Josh Gianelli, Max Cohn
Prof. Brewer
ECE 153a
13 December 2024

Final Lab: Chromatic Tuner

Introduction

For the final lab, we were tasked with creating a functional chromatic tuner using the Nexys A7 FPGA board. The chromatic tuner must be able to detect input frequencies in the range of 82.41Hz (E2) to 4200Hz (C7) and automatically decipher the closest note. From the closest note, the tuner must also produce the cent error between the read frequency and the expected frequency of the closest note. These values should be displayed nicely on the ILI9314 TFT LCD with a clean UI and UX. To accomplish this, we had to utilize all the tools taught to us throughout the course of the quarter and efficiently implement our design. Using QP-Nano to model our hierarchical finite state machine (HSM), and the kissfft library to compute a fast fourier transform (FFT) with minimal latency, we were able to construct a tuner that is fairly accurate considering the design and time constraints.

Getting Started and FFT Implementation

When beginning our project, we had to find a starting point. Using lab2b and lab3a we repurposed our HSM and FFT implementations respectively, making sure we were able to write to the LCD screen and run a FFT on the gathered samples from the onboard microphone. Once we had a platform to expand upon, our first task was to ensure our FFT was accurate and fast enough to detect notes within the specified frequency range. Initially, we were using our FFT from lab3a, but as we continued to test its latency and accuracy we realized that we could improve our latency by using the popular mixed-radix FFT library kissfft.

After importing kissfft into our project, we explored the different FFT versions available including the fixed point, floating point, and real-valued floating point implementations. The fixed point FFT was blazing fast, but was not very accurate. Floating point FFT yielded more accurate results but was noticeably slower. This latency wasn't going to cut it, so we then tried the real-valued floating point FFT. We decided this FFT algorithm was going to be the backbone of our chromatic tuner. Compared to the complex FFT evaluated with real values the real-valued FFT saves 45% CPU computation time, massively reducing latency[1].

The main concern at this point was accuracy, as to be in spec for the final project we had to ensure we were reading as close to the input frequency as possible (around ±1Hz). We tried many approaches to refine our FFT, using different sampling techniques, decimation values, and zero padding. Our FFT was fast enough to run twice for every note detection sequence, so we tried running a ballpark FFT to determine the general range of the input frequency, and then running another FFT with a predetermined decimation value depending on the initial frequency

reading. Using a 256pt FFT each time, and decimating from the maximum 4096 samples read by the stream grabber, we ended up sticking with the following scheme:

```
void run_fft(void) {
    float sample_f = 48828.125;// 3051.7578125 24414.0625, 48828.125;
    uint16_t sample_size = 256;
    sample_l2 = 8;
    decimation = 2;
    read_fsl_values(q, decimation, sample_size);
    frequency = fft(cfg, sample_size, sample_l2, sample_f / 2);

    if (frequency < 1525) {
        decimation = 16;
        sample_l2 = 8;
        sample_f = sample_f / decimation;
        sample_size = 256;
    } else if (frequency < 3050) {
        decimation = 8;
        sample_l2 = 8;
        sample_f = 6103.515625;
        sample_size = 256;
    } else if (frequency < 6104) {
        decimation = 4;
        sample_l2 = 8;
        sample_f = 12207.03125;
        sample_size = 256;
    } else {
        decimation = 2;
        sample_l2 = 8;
        sample_f = 24414.0625;
        sample_size = 256;
    }
    read_fsl_values(q, decimation, sample_size);
    frequency = fft(cfg, sample_size, sample_l2, sample_f);
}
```

From the output of our FFT, we then processed the closest note, then immediately after wrote all the pertinent information to global buffers accessible by our HSM, and eventually printed to our LCD screen the most up-to-date values.

LCD Screen

The Chromatic Tuner project focused on creating a practical and easy-to-use system to detect and display musical notes in real time. Using an LCD screen for output, we implemented key features like a tuner screen, spectrogram, and histogram to give users clear feedback on

frequency, octave, and tuning accuracy. To manage the different display modes, we implemented a hierarchical state machine (HSM). This ensured smooth transitions and a responsive interface. Careful attention was given to designing the display elements, updating them dynamically, and using color coding to make the interface intuitive and visually appealing.

The tuner screen is designed to display key information such as the current musical note, octave, frequency, and cents error in a clear and readable format. The lcdPrintBig() function is utilized to scale fonts for readability, ensuring that the note and frequency values are prominently displayed. For instance, the note and octave are printed as follows:

```
lcdPrintBig(note_buf, 20, 90);  // Display the note at specified
coordinates
lcdPrintBig(octave, 140, 140);  // Display the octave next to the note
```

Scaling the font made a big difference in our tuner screen's readability. By upscaling the provided SmallFont by a global integer factor named "scale" and incorporating it into the already existing print_char function we could make our note output legible and keep our display write logic minimal.

```
inline void print_big_char(u8 c, int x, int y) {
   u8 ch;
   int i, j, pixelIndex;

   setXY(x, y, x + (scale * cfont.x_size - 1), y + (scale * cfont.y_size - 1));

   pixelIndex = (c - cfont.offset) * (cfont.x_size >> 3) * cfont.y_size + 4;
   for (j = 0; j < (cfont.x_size >> 3) * cfont.y_size; j++) {
       ch = cfont.font[pixelIndex];
       for (i = 0; i < 8; i++) {
           if ((ch & (1 << (7 - i))) != 0) {
               fillRect(x + scale * i, y + scale * j, x + scale * i + 2, y +
scale * j + scale);
           } else {
               fillRectBG(x + scale * i, y + scale * j, x + scale * i + 2, y +
scale * j + scale);
           }
       }
       pixelIndex++;
   }
   clrXY();
}
```

