ECSE 325: LAB 5 REPORT

April 13, 2019

Imad Dodin (260713381)

Nada Marawan (260720514)

McGill University

Department of Electrical and Computer Engineering

0.1 Introduction

The task of this lab was to implement a musical synthesizer using ARM programming techniques. The implementation that we were required to develop was done in C, and interfaced directly with the ARM subroutines that we were provided. It was interesting to note that we were provided `.s.o` files, hiding the implementation details of these subroutines, but were provided with API documentation that allowed them to treat them as 'black-boxes' (although the implementation was required for previous labs).

0.2 Make Waves

We are provided with drivers for writing to the audio codec and a wavetable entitled `sine` that contains a single period of a 1 Hz sine wave that is sampled at 48000Hz. For easier comprehension, we consider this as a replacement for a sine function such as provided in Java or Python. We are then given a relation that allows us to compute the required signal to pass to the audio codec interface at time t, for a desired signal of frequency f:

$$index = (f * t) mod 48000$$

$$signal[t] = amplitude * table[index]$$

We can then use interpolation in the case that the computed index is not an integer. The example given in the lab manual is $table[10.73] = (1-0.73) * table[10] + 0.73 * table[11]$. We write a function to compute the above logic, this is essentially a case of rewriting the provided equations into C-code and casting between double and integer types, as required.

0.3 Control Waves

We use the PS/2 Keyboard as input for controlling the output sound waves from the synthesizer. This is fairly trivial to implement given the provided drivers. We read into a character pointer with the provided `read_ps2_data_ASM(char* string)` subroutine. This stores the last keystroke code (single byte) (if applicable) in the provided character pointer (and returns 1 or 0 as success/failure codes, respectively). We use an infinite `while` loop to continuously poll the PS/2 input. Every time we poll the input. we check the received code. If the code is a make code (i.e. not `0xF0`) that corresponds to one of {`A, S, D, F, J, K, L, ;, -, =`} keys used for controlling the synthesizer sound (and in the case of - and =, volume), we update the entry in the global integer array used to indicate current key-presses that corresponds to the detected keypress to 1. In the case of the - and = keys, we increase or decrease the volume variable's value. If we detect a break code (`0xF0`), we repeatedly poll the PS/2 interface for the next byte. This byte is the same code as for the previous key-presses, but now it indicates that the key has been released (a key-up, as it were). We perform the same check as before (but now we do not have to check for the volume buttons as the volume buttons execute a one-time action upon keypress, not a continuous output into the audio codec, as is the case for the other keys). This time instead of setting the value in the global keypress array to 1, we set it to 0, indicating that the key is not pressed.

We set a timer to trigger every 20 microseconds, which corresponds to the 48000 Hz rate. This timer's flag is checked on every iteration of the `while` loop, and when it is raised, the required signal output, corresponding to the pressed keys, is computed and stored into the audio codec interface using the provided driver function. Instead of continuously using the function described in Section 1 to compute the required signal for each pressed key, we construct a 8-by-48000 table that corresponds to the required signal output for each

sampling time for each of the 8 required notes. 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.
In general, no issues were faced in writing this section of code, however we faced difficulties when trying to modularize the code - due to our discomfort with handling 2D Array pointers etc. in C.

0.4 Displaying the Wave

Every 10 polls, we display a point on the screen via the VGA interface. We compute the x-coordinate of the output point by taking the poll-number and modulo-ing it by 320, the x-range of the screen. We then compute the y-coordinate of the point by taking 120, the mid-point of the vertical dimension of the screen and adding a scaled down version of the sample to this mid-point (we scale down by a factor of 60000, found by trial and error). The value is stored in an array of "previously drawn" points, as, before drawing a point on the screen, we need to clear the point previously corresponding to the x-coordinate from the screen. After clearing the previous point from the screen, we draw a new point to it. This is continuously done as we output signals to our Audio Codec, so we have a live representation of the output wave on the screen.
We note, again, that this section was not very difficult to implement. Additionally, while waiting for our turn to demo, we implemented dynamic colour changes (every point uses a different colour), while outputting. While this was, obviously, not a required feature, it was fun to do and was a nice way to end a great semester of labs!