Lab 5: Synthesizer

Thomas Hillyer - 260680811
Omar Yamak - 260662616
Group 47
Submitted: 2017/12/07

1 - Make Waves

For this part of the lab, we were to write C code that will take an input frequency and 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. This section was the main backbone of this project.

$$index = (f * t) \; mod \; 48000 \quad [1]$$
$$signal[t] = amplitude * table[index] \quad [2]$$

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.

We 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.

A 1Hz sine wave was provided, which contained 48000 samples. Initially, we implemented a for loop that looped through all 48000 points and calculated the sample at each, this method worked but was slow and did not sound good. Instead, timers were implemented where the samples could be precomputed and were only written when enough time had passed. The timer flag was raised every 20 microseconds, which was chosen because the FPGA samples at a rate of 48k/sec.

$$1/48000[/s] = 20.8 * 10^{-6}\ [s]$$

Writing anymore often than that would be too fast and could potentially skip values or result in fail to write because the audio registers are full. Writing slower than that would make the output sound like a lower frequency.

```
if(hps_tim0_int_flag == 1) { // check
    hps_tim0_int_flag = 0; // reset
    audio_write_data_ASM(signalSum, signalSum); // 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.

2 - 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.

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. Here is an example of the switch-case for the ‘A’ key:

```
case 0x1C: // A = C = 130.813Hz
    if(keyReleased == 1){
            keysPressed[0] = 0;
            keyReleased = 0;
    } else{
            keysPressed[0] = 1;
    }
    break;
```

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.

```
case 0xF0: //break code
    keyReleased = 1;
    break;
```

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.

```
case 0x41: // volume down
    if(keyReleased == 1){
            if(amplitude>0)
                    amplitude--;
            keyReleased = 0;
    }
    break;
```

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. 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. There are improvements that could be made in this area that are outlined below.

Initially, it was difficult to think of what to write within each of the switch statement, the algorithm to be used to acknowledge that a key, or multiple keys were pressed seemed difficult. Our initial implementation used a linked list to store the keys being pressed, this proved to be difficult to implement and cumbersome. It was deemed too much overhead so instead we created a fixed length array that held whether the key was currently being pressed or not.

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. Since there are 320x240 pixels on the screen and we only draw 320 (i.e. less than 1%) of them it was overkill to clear the entire display every time a new wave was drawn. Doing so actually massively slowed the down the entire drawing operation. Instead 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.

This was fairly simple to write and the only challenges were making the signal fit on the screen.

4 - Improvements

We implemented a volume display that showed the current volume level of signal from 0-10. We also implemented a pause button that would freeze the keyboard and hold the current sound wave constant until the pause was released.

When four or more keys are pressed at the same time on the keyboard the sound becomes distorted. This is due to the fact that each additional frequency sample is added to the

rest. Eventually the number becomes too large and starts to overflow and the wave begins to ‘clip’. To fix this is simple, each sample needs to be normalized and reduced before being summed. For example, if there are four keys being pressed currently each sample should be divided by four before being summed up to create the overall sample.

In terms of the timer, we calculate the signal on every pass through the loop even if t has not incremented. A better approach to this would be to only calculate the new signal if t has increased. This reduces the number of computations and increases the speed of the code.