The DiGuitar

Pronounced "Digi-Tar"
6.111 Final Project Report
Eric Pence & Ishaan Govindarajan

Project Overview

The DiGuitar receives an input from an electric guitar and digitize it to a Musical Instrument Digital Interface (MIDI) compliant serial datastream in real time. To do this, the DiGuitar takes advantage of the Artix-7's onboard ADC to sample the incoming audio waveform. Further digital signal processing (DSP) techniques, namely digital Finite Impulse Response (FIR) filtering and Autocorrelation are used to detect the frequency composition of the guitar input waveform. Further processing of the spectral content takes place to detect the particular notes being played. Note information is transmitted in a MIDI compliant format to be received by other devices for recording, playback, or manipulation.

Constraints and Requirements Exploration

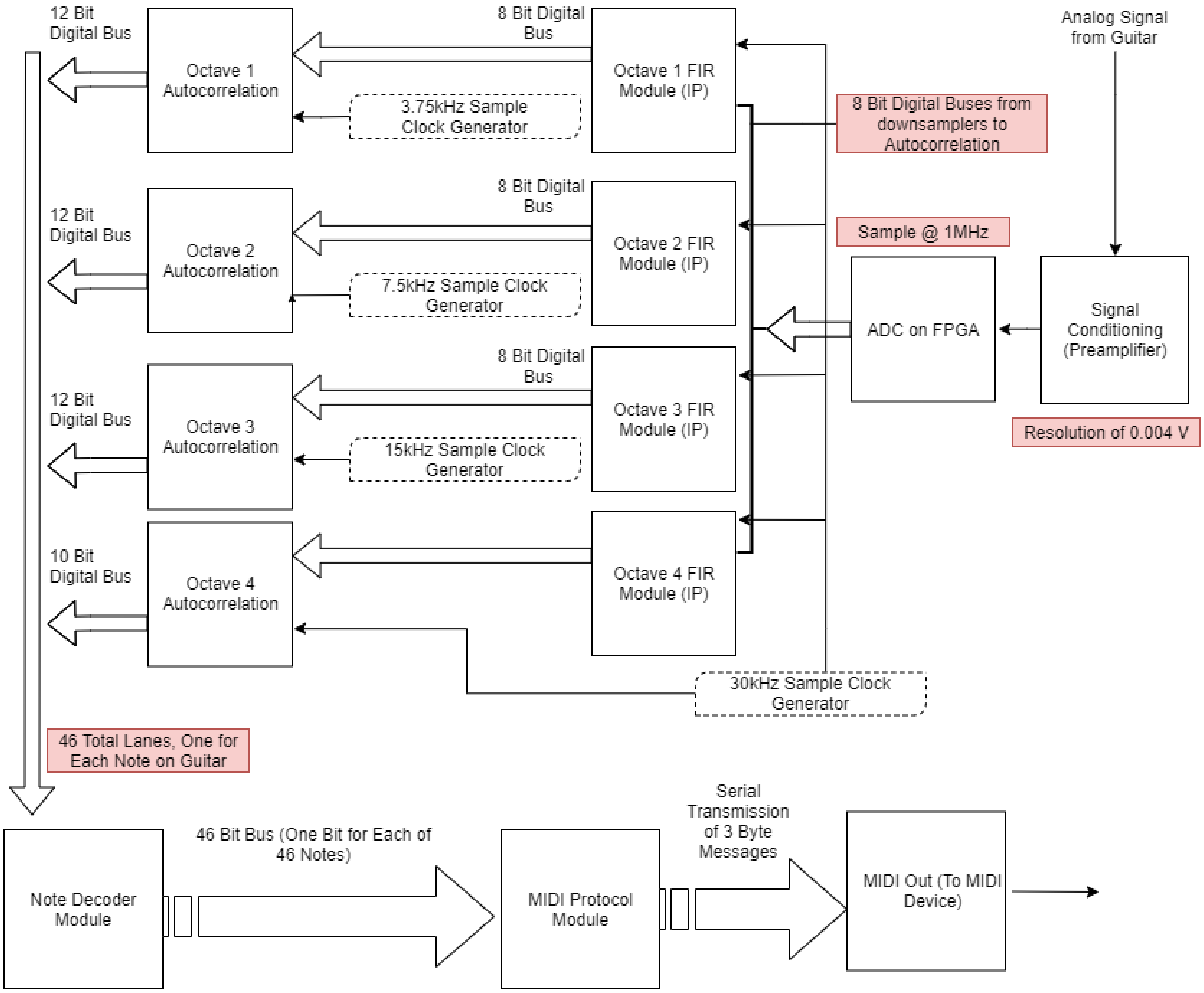
The key driving requirements of this project is that it must take a guitar audio input (in standard tuning) and be able to decode in real time. Since these requirements drive the design the rest of the digital system, it is important to specifically quantify what these requirements mean.

Requirements driven by Guitar Audio Input
Using a guitar as an input source primarily dictates the dynamic range of frequencies the DiGuitar must be able to process. The lowest note that a guitar can produce is an E2[1] (low E-string, MIDI code 40, 82.41 Hz[2]). The second lowest note that it can produce is an F2 (low E-string, MIDI code 41, 87.31 Hz[2]). These low notes in conjunction to our latency requirement drive the DSP strategy used to differentiate between these low notes. This will be further discussed later in the report.
The highest note that a guitar (Bullet Stratocaster) can play is a C$^\#$6 (high E-string fret closest to guitar body, MIDI code 85, 1108.7 Hz[2]). This dictates our Nyquist frequency, i.e. the **sampling frequency for the system must be ~2250Hz or higher.** Since high frequency harmonics are present in the audio input and can alias at low sampling frequencies, we chose to sample at 30kHz and downsample after using a low-pass FIR. This enables us to diminish the effects of aliasing before analyzing the frequency content of each octave of the guitar.

Requirements driven by Real-time Note Computation
One of the key motivations for this project was to design something that would be difficult to implement in a traditional programming language e.g. Python. One of the strengths of FPGAs is their parallel computing and real-time signal processing capability. Thus, the DiGuitar should ideally be a zero-latency system.

Zero-latency is effectively impossible, so a more realistic objective is to design a system with *imperceptible* latency. Different sources provide different latency targets for digital audio systems. For example, this source[4] recommends a <12ms latency for guitarists, this source[5] suggests that musicians can recognize latencies of 20ms, and a Wikipedia article about audio-video synchronization[6] notes that up to 45ms of latency is acceptable. We established a target latency of 75ms for our project. This latency target directly affected our selection of note detection strategies. Ultimately, we were able to detect the lowest frequency notes (E2 and F2) within ~70ms of note onset. This detection latency improved as frequencies increased, with C#6 having a detection latency of ~27ms, most of that being debouncing.

System Architecture/Block Diagram

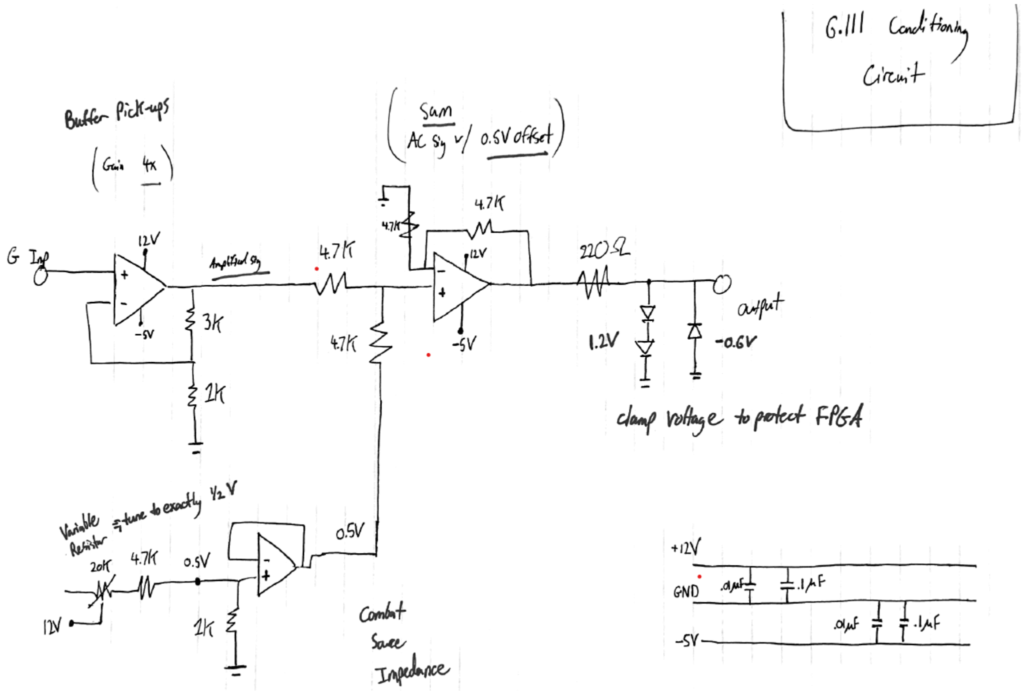


Module Descriptions

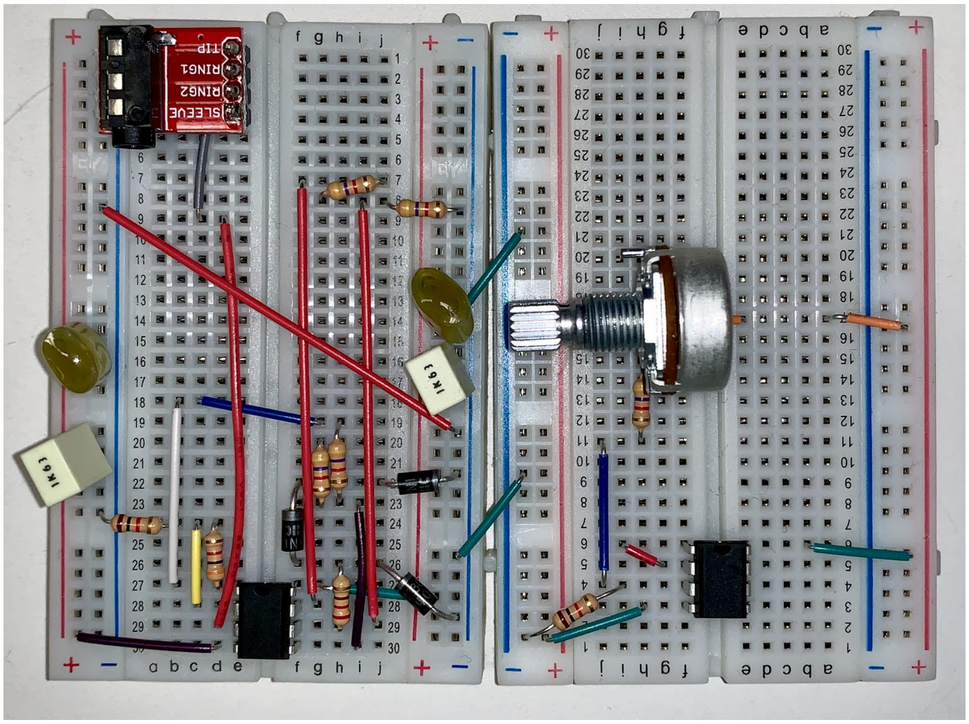
Preamplifier (EXTERNAL) [Eric]

Buffers and amplifies the guitar voltage output to something that can be read by the FPGA ADC (0-1V range). Amplification ratio was determined after measuring guitar pickup output voltage at +/- 100mV, and is set to 4x. Gain and offset requirements of this application are easily met by Op-amps available in EDS or lab. We selected the LM358 as our op-amp.

