COMP2300 A2: Polyphonic Sequencer

Rachel Schroder, u6301105

Introduction

I chose to extend my sequencer by giving it the capability to play multiple notes at the same time.

To create a waveform that sounds like multiple notes playing at once, the sequencer sums the triangle waves of each note. I chose to use a [illegible] wo reasons. Firstly, because I found a formula that would allow me to calculate the y v[illegible] ue, and secondly because I had already used a sine wave for assignment 1.

I have designed my sequence[illegible] can easily change the number of parts, song duration (in segments), and the song spe[illegible] ents per second). The maximum number of parts that can be played at once is 255, as this [illegible] yte. However, if each part has a large number of notes, this limit may be reduced due to the amount of memory available on the discoboard.

To demonstrate the capabilities of my sequencer, I chose to play the chorus of Carry on Wayward Son, by Kansas, as it demonstrates:

- Playing silence, in all or some parts
- Playing parts that have different rhythms
- Playing 3 notes at once
- Large pitch range (82-523 Hz)

My arrangement is altered from a medley called "Four Minute Fandoms" by Amber Balzar on Musescore (see Appendix, Figure 1). To convert the notes into frequencies, I used a table from Michigan Technical University's physics department.

Implementation

The overall structure of my implementation is shown in the Appendix, Figure 2.

The sections below provide more detail on the implementation and design of the main loops and functions.

Data storage

Information about the song to be played is stored in 3 types of array structure. I chose to use arrays because they are well suited for iteration.

The first, `song_info`, contains key information about the song. Its structure is shown in the Appendix, Figure 3. I designed this array to allow flexibility in song choice. The information in this array controls the song characteristics that the user can change. This structure also allows the user to change which parts they include in the playback without deleting them from the program. This allows storage of multiple songs and isolation of song parts.

The second type of array stores the notes that are played in each part. Each note is stored in a word, with the first halfword for the frequency (rounded to the nearest integer, or 0 for silence), and the second for the duration (in segments). Each part is stored in a separate array of this type.

The third array, `song_counters`, is used to track progress through each part for synchronisation. Each part's counter is stored in a word, the first halfword is the offset of the current note from the part's base address, and the second is the duration remaining for the current note. The counters are initialised at the start of each iteration of `song_loop`, and updated by `get_frequencies`. There needs to be at least one counter for each part so that the memory allocated for the audio library is not altered during playback.

Song loop

The purpose of this loop is to start the process that plays the full song. It first initialises the song counters, setting the offsets to zero and the durations to that of each part's first note. It then plays the number of segments specified in `song_info`, by starting `segment_for_loop`. The length of the song played is determined by the duration in `song_info`, not the number of segments in the part arrays.

Segment Loop

The purpose of this loop is to play n samples, where n is 48000/no. segments per second. The y value for each sample is calculated by summing the y values for each individual part's frequency, which is calculated by `get_frequencies`. The frequencies are not removed from the stack until the end of the segment, as they are needed for every sample.

The samples are played in segment groups so that the parts can be easily synchronised, while still allowing the part arrays to be as condense[illegible]egmentation also means that `get_frequencies` only has to be called once every n sampl[illegible]very sample, saving computation time.

The y value for each note is c[illegible]angle_wave function.

triangle_wave

This function, given a time va[illegible]) and frequency (f), generates a triangle wave. If the frequency is not zero, the y v[illegible]alue is calculated using Equation 1, where p = 48000/f.

A frequency of zero indicates silence, in which case the function returns zero before performing any calculations.

I implemented the `mod` and abs functions to calculate the result of this formula. The mod function was calculated using Equation 2:

Equation 1: triangle wave formula (Wikipedia, 2020)

$$y = {}^{2a}/_{p} * \left|(x \bmod p) - {}^{p}/_{2}\right| - {}^{2a}/_{4}$$

Equation 1: Modulus formula (Wikipedia, 2020)

$$a \bmod n = a - (n * \lfloor {}^{a}/_{n} \rfloor)$$

I needed to use a function to generate the triangle wave so that the waves would be continuous throughout each note, and thus reduce the amount of clicking caused by abrupt jumps in the y value. It also simplifies the process of calculating the y value as, unlike in part 1, I do not need to keep track of the previous value.

get_frequencies

This function calculates, for each part, the frequency in the current segment, and returns it on the stack. Its only argument is the number of parts being played.

