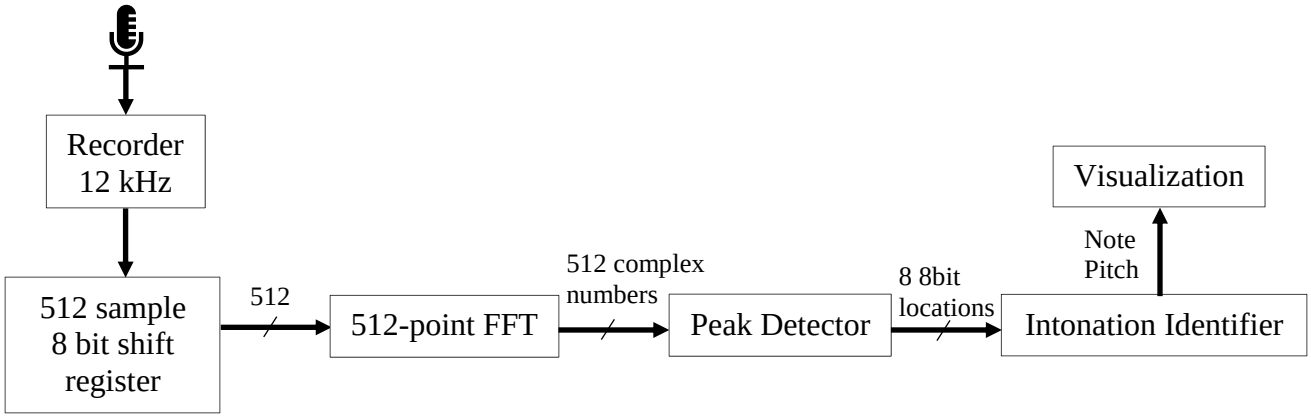
FPGA Tuner

Purpose

A tuner is a device musicians use to tune their musical instruments to an agreed upon reference pitch. In its most basic form, a tuner records short samples of sound, determines what pitch is being played, and displays the name of the note and its intonation (i.e. how sharp or flat the note is). This display is updated continuously in real time, giving the musician live feedback as they make music. Many tuners are, unfortunately, can have delays exceeding half a second which can make adjusting intonation properly difficult. By implementing this process in hardware, note identification process can be done much faster, giving feedback on intonation even during the swiftest musical passages.

High Level Overview



The tuner takes in input from a microphone, and outputs a note name and an indication of the intonation of the note on a monitor. To accomplish this, the recorder module takes an input from a microphone and samples it at 12kHz, and save the last 256 samples in memory. Between each sample, the FFT module computes the Fourier transform of the saved samples. The peak detector module takes the absolute value of the Fourier transform and identifies the positions of the peaks in the frequency spectrum. The intonation identifier uses the peaks to determine the pitch that the tuner is hearing, identifies the note closest to the pitch being played, and determines how many cents sharp or flat the note is. Finally, the visualization module displays the name of the note on a monitor and a sliding bar on a scale to display how many cents sharp or flat the note is.

Microphone

I intend to use the AC97 chip on the lab kit and several of the modules from lab 5 to facilitate communication between the AC97 chip and microphone and the rest of the design. I intend to purchase a better microphone to plug into the microphone jack. I will pay for it because I can use it later.

Recorder

This works similar to the recorder of lab 5. It will downsample the incoming data from the AC97 at 12 kHz (saving one of every four samples taken at 48kHz) at 8-bit resolution. This will

store the last 512 samples in a shift register to input into the FFT module. The range of notes I wish to detect is the range of a piano: from A^0, which has a frequency of 27.5 Hz, to C^8, which has a frequency of 4186 Hz. One full wavelength at 27.5 Hz will cover 436 samples, and a 12kHz sampling rate is over twice the target 4186 Hz. For testing purposes, I will start with a 6kHz sampling rate saving 256 samples. This will reduce my resolution in the higher registers, but will be significantly faster to build, test, and debug. Once I am confident the system as a whole is working, I can scale up the size of the memory and the sampling rate to improve accuracy and range of frequency operation.