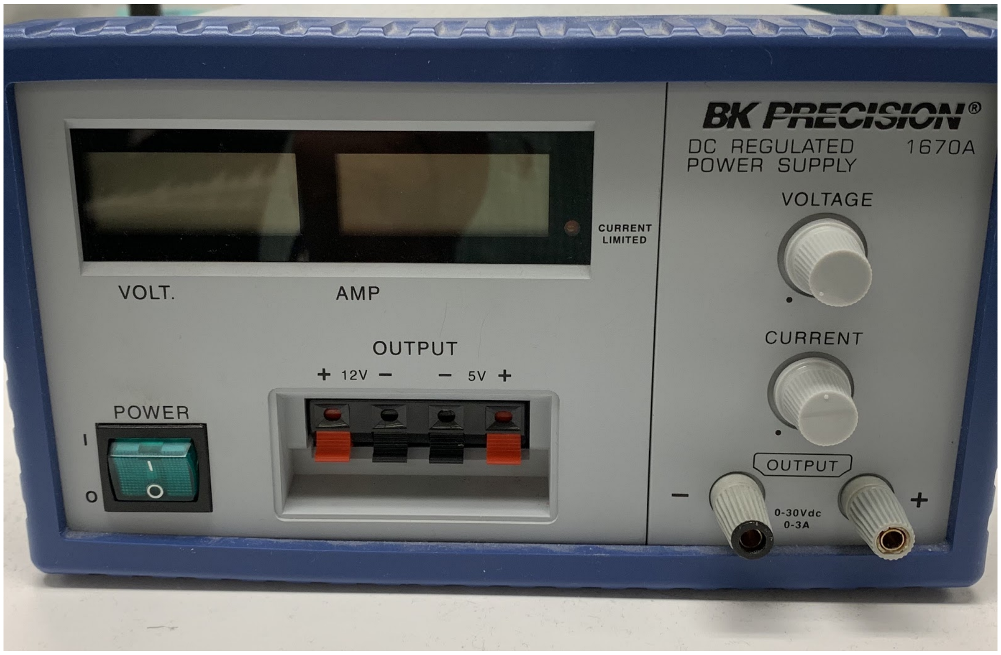
In the circuit below, we buffer the output of the guitar and amplify it by a factor of 4 using an LM358. We then bias the guitar signal by 0.5V, using a summing amplifier to add the AC signal with a 0.5V DC offset generated by a voltage divider. We chose to power our op-amps with +12V and -5V rails because the power supply at our bench had isolated +12V and +5V outputs. We put these outputs in series and were able to easily generate +12V and -5V supplies. Using 3 diodes, each with 0.6V forward voltages, we clamp the output of our summing amplifier between -0.6 and +1.2V in order to protect the ADC on the FPGA.



Circuit Diagram



Breadboard Circuit (with wires between power supplies removed for clarity)



Power supply used to create +12V, -5V

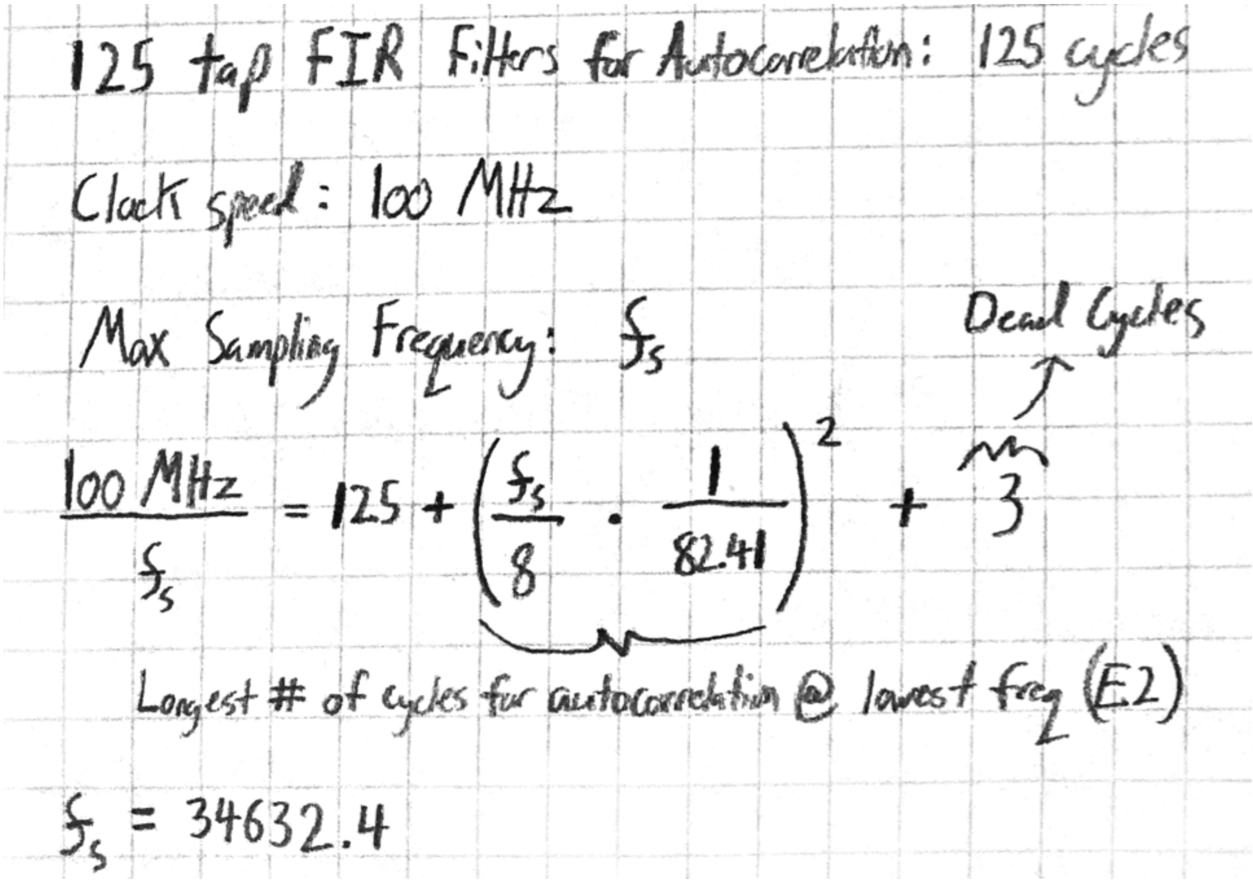
ADC

Xilinx IP block to digitize the input audio waveform. We only utilize the top 8 bits, and we flip the MSB. This has the effect of making the ADC signal signed two's-complement 8-bit, with the 0 value centered around 0.5V.

Sample Clock Generator [Eric]

A module that generates a sample clock signal for each of the octave decoders. We use this to generate four separate triggers that pulse high when the corresponding octave decoder should read in an 8 bit sample from the ADC. We down sample to decrease the data throughput to the rest of the note decoder system, eliminating unnecessary multiply/add cycles.

Since we fixed the size of the FIRs in the autocorrelation module at 125 taps and our system clock is set at 100MHz, we have an upper bound on our sampling frequency as shown by the relation below. If we violate this upper bound, autocorrelation will not complete before we receive the next audio sample.



125 tap FIR Filters for Autocorrelation: 125 cycles

Clock speed: 100 MHz

Max Sampling Frequency: f_s

$$\frac{100\text{ MHz}}{f_s} = 125 + \underbrace{\left(\frac{f_s}{8} \cdot \frac{1}{82.41}\right)^2}_{\text{Longest \# of cycles for autocorrelation @ lowest freq (E2)}} + \overbrace{3}^{\text{Dead Cycles}}$$

$f_s = 34632.4$

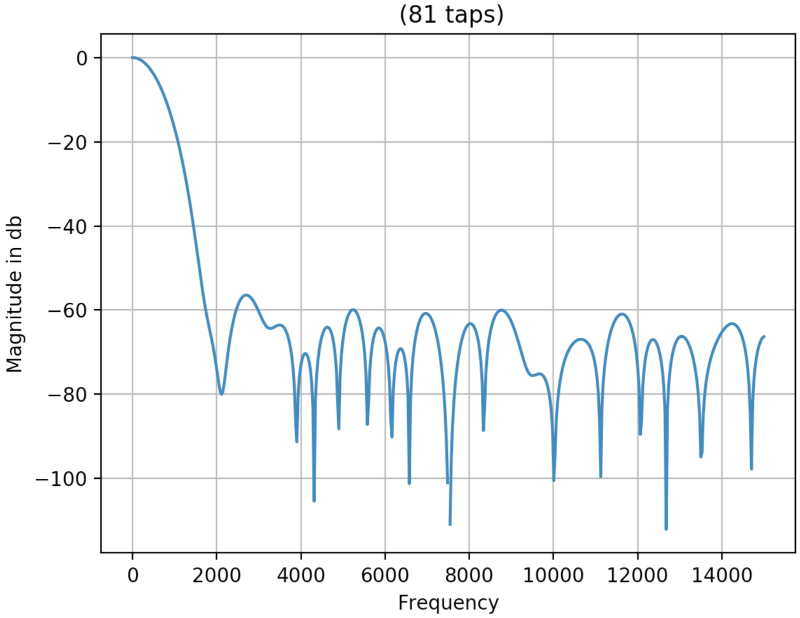
As shown above, the upper bound on our sampling frequency is 34632 kHz. To meet this constraint, we used this module to generate 30kHz, 15kHz, 7.5kHz, and 3.75kHz sampling frequencies. We selected our highest frequency such that it would fall just below the upper bound, maximizing resolution while meeting timing requirements. We selected each sampling frequency to be twice that of its neighboring frequency so that we could reuse 12 FIR modules (one for each note in an octave) for all 4 octaves. This is explained further in the discussion of the Octave Frequency Decoder Block below.

As can be seen in the Verilog at the end of this report, the sample clock module is parameterized by a maximum 16 bit count value. This module is fed the 100MHz system clock and it counts clock cycles from zero up to the specified 16 bit count value. Once the specified count is reached, the trigger is pulsed high and the count resets. To generate these frequencies with a 100MHz clock, we must count 3333, 6666, 13332, and 26664 clock cycles, respectively, between each pulse.

Digital LP Filters [Eric]

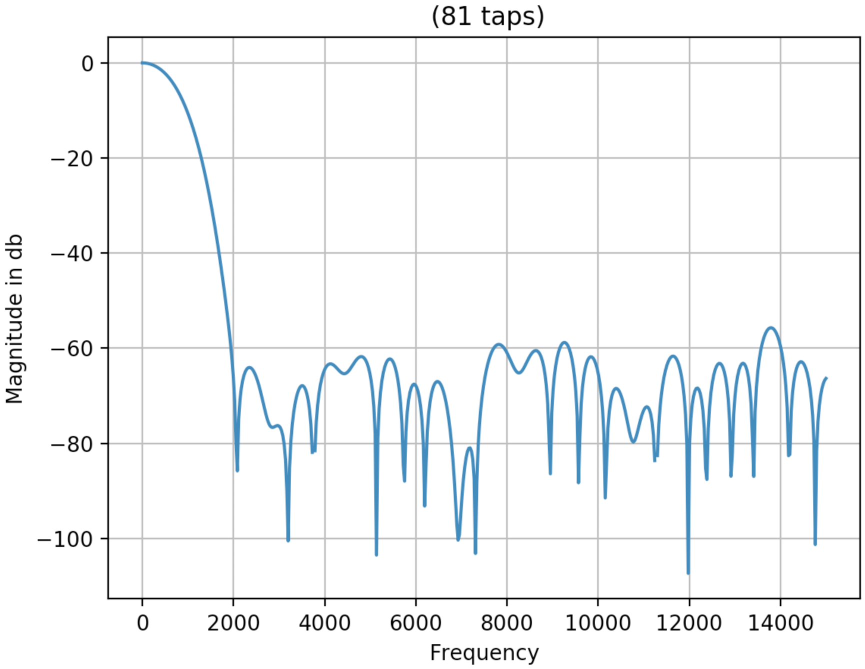
IP modules that digitally filter (FIR) high frequency harmonics of the input audio. There is one FIR for each of the four octaves on the guitar. Each FIR is designed to pass frequencies in a particular octave and stop frequencies at and above the nyquist of its corresponding sample clock. Output is 9 bits of signed data. The design of these filters is discussed in depth in the Design Process section of this report.

There is one FIR filter for each of the four octaves on the guitar. Each FIR filter is composed of 81 taps. Using the SciPy library to generate prototype filters, we found that 81 taps provide a sharp enough cutoff for the purpose of removing nyquist frequencies and above while retaining the frequencies of interest in a given octave. 81 tap filters also ensure a sub-3ms latency, which is necessary given our 75ms target latency and the time consuming nature of autocorrelation. For the lowest Octave 1 (E2-D#3), the cutoff frequency is set at 500Hz. This guarantees that D#3 is attenuated to no less than -.8dB and the nyquist frequency of the corresponding 3.75kHz sample clock falls inside the stop band.



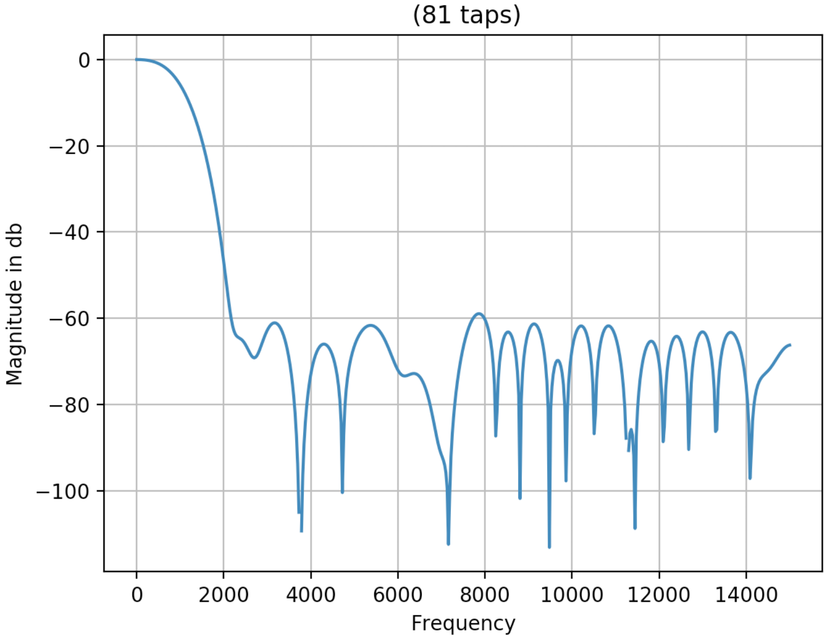
Octave 1 FIR

The cutoff of Octave 2 (E3-D#4) is set at 750Hz, ensuring D#4 will pass with minimal attenuation and nyquist of the 7.5kHz sample clock will fall inside the stop band.