The frequencies are returned on the stack because the number of parts changes, and thus it is not known how many registers would need to be allocated. Using the stack also makes it easier to retrieve the frequencies when they are used in the `additive_loop`.

Importantly, this function ensures that the parts are synchronised by updating the song counters. Without the ability to synchronise parts, the sequencer would not be able to play the notes together at the correct time.

If a note has finished (duration counter = 0), the offset counter is moved to the next note, and the duration is updated to reflect the full length of the new note.

After checking if the note has finished, the function pushes the frequency for the current note to the stack and decrements the duration counter to indicate that the segment has been played. The duration has to be stored in a counter because the original array needs to be maintained for the next loop of the song. It is also much more memory efficient than storing a whole song's worth of durations in a separate array.

Possible Improvements

The simplest improvement I would make is to write functions for loading and storing the offset and duration counters. This would make the code easier to both read and write, as there would be less time spent interpreting and fiddling with registers.

To improve the usability of the program, I could have stored the notes using midi numbers rather than frequencies. However, I ran into precision problems when implementing the midi number to frequency formula. I decided that this problem was too complex to solve, and not the best use of my time as it only marginally improves usability.

I could create a dictionary-like structure that stores pairs of all the midi numbers and frequencies, however it would be infeasible to implement the full range of frequencies.

References

Balzar, A., n.d.. *Four Minute Fandoms.* s.l.:Musescore.

Michigan Technical University Physics Department, n.d.. *Frequencies of Musical Notes, A4 = 440Hz.* [Online]
Available at: https://pages.mtu.edu/~suits/notefreqs.html
[Accessed 14 May 2020].

Wikipedia, 2020. *Modulo operation.* [Online]
Available at: https://en.wikip[illegible]lo_operation
[Accessed 11 May 2020].

Wikipedia, 2020. *Triangle wa*[illegible]
Available at: https://en.wikip[illegible]le_wave
[Accessed 11 May 2020].

Appendix

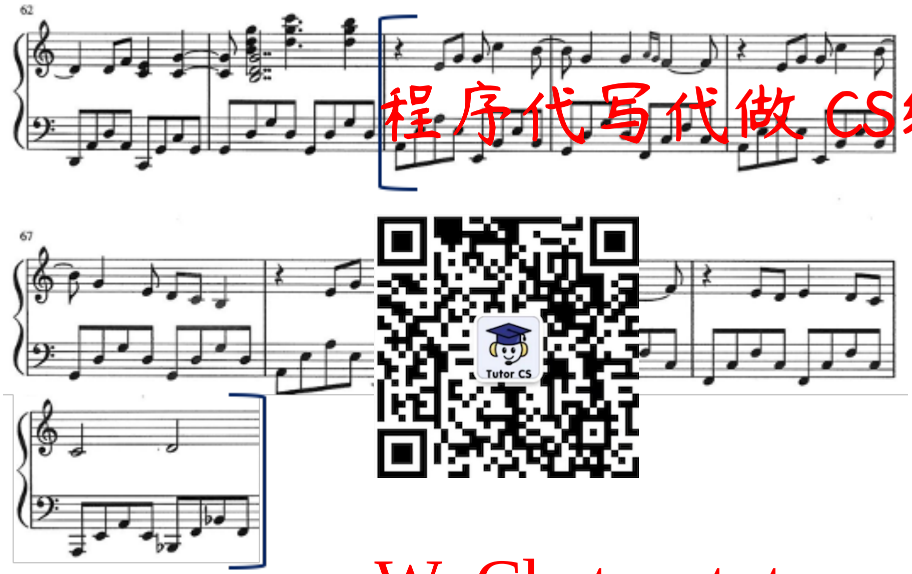

Figure 1: Arrangement of Carry on Wayward Son (Balzan, n.d.)

song_loop

- Runs indefinitely, once for every play through
- Initialises counter variables

segment_for_loop

- Runs once for every segment in the song (specified by song duration)
- Calls get_frequencies to generate frequencies that need to be played during the segment

play_sample_loop

- Runs once for every sample in the segment
- Adds the y values of the frequencies for this segment to generate the sample y value using additive_loop
- Plays the sample

Figure 2: Overview of program structure

Description	Element size
Number of parts (n)	byte
Segments/second	byte
Duration (number of segments)	halfword
Part address 1	word
Part address 2	word
⋮	
Part address n	word

Figure 3: song_info array structure