Lab Report 5

Group 17

He Menglin - 260863025

Mohammad Noor - 260782491

Work submitted to

Prof. Katarzyna Radecka

ECSE 324

Computer Organization

Fall 2019

McGill University

December 3rd , 2019

<u>Introduction:</u>

As the last lab for the semester, lab 5 is the application about the materials we learned during the lecture and the lab. Also, by looking at the lab description, we found out that the lab 5 is an extension of lab 4.

There are three parts in this lab, which are audio, make waves, and control waves.

Note: all the files for the driver are given by the instructor, so the main implementation in this lab is to implement the *main.c file*, which will call the corresponding subroutines declared in the driver folder, to have the accurate logic.

<u>Part 0: Audio</u>

In this part, the main work is to find the sampling rate of the DE1-Soc Board and calculate the number of samples for each half cycle of the square wave.

From the *<u>DE1-Soc Computer System with ARM Cortex-A9</u>* manual, the default setting for the sampling rate is 48K samples/sec. A 100 Hz wave is to be played, that means there are 100 complete cycles of the wave contained in 48,000 samples, so for 240 samples a ‘1’ should be written to the FIFOs, and for 240 samples a ‘0’ should be written to the FIFOs. We wrote 0x00FFFFFF and 0x00000000 to the FIFO instead of ‘1’ and ‘0’

<u>Part 1: Make Waves</u>

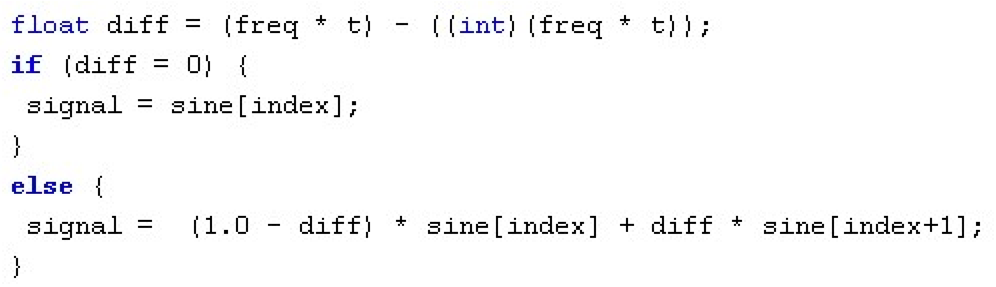
In this part, we wrote the C code which will take an input frequency and sample point then calculate the index and the signal at sample point t.

$$\text{index} = (f * t) \bmod 48000,$$
$$\text{signal}[t] = \text{amplitude} * \text{table}[\text{index}].$$

Figure 1: Signal calculation formulas given by the instructor in the lab 5 description document

If more than one note is played at the same time, then the notes should be added together and will have a different frequency by summing up each given frequency. However, our implementation gave out noise when you try to press more than one keys. Different approaches have been tested out, however, we were not able to find a property solution.

Meanwhile, if the index is not an integer, then the table[index] in the second formula of Figure 1 should be calculated by liner approximation using the floor and ceiling value of the non-integer index, ie, table[10.73] should be calculated with (1 – 0.73) * table[10] + 0.73 * table[11]. This part has been taken care with the following code in *Figure 2*.



```
float diff = (freq * t) - ((int)(freq * t));
if (diff = 0) {
 signal = sine[index];
}
else {
 signal =  (1.0 - diff) * sine[index] + diff * sine[index+1];
}
```

Figure 2: Fragment of the code in main.c file for the non-integer index calculation

In next part of this lab, we will use this method to calculate the signal value of the waves with the given 8 frequencies in the lab description document.

By calculating the timer flag rising time from the De1-Soc Board samples (48k samples/sec), we obtain the followings: 1/48000s = 20.8 * 10^{-6} s as the *hps_time.timeout* value with an error of 0.8*10^{-6} s.

Challenges faced:

During the demo period, the TA pointed out that when we press more than one keys simultaneously, it could not generate note, but instead, we have the noise. We tried to modify our C code, but considering the time manner, we were not able to correct implement this part. Also, the liner approximation part was tricky, the logic behind the calculation is the key to the success.

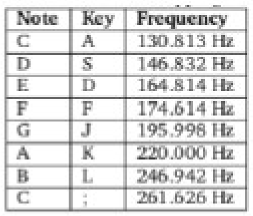
Possible improvements made/could have made

We use the timeout value of 20 instead of 20.8, which is the exact value in our design approach, the notes will be more precise and clearer if we choose to use the exact value. But considering about the liner approximation part in the design, which also causes the unprecise of the implementation, we stayed with the approximate value.

Part 2: Control Waves

In this part, by pressing the key on a P/S2 keyboard, the corresponding note shall be played through the audio port.

First, in this part, we were given the notes with each key on the keyboard with the corresponding frequency, please refer to *Table 1*.