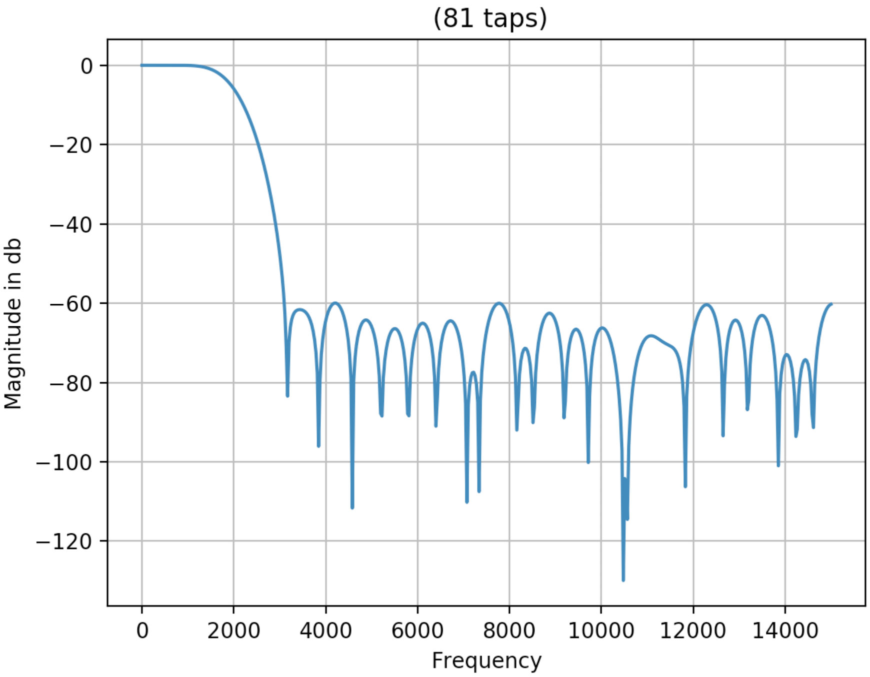
Octave 2 FIR Filter

The cutoff of Octave 3 (E4-D#5) is set at 1kHz, such that a 622Hz D#5 will pass with minimal attenuation but the nyquist of the 15kHz sample clock will be cutout.



Octave 3 FIR Filter

The cutoff of Octave 4 (E5-C#6) is set at 2kHz, such that a C#6 will pass with minimal attenuation but nyquist frequency and above of the 30kHz sample clock will be removed.



Octave 4 FIR Filter

Given the constraints imposed by the frequency bands each filter had to pass and remove, these filters were developed through an iterative process of guess-and-check. We made use of the SciPy library and NumPy to generate and visualize the frequency response of each filter. Here is the code used:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import cos, sin, pi, absolute, arange
from scipy import signal
from scipy.signal import kaiserord, lfilter, firwin, freqz
from pylab import figure, clf, plot, xlabel, ylabel, xlim, ylim, title, grid, axes, show


#Input Sampling Frequency:
fs = 30000.0 #30kHz
nyq_fs = fs/2.0

#FIR Filtering by Octave:
#Octave 1: We sample at 3.75kz and cutoff out everything above 500Hz
#Octave 2: cutoff = 750Hz
#Octave 3: cutoff = 1kHz
#Octave 4: cutoff = 2kHz

#Variable to be adjusted for each FIR Filter
filter_cutoff = 500 #VARIABLE

#Specify number of Delay and number of taps:
delay = .002 # 2ms
```

```
N = 81 #81 taps (2ms delay)

# Use firwin with a BlackmanHarris window to create a lowpass FIR filter.
taps = firwin(N, filter_cutoff/nyq_fs, window = ('blackmanharris')) * (2**12 -1)
taps = np.round(taps)/(2**12 -1) #convert to fixed point

# Use lfilter to filter x with the FIR filter.
filtered_x = lfilter(taps, 1.0, x)

#------------------------------------------------
# Plot the FIR filter coefficients.
#------------------------------------------------

figure(1)
title(' (%d taps)' % N)
grid(True)

#------------------------------------------------
# Plot the magnitude response of the filter.
#------------------------------------------------

w,h = signal.freqz(taps,1)
h_db = 20*np.log10(np.abs(h))
plt.plot(w/max(w)*fs/2,h_db)
plt.ylabel("Magnitude in db")
plt.xlabel(r'Frequency')
plt.show()

show()
```

The 81 taps for each of the FIRs are as follows.

Octave 1 FIR:

```
 [-0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0002442002442002442, -0.0002442002442002442, -0.0002442002442002442,
-0.0004884004884004884, -0.0004884004884004884, -0.0007326007326007326, -0.0007326007326007326,
-0.0009768009768009768, -0.0007326007326007326, -0.0007326007326007326, -0.0004884004884004884, 0.0,
0.0007326007326007326, 0.0017094017094017094, 0.0031746031746031746, 0.004884004884004884, 0.006837606837606838,
0.009523809523809525, 0.012454212454212455, 0.015873015873015872, 0.019536019536019536, 0.023687423687423687,
0.02783882783882784, 0.03199023199023199, 0.036141636141636145, 0.040293040293040296, 0.04395604395604396,
0.04713064713064713, 0.04981684981684982, 0.051770451770451774, 0.05299145299145299, 0.053235653235653234,
0.05299145299145299, 0.051770451770451774, 0.04981684981684982, 0.04713064713064713, 0.04395604395604396,
0.040293040293040296, 0.036141636141636145, 0.03199023199023199, 0.02783882783882784, 0.023687423687423687,
0.019536019536019536, 0.015873015873015872, 0.012454212454212455, 0.009523809523809525, 0.006837606837606838,
0.004884004884004884, 0.0031746031746031746, 0.0017094017094017094, 0.0007326007326007326, 0.0,
-0.0004884004884004884, -0.0007326007326007326, -0.0007326007326007326, -0.0009768009768009768,
-0.0007326007326007326, -0.0007326007326007326, -0.0004884004884004884, -0.0004884004884004884,
-0.0002442002442002442, -0.0002442002442002442, -0.0002442002442002442, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0]
```

Octave 2 FIR:

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0002442002442002442, 0.0002442002442002442, 0.0002442002442002442,
0.0002442002442002442, 0.0, -0.0002442002442002442, -0.0004884004884004884, -0.0007326007326007326,
-0.001221001221001221, -0.0019536019536019536, -0.002686202686202686, -0.003418803418803419,
-0.004151404151404151, -0.00463980463980464, -0.004884004884004884, -0.00463980463980464,
-0.003663003663003663, -0.0017094017094017094, 0.0009768009768009768, 0.004884004884004884,
0.009768009768009768, 0.01562881562881563, 0.022466422466422466, 0.030036630036630037,
0.0380952380952381, 0.046153846153846156, 0.05396825396825397, 0.06105006105006105, 0.06691086691086691,
0.0713064713064713, 0.07423687423687424, 0.07521367521367521, 0.07423687423687424, 0.0713064713064713,
```

0.06691086691086691, 0.06105006105006105, 0.05396825396825397, 0.046153846153846156, 0.0380952380952381,
0.030036630036630037, 0.022466422466422466, 0.01562881562881563, 0.009768009768009768,
0.004884004884004884, 0.0009768009768009768, -0.0017094017094017094, -0.003663003663003663,
-0.00463980463980464, -0.004884004884004884, -0.00463980463980464, -0.004151404151404151,
-0.003418803418803419, -0.002686202686202686, -0.0019536019536019536, -0.001221001221001221,
-0.0007326007326007326, -0.0004884004884004884, -0.0002442002442002442, 0.0, 0.0002442002442002442,
0.0002442002442002442, 0.0002442002442002442, 0.0002442002442002442, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Octave 3 FIR:

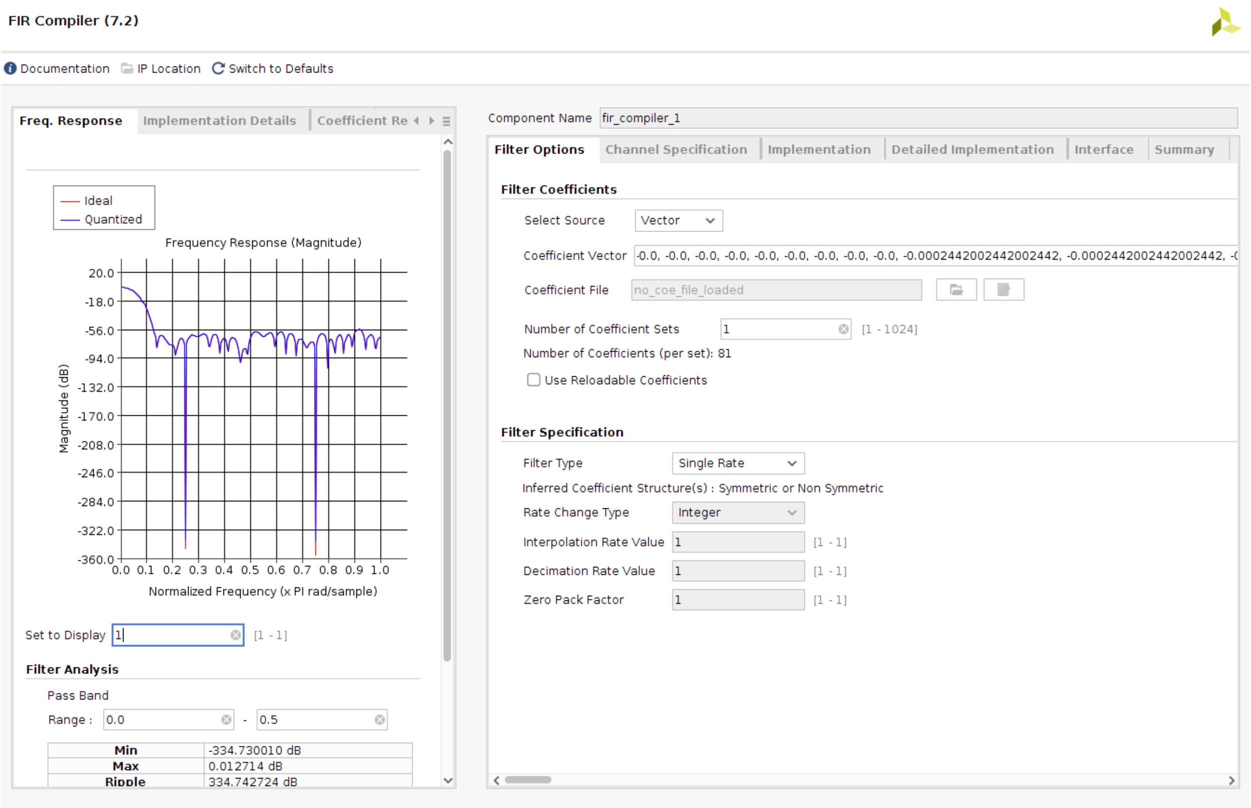
[-0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, 0.0, 0.0, 0.0002442002442002442, 0.0004884004884004884,
0.0007326007326007326, 0.0009768009768009768, 0.001221001221001221, 0.0014652014652014652,
0.001221001221001221, 0.0009768009768009768, -0.0, -0.001221001221001221, -0.0031746031746031746,
-0.005128205128205128, -0.007326007326007326, -0.00927960927960928, -0.010500610500610501,
-0.010744810744810745, -0.00927960927960928, -0.005860805860805861, 0.0, 0.00805860805860806,
0.01855921855921856, 0.03076923076923077, 0.044444444444444446, 0.05811965811965812, 0.07155067155067155,
0.08302808302808302, 0.09230769230769231, 0.09792429792429792, 0.10012210012210013, 0.09792429792429792,
0.09230769230769231, 0.08302808302808302, 0.07155067155067155, 0.05811965811965812, 0.044444444444444446,
0.03076923076923077, 0.01855921855921856, 0.00805860805860806, 0.0, -0.005860805860805861,
-0.00927960927960928, -0.010744810744810745, -0.010500610500610501, -0.00927960927960928,
-0.007326007326007326, -0.005128205128205128, -0.0031746031746031746, -0.001221001221001221, -0.0,
0.0009768009768009768, 0.001221001221001221, 0.0014652014652014652, 0.001221001221001221,
0.0009768009768009768, 0.0007326007326007326, 0.0004884004884004884, 0.0002442002442002442, 0.0, 0.0, -0.0,
-0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0]

Octave 4 FIR:

[-0.0, -0.0, -0.0, -0.0, -0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.0, -0.0002442002442002442, -0.0004884004884004884,
-0.0004884004884004884, -0.0004884004884004884, 0.0, 0.0007326007326007326, 0.0017094017094017094,
0.002197802197802198, 0.0017094017094017094, -0.0, -0.002442002442002442, -0.004884004884004884,
-0.006105006105006105, -0.004395604395604396, 0.0, 0.006593406593406593, 0.012454212454212455,
0.014896214896214897, 0.01098901098901099, -0.0, -0.01562881562881563, -0.030036630036630037,
-0.036141636141636145, -0.02735042735042735, 0.0, 0.0442002442002442, 0.09768009768009768,
0.1492063492063492, 0.18632478632478633, 0.2, 0.18632478632478633, 0.1492063492063492, 0.09768009768009768,
0.0442002442002442, 0.0, -0.02735042735042735, -0.036141636141636145, -0.030036630036630037,
-0.01562881562881563, -0.0, 0.01098901098901099, 0.014896214896214897, 0.012454212454212455,
0.006593406593406593, 0.0, -0.004395604395604396, -0.006105006105006105, -0.004884004884004884,
-0.002442002442002442, -0.0, 0.0017094017094017094, 0.002197802197802198, 0.0017094017094017094,
0.0007326007326007326, 0.0, -0.0004884004884004884, -0.0004884004884004884, -0.0004884004884004884,
-0.0002442002442002442, -0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.0, -0.0, -0.0, -0.0, -0.0]

When generating the FIR IP, we expressed our taps as 12 bit fixed point numbers because this provides us with enough tap resolution to minimize stop band ripple while still having relatively small sized coefficients.

