Part-2 design document

Overview:

My program has two different parts. The first part of the program, I create a sequencer to play some melodies that can change tempo and pitch continuously over time. Another part, I create percussive sounds to build a drum machine set.

The frequency of drum vibration I set is *100hz*, the vibration frequency of bass I choose is *80hz*, the vibration frequency of snare I choose is *240hz*, the vibration frequency of Toms I set is *210hz*, and the vibration frequency of cymbal I set is *4khz*.

Percussion Instruments (things you hit)	
Instrument	Fundamental
Drums (Timpani)	90Hz - 180Hz
Bass (Kick) Drum	60Hz - 100Hz
Snare Drum	120 Hz - 250 Hz
Toms	60 Hz - 210 Hz
Cymbal - Hi-hat	3 kHz - 5 kHz
Xylophone	700 Hz - 3.5 kHz

```
/*
  r4 is a counter related to r6
  r5 is a counter related to r7
  r6 is a number which stands for how many times to run in the semiperiods
  r7 is a number which controls the tempo
  r8 is a number which controls the frequency of the note fa
  r10 is the label memory address of the instruction
*/
```

Implementation:

Pitch

As it is told to change one of the melodies, it is impossible to change R6 directly so that the pitch of other melodies will be affected, hence another register should be introduced. R8 is set to control the frequency of the note Fa. At first, initialize the r8 value to #40 and R8 increments by one and move r8 to r6, and when it is incremented to value 57, it resets an original initial value, which continually loops.

Tempo

In this part, I was considering moving values to a register but without adding an additional register. Therefore, I, at first, move r7 to r0 and store the r0 register to the stack.

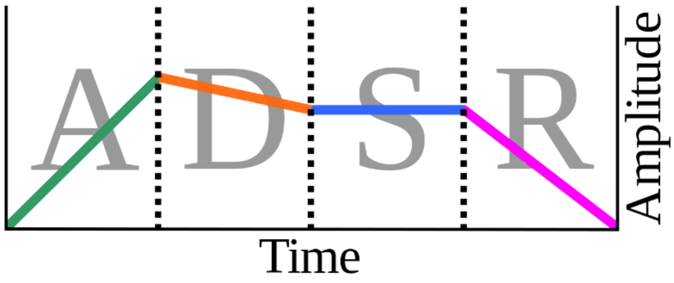
R7 presents how many times of the musical notes play. If r7 is changed, then the tempo of the music is also varied. My intention is to let r0 decrement 1000 by one. When decrementing to #0, r0 is re-given back to the original initial value and store r0 back to the stack memory, and it loops continuously.

Build a drum machine set

I was inspired by the first assignment. The sine waveform in the first assignment can be generated by the hexadecimal value of the sine wave table stored in the SRAM and the hexadecimal value can be loaded to the register r0 every 2 memory cells.

I utilize java programming to calculate the half-word 16-bit data table of the envelope wave, and then load the data into the register every two memory cells.

The data is the output in java programming according to the ADSR principle of the envelope wave.




```java
for(int i=0;i<N;i++) {
        float p = C*i;
        double div = Math.PI/wf;
        int result = (int) (p*Math.sin(i*div));
        String hex = Integer.toHexString(result);
        while(hex.length()<8) {
          hex = "0"+hex;
        }
        System.out.print("0x"+hex.substring(4,8)+",");
}

for(int i=N;i<M;i++) {
        float p = (-C1)*(i-C2);
        double div = Math.PI/wf;
        int result = (int)(p*Math.sin(i*div));
        String hex = Integer.toHexString(result);
        while(hex.length()<8) {
          hex = "0"+hex;
        }
        System.out.print("0x"+hex.substring(4,8)+",");
}

for(int i=M;i<O;i++) {
        float p = C;
        double div = Math.PI/wf;
        int result = (int)(p*Math.sin(i*div));
        String hex = Integer.toHexString(result);
        while(hex.length()<8) {
          hex = "0"+hex;
        }
        System.out.print("0x"+hex.substring(4,8)+",");
}

for(int i=O;i<=P;i++) {
        float p = (-C1)*(i-C2);
        double div = Math.PI/wf;
        int result = (int)(p*Math.sin(i*div));
        String hex = Integer.toHexString(result);
        while(hex.length()<8) {
          hex = "0"+hex;
        }
        System.out.print("0x"+hex.substring(4,8)+",");
}
```