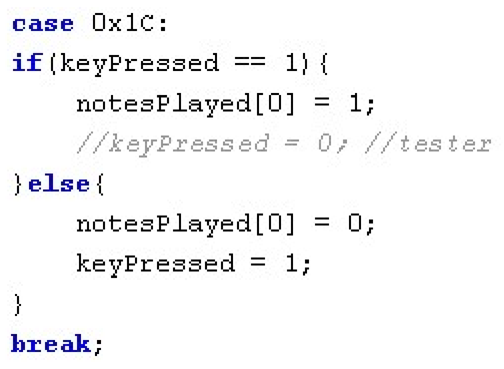


Note	Key	Frequency
C	A	130.813 Hz
D	S	146.832 Hz
E	D	164.814 Hz
F	F	174.614 Hz
G	J	195.998 Hz
A	K	220.000 Hz
B	L	246.942 Hz
C	;	261.626 Hz

Table 1: The note and frequency of the corresponding key on a P/S2 keyboard

We store the eight frequencies in an array table so we are able to find the right note-frequency combination from the key pressed. Considering the number of keys we have, a switch case statement was used instead of if else statement. Two variables are being used in each switch: (1) keyPressed which takes value 1 or 0 to indicate if the key has been pressed or not (2) notesPlayed which is an array of length 8 where each slot in the array corresponds to one of the eight keys used on the keyboard. If a specific key was pressed, its corresponding position in the notesPlayed array was set to 1. This notesPlayed array was then passed to the *makeSignal* function to generate the signal for each key pressed.

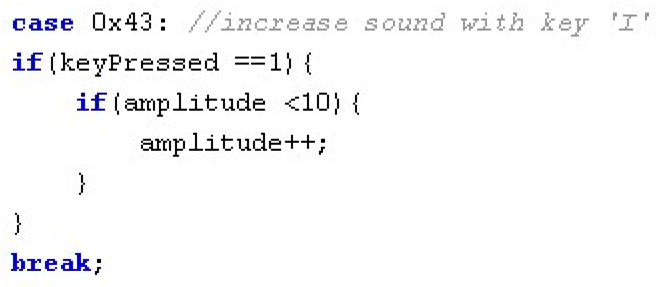
Below is an example of the switch case for key A being pressed, see *Figure 3.*



```c
case 0x1C:
if(keyPressed == 1){
    notesPlayed[0] = 1;
    //keyPressed = 0; //tester
}else{
    notesPlayed[0] = 0;
    keyPressed = 1;
}
break;
```

Figure 3: Fragment of the code in main.c file for the switch case for key A being pressed

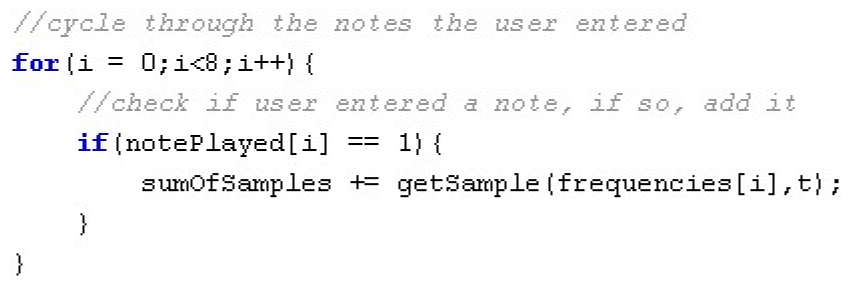
Also, in the lab description, we are required to implement the volume up and volume down by pressing some keys on the keyboard. The volume of the notes is related with the amplitude, so by incrementing or decreasing the amplitude, we are able to control the volume. We used the key I for increasing and R for reduce the volume. The code shown in *Figure 4* is our approach.



```c
case 0x43: //increase sound with key 'I'
if(keyPressed ==1){
    if(amplitude <10){
        amplitude++;
    }
}
break;
```

Figure 4: Fragment of the code in main.c file for the switch case for controlling the volume

With all the switch cases implement, we can control the key being pressed, the next step is to successfully generate the note with the right frequency. With the help of the formulas defined in **Part 1** (see *Figure 1* and *Figure 2*), a method called *makeSignal* takes into two argument, the *notePlayed* and *t* to generate the note from the pressing key.



```c
//cycle through the notes the user entered
for(i = 0;i<8;i++){
    //check if user entered a note, if so, add it
    if(notePlayed[i] == 1){
        sumOfSamples += getSample(frequencies[i],t);
    }
}
```

Figure 5: Fragment of the code in main.c file for the generation of the signal based on the key pressed

In order to capture the functionality of pressing multiple keys, a for loop in introduced (see *Figure 5*) to sum up all the frequencies through *getSample(frequencies[i], t)*, however, as descripted in Part 1, this function gives noise instead of actual notes.

<u>Challenges faced:</u>

The challenges here is to find a way to associate the key pressed with the frequency and play the notes, we managed to implement this part. Also, the pressing multiple keys at the same time was and still is a challenge for our group.

<u>Possible improvements made/could have made</u>

We might need to have a more precise time for our system, this might fix our problem with the multiple key pressing part.

<u>Conclusion</u>

With the end of lab 5, we finish the lab section of ECSE 324. The application of the theory learned in the lecture and the understanding of the assembly code is the key concept for the lab. Also, allocate time outside lab section is necessary for finishing the lab on time.