Giuseppe Lipari (260742811)

Alex Masciotra (260746829)

Monday April 16, 2018

ECSE 324

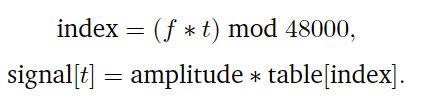
**Lab 5 – Synthesizer**

Introduction:

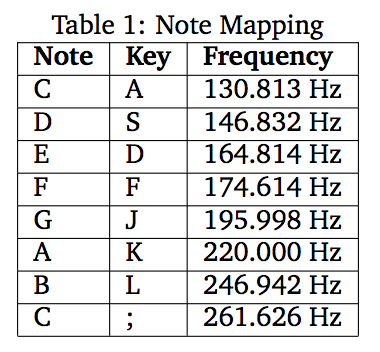
The purpose of this lab was to use all our knowledge that we have acquired throughout the semester about ARM programming in order to implement a music synthesizer. Within this lab, there were three different sections. The first section was the “Make Waves” section which was essentially making sound waves. The second section was the “Control Waves” section which was controlling the sound waves using the buttons on a keyboard. Finally, the last section was the “Display Waves”, which displayed the sound waves that were being played on a computer monitor. We found this lab to be quite challenging, since it required us to use many different techniques from previous labs, such as polling and interrupt timers. In addition, a lot of code needed to be written from scratch and the instructions were more ambiguous.

Make Waves:

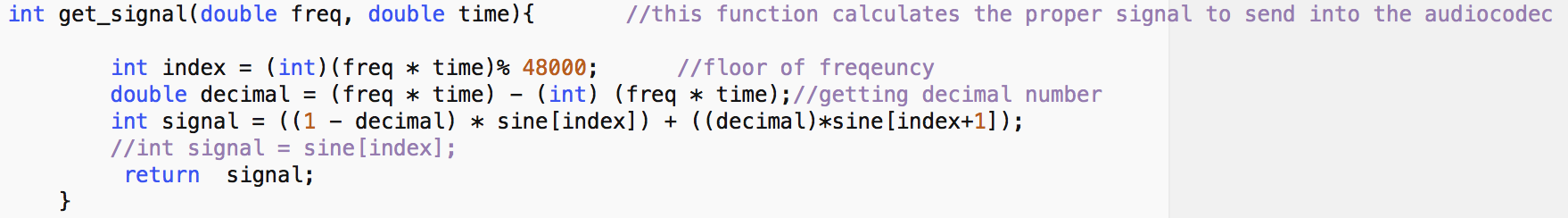
For this first section of the lab, we were tasked with writing a function that computes the sound signal for a given frequency. More specifically, we needed a function that computed the sound signal for eight different frequencies (8 different notes that can be seen in Table 1). In order to calculate these sound signals we used two formulas from the lab document given to us:



The 48,000 comes from the fact that DE1-SoC Computer System works at a sampling rate of 48k sample/sec. In addition, a table was given to us showing what frequency plays what musical note as can be seen below in Table 1:



This section of the lab proved to be quite challenging. One challenge that we faced was to deal with what data type was needed to cast the linear interpolation. This is when the computed index is not an integer value. In essence, this means that the value is between two samples on the wavetable. In order for us to implement the interpolation and the sound signal we coded the following snippet of code as can be seen in Figure 2:



**Figure 2: Interpolation**

We had casting issues since modulus operators could not be used on doubles. In other words, we had to find a way to retrieve the proper decimal value. We do this by subtracting the value of the frequency multiplied by the time to the same value casted by an int.

Control Waves:

Our next task was the implementation of the keyboard characteristics of the synthesizer. We used the PS/2 keyboard keys in order to play the eight different notes. The eight different notes that we played were done by using the A, S, D, F, J, K, L and ; keys. In addition the plus and minus keys on the right side of the keyboard were also used to increase or decrease the amplitude (the sound). Fortunately, the drivers for the PS/2 keyboard were provided for us.

In order to implement this code, the idea of a make code and break code needs to be understood. A make code is produced when a button is pressed and a break code is produced when it is released, which consists of F0 and the make code of the button that was released.