(Java code which generates the sine data table)

① A: the first part is C*x*sin(pi*x/w), w is the semi-periods of sin wave
② D: the second part is (-C1) * (x-C2) * sin(pi*x/w), w is the semi-periods of sin wave
③ S: the third part is C*sin(pi*x/w), w is the semi-periods of sin wave
④ R: the fourth part is (-C1) * (x-C2) * sin(pi*x/w), w is the semi-periods of sin wave

I tried some numbers to let the four parts of the function meet ADSR.

For Timpani:
①C:85 ②C1: 15, C2: 2640 ③C:27024 ④C1: 32, C2: 2640, w:240

For Bass:
①C:70 ②C1: 15, C2: 2700 ③C: 27923 ④C1:15, C2:2700, w:300

For Snare:
①C:210 ②C1:15, C2:2300 ③C: 23255 ④C1:31, C2:2300, w:100

For Toms:
①C:185②C1:15, C2:2394 ③C: 23091 ④C1: 27, C2:2394, w:114

For Cymbal:
①C:100 ②C1: 15, C2: 2190 ③C: 24435 ④C1:63, C2:2190, w:6

Reflection

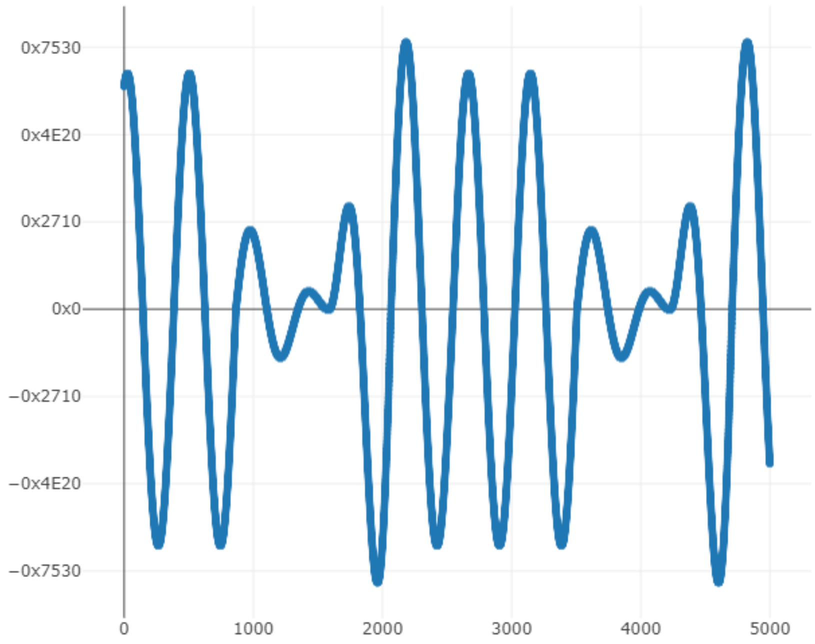
Is there anything you'd do differently if you knew more about assembly language & programming your discoboard?

I was informed that it is possible to use Taylor Series to write a sine function. The first two terms of Taylor Series are x-x^3 /3!. However, the frequency of the sine wave is relatively small, and the amplitude of the sine wave is relatively large ($A * (\frac{x}{w} - \frac{\frac{x}{w}^3}{3!})$ A is the amplitude of sine function and w is the semi-periods of the function). Without using FPU stack, the sine function will be inaccurate because of all float numbers are rounded. If I can know how to use FPU stack, I can probably do sine function by using FPU stack registers instead of storing data to the register.