We used the following specs when generating IP for all FIRs used throughout the project:



FIR Compiler (7.2)

Documentation IP Location Switch to Defaults

Freq. Response | Implementation Details | Coefficient Re

Set to Display 1 [1 - 1]

Filter Analysis

Pass Band

Range : 0.0 - 0.5

Min	-334.730010 dB
Max	0.012714 dB
Ripple	334.742724 dB

Component Name fir_compiler_1

Filter Options | Channel Specification | Implementation | Detailed Implementation | Interface | Summary

Filter Coefficients

Select Source Vector

Coefficient Vector -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0, -0.0002442002442002442, -0.0002442002442002442, -(

Coefficient File no_coe_file_loaded

Number of Coefficient Sets 1 [1 - 1024]

Number of Coefficients (per set): 81

□ Use Reloadable Coefficients

Filter Specification

Filter Type Single Rate

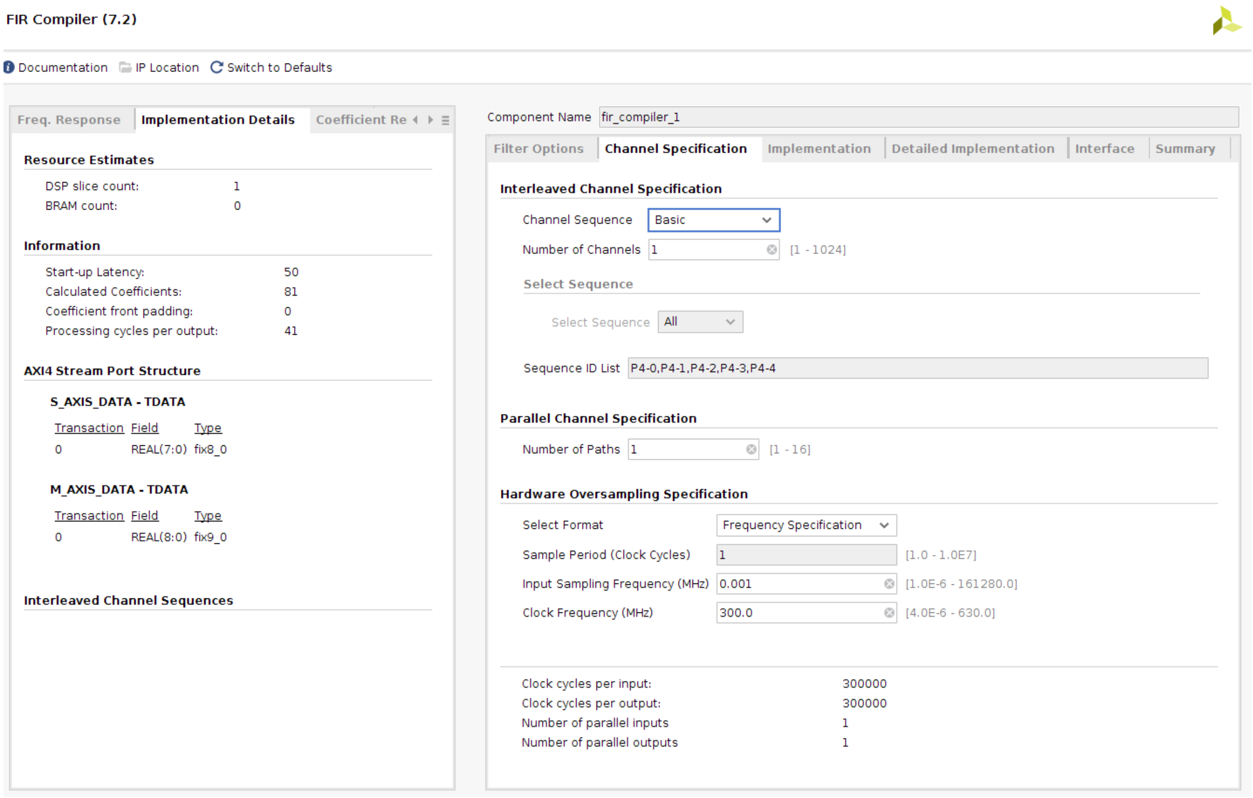
Inferred Coefficient Structure(s) : Symmetric or Non Symmetric

Rate Change Type Integer

Interpolation Rate Value 1 [1 - 1]

Decimation Rate Value 1 [1 - 1]

Zero Pack Factor 1 [1 - 1]



FIR Compiler (7.2)

Documentation IP Location Switch to Defaults

Freq. Response | Implementation Details | Coefficient Re

Resource Estimates

DSP slice count: 1
BRAM count: 0

Information

Start-up Latency: 50
Calculated Coefficients: 81
Coefficient front padding: 0
Processing cycles per output: 41

AXI4 Stream Port Structure

S_AXIS_DATA - TDATA

Transaction	Field	Type
0	REAL(7:0)	fix8_0

M_AXIS_DATA - TDATA

Transaction	Field	Type
0	REAL(8:0)	fix9_0

Interleaved Channel Sequences

Component Name fir_compiler_1

Filter Options | Channel Specification | Implementation | Detailed Implementation | Interface | Summary

Interleaved Channel Specification

Channel Sequence Basic

Number of Channels 1 [1 - 1024]

Select Sequence

Select Sequence All

Sequence ID List P4-0,P4-1,P4-2,P4-3,P4-4

Parallel Channel Specification

Number of Paths 1 [1 - 16]

Hardware Oversampling Specification

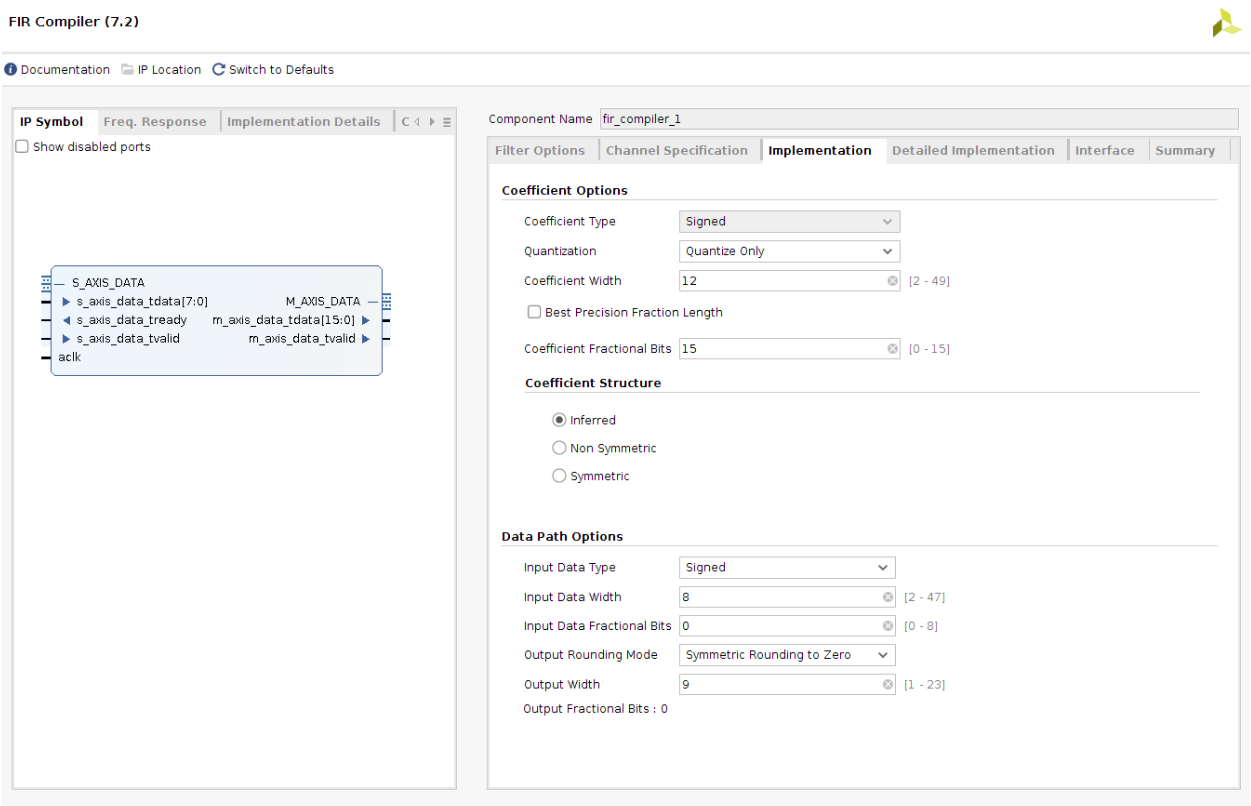
Select Format Frequency Specification

Sample Period (Clock Cycles) 1 [1.0 - 1.0E7]

Input Sampling Frequency (MHz) 0.001 [1.0E-6 - 161280.0]

Clock Frequency (MHz) 300.0 [4.0E-6 - 630.0]

Clock cycles per input:	300000
Clock cycles per output:	300000
Number of parallel inputs	1
Number of parallel outputs	1



FIR Compiler (7.2)

Component Name fir_compiler_1

Coefficient Options

Coefficient Type: Signed

Quantization: Quantize Only

Coefficient Width: 12 [2 - 49]

□ Best Precision Fraction Length

Coefficient Fractional Bits: 15 [0 - 15]

Coefficient Structure

● Inferred

○ Non Symmetric

○ Symmetric

Data Path Options

Input Data Type: Signed

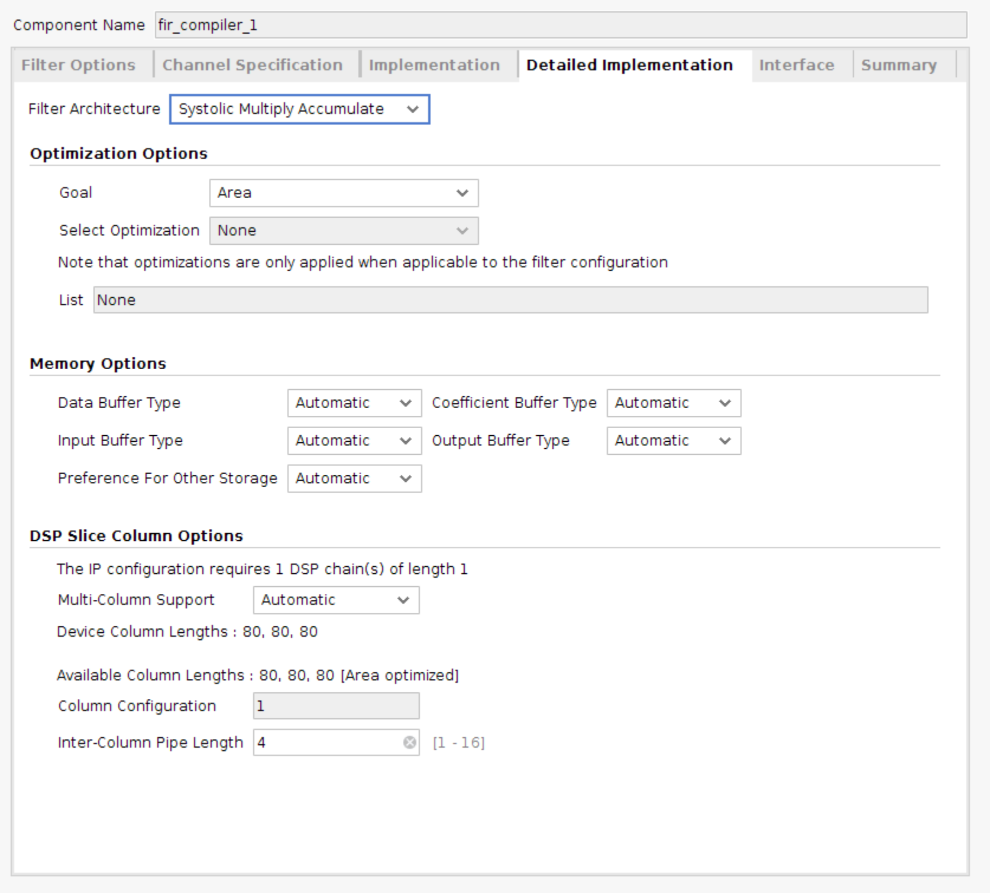
Input Data Width: 8 [2 - 47]

