ECSE 324 – Computer Organization (Fall 2018)

Lab 5: Synthesizer

December 4th, 2018

<u>Group 12</u>

Alain Daccache - 260714615

Michel Abdel Nour - 260725050

About this lab

In this lab, we combined the low-level ARM programming techniques we have acquired in the course by implementing a musical synthesizer (or "synth") that can play 8 musical keys. The different I/O devices used were the Keyboard, the Audio, and the VGA display.

McGill

Part I: Make Wave

Given a wavetable that consists of a sine with with f = 1Hz, and a sampling rate of 48000, the wavetable contained 48000 samples. We could write a function get sample that took as input a frequency and instant, and outputs the corresponding signal. Using the equations provided in the lab module, we calculate the index as follows:

$$Index = (f * t) \, mod \, 48000$$

Then, we calculate the signal with the following equation, using the previous index computed:

$$Signal[t] = Amplitude * Table[index]$$

If the index is not an integer, we use linear interpolation of the two nearest samples. For instance, getting the signal at index 10.73 was done so

$$Signal[t] = (1 - 0.73) * Table[10] + 0.73 * Table[11]$$

In the table below, the keys are mapped to their respective frequencies. We therefore initialize an array that represent those frequencies.

When we needed to get samples for waves of different frequencies, for example 2 Hz, there had to be tweaks to be made in order to get values of samples. For f = 1 Hz, we take all the values from the array provided in wavetable.s. However, if f = 2 Hz, then we need to extrapolate the values by incrementing the index by 2 each time.

The increments depend on the value of the frequency f since the values in the sine wave are constant : -1 < sine < 1.

Challenges Faced

We also discovered that using the linear interpolation method, while technically more precise, introduced **unnecessary computational overhead** because in the end the sample had to be cast to an int in order to use the provided audio_write_data_ASM() method. Casting to an int would only be able to make the output different by 1Hz, which we decided was an acceptable error since the human ear cannot distinguish sounds that close together. Since equation [1] and [2] are less complex computations we used them to increase the efficiency of our code. Both methods have been left in the code base for comparison.

```
if ( hps_tim0_int_flag == 1 ) { // check
hps_tim0_int_flag = 0 ; // reset
audio_write_data_ASM ( signal_sum , signal_sum ); // write
t ++; // increment
}
```

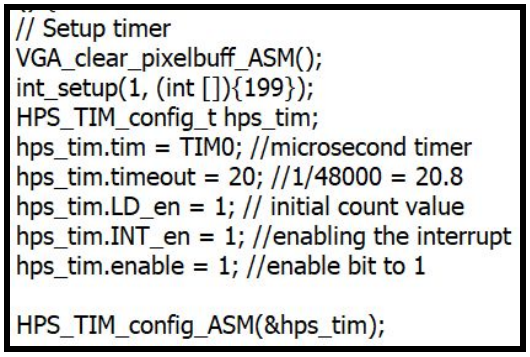
The timer flag, hps_tim0_int_flag , is continuously polled and goes high every 20µs. The sine wave index, t, is only incremented when the audio sample is successfully written so as not to miss any values.

The difficulty found in this component was to figure out how to use a timer to feed the generated samples to the audio codec, looking at previous labs for reference helped with the correct implementation of timers. The code had to be refactored from inputting a frequency and calculating all the sample values at that frequency to the opposite method where a counter was implemented and the audio sample was computed and written only when the timer allowed.

Part II: Control Waves

For the second part of the lab it was expected to be able to control the waves using the keyboard buttons A, S, D, F, J, K, L, and ;. Each of these buttons represented a different note of the octave from C - C, and had a corresponding frequency.

We used the HPS Timer which is set to 1 after a period of 20 microseconds, due to the fact that the sampling rate was 48000 samples/sec. By getting its inverse (~ 20 microseconds), we can write each sample at that time interval (and increment instant t), then set the timer flag to 0.



