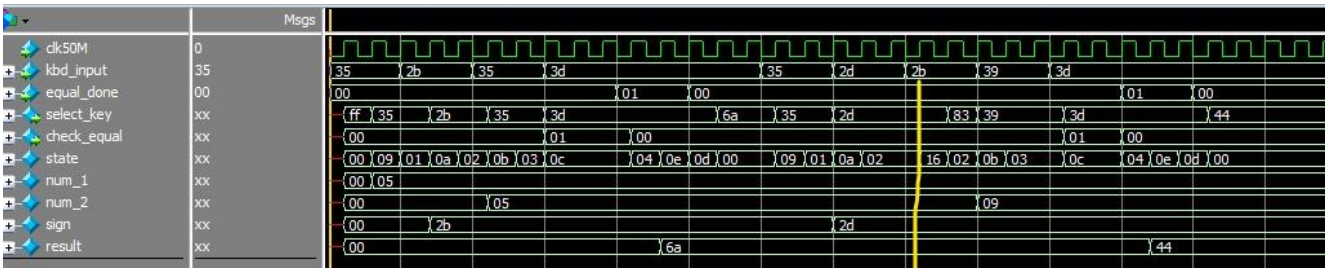


Figure d. The overall version of the Flash_handle simulation

In general, the Picoblaze starts first to give the FSM an address. The FSM starts after the picoblaze is done, and the picoblaze starts after the FSM is done.

In addition, as shown in **Figure d**, the ‘flash_mem_readdata’ and ‘hz_72’ signals look like ‘glitch’. However, they are NOT ‘glitch’. They look like that because the scale of the graph is too large. You can double check in **Figure c**, the zoomed version shows that these two signals are the useful signals instead of ‘glitch’. This simulation matches my expectation which indicates the design is fine.

(d) Keyboard



As shown above, the initial state is 'idle' waiting for an input. Then a '5 (35 in hex)' is inputted, the state goes to 'idle_out (09)' (the num_1 captures the input at this time) and 'second (01)' to wait for the second input. Then a 'plus (2B)' is inputted, the state goes to 'second_out (0A)' and 'third (02)' to wait for the third input. Then a '5 (35)' is inputted, the state goes to 'third_out (0B)' and num_2 captures the input. The state then goes to 'fourth (03)' and wait for the input 'equal(3D)'. When the 'equal' is inputted, 'check_equal' is 1. At this moment, this FSM is paused and the picoblaze starts to talk 'equal'. When the talking process is done, 'equal_done' is 1, and the state goes to 'done (04)' to do the calculation. Now the we got the result and output the result to the picoblaze so that it can give the flash_handle FSM a signal and correct address to talk. Then the state goes back to idle and wait for another input, and repeat the process.
This time, we input a '5' and a 'minus (2D)'. The next input should be a number. However, we input a 'plus (2B)' to test the error. When the input is 'plus' (at the yellow bar), the state goes to 'our_error_3_plus (16)' from 'third (02)'. Then the selecy_key is '83' which is 'error' signal in picoblaze. Then we input a '9 (39)', The state goes to 'third_out (0B)', 'fourth (03)', 'fourth_out (0C)' and 'done (04)'. At the done state, the algorithm gives the correct result '44 (hex)' which is 'minus 4' in picoblaze. This indicates the minus-result also works well. Then the state goes back to idle and wait for the next input. This simulation matches my expectation which indicates the design is fine.