Input Data Fractional Bits: 0 [0 - 8]

Output Rounding Mode: Symmetric Rounding to Zero

Output Width: 9 [1 - 23]

Output Fractional Bits : 0



Component Name fir_compiler_1

Filter Architecture: Systolic Multiply Accumulate

Optimization Options

Goal: Area

Select Optimization: None

Note that optimizations are only applied when applicable to the filter configuration

List: None

Memory Options

Data Buffer Type: Automatic

Coefficient Buffer Type: Automatic

Input Buffer Type: Automatic

Output Buffer Type: Automatic

Preference For Other Storage: Automatic

DSP Slice Column Options

The IP configuration requires 1 DSP chain(s) of length 1

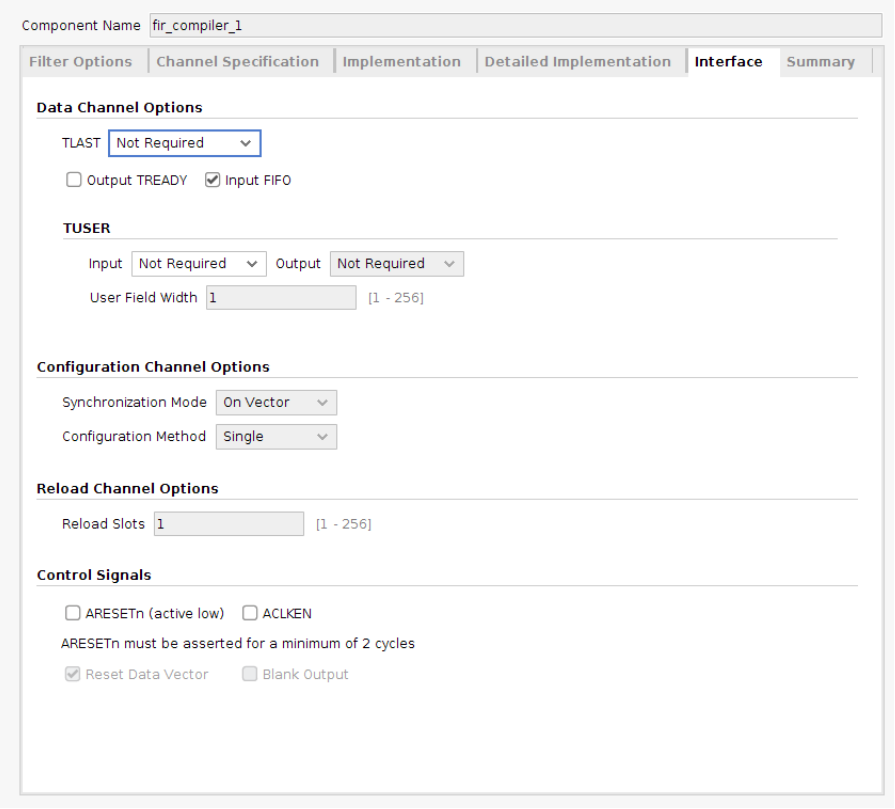
Multi-Column Support: Automatic

Device Column Lengths : 80, 80, 80

Available Column Lengths : 80, 80, 80 [Area optimized]

Column Configuration: 1

Inter-Column Pipe Length: 4 [1 - 16]



Component Name fir_compiler_1

Data Channel Options

TLAST Not Required

□ Output TREADY ☑ Input FIFO

TUSER

Input Not Required Output Not Required

User Field Width 1 [1 - 256]

Configuration Channel Options

Synchronization Mode On Vector

Configuration Method Single

Reload Channel Options

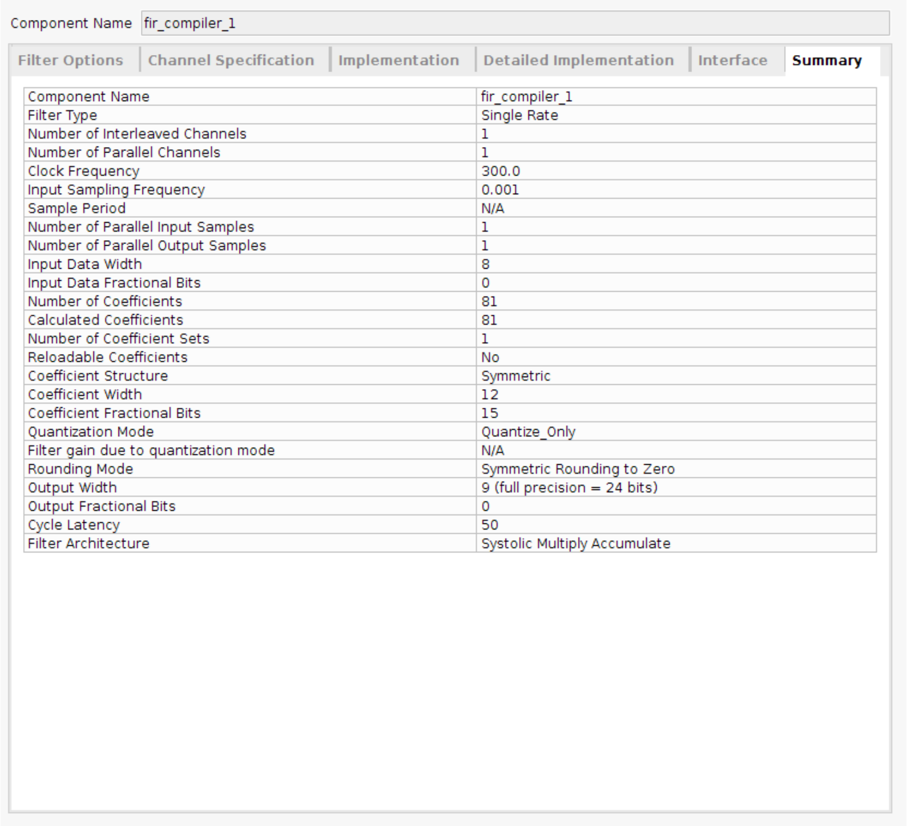
Reload Slots 1 [1 - 256]

Control Signals

□ ARESETn (active low) □ ACLKEN

ARESETn must be asserted for a minimum of 2 cycles

☑ Reset Data Vector □ Blank Output



Component Name fir_compiler_1

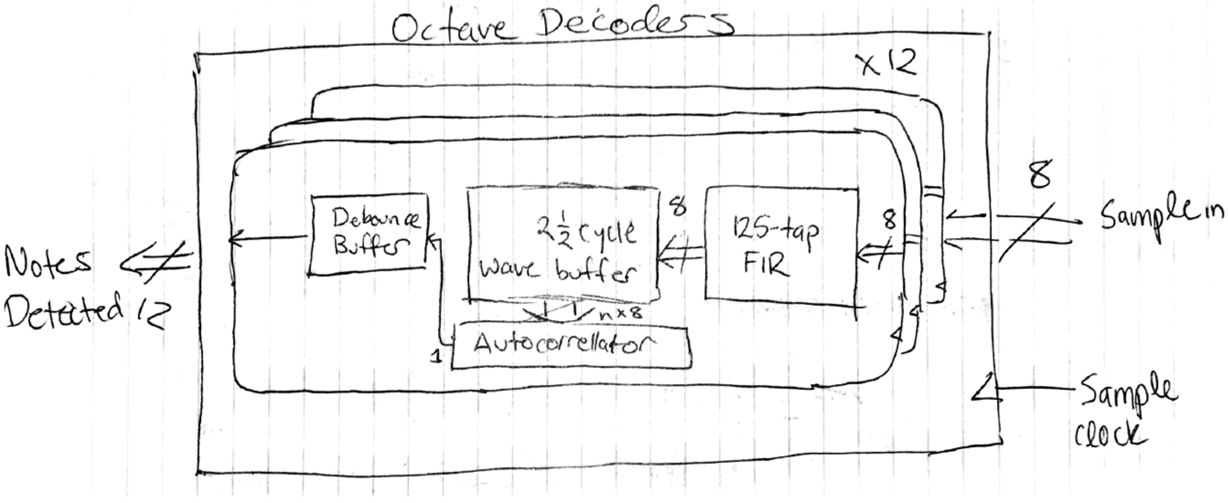
Component Name	fir_compiler_1
Filter Type	Single Rate
Number of Interleaved Channels	1
Number of Parallel Channels	1
Clock Frequency	300.0
Input Sampling Frequency	0.001
Sample Period	N/A
Number of Parallel Input Samples	1
Number of Parallel Output Samples	1
Input Data Width	8
Input Data Fractional Bits	0
Number of Coefficients	81
Calculated Coefficients	81
Number of Coefficient Sets	1
Reloadable Coefficients	No
Coefficient Structure	Symmetric
Coefficient Width	12
Coefficient Fractional Bits	15
Quantization Mode	Quantize_Only
Filter gain due to quantization mode	N/A
Rounding Mode	Symmetric Rounding to Zero
Output Width	9 (full precision = 24 bits)
Output Fractional Bits	0
Cycle Latency	50
Filter Architecture	Systolic Multiply Accumulate

The output of each FIR module is 9 bits, so taking the lower 8 bits results in an 8 bit signed output that is scaled by a factor of 2. This factor of 2 is critical because the FIR module scales amplitudes by ½. We discovered this through testing, as described in the Testing section below.

Octave Frequency Decoder Blocks [Ishaan]

For note detection, an autocorrelation with FIR pre-filtering strategy was used. Before settling on that strategy, we had been experimenting with using high-Q infinite impulse response (IIR) peak filters[9]. However, these IIR peak filters suffered from very large group delay exceeding our latency requirement. This led us to the FIR filter strategy, due to their very predictable latency and ease of generation with Python libraries[13].

A single octave decoder block detects whether the fundamental frequencies of E -> Eb of a certain octave are present. The input to one of these modules are signed 8-bit audio samples, and the output is a 12-bit bus, with each bit representing whether a particular note is detected, i.e. if the LSB is asserted (set to 1), the frequency corresponding to an E is present, and if the 5th bit (zero indexed) is present, the frequency corresponding to an A is present.



The input to the octave decoder (block diagram above) is passed to 12 independent frequency detectors. Each frequency detector is comprised of three parts.

Harmonic-stop FIR filter

Each detector consists of a 125-tap FIR filter designed to filter out any harmonics of a particular note's fundamental frequency (i.e. for the E2 detector, the FIR module will let the 82Hz fundamental pass but filter out 160Hz+ harmonics). At our lowest sampling frequency (3.75kHz), this 125-tap FIR module incurs a 16.5ms delay[9]. The FIR output then gets queued into a buffer that corresponds to the length of 2.5 wave cycles, i.e. for the E2 detector (82Hz) sampled at 3.75kHz, this buffer would be 113 samples long, corresponding to 30.3ms of waveform data. To generate the FIR filter taps for the frequency detectors, the following python script was used:

```
pb_start = center_freq/(2**(PASS_BAND_WIDTH/24))
pb_end = center_freq*(2**(PASS_BAND_WIDTH/24))
fir = signal.firwin(filter_len, cutoff = [pb_start*2/fs, pb_end*2/fs],
                    window = 'blackmanharris', pass_zero = False)
```

Autocorrelator

The heart of the frequency detector is the autocorrelator. The autocorrelation operation takes the first cycle's worth of wave data and effectively "template-matches" it against the rest of the waveform buffer. Template matching is accomplished through an operation similar to convolution, where each sample from the template sequence is multiplied by a corresponding sample in a subset of the buffer. The products are summed together and returned as a datapoint in the autocorrelation. Autocorrelation is explained better by sources [10] and [11]. After the autocorrelation is completed, the location of the best template match is identified through peak picking; the location of this autocorrelation peak relates to the period of the incoming waveform to the frequency detector. If this calculated period is close to the expected period of the frequency detector, then a 'note_detected' line is asserted high. Parameters for the autocorrelation module were also generated with python scripts.