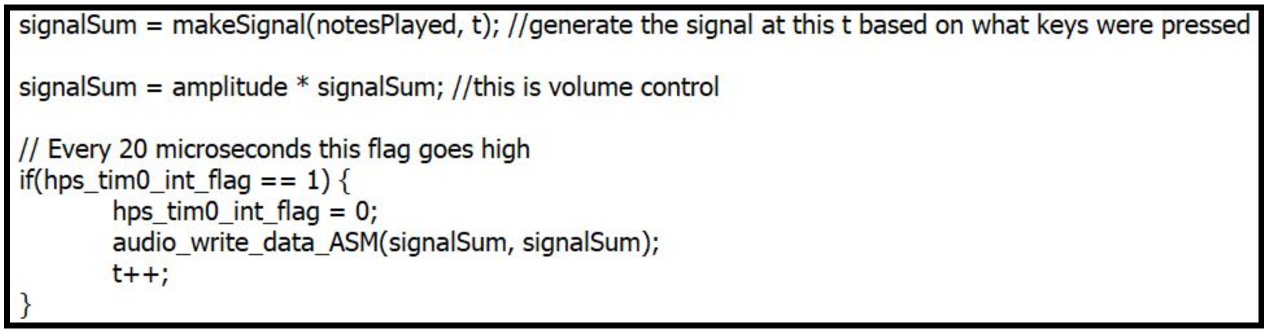
```
// Setup timer
VGA_clear_pixelbuff_ASM();
int_setup(1, (int []){199});
HPS_TIM_config_t hps_tim;
hps_tim.tim = TIM0; //microsecond timer
hps_tim.timeout = 20; //1/48000 = 20.8
hps_tim.LD_en = 1; // initial count value
hps_tim.INT_en = 1; //enabling the interrupt
hps_tim.enable = 1; //enable bit to 1

HPS_TIM_config_ASM(&hps_tim);
```

We created an array key_pressed of “one-hot encoded bits” to represent the keys that were pressed (value of 1) or not (value of 0). This way, we can deal with the edge case of having more than one key pressed at a time, as we later send this array to the makeSignal function described previously

We implemented a switch statement that takes 12 cases depending on the key that has been pressed. We had one case for the **make-code** of each key pressed, one case for **volume up**, one case for **volume down**, one case for the **break-code**, and one case for **default** (to clear break code).

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. This piece of code is shown below.

At the end of the switch statement, we collected keys pressed, convert them to a signal, and when audio is ready to accept input, we write into it:



```
signalSum = makeSignal(notesPlayed, t); //generate the signal at this t based on what keys were pressed

signalSum = amplitude * signalSum; //this is volume control

// Every 20 microseconds this flag goes high
if(hps_tim0_int_flag == 1) {
        hps_tim0_int_flag = 0;
        audio_write_data_ASM(signalSum, signalSum);
        t++;
}
```

Challenges Faced

To generate the actual wave we use makeSignal(char notes_played, 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 key_pressed array. frequencies[] holds the values for the frequencies of each note. makeSignal() looped through the array of keys pressed and summed up all the samples from getSample(float f, int t) corresponding to each frequency. Currently, pressing any more than three keys at a time results in some form of overflow and does not sound good. A way to fix this is explained below.

The timer was used after the signal was determine if it was time to write the audio to the output. The 'time' variable, t, was only incremented if the audio signal was written so that no samples were missed.

Improvements

We initially went with the idea of a fixed length array that held whether the key was currently being pressed or not which worked well throughout the implementation process.

Part III: Display Waves

In this part, we display the waves generated, onto the screen. We keep track of an index that represents instant t. At each 20 microseconds, the timer writes into the audio, and increments the instant t. Therefore, it makes sense to use t as our x-axis, and at each instant, draw the sample at an amplitude y given by the wavetable. We thus created an array of magnitudes for the sample at time t, which were to be mapped to the y and x- axis, respectively.

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. We divided the signal sum value by an arbitrary value in order to compact the sine wave and make it observable on the screen. To center the signal on the screen the signal with another arbitrary number. Those numbers were decided on by trial and error

Challenges Faced

We spent time figuring out how to use a timer to feed the generated samples to the audio codec. Looking at previous labs for reference helped with the correct implementation of timers.Initially, we implemented a for loop that looped through all 48000 points and calculated the sample at each. Since it was not efficient, we used timers, where the samples could be precomputed and were only written when enough time had passed.

Improvements

We could have used another method than linear interpolation when computing the signal. In fact, casting the sample to an int to call audio_write_data_ASM() made the output less precise (~ 1Hz), despite indistinguishable.

Instead of checking each case in a huge switch statement, we could create an array of make codes, and used a loop, iterating through to check which note(s) was/were pressed.

Finally, we could implement an interrupt for the keyboard to save processing time.