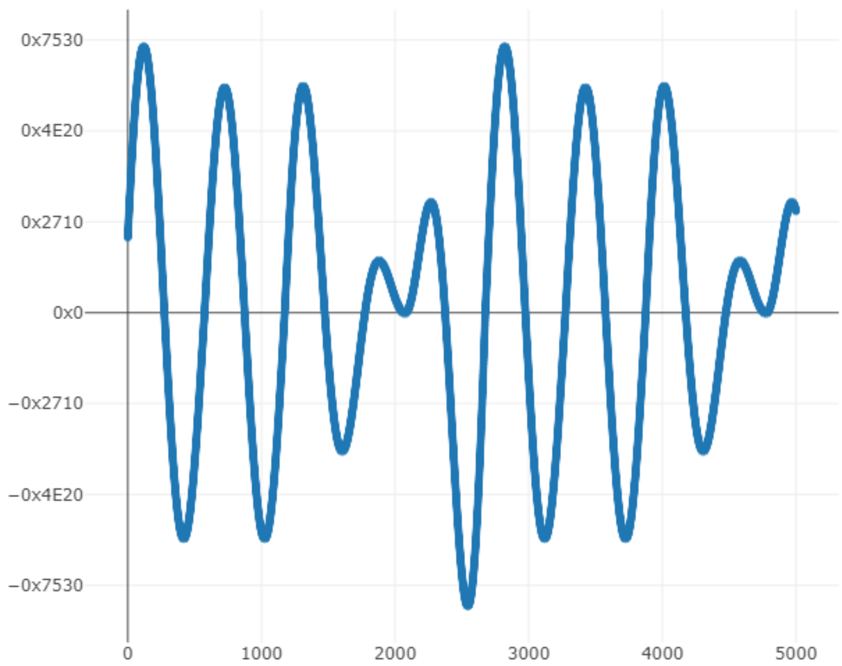
What did you learn?

In this assignment, I know how to generate percussive sounds according to ADSR rules by storing the data to the ram and loading data from memory to the register.

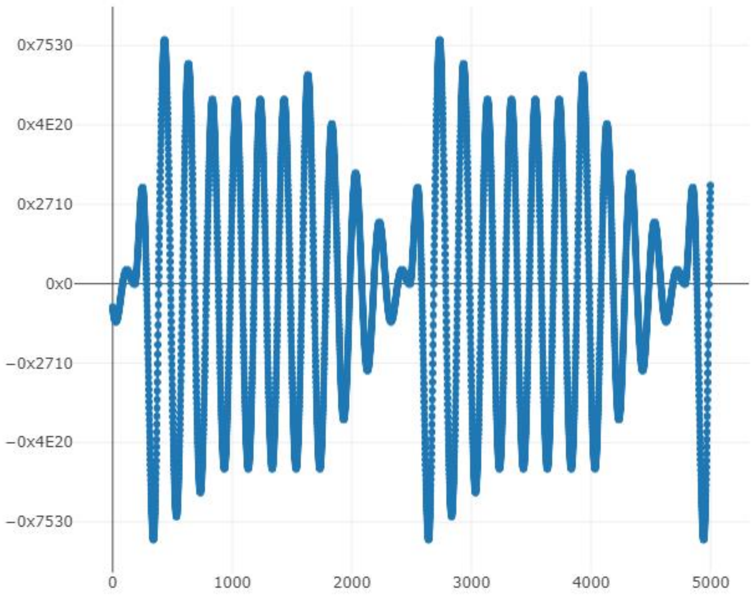
Appendix:



