LAB 5
SYNTHESIZER

Group 29

INTESARUL QAYYUM KHAN -26079991

SHIRAAZ DILAWAAR - 260577486

Introduction

In this lab, low-level ARM programming techniques are combined to implement a musical synthesizer. The lab consisted of three parts, which were:

1. Make Waves

2. Control Waves

3. Display Waves

Description:

The goal of this lab was to design a synthesizer capable of playing 8 notes (C, D, E, F, G, A, B and C). The user had to be able to play those notes by pressing the following keyboard keys: 'A', 'S', 'D', 'F', 'J', 'K', 'L' and ';'. These keys respectively matched the notes listed before. The supported frequencies are those corresponding to a limited range of musical notes. The specific note to be played is controlled by holding down its associated key (see Table 1) on the PS/2 keyboard, which is connected to the DE1-SoC via its PS/2 port. The synthesizer should be able to play multiple notes at once by superposing their respective soundwaves. The synthesizer also had to be able to display the soundwave of the note being played on a screen connected to the Altera board via a VGA cable. Finally, the synthesizer had to allow the user to change the volume of the sound being played.

Table 1: Representation of Musical Notes

PS/2 Keyboard Key	A	S	D	F	J	K	L	;
Musical Note	C	D	E	F	G	A	B	C

1 – Make Waves

For this part of the lab, we were to write C code that will take an input frequencyand time and return a signal[t], a sample, as shown in equation [2] below. If more than 1 note was played it was expected to add the notes together by summing the samples for each given frequency. The signal was generated using the equations shown below. The table array was a file provided that consisted of a 1Hz sine wave.

$$index = (f * t) mod\ 48000 \quad [1]$$

$$signal[t] = amplitude * table[index] \quad [2]$$

In addition, if the index was not an integer, we were expected to interpolate the two nearest samples and add them together as shown in example equation [3]:

$$sample[10.73] = (1 - 0.73) * sinewave[10] + 0.73 * sinewave \quad [3]$$

The initial approach we took was to first write a method generateSignal that would return the sample of the wave, using equation [1] [2] and [3]; the method would accept an input of frequency and time and return the signal sample at that index. This was very simple as the equations directed how the code was to be written; only simple computations were done.

The signal calculated is fed to the audio codec using an IRS with a timeout of 20 microseconds. When the audio flag is 1 the program using multiple if statements checks which boolean variables associated with a keyboard button is 1 and get the sound sample using the makeWave function for that button. If more than one button is pressed at the same time the final sample is calculated by adding the different samples as mentioned in the Generate signal method using the formula.

2 – Control Waves

As mentioned in the description this part was about mapping the particular keys to their corresponding Note. Firstly, an array of size 8 called keyPressed[ ] (corresponding to each of the buttons) filled with 0's was declared, this was used to store which buttons were currently being pressed. A variable called keyReleased was used to determine that a key had been released but did not store which key that was. To determine which key had been pressed, a switch statement with 12 cases was made, 8 of which correspond to a 'key' button being pressed, 2 of which allow to alter the volume, 1 for the break code and the final one was a default statement that just cleared the break code. For the 8 cases that corresponded to the make code of each of the characters, the code consisted of an if-else statement. If keyReleased equals 1, set the KeyPressed array corresponding to the specific button to 0 and then reset keyReleased to 0. Otherwise, the key has not been released but pressed and the corresponding entry in KeyPressed should be set high, to 1. For the break code case, since all of the characters started with the same break code 'F0', we would simply set keyReleased to 1 and then it was known that the next character that came up had been released. The volume keys, < and >, were chosen because they had the same break code as the note keys which made the code simpler, the graphics on them are also conducive to represent volume changes. There was a max limit of 10 for the amplitude and a minimum of 0 so as not to overflow. To generate the actual wave we use generateSignal(char* keys, int t) to continuously generate the sample of the signal according to which of KeyPressed, this method would take input an array of which keys were pressed. There is another array, frequencies[], that is index-bound to the keyPressed array. frequencies[] holds the values for the frequencies of each note. generateSignal() looped through the array of keys pressed and summed up all the samples from getSample(float freq, int t) corresponding to each frequency.

3 – Display Waves

The third component of the lab was to display the waves generated onto the screen. To do this, it was simply expected to basically map the value of t (time) onto the x value on the screen and map the magnitude of the sample the y value to the at that corresponding t value. We implemented an array that stored the y values for each point that had been drawn on the previous pass through. The index for this history array was the x value. Since it was not necessary to draw

a point for all 48000 samples, and since there are only 320 horizontal pixels anyway we only draw a value of t if it is a multiple of 10. To center the signal on the screen the number 120 was added to the signal. The signal was also divided by 500,000 in order to fit on the screen.

Challenges faced and solutions:

Throughout the lab, we faced several challenges while implementing the code, all of which are outlined below:

One of the biggest challenges faced in this lab were the use of the interrupts and the timeout values. We solved this problem by using a timeout value of 20 microseconds for the audio which has a frequency of 48000 Hz. As for the keyboard timer we realized we needed to use another timer with a lower frequency. While testing out the audio using the keyboard we found out that if 3 or more keys are pressed at the same time the sound becomes distorted. Initially we were simply adding the samples if more than one key was pressed however by taking the average of the sample we reduced the distortion. For the display the two challenges we faced were finding a suitable sampling and finding the scaling factor for the wave. Both these problems were solved by trail and error as we tried different numbers until we produced and acceptable wave.