ECSE 324
Computer Organization
Fall 2019

Date: December 03 2019

Lab 5 Report

Group 1

Preyansh Kaushik : 260790402
Elie Elia : 260759306

Lab 5 continues off from Lab 4 by first tasking us to create a driver for the audio port, to play a 100 Hz square wave on audio out. After this, we were to create a musical synthesizer by concurrently utilising ARM assembly code and C. Eight keys from the keyboard correspond to eight individual musical notes and the combination of keys and hence allowing for eight factorial of possible played notes. We mapped specific frequencies each to a key on the keyboard. By pressing one of these mapped keys, a wave would be created using the frequency and time values in real time. Multiple keys pressed concurrently allows for the combination of waves. The volume of the wave can also be modified through key presses hence modifying the amplitudes of generated waves.

1. Audio Driver

The audio component of this lab utilizes registers to hold the data being outputted before actually outputting the data. The registers holding the I/O data have an additional register R2 that stores how much space there is. The two registers left and right contain the output data corresponding to the left and right audio channels.

The fields WSRC and WSLC give the number of words currently available (that is, unused) for storing data in the right and left audio-out FIFOs. When all FIFOs in the audio port are cleared, the values provided in the Fifospace register are RARC = RALC = 0 and WSRC = WSLC = 128. This information was retrieved from the Altera University Program DE1-SoC Computer Manual.[1] In our code, we checked the current values of WSRC and WSLC to ensure that free space was available in our left and right registers, to which we would then write data to in order to indicate audio out.

The C portion of the audio driver incorporated the assembly routine detailed above, where the 48k samples per second default sampling rate of the CODEC was used. Again, the DE1-SoC manual was referenced for this rate. Calculations were then performed to determine the rate at which values had to be written to registers to output a 100Hz audio signal.

The laboratory instructions indicated that a 100Hz square wave should be played on the audio out port, where the frequency can be achieved by knowing the sampling rate of the audio DAC. With a requirement of a 2Hz square wave, there are two complete cycles of the wave in 100 samples. This results in 50 samples per cycle where 25 samples are ‘high’ and the remaining 25 samples are ‘low’. The same calculations were applied to the desired 100Hz signal at 48k samples/second where it was calculated that 48000/100 indicated 480 samples per cycle. Therefore 240 ‘high’ samples and 240 ‘low’ samples.

Our C code implements this calculation through two while loops that continue functioning as long as our counter is less than the 240 samples calculated previously. Our code either writes 0x00FFFFFF to indicate ‘high’, or 0x00000000, to indicate ‘low’. As long as the counter is not 240 (indicating that there is space) a write is successful, otherwise the counter is reduced by one in an attempt to write again successfully. A successful writing attempt receives a return value of 1 whereas an unsuccessful attempt at writing returns 0.

[1] Altera University Program DE1-SoC Computer Manual.
http://www-ug.eecg.toronto.edu/msl/handouts/DE1-SoC_Computer_Nios.pdf

Challenges:

The difficulties we faced were in attempting to check whether there was space available in WSRC and WSLC in order to be able to write. We initially attempted to peruse them stacking the register and moving right by 16 bits to dispose of the RARC and RALC bits, at that point contrasting the rest of the incentive with a consistent. We would then keep in touch with the two registers by utilizing counterbalance addresses.

2. **Making Waves**

Since we need to output the desired frequency, we first have to calculate the desired frequency as a function of the frequency f and the time t. An integer t will keep track of the time, and is a variable that grows from 0 to 48,000 milliseconds. We have this range given the sine wave has a sampling frequency of 48,000 Hz.

Given that the wave table provided contains one period of 1Hz and the sampling frequency is 48,000 Hz, we can estimate the sampling time of the board. By our estimation and interpolation, we estimated the sampling time of the board based on this information and made a design choice to implement a timer of 20 microseconds. The 20 microseconds would therefore be our sampling time of the board.

This sampling time is of key significance as we use this sampling time to set the interval of signal calculations. The calculations for the sine wave are done in the following manner:

1. First, calculate the index of the signal for which we take the sine value for the current time value.

   index = (frequency * time) mod 48000

2. If the index satisfies the condition of being an integer, we can get the value at the index in the sine array that is provided. This is the straightforward case.
3. However, if the index is not an integer (i.e. NaN), we have to use another formula to estimate the value:

   (1- mantessa of index) * sine[index] + mantessa of index * sine[index+1]

Using this formula, we estimate the value of the sine wave at that index. This contributes to uncertainty in our results.

Challenges:

The biggest challenge we faced in this section is finding a way to verify our calculated values where appropriate to the input provided. Because every frequency has 48000 possible outputs these means we couldn't check if everyone of the cases will work.

Improvements:
As an improvement we could use a bigger sample wave in order to get more precise values in case of a non-integer index. Because we won't have to estimate the values using the provided equation which is prone to errors. Instead we would use the exact values.

3. **Control Waves**

This section of the lab compromises the mapping of frequencies and created waves to keys on the keyboard. More specifically, the keys A, S, D, J, K, L and ; are respectively mapped to the provided frequencies. To achieve this we first create an array called “notesPlayed” that will keep track of what notes are currently being played be it a single one or many that will be summed. We have a continuous while loop that checks first that all switches are set to low, next reads the inputted keyboard value from the keyboard using “read_ps2_data_ASM” and a switch statement is used to determine what frequency to assign to the key pressed. Within each case of the switch statement, the flag denoting if a key has been pressed is checked and if true the note corresponding to the key pressed is added to the array mentioned above. If the key pressed flag has not been set then the slot in the array corresponding to that note is filled with a ‘ ‘ character so as to prevent it being included in the list of notes that have been played. Similarly, in this switch statement the pressing of “I” and “R” are checked and increment and decrement the amplitude respectively within a range of 0-10. Finally, the array of notes played is passed into the logic used to create the wave described above and a summative wave of all notes played has been created and on it’s way to be displayed. Our biggest challenge when completing this section was our attempting to complete it using individual data fields for each note rather than an array. Only after wasting valuable energy and time did we attempt the array approach and realize it was more effective and much easier to use during the implementation of the make wave section.

Improvements:

In terms of improvements we could significantly shorten our code by creating a function

checkKeyPressed(int index, char note){...}

that would take in the arguments of the note value and the index of the array to be passed written into. With each case in the switch statement this function would simplify the repetitive if and else cases and make for better coding practice.