COMP 2300 Assignment 1

Part 2 - Design Document

Aditya Sharma - u6051965

Sequence

The sequence this program generates is the Pokemon Center Theme, namely from Pokemon Red/Blue/Green/Yellow as all the music in these games was made in assembly also. Personally, Pokemon has been a big childhood influence in my life and is widely regarded in popular culture as having excellent music scores and even better gameplay, which is the reasoning behind this choice.

Although the theme itself has several channels of audio which are used to play true harmony rather than additive waveforms (two notes at a time) - which gives it more depth and somewhat of a backing bassline, I have decided on doing solely the melody (the main sequence of notes) as additive synthesis was not within my scope for this assignment. This was because I wanted to work more on creating the melody and the right sequence of notes in proportionally adequate timing when compared to the original, regardless of beats per minute (BPM) increasing or decreasing. This means that the BPM may not exactly be the same, but the notes still sit close to where they would on sheet music regardless of the BPM - half notes would still be half notes, triplets would be triplets, etc. As the playback is time based, calculating triplets and half notes is difficult by ear, so the final result may not be exact.

I decided to use these tabs available for free on Ultimate Guitar in order to create the final melody. I first played the tabs on guitar to ensure they sounded like the original song, then translated the frequencies using a free online chart by the Michigan Technological University. (https://pages.mtu.edu/~suits/notefreqs.html)
The final notes, translated to frequencies, have been commented out and listed at the bottom of the code.

Structure

Reflecting on Assignment 1 - Synth, the main learning outcome was learning how to make a shape as a part of a waveform, then repeating this shape infinitely in order to create the wave itself. There were two main discrepancies between the two assignments:

- There was no need for timing waveforms in the first assessment, and;
- There was no need for changing frequencies in the first assessment.

Thinking logically, I felt that this code needed to be built to cater to these two inputs - timing and frequency. My thinking is as follows:

The issue here is that we have multiple frequencies and multiple timings - half seconds and quarter seconds in the first part of the assignment, and full seconds and fractions of a second for this melody specifically. The conclusion I drew from this was that I would need a sort of function or loop which could make these waveforms at **x frequency** for **y amount of time.** Abstracting this thought process down, I felt that the best way to apply this in assembly would be to use a macro which takes in two inputs: frequency and time. As the sawtooth shape had already been made in the first assignment, I simply needed to make a macro that is repeatable

and will play this sawtooth wave repeatedly until the amount of time has elapsed. However, the frequency changes according to the time the wave goes on for, it is, after all, the *frequency of the shape* in **relation** to the *time it is played* for.

Thinking back to the macro, I thought it best to create it first, as debugging may be easier when the code is less congested. Initially, the macro was a label, using r4 and r5 as inputs for frequency and amount of time to enable efficient debugging, as macro breakpoints are not as effective and are more complex than a label. To figure out the values for time, remembering that the headphone jack runs at 48000 Hz per second, we can simply define 48000 as one second in terms of operations. This means that 24000 would be half a second, and 12000 a quarter. As we have the frequencies for our melody already, we now must make a macro which does what we want and can be used repeatedly.

Similar to the first assignment, we needed to figure out the **n** amount of increments required for the dynamic range, to step from one value in the range to the next to create our needed waveform in proportion to the frequency. This is equal to total hertz / frequency. 48000 / frequency. We can store this in a register and use it later to make our shape. As this value is n and is therefore a counter, it is decremented when used as a counter for the loop when making a shape, and so it may also be in good practice to store this in another register so we can reset our n counter register to our n increments once we have made our shape.

The next issue is the length, the length of time we want to play a shape means that a shape must be repeated a certain amount of times in a certain span of time. If a shape has n increments, and we have y amount of time, we can simply do y divided by n (y/n) to get the amount of times we must repeat the shape in our given time. This can also be stored in another register which we can decrement each time we make our shape, then keep making shapes until we have no more left to make.

All that's left now is the loop in the macro. As the shape-making function from assignment 1 uses r0 for the lower bound of the amplitude, and r1 for the amount to add to r0 per increment, we can set r0 in this loop equal to the lower bound of the amplitude and r1 equal to the total amplitude divided by the amount of increments we have from before. At this point, we must make the shape, continuously until our register storing y/n is exhausted to 0, at which point we have played a frequency for y amount of time.

Code Abstraction and Complexity

There was not much code abstraction done besides the macro above, but the reasoning for the macro in the first place was because the code may have been too congested with continuous and long labels for each frequency. The only other abstraction is listed in the main loop where all audio is played, where instead of putting values in registers that the macro uses then playing the macro, the common frequencies and timings are put in labels then call the macro within themselves to play the sound.