ECSE 324 – COMPUTER ORGANIZATION

LAB 5 – REPORT

Group 34

Jia Wei Sun

260866206

Karl Michel Koerich

260870321

FALL 2019

Table of Contents

1 INTRODUCTION

In this Lab, we combined the low-level ARM programming techniques we have acquired in the course and implemented a musical synthesizer. To do so, firstly we wrote a driver containing one subroutine that would either write data in both FIFOs and return 1, or write nothing and return zero. Secondly, we used the provided drivers and files to play notes of different frequencies. To do so, we wrote a function that takes as input f and t and returns a signal at t. Finally, we implemented the functionalities of an actual musical synthesizer by mapping notes of certain frequencies to keys of a PS/2 keyboard. We also implemented a volume control.

2 APPROACHES

Given the fact that we were given a lot of the base code to work as starting point, our approach when developing the synthesizer consisted on working around the already existing subroutines, and using the De 1-SoC Computer Manual and the lectures slides to guide us.

2.1 Designing the Subroutine

When designing the subroutine, we wanted it to be highly reproducible to avoid repetitions in the code. We found a snippet of code in C that checks to see when the depth of either the left or right Read FIFO has exceeded 75% full, and then moves the data from these FIFOs into a memory buffer. We based ourselves in this logic and implemented a subroutine that checks for space in both FIFOs, if there is, it writes an integer argument and returns **1**, if not, it returns **0**.

2.2 Writing Useful Functions

We wrote two functions getSignal(float freq, int time) and createSignal(). We wrote the 1st one using the following formula provided in the Lab 5 instructions,

$$index = (f * t) mod 48000 \tag{1}$$

where the f is the frequency in Hertz, and t is sampling instant. The signal at a specific time t is then calculated by

$$signal[t] = amplitude * table[index] \tag{2}$$

where the amplitude is a set value initiated by the program but that can be altered by the user by lowering or increasing the volume, and table[index] refers to the values provided to us in the instructions (seen in Table 1). Finally, the function returns the signal at the end of the function call.

Table 1: Note Mapping

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

The 2nd function we wrote iterates through all frequencies of the keys pressed and performs an addition of all the frequencies of the keys pressed, which means that pressing multiple keys at the same time will result in a different frequency that is not in the table initially. If the resulting index is not an integer, we calculate the table[index] by linear interpolating the two nearest samples. Because the table does not cover a lot of notes, we noticed that when pressing multiple keys, the audio resulting is very weird and somehow disturbing to hear.

2.3 Writing the Main Logic

We wanted to write a main function that is capable of recognizing the keys pressed and output the corresponding frequency. We first check if a keyboard key is pressed using a big condition (while) loop that only runs if a valid key is pressed, which means that one of the 10 keys that we are using for this lab. Since there are only 10 valid keys, we chose to use the switch case to correctly link each frequency to them. At the end of the switch statements, we create the signal using the 2nd function mentioned earlier, and we multiply it by a higher or lower volume value depending on whether we want it louder or softer.

We used switch cases to associate frequencies to individual keys. For example, the 1st frequency in the wavetable (C note) is associated with the A key, the 2nd frequency (D note) is associated with the S key, etc. We used the PS/2 keyboard scan codes to correctly label each key. Every case is associated with a specific input key such that if the A key is pressed, we proceed with case 0x1C since it is A's scan code. We also need 2 keys for volume adjustment; the process is very similar to that for frequencies. We associated the scan code of that key to a case, and we increase or decrease the volume at each click. As mentioned before, the total signal is outputted at the end of the switch statements with its volume adjustment applied to it.

3 ISSUES

There were 2 main issues encountered when implementing a musical synthesizer. The first issue we encountered was when writing the subroutine to check if there was space on the FIFOs or not. We were told to refer back to Section 4.1 (pp. 39-40) of the De 1-SoC Computer Manual. There, we found a snippet of code in C that checks to see when the depth of either the left or right Read FIFO has exceeded 75% full, and then moves the data from these FIFOs into a memory buffer. Using this example, we were able to understand how the audio port registers work, and then we wrote the subroutine that takes an integer argument and writes it to both the left and the right FIFO only if there is space in both FIFOs.

The second issue we encountered was to perform the linear interpolation between samples when more than 1 key was pressed at the same time. According to the Lab instructions, if the index was not an integer, we should calculate table[index] by linear interpolation using the two nearest samples. To solve this issue, we counted with the help of TAs who instructed us to write a function that does that, this way we would avoid repetition and the code would look cleaner. At every loop, this is called to generate the resulting signal. It calculate the interpolation of the 2 nearest samples at a time t. This way, no matter how many keys are being pressed at the same time, the function would still get the resulting index and do the linear interpolation, what results sometimes in weird noises being played.

4 IMPROVEMENTS

This section describes the improvements we added to the synthesizer and futures we would like to implement if time was not a constraint.

4.1 Implemented Improvements

A unique feature we added to the music synthesizer was a volume control strategically placed underneath the note keys. Also, we made sure that the volume can be increased or decreased while holding other keys by adding it on the same loop as the note keys pressed. This way, the user can change the volume using their thumb and while holding the notes, making changing the volume more smooth.

4.2 Further Improvements

One of the main improvements we could implement to the musical synthesizer would be fixing the problem of getting weird noises when multiple (5 or more) keys were pressed at the same time. We noticed this issue while testing and were not able to verify its causes. We believe the problem probably arises from the linear interpolation of the 2 nearest samples when the resulting index is not an integer, and its possibly caused by the fact that our table of frequencies does not account for multiple notes. We would gave to further investigate the causes to come up with a solution.

Another improvement we could make to our synthesizer to make it more realistic and useful would be adding more musical notes, such as sharps and flats, to account for a higher range of possibilities. This would greatly help the issue above since there a difference between the nearest known samples would be diminished.

Also, we noticed that after a long time using the synthesizer, some keys will gradually stop working. This problem would also require more time to be investigated and solved.

5 CONCLUSION

In this Lab, we implemented a musical synthesizer with the techniques we have acquired in the course. A lot of the initial code had already been given to us, what facilitate our implementation. Our main tasks were to design a subroutine to handle writing data to the FIFOs, play notes of different frequencies according to the keys being pressed, and handle edge cases when multiple keys are pressed. We had a lot of freedom to choose how the synthesizer would work, and with more time, we would have definitely come up with more complex and interesting implementations. Overall, this Lab helped us learn the material presented in class related to memory, and indirectly increased our interest in low-level programming.