Kai Hirota

SID: u7233149

Musical Sequencer with MIDI to frequency converter

Design

The program reads musical notes in MIDI format, converts the MIDI byte numbers to frequencies, and generates sounds at appropriate interval to achieve the desired notes and durations. Each MIDI sound consists of three 1 byte components: function, note number, and velocity. For simplicity, the program uses a single channel and constant amplitude (velocity). For our purpose, we will only use the note number, which has the range 0 to 127. To convert MIDI note number, d, to frequency, f

$$f = 2^{(d-69)/12} \times 440 \text{ Hz} \tag{1}$$

The conversion involves calculating $2^{(69-d)/12}$, which is a positive or negative fractional exponent of base 2. The program achieves this by using an identity of the n-th root:

$$\sqrt[n]{a^m} = (a^m)^{1/n} = a^{m/n} = (a^{1/n})^m = (\sqrt[n]{a})^m \tag{2}$$

Therefore, $2^{(d-69)/12} = \sqrt[12]{2^{d-69}} = (\sqrt[12]{2})^{d-69}$. The 12th root of 2, $\sqrt[12]{2}$ is pre-computed and encoded as numerators and denominators in the code. The 12th root of 2 is approximately $1\frac{2973}{50000}$, or 1.05946. Floating point unit was used to store and process floating point numbers.

Implementation

Frequency and duration in seconds are multiplied by 100 to preserve 2 decimal places, so that the generated frequency is close to the target.

Using Floating Point Unit

At startup, the program will initialize and enable the peripherals for generating sound and floating point unit for storage and processing. This enables floating point registers `s0-s31`, which can either store 32 bit or 64 bit floating point numbers. Enabling storage and processing of floating point numbers alone was not adequate to convert MIDI note to frequency, as there are no instructions for performing fractional exponents, at least in ARM architectures.

Converting MIDI byte to frequency

As mentioned in the previous section, the program uses the 12th root of 2 to compute $(\sqrt[12]{2})^{d-69}$, which is the same as $\sqrt[12]{2^{d-69}} = 2^{(d-69)/12}$. The floating point unit was used to store the 12th root of 2, which is approximately $1\frac{2973}{50000}$, and the resulting frequency. The resulting frequency was then multiplied by 100 and converted to integer.

Achieving target frequencies

The program achieves desired frequency by calculating the wavelength using the 48,000 Hz clock rate of the cpu. For example, if we want to play MIDI byte number 62, converted to frequency of 293.66 Hz for 0.15 seconds, that means a sound needs to be generated 293.66 times per second with equal interval. Calculating $48000/293.66$ gives the wavelength. This will be the approximate length, or the gap between each sound, in terms of cpu cycles. The program uses a recursive function to iterate through each note in the music, convert the note from MIDI to frequency, then convert the frequency to wavelength. Finally, the program calls the `play_sound` function to enter a loop, and makes sound at the computed interval to achieve the target frequency for specified duration of time.

Measuring time to achieve desired duration

We know that the cpu on the discoboard processes around 48000 instructions every second. This means that 0.01 second can be measured by entering a loop and iterating 480 times. The desired duration in terms of hundredths of a second, for example 0.15 seconds, can then be expressed as the number of cpu cycles as $15 \times 480 = 7,200$.

Playing each note

Playing musical notes is done by using a recursive `play_note` function which accepts array size and memory address of the first element of the array. `play_note` reads the next note in MIDI byte number and duration (0.15 seconds will be encoded as 15), converts the note to frequency, and calculates mainly three important values which will be passed to `play_sound` function, which are

1. `wavelength`: The number of cpu cycles between each call to `play_sound` function to achieve the target frequency.
2. `on_length`: Wavelength multiplied by the duty cycle. This is for generating a square wave with on and off duty.
3. `cutoff`: The desired duration of the musical note, expressed in terms of number of cpu cycles.

Once the above three are calculated, `play_sound` will be called, whic his also a recursive function that calls itself until the `cutoff` is reached. `play_sound` uses 2 separate counters to keep track of iterations. One o the counters is used to measure the wavelength, and gets reset each time the wavelength is reached. The other counter is for measuring the duration of the note, and when reached will return to the previous function, `play_note`, which will then load and process the next note in the array.

Analysis

Using floating point unit was essential because using integer division, the error would be too large. For example, MIDI note 62 is 293.66Hz. $2^{(-7//12)} \times 440 = 220$, which is far from the target frequency. However, enabling the floating point unit unveiled the next challenge, which was to compute a positive or negative fractional exponent. Using the identity of the n-th root,

$$\sqrt[n]{a^m} = (a^m)^{1/n} = a^{m/n} = (a^{1/n})^m = (\sqrt[n]{a})^m \tag{3}$$

therefore

$$2^{(d-69)/12} = \sqrt[12]{2^{d-69}} = (\sqrt[12]{2})^{d-69} \tag{4}$$

This changes the problem to a problem of solving for integer exponent of a constant. Integer exponent can be easily implemented as a recursive function that tends to be easier to read. One of the key functions is `compute_2_exp_frac`, which is called every time `convert_midi_to_frequency` is executed. It calculates $2^{(d-69/12)}$ as an integer exponent of the 12th root of 2.

The execution flow consists mainly of three layers of loops. First, the program continues to play the music from start after finishing the song. Second, `play_note` is called recursively to process each note of the song, or array. For each note, data is converted from MIDI byte number to frequency, and the calculated frequency and duration is passed as arguments to `play_sound`. `play_sound` then recursively calls itself and turns on the sound on or off to achieve the target frequency and duty cycle. The aim behind this is to make the code easier to understand, read, and manage.