LAB 5 REPORT

Muhammad Anza Khan
260618490
Ayden Malik
260627064

Summary

For the final lab we were asked to build upon the simple 100Hz square wave we made in Lab 4 by building a fully functional synthesizer, which was very reminiscent to the kind heard in movies from the 1980’s, such as Blade Runner. We were given a table that had specific frequencies that mapped to certain keyboard keys, for example note C having a frequency of 130.813 Hz was mapped to key A.
Using this as a starting point, we designed our code so that all the possible notes were mapped to their respective keys and when the user pressed any key, it would generate a sound matching that specific note’s frequency.
We were also able to add multiple notes, which we display as colored sine waves, and we can manipulate their volume by changing the amplitude.

Make Wave:

The first part starts by taking an input frequency and time and returning a signal that is a function of time. It is calculated using the following equations, provided in the lab handout:

1. Signal[t] = amplitude*table[index]

The index number is calculated using the following formula:

2. Index (frequency*t)%48000

If the index is not an integer, we calculate it using the following interpolation formula provided
If more than one note was pressed, then the samples for each frequency were added together.

To generate our wave, we use the getSample function using the equation provided below:

3. Sample[]=(1-0.73)*sinewave[] +0.73*sinewave

Using the 1hz sine wave file provided which consisted of 48000 samples. We set the timer flag every 20 microseconds which is basically the sampling time period of the board.

After every 20 microseconds we send the current value of signalSum to the audio register and set the flag to 0 to be polled again and the sine wave index, t, is incremented after signalSum is passed.

Control Waves

For this part of the lab we were expected to to able to produce different notes from certain character of the keyboard, namely, A, S, D, J, K, L and ; , to see which character was pressed. To do this, we decided to create an integer array called notesPlayed. This array stores whatever note the user entered. We initially tried using individual data fields for the notes but that was not working, and after speaking to Ali, the TA we decided using an array was simpler because we simply store the notes in continuous memory locations and accessing them becomes easier for future use.

To check whether a note is pressed, we have implemented a series of switch cases, which test for the scan code of all possible notes that we were asked to play. If indeed a note matches to a scan code we enter into a logical block that will either store the note played or not play it.

This 'on' 'off' logic is implemented by using a flag called keyPressed which when set to 1 indicated a note has been pressed and if so stores the note in the notesPlayed array, if keyPressed is 0, than that means the note has not been played and we store a 0 in the array meaning we will not generate the sound of that note.

We added two extra cases for volume control, which is controlled by key I (I for increase) and R (R for reduce).

Display code

Here we were required to display the wave generated on the screen. So here we implement an array which contains the magnitude of the sample at a particular time. So the time t is mapped onto the x-axis of the screen and the magnitude i.e. valToDraw onto the y-axis.
Since we cannot display all 48000 samples onto the screen, to center the wave onto the screen and to reduce data handling we create an array called old_wave that contains values for valToDraw and is indexed using drawIndex. drawIndex is the index generated by checking to see which t a multiple of 10 and then dividing it by 320 and storing it.
To scale the wave onto the y axis we divide the signalSum by an arbitrary number, in this case 500000, and add 120 to it and store it in valToDraw.

Main

Our main function begins by first ensuring the screen is clean, this ensures that unwanted pixels do not corrupt our waveform; we do this by calling the relevant clear functions from the display files.

We calibrate HPS timer to 20 microseconds and the interrupt enable bit to 1, meaning we are servicing interrupts for the timer.

We than create the flag called keyPressed, which as mentioned above checks whether a note has been pressed or not.

An array called old_wave of size 320 that holds the value of valDraw. valDraw is initially set to 0.
Then we initiate a while loop which is explain above under the section 'Control Waves'.

Next, signalSum is a double which we use to store the results of makeSignal, which calculates the signal we will output based on the samples we get. Then it is multiplied by the value of volume at that time.

We set our sampling time to 20us so that our samples could be compute before hand and were only written when enough time had passed. Any faster than that, we could skip values and we would not be able to write anything into our audio registers since they would be full. Slower than that would distort our frequency and make it seem like it is a lower frequency.
Every 20 microseconds the flag goes high and we pass value of signalSum to the subroutine audio_write_data_ASM.

GetSample

This function was used to generate the signal using the 3 equations provided below and using the table array file provided that was mainly a 1Hz sine wave.

Also, if the index generated was not a whole number, then we use equation 3 to interpolate the samples using adjacent samples and adding them together.

1. Signal[t] = amplitude*table[index]

2. Index (frequency*t)%48000

3. Sample[]=(1-0.73)*sinewave[] +0.73*sinewave

valToDraw

As explained above this function is used to generated values y-axis values to put into the old_wave array.

Improvements

As pointed out by our TA during the demo, we could have implemented a lookup table for the character display on the screen.

Secondly, when we display the characters being pressed onto the screen, the sound output was distorted. We concluded that it was basically because of the boards internal hardware which caused the ISR to be delayed, the hardware in question we believe to be the pixelbuffer.

Another improvement we could have done was instead of checking each case iteratively, we should have had an array that stores all the scan code we needed for the notes and implemented a for loop that would cycle through each entry of the array checking which note was entered by the user.

We could have implemented interrupts for the keyboard but we felt that would have been overkill as already the ISR for the HPS timer causes such a delay that when attempting to display the notes the user entered it caused the sound to be extremely noisy.