(e) Led_control

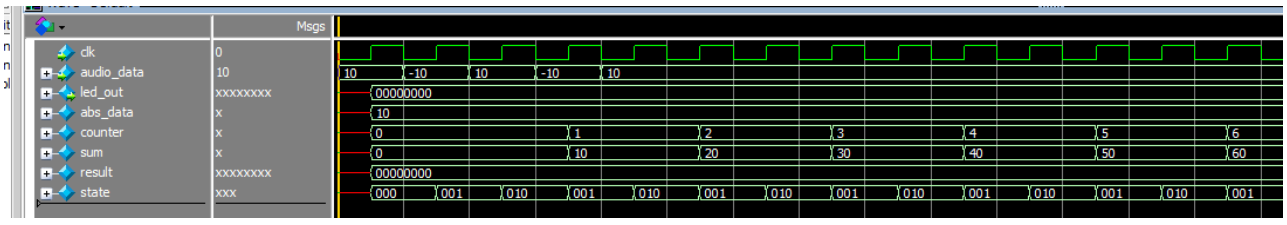


Figure e. The initial part of the led_control simulation

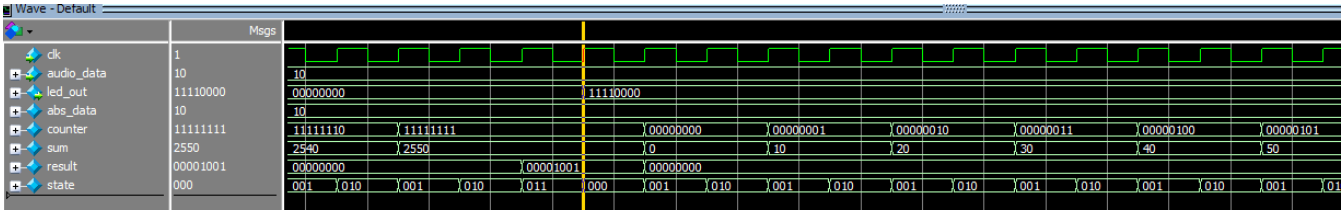
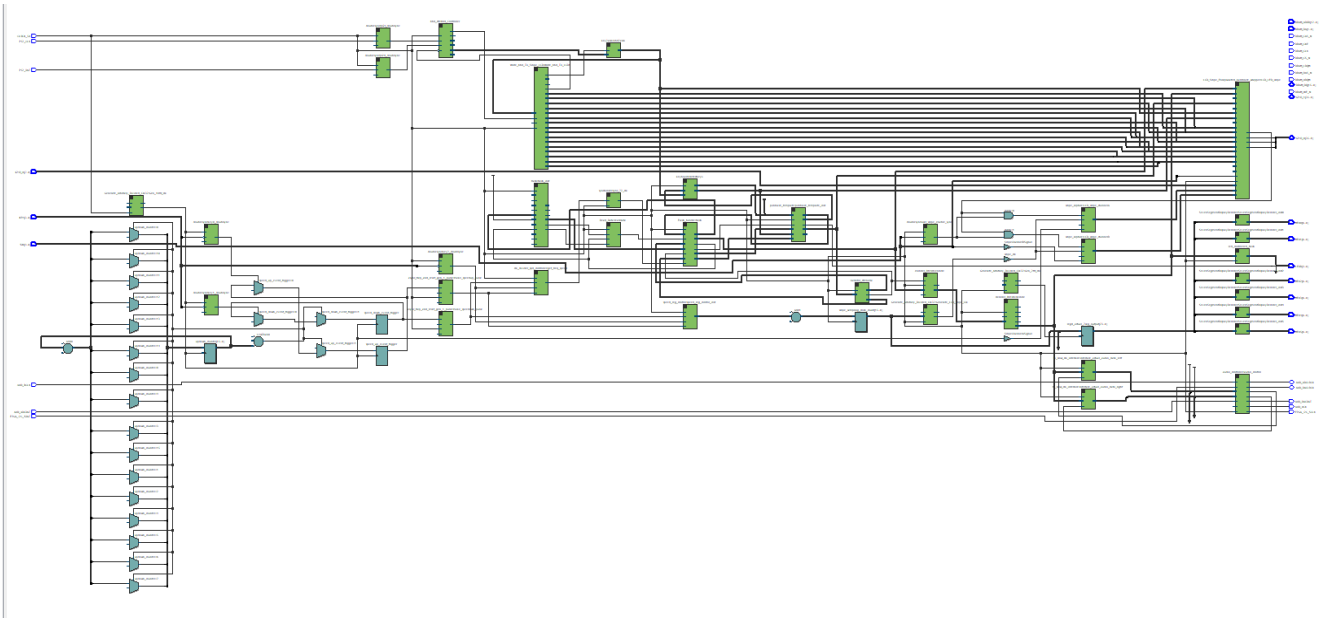


Figure f. The first flash part of the led_control simulation

Figure e shows the beginning of the led_control module. At the beginning, the initial state is 'init (000)' and counter, sum, result are all 0 (default). The first audio_data is 10 (decimal), the state then goes to find_mbs (001) to check if it is negative. Then the state goes to sum_up (010) and the counter starts to be increased by 1. The second audio_data is -10(in decimal), when the state goes to find_mbs (001), we got the absolution value so the abs_data is always 10, and then the counter is increased by 1 again. When the counter is 255 (decimal), we do the **averaging**. As shown in **Figure f**, when the counter is 255, we do the average and get the result. The result is 1001 in binary. Then we go to the display_led (011) state to check the most significant bit. As shown above, the most significant bit is result[3], therefore the led_out is 11110000 (light up from left to right as required.) Then the sum, counter and result are reset to 0. The process starts again. This simulation matches my expectation which indicates the design is fine.

RTL viewer of the design



4. How to run the simulations:

The testbenches for the clk_divider_spd_controller, Read_Data, Flash_handle, keyboard and led_control are called tb_clk_divider_spd_controller.sv, tb_Read_Data.sv, tb_Flash_handle.sv, tb_keyboard.sv and tb_led_control respectively. These testbenches are located at:
\rtl
I run the simulation in **ModelSim 10.3c** by clicking the 'simulation' button on the tool bar.
If you have my do. files, to simulate, click the 'start simulation' and choose the corresponding testbench. Then input the command 'do <filename>'. Then click 'run all' to get the waveforms.