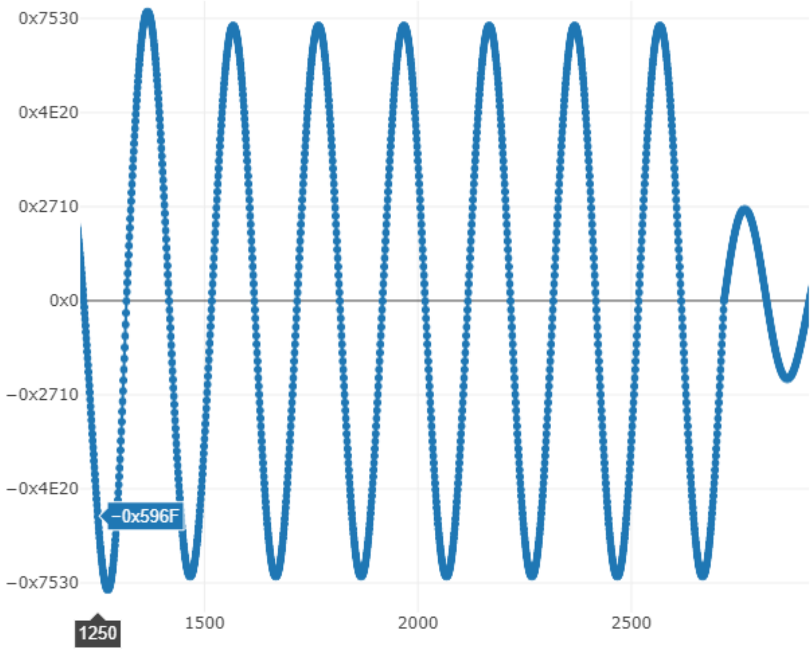
Regular drum waveform



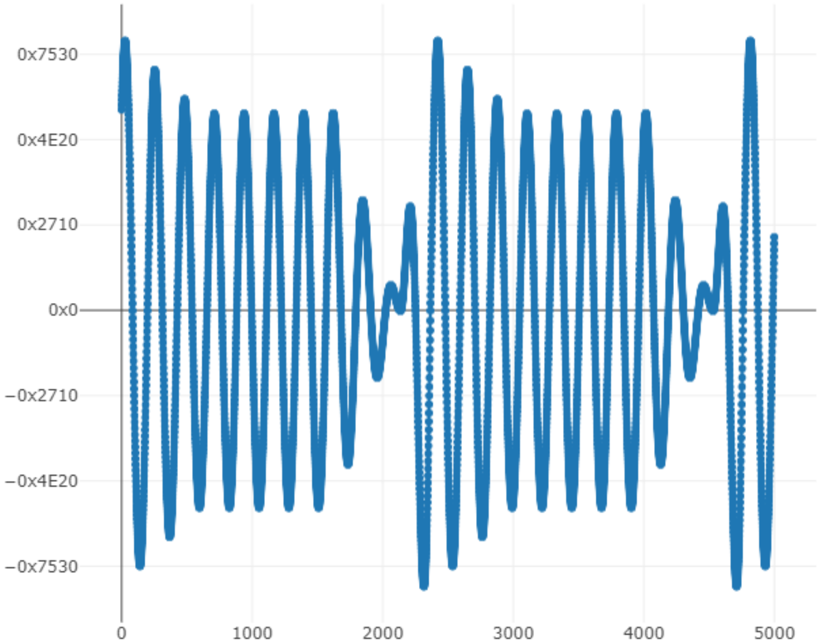
Bass waveform



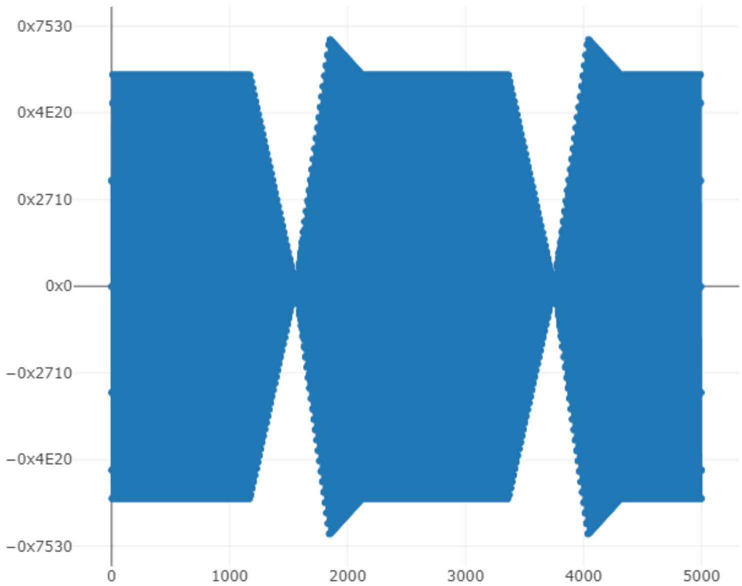
Snare waveform



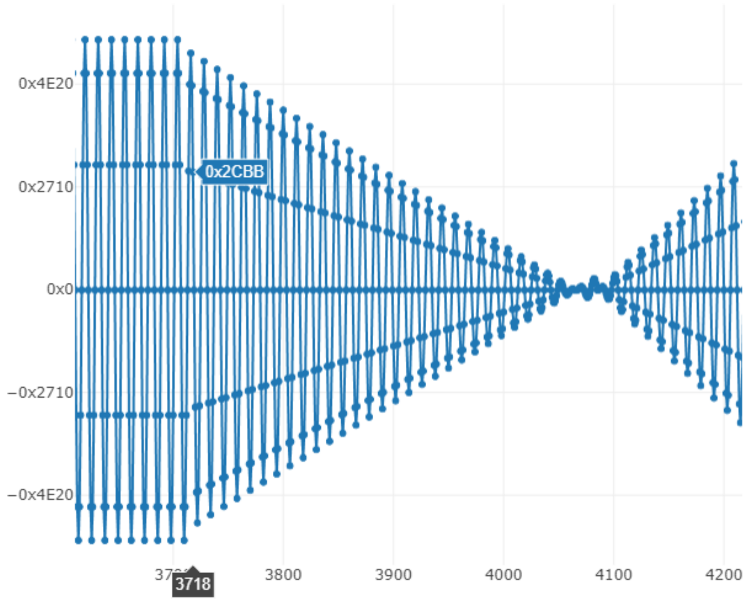