FFT

To determine the frequency content of the recording, I will take the Fourier transform of the samples saved in memory. At a 12kHz sampling rate and a 27MHz clock, I will have 2250 clock cycles to perform the remaining computations. The FFT module will likely take the most clock cycles out of any of the modules. I intend to make a recursive, parameterizable FFT module that computes the radix 2 DIT FFT. Each stage will be pipelined with complex multiplications. The basic form will compute all points in parallel. If this can be successfully recursively designed, I can test the FFT at small sizes and then arbitrarily scale up the size of my FFT by powers of 2. If hardware usage is an issue, or if I have extra clock cycles and time to implement it, I can attempt to serialize the computation between its even and odd components, doubling the amount of clock cycles used and halving the hardware used.

This module takes in 512 complex numbers that are broken into two parts: an 8-bit real part and an 8-bit odd part. For the fully pipelined version, there are 9 layers of FFT modules, each of which has 2 pipelined stages, which means this module has a grand total of 20480 8-bit registers and the FFT will take 27 clock cycles to compute.

Peak Detector

This module first takes the magnitude of the complex output of the FFT module. Since only the location of the peaks matter, and not the correctness of magnitude, I can take the sum of the squares of the real and imaginary components and ignore the square root. While mathematically incorrect, this method will still produce prominent peaks the rest of the module can use.

For peak detection, there a few different possible methods. The simplest I will implement first is to search sequentially from the lowest pitch to the highest pitch until it finds a peak that exceeds a threshold of intensity and is within the bounds of thinness (to be determined experimentally). In theory, such a peak will be the fundamental pitch of the note played, and thus the pitch can be determined by the lowest peak. Alternatively, the peak detector can find multiple peaks and average the differences between adjacent peaks to find the pitch. If only one note is being played, peaks should be spaced regularly at multiples of the frequency of the fundamental pitch. If time permits, aspects of this second method can be expanded to detect multiple pitches simultaneously.

Taking the magnitude will take two pipelined stages. It takes 512 complex numbers as inputs and the two stages store 512 16-bit real numbers each. Afterwards, this stage will take a maximum of 512 clock cycles to identify a peak, searching from 0 frequency.

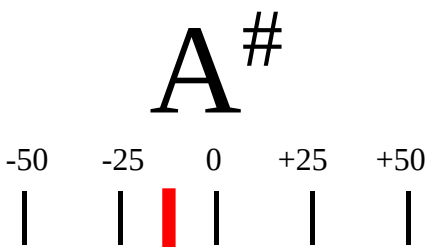
Intonation Identifier

This module computes the following operation:

$$cents\ above\ a\ reference\ pitch = 1200 \cdot \log_2 \frac{identified\ pitch\ (in\ Hz)}{reference\ pitch\ (in\ Hz)}$$

This computation divides the octave into 1200 “cents”, which are logarithmic units of pitch similar in concept to decibels. The frequency of a pitch an octave higher than another is double the frequency of the lower note. That factor of two is divided into 1200 logarithmic cents, so each half step is 100 cents wide.
To make this computation easier, I will use the reference pitch of 1 Hz, rendering the division moot. I will use a look-up table of pitches in terms of cents above 1 Hz to identify the pitch and how many cents above or below the recorded pitch is. To find it, I’ll either use division, binary search, or some other method based upon how nice the cent numbers end up being.
Alternatively, I can devise a look-up table of the bounds of each note. Once the note is identified, the difference between the target frequency and the recorded frequency is computed, and a linear approximation of the logarithmic function can be used. This removes the need for a logarithm module, and instead uses multiplication with Taylor series.
This module is the most open ended in its implementation. I believe will be the most challenging to implement and debug, only potentially superseded by the FFT module if recursive definition is not supported by this version of Verilog. This will be the module I budget the most time for.

Visualization



The visuals will interface with a monitor using VGA to display the identified note name (images of characters will be saved as ROMs) and a sliding bar (the red box pictured above) that will indicate how flat or sharp the note is in cents (+ is sharp, - is flat). The information from the Intonation Identifier will tell this module which note to display and the position of the bar. If no note was detected (i.e. no peak exceeded the threshold in the Peak Detector), the bar and note name will not be displayed. If the bar proves too jittery, I can make the position of the bar a time domain average of the last 10 or so positions the Intonation Identifier outputs. Additionally, the system as a whole can potentially be expanded to identify and tune multiple pitches at once, which requires duplication of the display in different corners of the screen.
This will store images of the letters A, B, C, D, E, F, G, with the icon for #, and the numbers -50, -25, 0, +25, and +50.