Design Document of Assignment 2

COMP2300 – Danny Feng (u6611178)

Version 1, 30 Apr. 2019

1 Overview

For this assignment of creating music sequencer by assembly code on the STM32L476G discovery board, I spent many efforts and successfully implemented the powerful sequencers that can play both harmony and amplitude envelope. Moreover, they are straightforward to use for different music. Just pass in different notes' frequency and duration as parameters. My final submissions for part 2 include three branches, "Happy-Birthday-Song", "Billie_Eilish_bad_guy_harmony", and "part-2" which is actually the ADST envelope version of the Happy Birthday Song. This report is structured based on these three branches.

2 Branch Happy-Birthday-Song

2.1 Judgement of Using Arrays

This branch is based on my earlier version of part-1. I do not use an array for storing notes in this branch, which I believe is a slightly faster and the most manageable way. When using arrays, I have to use load and save instructions that take two CPU cycles per each operation. In this branch, I repeatedly call the function with the parameters moved into registers. However, such speed benefits are not practical because the BSP call will always take 1/48,000 seconds. You might argue that the code is not clean by copy and paste the function call many times. However, it makes the reader more comfortable to see the mapping between the current note and its duration. For using arrays, the code writer is very likely to lose track of such relation. That is exactly what I experienced when writing them. I used arrays in part-1 and part-2 branch in order to make the marker happy since it looks more concise.

2.2 Syntax Diversity

Besides the song and array difference from part-1, in this branch, I use jumper label instead of IT block, and stmdb/ldmia operation instead of push/pop just to make myself familiar with all kinds of methods.

3 Branch Billie_Eilish_bad_guy_harmony

3.1 Functionality

In this branch, I implemented the harmony by taking an average of two waves. Load-twiddle-store pattern, as well as the divide-and-conquer idea is being used. For example, the divisible checking helper function I wrote is to make the big task simpler. I chose this song for its intense rhythmic and popularity. In my implementation, the single notes and harmony notes play by turns. My harmony sequencer takes two frequency and their duration, then play them at the same time. So, I use it to play two octaves in this music.

3.2 Algorithms and Implementations

The wave averaging algorithm I created for harmony is not complicated. A cycle of a square wave contains an upper part and a lower part. Each BSP_AUDIO_OUT_Play_Sample call is to draw a single dot. Those dots will make up a wave. When we take the average of two dots

from square waves, the resulting dot only appears in three positions. Upper, lower, and middle. If both two original dots are currently on the upper or lower part, the average dot should be on the same place. If two original dots are on the opposite position, the harmony dot for this point of time ought to be in the middle. By tracking the current dot count and checking if it is divisible by the total dots in the half cycle, we can know that if the current dot is the turning point (next dot will change from the upper to lower, or vice versa). I used two registers as flags that represent the dot position status for two original waves. 0 refers to upper and 1 means lower. When the current dot is a turning point for an initial wave, the corresponding flag will be flipped, which indicates the changing position of that wave. The Exclusive OR of two flags points out whether the average wave should be in the middle or not. If it is not in the middle, then it must be at the same position as the original waves.

3.3 limitations

The major disadvantage of this method is lack of flexibility. As the harmony function can only play two notes with the same duration at the same time.

4 Branch part-2 (Happy-Birthday-Song-ADSR-Envelope-Version)

The technical of memory allocation, if-else conditions, memory store and load, arrays have been used in this branch. This sequencer will allocate 30%, 20%, 50 % of the whole note time to Attack, Decay, and Sustain part respectively. What's more, 20 % of the extra time will be given to the Release. The highest amplitude is set at 0xFFF5 while the lowest is 0x8011. This value is slightly less than the maximum range that the BSP function supports in order to avoid overflow. Starting from the amplitude of ±10, the code calculates the growth per loop dynamically so that it reaches the target amplitude I set above smoothly. Those ADSR ratios can be changed by edit the divisor. The program keeps track of the current wave number as well as dot number. More importantly, the sequencer function still follows the ARMv7 Architecture Procedure Call Standard, r0 to r3 used as parameters, and the rest of the registers remain untouched when the function returns. The sound it creates is not only clear, precise but also beautiful.

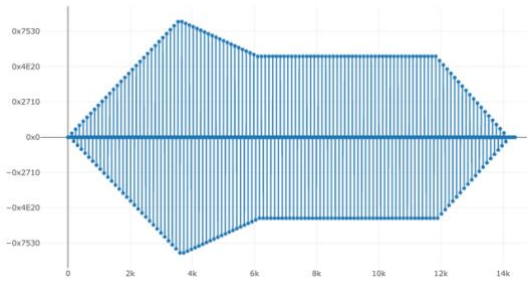

Fig.1 ADSR Envelope wave the sequencer generated

5 Summary and Reflection

Each of the branches above uses different methods and techniques to achieve different goals. This is meant to be challenging and separated so that I can practice more. However, I met several similar problems when developing those sequencers. The most common one was: Forgetting to store vital registers like link register and r0 to r3. The value in these registers is likely to change when calling other function. That will lead to nondeterministic of the code or get lost. Last but not least, the symmetry of push and pop operation is also crucial. Otherwise, the stack pointer might be in an unknown position and brings disaster for later stack operation.