ECSE 324

Lab5

Saturday, November 30, 2019

Group 66

Gengyi Sun 260768270

Liang Zhao 260781081

Introduction

The task for this lab was to implement a synthesizer using ARM. We were required to write C programs and Audio ARM code to interact directed with the provided ARM subroutines.

0. Audio

The audio part of our lab 5 was very similar to the keyboard part of Lab 4. It used registers to hold the output data before actually outputting the data. The registers that held the input or output data had another register that stores how much space there was. There were two registers that hold the left and right output data channels. There were two 8-bits sections of the FIFO space register called WSLC and WSRC, which were used to store the amount of remaining space in the output registers.

The ARM code for this part was to check the values in WSRC and WSLC. If both two registers that hold the left and right output data channels were represented free space, then we wrote a value to both of the two data registers.

For the C code, we had to use the default sampling rate of the CODEC, which was found in the DE1-SoC manual to be 48K sample per second and determine what rate we had to write values to the registers to output a 100HZ audio signal in forms of a square wave. From the lab we knew that if the codec sampled at 100 samples per seconds and we wanted a 2HZ signal then two full cycles would need to occur in 100 samples. Therefore, in 50 samples per cycle, we needed 25 to be ‘high’ and 25 to be ‘low’. For getting 100HZ signal at 48K samples per seconds, we would need 48K/100 = 480 samples per cycle, which were 240 ‘high’ and 240 ‘low’. We did this by writing a C code in which would run two loops counting up two 240 that wrote 0x00FFFFFF high, and 0x00000000 low. Each loop would check if there was a space. If there was a space then it meant the write was successful, otherwise try again by decrementing the counter. The loops also checked if the write was successful by detecting the value returned, for which 1 represented a successful write and a 0 represented a failed write.

We didn't find any challenges when we were doing this part.

1. Make Waves

For this part of lab 5, we were to write C code that would take an input frequency and time and return a signal[t], the signal equation was shown in [2] below. Additionally, if more than one note is played at the same time, it would compute the sample for each note and add them together under each given frequency.

The signal was computed by using the equations below. The table array was a file which consisted of a 1HZ sine wave.

$$\boldsymbol{index = (f * t)\ mod\ 48000} \textbf{ - [1]}$$
$$\boldsymbol{Signal[t] = amplitude * table[index] - [2]}$$

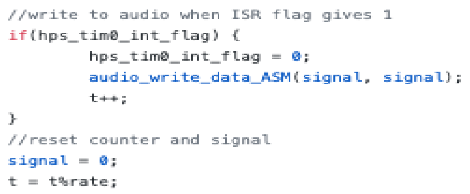
Furthermore, if the index was not an integer, we would need to interpolate the two nearest samples and add them together as shown in equation [3].

$$\boldsymbol{table[10.73] \coloneqq (1 - 0.73) * table[10] + 0.73 * table[11] - [3]}$$

The method we used was to first write a method which would return the sample of the wave. The method would take an input of frequency and time and return the signal sample at that index. This part was straightforward and relatively easy to implement as the equations [1], [2] and [3] was already given.

We didn't use the linear interpolation method, this is because although by using the interpolation method, the calculation could be more precise, it would cause even more unnecessary calculations which would slow down the whole process. In addition, the sample had to be cast to an integer in order to use the audio_write_data_ASM() method. The error of casting the sample to an integer would be less than 1HZ, which we thought it was acceptable because it was very hard to be distinguished by human ears. The equations [1] and [2] were less complex so we used them only to increase the efficiency of our codes.

A 1HZ sine wave was given, which contained 48K samples. We used timers to compute the samples and write the samples when enough time had passed. The timer flag was raised every 20 microseconds because the FPGA samples at a rate of 48k per second. Writing more often than that would be too fast and might skip values or result in failure because the audio registers were full while writing slower than that might make the output sound like a lower frequency.



```
//write to audio when ISR flag gives 1
if(hps_tim0_int_flag) {
        hps_tim0_int_flag = 0;
        audio_write_data_ASM(signal, signal);
        t++;
}
//reset counter and signal
signal = 0;
t = t%rate;
```

The hps_tim0_int_flag went high every 20 microseconds and the sine wave index t was only incremented by 1 when the audio sample was written so it would not miss any values

The challenges we faced in this part was to decide not to use the linear interpolation method. We were confused when we implemented the interpolation method because the process was slow and sometimes the waves didn’t appear to be correct.

3. Control Waves

For this part of lab 5, we needed to implement program which could control the waves using the keyboards A, S, D, F, J, K and L.

We used an array to store which buttons were currently being pressed. To determine which key had been pressed, we used a switch statement with 11 cases, of which 7 cases were used to determine which key buttons had been pressed and 2 cases were used to control the volume (G and H). The other two cases were used to break and clear the code.

For each note that is currently pressed, we retrieve the appropriate signal output for the sample instant (that we keep track of using a counter that resets back to 0 after reaching 48000, indicating the end of one sampling period), and sum them together. We multiply this sum by the current volume (to modify amplitude for volume output) and store the result in the audio codec interface.

The challenges we had faced was that there were too many switch cases and it was very easy to make mistakes.