Snare waveform (Zoom)



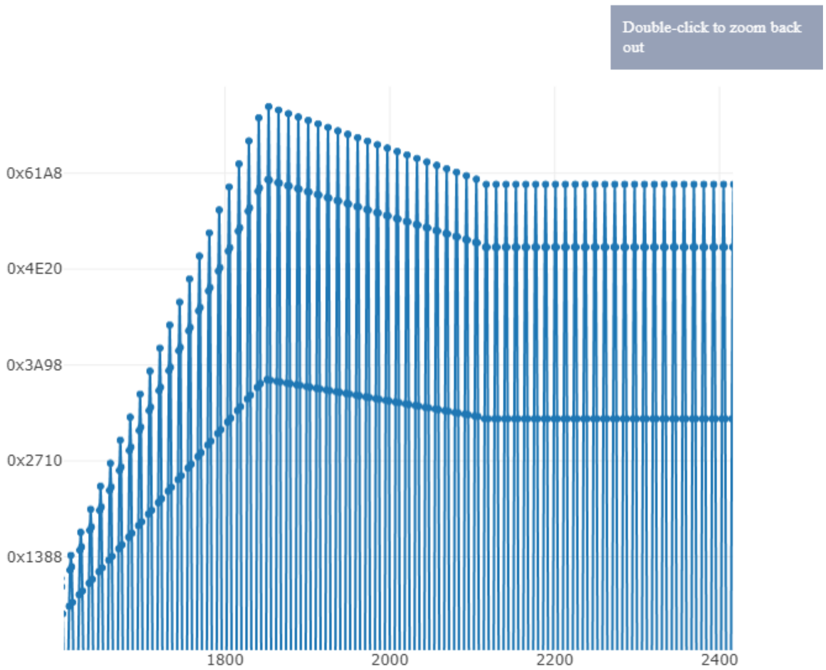
Toms waveform



Cymbal waveform



Cymbal waveform (Zoom)



Cymbal waveform (Zoom)

Xuecheng Zhang
u6284513

Due date:

11:59 PM, Friday 3rd of May 2019 (Friday of week 8)