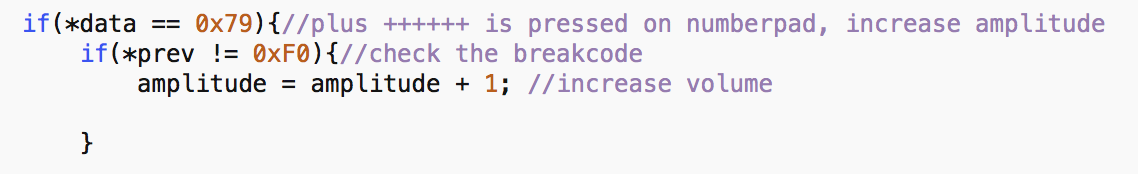
The make codes of each corresponding button of the keyboard were found by referring to Figure 3 below:



**Figure 3: Keyboard (exact representation of the keyboard used)**

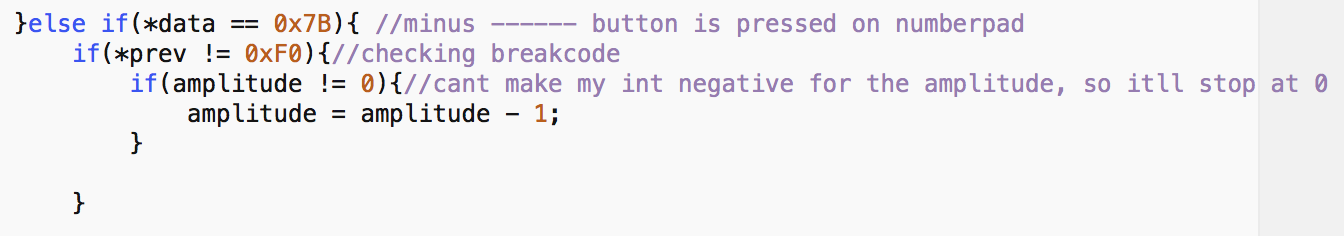
We used the read\_PS2\_data\_ASM() function provided in the drivers to retrieve the keyboard information. This function is called every clock cycle to gather the information about what key is being pressed and what key is being released. In order to save the keyboard information every clock cycle, we declared a global array to hold the status of the eight different keys representing the notes. Essentially, when a key is pressed, the value inside the array at its respective index is updated to 1, otherwise it is 0. To determine if the button was released or pressed, the keyboard queue was checked to see what was the previous code. If it was the break code FO, then the button was being released, and the array element was set to 0. If it wasn’t F0, then the button was being pressed and the element in the array was set to 1. This was repeated for each possible key.

In order to implement the volume keys, very similar logic was used. One new element that was added in this section was the idea of amplitude which is what ultimately controls the volume. By changing the amplitude, we change the amplitude of the sound wave which either makes the volume louder or softer. The amplitude was multiplied with each sample before being sent to the audio. We coded this volume section so that even if the volume key was held it would continuously grow or decrease in amplitude until max volume or min volume (no sound) is reached. To increase the amplitude the following code in Figure 4 is used:



**Figure 4: Volume up**

This quite simply checks if the plus button is being pressed and increments the amplitude accordingly. In order to code the volume down there was one little catch that can be seen in Figure 5:

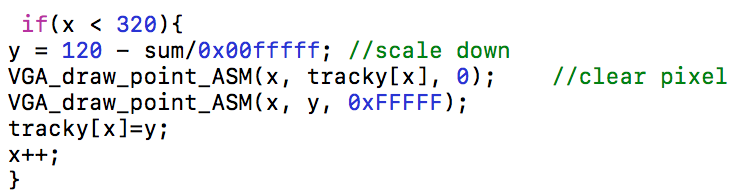


**Figure 5: Decrease the volume**

Since the volume could not be negative, there needed to be a condition checking if the amplitude was not equal to zero prior to decrementing the amplitude.

Display Waves:

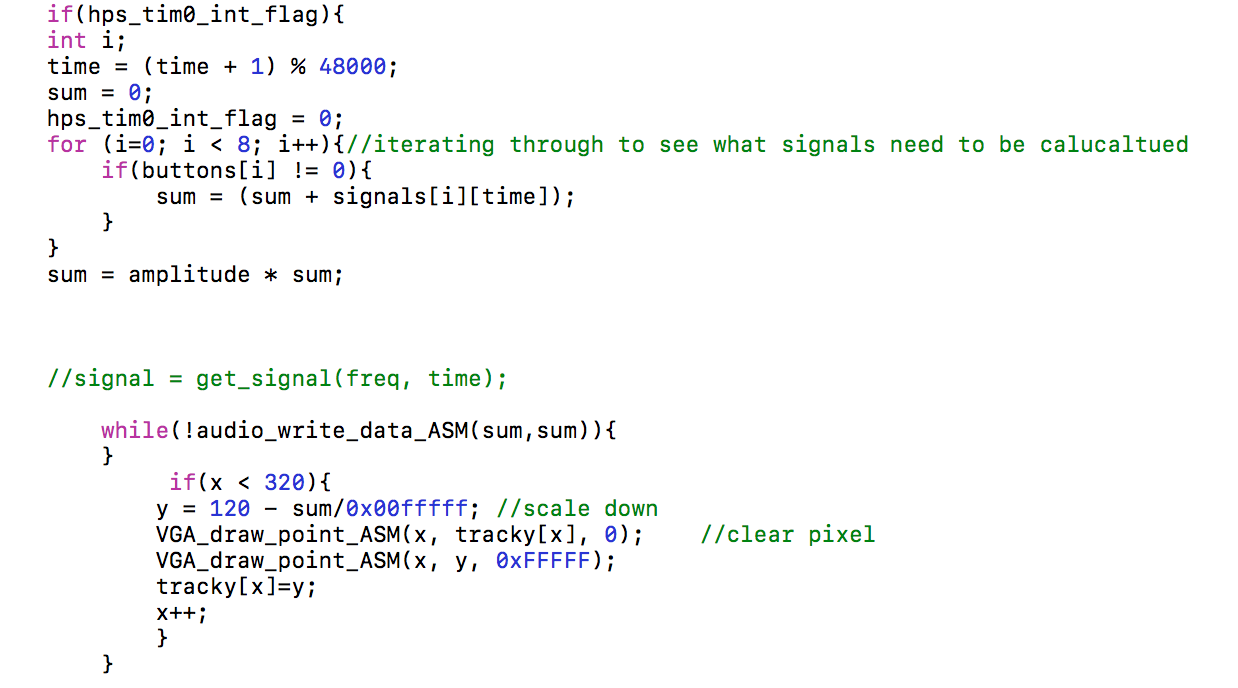
The third section was perhaps the trickiest. We were tasked with implementing the corresponding sound waves on the VGA monitor. The display subroutines were fortunately given to us. Our task was to display the x coordinate on the screen which represented the time and the y coordinate which represented the signal that is to be displayed. We wanted to produce a sinusoidal wave. However, in order to properly display the sound waves, the signal values need to be scaled so that they would fit on the screen and be centered as well as updated every time a button was pressed; if two signals needed to be added. Figure 6 shows how this was done.



**Figure 6: Draw**

As seen from the figure above, the entire draw was written in the main part. As the screen is 320 by 240 pixels, the y needed to be set at the middle of the screen. This is why the 120 is present, the sum represents the signal that is being played and it is scaled down in order to fit on the screen. Using the VGA Draw Point, the signal was drawn on the screen until the x reached 320. Afterwards, it would wait until another signal or same signal was being played, which set the x back to 0 in order to perform this loop. The x was set to 0 in another loop within the main that checked the input to the keyboard, and when another key was pressed, it saved the current key and reset the x to 0. In order to clear the display, using the CLEAR\_ASM took too long and did not work. Instead, the value of y was stored in an array of size 320, called tracky. This would track the y and save the y coordinate for every x coordinate in a certain pass. When the x was reset, the old signal was cleared by writing the color black (0X0) into the pixel location where a previous pixel was shown. Afterwards it drew the new pixel at the new y coordinate in white. This new y coordinate was then saved in the array for the next pass through.

Finally, we needed to put everything together. An interrupt timer was used with a timeout set to 20. This was the inverse of the 48000 sampling rate. In the main, we first calculated and stored all the signals in an 8 by 48000 2D array. This was done to save time when the program was running and be able to produce nicer sounds. The way it saved time, was that instead of calling the get\_signal function every time a signal needed to be played with corresponding frequencies, and using the processor to calculate a signal; all the possible signals were already calculated and stored. Instead of calculating a new signal every interrupt, it would just access the array and take what signals needed to be added together corresponding to which buttons are pressed and what the value of the timer is. Once these signals were stored, the keyboard function was called to be able to read the data from the keyboard in order to know what signals needed to be played. We were polling the interrupt timer flag in order to execute the rest of our code, which consisted of incrementing the timer variable and making sure it does not exceed 48000. Afterwards, we accessed the 2D array corresponding to the buttons that were being pressed and the value of the timer in order to calculate the correct signal stored in the sum. This sum was multiplied by the amplitude and then written in the write\_audio\_ASM as long as there is space in the data register (returning a value of 1). This is why it is in a while loop. Then this signal was drawn on the screen explained above. The way this was written can be seen in Figure 7:



**Figure 7: snippet of the main**

Conclusion:

To conclude, this lab proved to be challenging. Many different elements needed to be done correctly in order for the full project to work. This was a good concluding lab since previous knowledge from lab 3 and 4 was incorporated. All the labs were very helpful and all in all a lot was learned about assembly coding. We’ve never coded in a language so close to the hardware before. In addition, the material learnt during these labs was very helpful with the course material considering that often it was too abstract to understand without actually doing it.