Debouncer

Before being outputted from the module, the 'note_detected' line is debounced for approximately 20ms. This is accomplished by registering the 'note_detected' outputs from the past 20ms and performing a logical AND on those previous outputs. The result of the logical AND is the output from each frequency detector module.

All 12 frequency detector outputs are stitched together to form the output of the octave decoder module. Through clever selection of sampling frequencies, four identical copies of this octave decoder module can be used to detect all notes a guitar can produce. Since the ratios of frequencies between different octaves is 2:1 (i.e. an E2 is ~82.5Hz and an E3 is 165Hz), doubling the sample clock to a higher octave decoder will make higher octave notes appear identical to lower octave notes sampled at half the frequency. This effect is exploited in our note detection strategy.

Note Decoder [Ishaan and Eric]

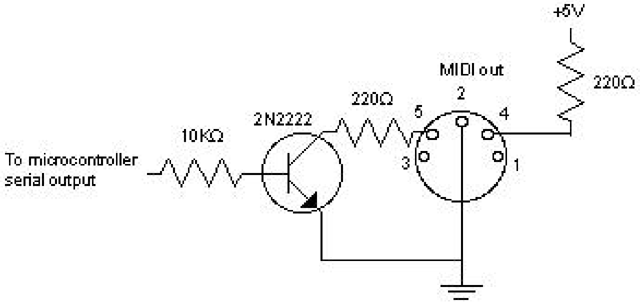
Since an individual guitar note is composed of a fundamental frequency and several harmonics, the output from the frequency decoders must be further processed. This is done through the note decoder module. Since we are only designing for single note detection. The note decoder simply needs to select the lowest frequency detected by all octave frequency decoder blocks. Inputs and outputs to the Note Decoder are synchronized to the slowest sample clock (3.75kHz) to ensure no transient glitches when playing low frequency notes (i.e. if the harmonics of low frequency notes get detected before their fundamentals).

MIDI Protocol Layer [Eric]

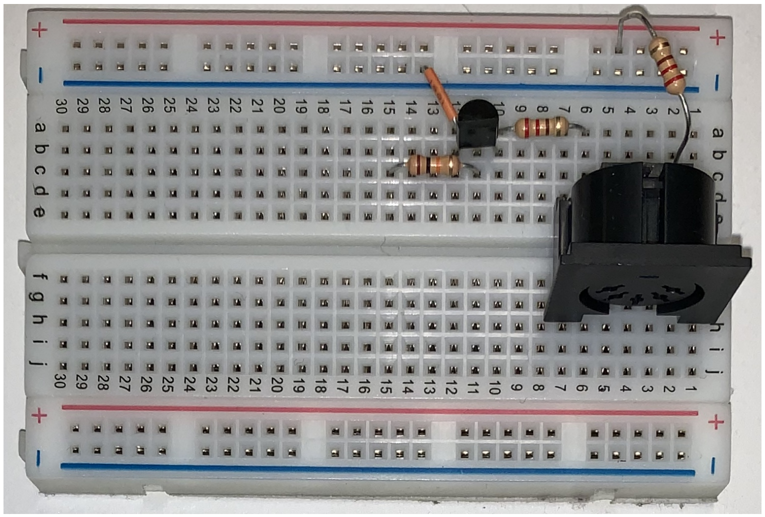
The Protocol Layer transmits the MIDI messages in a serial format. The MIDI protocol is a relatively straightforward UART-style serial communication protocol running at 31250 baud. This source[8] along with countless others on the internet describe the specifics of the MIDI communication protocol and hardware. The input to this module is a 46 single-bit-wide input channels (1 per note), with a 0-1 transition describing a NOTE_ON for that particular channel and a 1-0 transition describing a NOTE_OFF. At this stage, different note velocities are not supported. The output is a single pin with inverted logic for the MIDI Line Driver.

MIDI Line Driver (EXTERNAL)

This is a simple single-transistor driver for the MIDI transmitter. The schematic is the following (from this source[8]).



MIDI Line Driver Schematic



Breadboard Circuit for MIDI Line Driver

This circuit requires inverted logic from the FPGA, which is easily accomplished by inverting the output from the MIDI Protocol Module before sending it to the MIDI output.

Challenges and Advice

Our greatest challenge was developing a real time note decoding scheme. We knew that FFTs would not be fast enough to differentiate between all notes in our lowest octave in real time, but we did not have sufficient knowledge of alternative methods. Our first thought was IIR filtering, but this proved to be a deadend after substantial testing in Python. We then settled on FIRs, which we had explored in Lab 5a, and Autocorrelation, which was entirely new to us. Once we familiarized ourselves with these approaches and prototyped them in Python, we were confident that they were our best bet. We proceeded to reproduce them in Verilog, and this was definitely the easier part of the project.

Another challenge we faced was the extremely long simulation time when running test benches on our FIRs. Because we were simulating with frequencies as low as 100Hz, our test benches could take up to 45 minutes to complete just 2.5 cycles of a waveform. We would not recommend this testing approach to future 6.111 teams. To verify the outputs of filters, we strongly recommend using an ILA, if possible. ILAs can save groups immense amounts of time, and they were very effective in helping us to debug.

Our advice for future groups is to start the project early. Our final build times were approximately 40 minutes and we needed to finely tune our autocorrelation parameters. This tuning process became very tedious, and we were extremely glad that we had not left things down to the wire. We also highly recommend using Python (and SciPy) as a simulation tool for any groups interested in signal processing. The ability to quickly and easily prototype our designs was critical for during the early stages of our project.

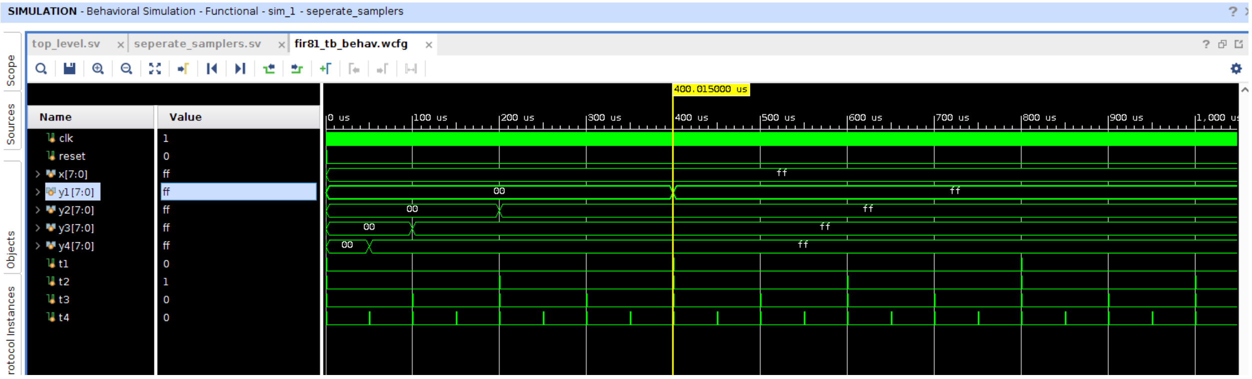
Testing

Preamplifier Testing

For preliminary testing of the preamplifier circuit, we generated a waveform with a function generator with the expected amplitude from the guitar pickups (~100mV nominal), and verified that the output of the amplifier was centered around 500mV and remained within 0-1V. In this test, we noticed a slight distortion of our test waveforms near the zero-crossing of the input signal. We attribute this effect to the quality of our op-amps used in our design, but didn’t think it would pose a significant challenge toward note detection. Following that test, we replaced the line input to the preamplifier with a signal from an actual guitar. The guitar line input reached peak amplitudes higher than what we expected (~200mV), so we decreased the gain of the preamplifier accordingly. Once the performance of the preamplifier circuit was satisfactory, we connected the line output from the preamp to ja[0] on the Artix-7 board.

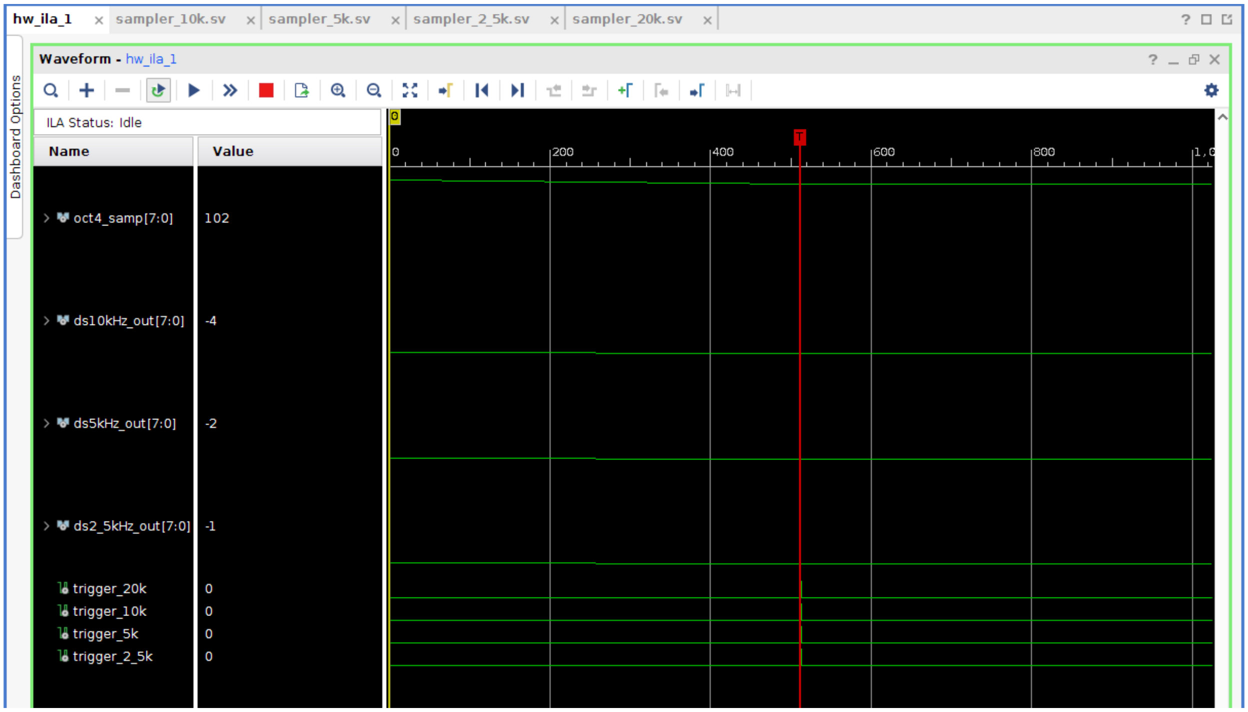
Sample Clock Testing

We developed a test bench to verify the timing of triggers (t1-t4) produced by our sample clocks. We ran our testing with target sampling frequencies of 20kHz, 10kHz, 5kHz, and 2.5kHz. As seen in the plot below, triggers occur every 50us, 100us, 200us, and 400us. This validates our sample clock module.



Name	Value
clk	1
reset	0
x[7:0]	ff
y1[7:0]	ff
y2[7:0]	ff
y3[7:0]	ff
y4[7:0]	ff
t1	0
t2	1
t3	0
t4	0

In order to verify that our triggers were functioning properly in real time, we created an ILA IP to probe the triggers. Initially, we found that they were out of synch and discovered an off-by-one-error in the counting method of our module. As a result, the triggers got farther out of synch as time went on. We then corrected our sample clock module and achieved proper alignment of our sample triggers as seen below.