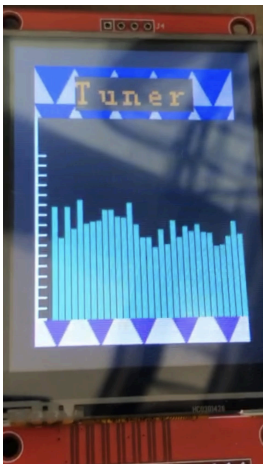
The screen refreshes periodically to ensure that the displayed values stay updated with real-time input. A dynamic bar graph at the bottom visually represents the cents error, with its color and position recalculated based on the error magnitude. Additionally, the background and tuning reference (e.g., A4 value) are updated to reflect any adjustments made by the user through the rotary encoder. By structuring the screen with prominent, large fonts and well-placed visual

indicators, the tuner screen provides users with immediate and accurate feedback. The use of color-coded elements for cents error enhances usability, making it easy to identify whether a note is sharp, flat, or perfectly tuned.

Main Tuner Display Screen:

The histogram provides a graphical representation of frequency bins, updating periodically to reflect the latest FFT results. Bin values are calculated using logarithmic scaling to normalize differences in frequency amplitudes for better visual balance. Dynamic bar heights are drawn with fillRect(), using gradient colors mapped from blue to green to red based on normalized bin values. The periodic refresh of the histogram ensures it remains accurate and visually consistent without disrupting the overall UI flow. This representation allows users to analyze frequency distribution effectively and identify dominant tones or anomalies.

Histogram Display:



The spectrogram offers a visual representation of how frequencies vary over time, using a color-mapped grid that updates dynamically with incoming signal data. Each frequency bin is assigned a color based on its intensity, transitioning smoothly to represent amplitude variations.

The spect() function calculates the bin values, applies logarithmic scaling, and maps them to colors for display:

```
int value = log2(temp[i] + 1) * 10; // Normalize bin values logarithmically
int r = 0, g = 0, b = 0;
if (normalized <= 127) {
    g = normalized * 2;   // Gradual transition from blue to green
    b = 255 - g;
} else {
    r = (normalized - 127) * 2; // Gradual transition from green to red
    g = 255 - r;
}
setColor(r, g, b);  // Set color based on normalized value
```

The spectrogram is refreshed periodically to reflect the most recent frequency data for a seamless visualization of real-time sound patterns. The logarithmic scaling and color mapping make it easy to distinguish variations in intensity, offering both functionality and aesthetic.

Spectrogram Display:

Testing and Debugging

To test our design, we obeyed the sentiment that small changes of our program required extensive testing. We did this by timing our FFT, getting a running output of the current HSM state and event queue, playing note frequencies using a bluetooth speaker to correctly broadcast low octave notes, and utilizing the built-in GDB debugger when values weren't being updated correctly.

Hierarchical State Machine Implementation

For this project, we used a hierarchical state machine (HSM) built with QP Nano to manage the different modes of the Chromatic Tuner. The HSM that we implemented organized the system into a parent state with three child states: tuner screen, histogram, and spectrogram. Each state handled specific parts of the display and was responsible for updating the LCD with the right information based on signals sent from the button interrupts. The parent state acted as a central hub to initialize the system and manage transitions between the different modes.

We defined several signals to handle events and trigger state changes. For example, BTN_SIG was used to detect button inputs and move between states. The DRAW_SCREN_SIG was a periodic signal that updated the LCD every 0.5s, ensuring the display stayed responsive and synchronized with the incoming data. Other signals included compute_fft_sig to start the FFT process and encoder_up_sig/encoder_down_sig to adjust the tuning frequency (A4) when the rotary encoder was used. This signal-based structure made it easy to expand functionality without breaking the existing code.

Each child state handled a specific part of the display. The left state (tuner screen) displayed the note, octave, frequency, cents value, and a bar graph showing the cents error. It used DRAW_SCREN_SIG to redraw the screen periodically and update the bar graph based on real-time data. The right state (histogram) visualized the FFT output as a bar graph of frequency bins, with dynamic updates that reflected changes in frequency intensity. The up state (spectrogram) showed frequency changes over time by mapping frequency intensities to colors on a grid, making it easy to spot patterns or trends.

Transitions between states were managed through signals and button inputs. For example, pressing the left button (btn_sig with left) from the parent state transitioned the system to the tuner screen state. Each state also handled its own entry signals (q_entry_sig) to initialize its display when activated. This structure ensured the system was modular, with each part of the display fully isolated and easy to debug.

The HSM made the system more efficient and easier to manage, especially with the periodic updates and the ability to handle events like button presses and encoder input. By keeping the logic for each state separate and focused on its specific task, the HSM design kept the project organized and allowed us to build a responsive and interactive tuner interface.

While our project turned out great in our eyes, there was still a lot of room for improvement. We should have spent more time making sure our FFT was accurate, incorporating a Hann window or other techniques to reduce spectral leakage. We also wanted our debug screens to update quicker, but couldn’t with the way we implemented our HSM and FFT calculation signal. In totality, we thoroughly enjoyed our time building the chromatic tuner. Not only did we learn a lot, but we also gained valuable experience in developing software for embedded systems.

References:

- [1] https://github.com/mborgerding/kissfft