4. Additional information:

- To make the picoblaze connection easier, I changed the .psm file name from speech.psm to ***pracpico.psm*** (the same name as lab 2 but the contents are different) so that I don't need to change the port name of template. **The .psm file is located at /doc/KCMSP3/Assemble Name: pracPICO.psm**

-The speed control is the same as solution, which is not very fast changing. Therefore, if you want to see obvious speed change, please hold the KEY0/KEY1 for a little bit long time. Also, KEY 2 is reset the speed.

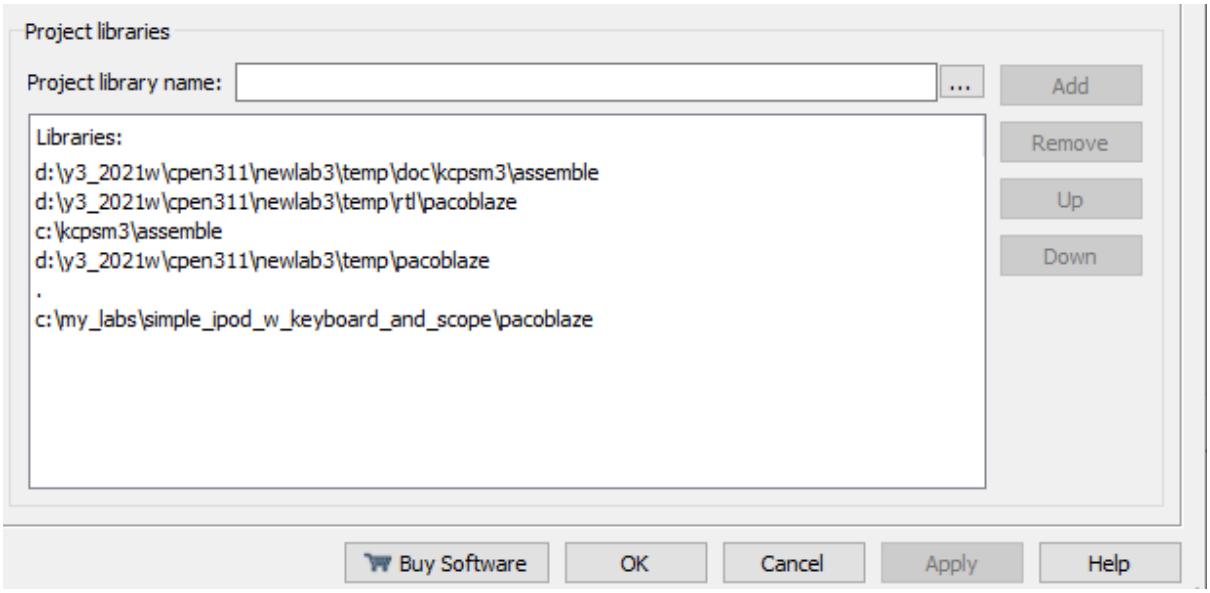
-The 8b10b encoing/decoding works well and I put the 'silent' signal as the control_character as required.

As shown above, the last 4 bytes of Line 2 represent the audio_data before and after encode/decode. The green part is the audio_data after encode/decode, and the blue part is the audio_data before the encode/decode. The green part exactly equals to the blue part, which indicates the encode/decode works well.

-All the code files and simulation files (do. Files) are inside **rtl** because I simulate the design using **ModelSim**, which requires all the project files under the same dictionary.

-The code files (.v) are in **rtl**, the KCPSM3 (.psm) files are in **doc**.

-To compile in Quartus, you should first set up the Library in Quartus as following:



(Please ignore the all **c:**/since I put KCPSM3 in C when I did the lab. When I submit the lab, I put all the files in D:/). If you download my files in C drive, you should change to the corresponding location which include the pracpico folder.

Tips to run:

After you download the file on your computer (let's assume you download it into C drive), the first thing is to set up the DOSBox. The default drive for the DOSbox is Z:/, so we need to do:

1. mount c c:/
2. cd "*the location of the '**Assemble**'*"
3. KCPSM3 pracpico > compile.log

Make sure to use the **left shift** for the '>'.

After a few minutes, a .MEM file will be generated. To check if it is successful, click the COMPILE.LOG and check if the bottom line shows the 'successful'. If so, copy the .MEM file and paste into the **rtl**/, and start the compilation. It should work.

-I really appreciate all helps from the professor and TAs. Even though this course is hard, it is also my favorite course. Thank you for the exciting term! If you have any questions about my project please let me know via piazza or email ruyizhou0711@gmail.com or 1820330027@qq.com.