ILA Status: Idle

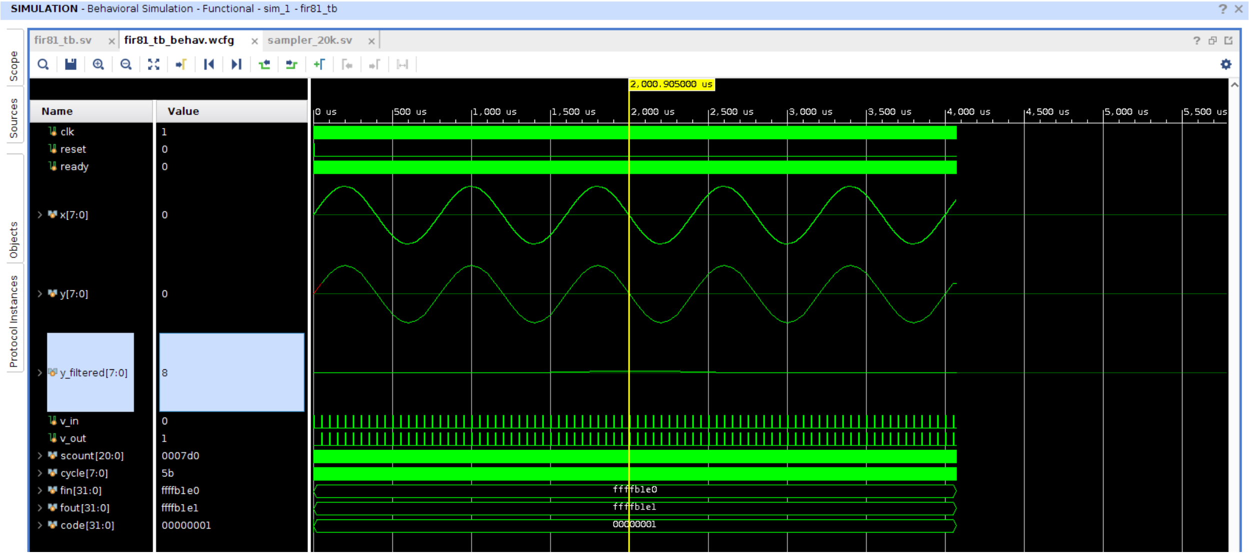
Name	Value
oct4_samp[7:0]	102
ds10kHz_out[7:0]	-4
ds5kHz_out[7:0]	-2
ds2_5kHz_out[7:0]	-1
trigger_20k	0
trigger_10k	0
trigger_5k	0
trigger_2_5k	0

Trigger Alignment with ILA

Anti-aliasing FIR Filter Testing

We took the same FIR testing approach introduced in Lab 5a. We created waveform files in Python. Using these files, we were able to verify that the FIRs for each octave passed the frequencies contained in that octave with minimal attenuation and stopped frequencies at and above the nyquist frequency. The sampling frequencies we used during these tests were 20kHz, 10kHz, 5kHz, and 2.5kHz. The following screenshots depict the results of each test:

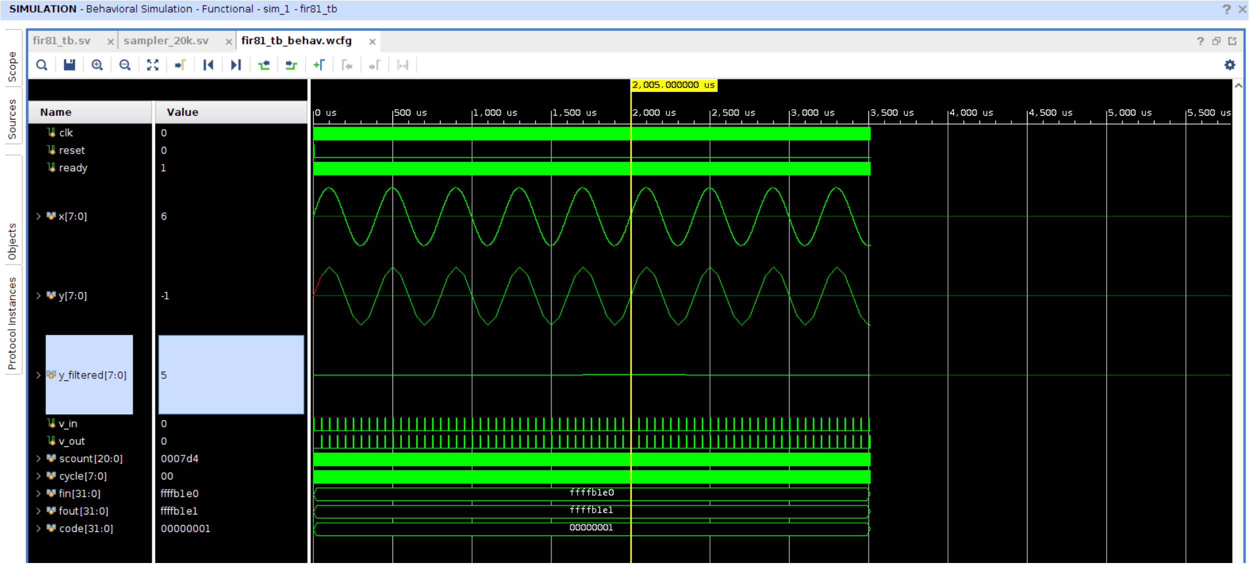
Octave 1 FIR Stopping 1250Hz Wave:



SIMULATION - Behavioral Simulation - Functional - sim_1 - fir81_tb

Name	Value
clk	1
reset	0
ready	0
x[7:0]	0
y[7:0]	0
y_filtered[7:0]	8
v_in	0
v_out	1
scount[20:0]	0007d0
cycle[7:0]	5b
fin[31:0]	ffffb1e0
fout[31:0]	ffffb1e1
code[31:0]	00000001

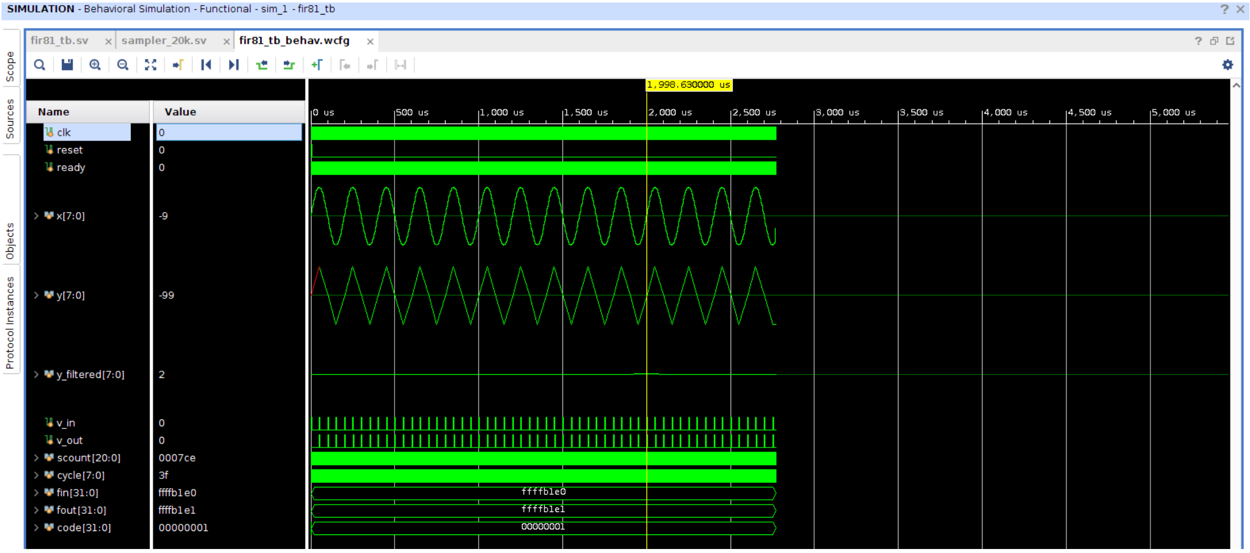
Octave 2 FIR Stopping 2500Hz Wave:



SIMULATION - Behavioral Simulation - Functional - sim_1 - fir81_tb

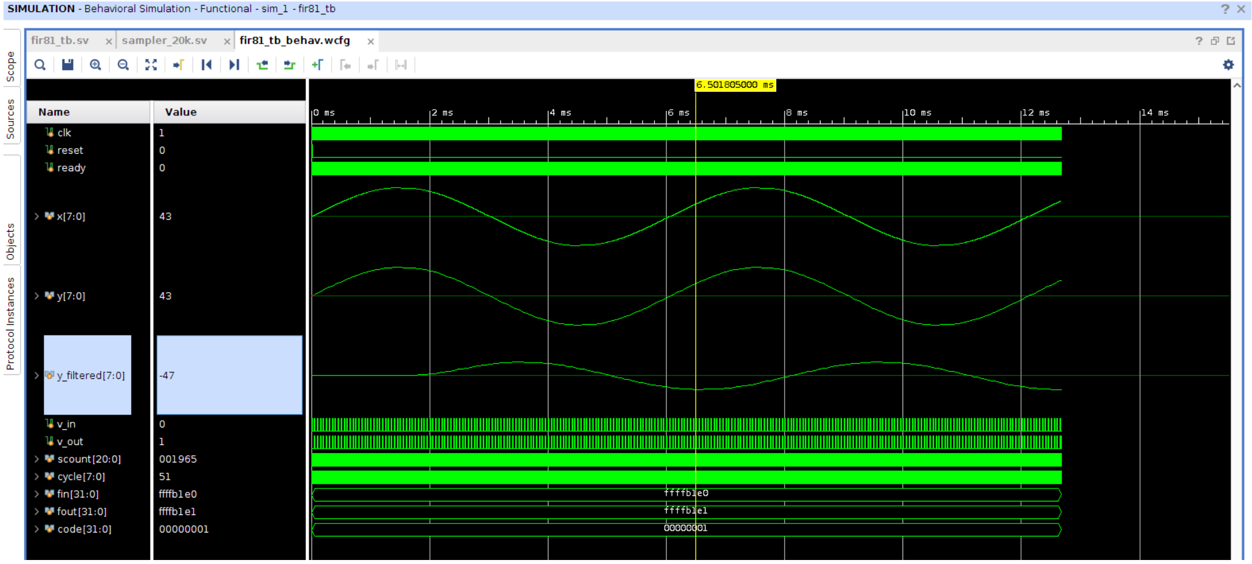
Name	Value
clk	0
reset	0
ready	1
x[7:0]	6
y[7:0]	-1
y_filtered[7:0]	5
v_in	0
v_out	0
scount[20:0]	0007d4
cycle[7:0]	00
fin[31:0]	ffffb1e0
fout[31:0]	ffffb1e1
code[31:0]	00000001

Octave 3 FIR Stopping 5000Hz Wave:



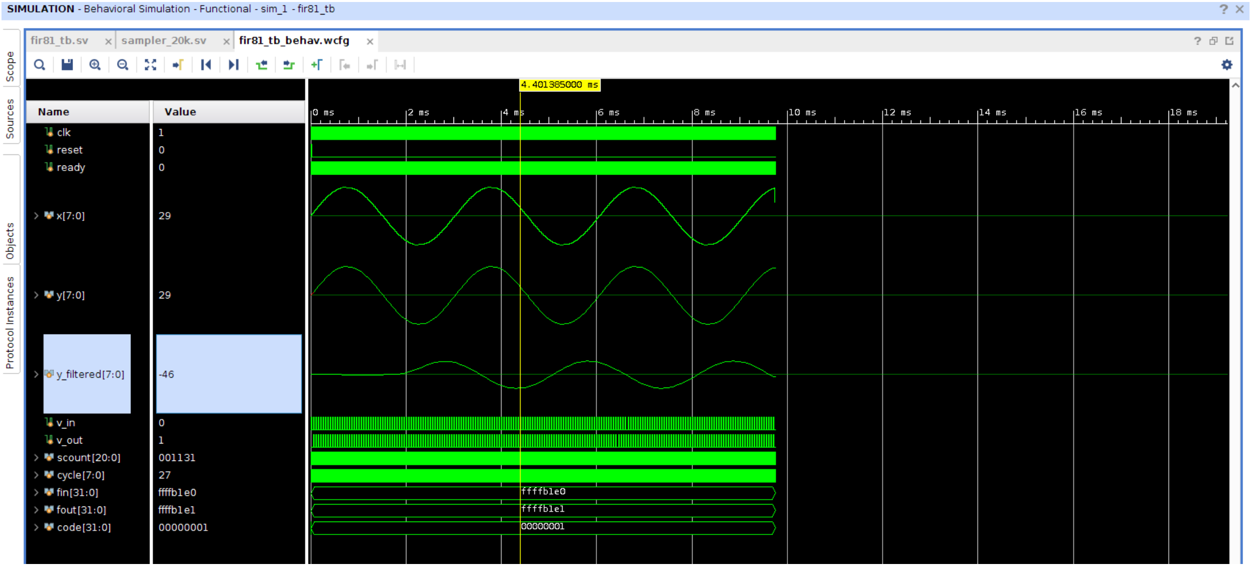
Name	Value
clk	0
reset	0
ready	0
x[7:0]	-9
y[7:0]	-99
y_filtered[7:0]	2
v_in	0
v_out	0
scount[20:0]	0007ce
cycle[7:0]	3f
fin[31:0]	ffffb1e0
fout[31:0]	ffffb1e1
code[31:0]	00000001

Octave 1 FIR Passing 165Hz Wave:



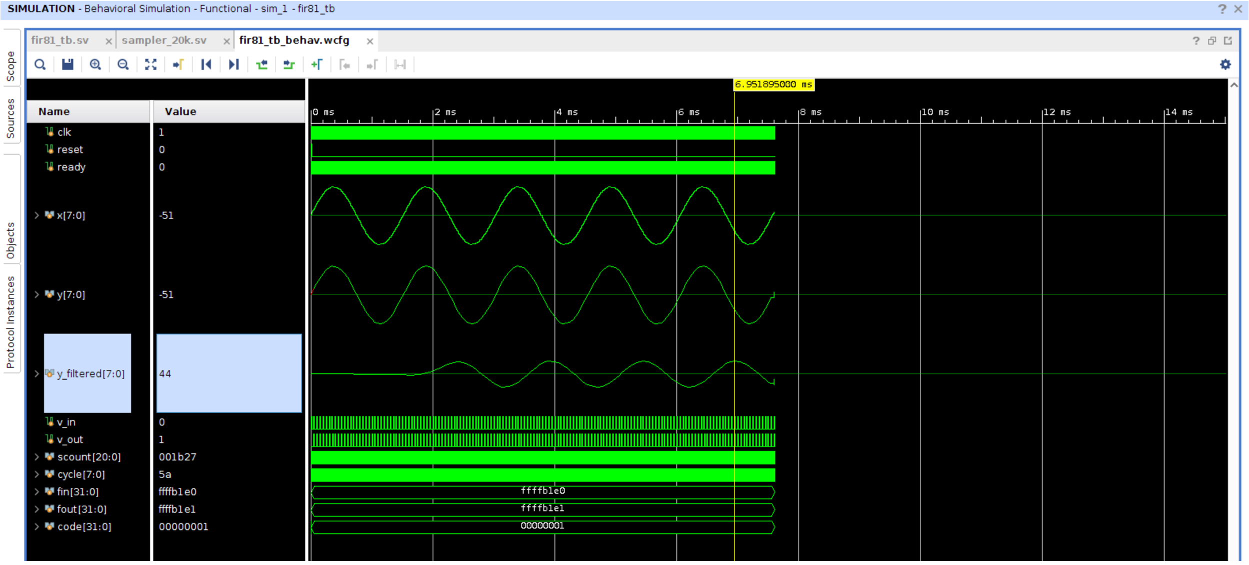
Name	Value
clk	1
reset	0
ready	0
x[7:0]	43
y[7:0]	43
y_filtered[7:0]	-47
v_in	0
v_out	1
scount[20:0]	001965
cycle[7:0]	51
fin[31:0]	ffffb1e0
fout[31:0]	ffffb1e1
code[31:0]	00000001

Octave 2 FIR Passing 330Hz Wave:



Name	Value
clk	1
reset	0
ready	0
x[7:0]	29
y[7:0]	29
y_filtered[7:0]	-46
v_in	0
v_out	1
scount[20:0]	001131
cycle[7:0]	27
fin[31:0]	ffffb1e0
fout[31:0]	ffffb1e1
code[31:0]	00000001

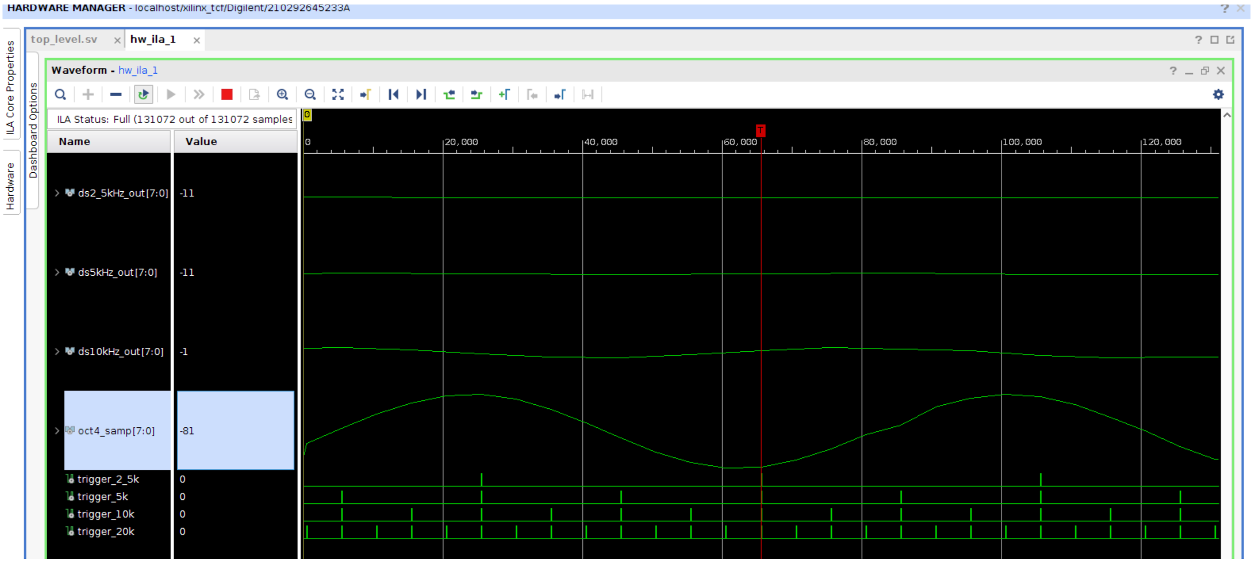
Octave 3 FIR Passing 660Hz Wave:



Name	Value
clk	1
reset	0
ready	0
x[7:0]	-51
y[7:0]	-51
y_filtered[7:0]	44
v_in	0
v_out	1
scount[20:0]	001b27
cycle[7:0]	5a
fin[31:0]	ffffb1e0
fout[31:0]	ffffb1e1
code[31:0]	00000001

As seen in the plots above, the outputs of the FIR modules are scaled by a factor of ½. To compensate for this, we effectively left shift the outputs of each FIR module in Verilog.

In order to visualize the outputs of our FIR modules in real-time, we created an ILA to probe the signals. We observed the plot below.



ILA Status: Full (131072 out of 131072 samples

Name	Value
ds2_5kHz_out[7:0]	-11
ds5kHz_out[7:0]	-11
ds10kHz_out[7:0]	-1
oct4_samp[7:0]	-81
trigger_2_5k	0
trigger_5k	0
trigger_10k	0
trigger_20k	0

FIR Outputs in Real Time with ILA

We noticed that the samples for Octave 4 were out of phase with the samples in the other 3 octaves. This was due to the fact that we initially were not applying an FIR filter to Octave 4. In response, we developed an FIR for Octave 4 so that all outputs would be in phase. While this was not critical, we made this change to reduce the possibility of error downstream in the system.

Verilog

Top level

```
module top_level(   input clk_100mhz,
                    input btnc,
                    input vauxp3,
                    input vauxn3,
                    input vn_in,
                    input vp_in,
                    output logic midi_tx,
                    output logic[15:0]      led
    );

    //========================== ADC and sampling ==============================

    logic [15:0] adc_data; //for some reason the ADC is 16 bits?
    logic [7:0] two_comp_samp; //signed two's complement 8-bit representation of adc sampled data
    logic adc_ready;

    assign two_comp_samp = {~adc_data[15], adc_data[14:8]};//convert ADC sample to 8 bit twos complement

    //configuration more or less taken from lab 5A
    xadc_wiz_0 my_adc ( .dclk_in(clk_100mhz), .daddr_in(8'h13), //read from 0x13 for a
                    .vauxn3(vauxn3),.vauxp3(vauxp3),
                    .vp_in(1),.vn_in(1),
                    .di_in(16'b0),
                    .do_out(adc_data),.drdy_out(adc_ready),
                    .den_in(1), .dwe_in(0));

    //======================= Sample Clock Generators ========================

    // generate a single-cycle 30kHz, 15kHz, 7.5kHz, and 3.75kHz pulse train respectively
    // these sample clocks drive the FIR filters and the downstream octave decoders

    logic trigger_3_75k; //2.5kHz
    logic trigger_7_5k; //5kHz
    logic trigger_15k; //10kHz
    logic trigger_30k; //20kHz

    //Generate 30kHz Pulse, Indicates when to read from ADC and when to read from FIR4
    //also "clocks" the highest octave decoder block
    sample_clock_gen #(.COUNT('d3333)) my_samp4 (
                        .clk(clk_100mhz),
                        .rst(btnc),
                        .trigger(trigger_30k));

    //Generate 15kHz Pulse, Indicates when to read from FIR3
    //also "clocks" the second-highest octave decoder
    sample_clock_gen #(.COUNT('d6666)) my_samp3 (
                        .clk(clk_100mhz),
                        .rst(btnc),
                        .trigger(trigger_15k));

    //Generate 7.5kHz Pulse, Indicates when to read from FIR2
    //also "clocks" the second-lowest octave decoder
    sample_clock_gen #(.COUNT('d13332)) my_samp2 (
                    .clk(clk_100mhz),
                    .rst(btnc),
                    .trigger(trigger_7_5k));

    //Generate 3.75kHz Pulse, Indicates when to read from FIR1
    //also "clocks" the lowest octave decoder
    sample_clock_gen #(.COUNT('d26664)) my_samp1 (
                    .clk(clk_100mhz),
                    .rst(btnc),
                    .trigger(trigger_3_75k));

    //========================== Antialiasing FIR Filters ===========================

    //filter the sampled ADC data to prevent frequency aliasing before downsampling
    //downsampled ADC data gets sent to their respective octave decoder modules

    logic [8:0] fir1_out; //9 Bit output of FIR Modules
    logic [8:0] fir2_out;
    logic [8:0] fir3_out;
    logic [8:0] fir4_out;

    //have to multiply the FIR outputs by 2 to get an output that's scaled appropriately
    logic [7:0] oct1_samp; //8 Bits of FIR Oct 1
    logic [7:0] oct2_samp; //8 Bits of FIR Oct 2
    logic [7:0] oct3_samp; //8 Bits of FIR Oct 3
    logic [7:0] oct4_samp; //8 Bits of FIR Oct 4

    //not really using these valid lines from the FIRs
    //by the time we sample, the FIRs should have plenty of time to compute
  //   logic fir1_valid_out; //Valid Out of FIR X
  //   logic fir2_valid_out;
  //   logic fir3_valid_out;
  //   logic fir4_valid_out;

    //multiplying the FIR outputs by 2 effectively
    assign oct1_samp = fir1_out[7:0]; //Take Lower 8 bits of FIR X Output
    assign oct2_samp = fir2_out[7:0];
    assign oct3_samp = fir3_out[7:0];
    assign oct4_samp = fir4_out[7:0];

    //High frequency filter for the highest-octave decoder module
    //honestly not *super* necessary, but keeps the audio to the highest decoder in phase with
    //audio going to the other octave decoders
    //81 taps, passes ~1300Hz and rolls off to <-60dB close to nyquist
    fir_compiler_4 my_comp4(
      .aclk(clk_100mhz),
      .s_axis_data_tvalid(trigger_30k), //30kHz trigger
      .s_axis_data_tdata(two_comp_samp),
      .m_axis_data_tdata(fir4_out)); //9 Bit Value
      //.m_axis_data_tvalid(fir4_valid_out)); //Octave 4 Valid out pulse

    //Antialiasing FIR to filter audio data going to the second-highest octave decoder module
    //81 taps, passes 660Hz and rolls off to -60dB around 7.5kHz
    fir_compiler_3 my_comp3(
      .aclk(clk_100mhz),
      .s_axis_data_tvalid(trigger_30k), //30kHz trigger
```

```
        .s_axis_data_tdata(two_comp_samp),
        .m_axis_data_tdata(fir3_out)); //9 Bit Value
        //.m_axis_data_tvalid(fir3_valid_out)); //Octave 3 Valid out pulse

    //for second-lowest octave decoder bank
    //81 taps, passes 330Hz and rolls off to -60dB around 3.75kHz
    fir_compiler_2 my_comp2(
        .aclk(clk_100mhz),
        .s_axis_data_tvalid(trigger_30k), //20kHz trigger
        .s_axis_data_tdata(two_comp_samp),
        .m_axis_data_tdata(fir2_out)); //9 Bit Value
        //.m_axis_data_tvalid(fir2_valid_out)); //Octave 2 Valid out pulse

    //for lowest octave decoder bank
    //81 taps, passes 330Hz and rolls off to -60dB around 1.875kHz
    fir_compiler_1 my_comp(
        .aclk(clk_100mhz),
        .s_axis_data_tvalid(trigger_30k), //20kHz trigger
        .s_axis_data_tdata(two_comp_samp),
        .m_axis_data_tdata(fir1_out)); //9 Bit Value
        //.m_axis_data_tvalid(fir1_valid_out)); //Octave 1 Valid out pulse


    //========================== Octave Decoder Modules==========================
    //do the note frequency detection
    //so the cool thing is that since after an octave, frequencies are doubled,
    //we can reuse basically everything if we adjust our sampling frequency accordingly
    //i.e. we can detect a higher octave by doubling our sampling frequency and keeping almost everything else the same
    //which is what we do exactly

    //Side Note: we had some synthesis issues with the reset lines (something about distribution and that we should declare it as a
clock)
    //  so we just kinda commented it out lol; synthesizes fine i guess

    logic [11:0] first_octave_notes;
    logic [11:0] second_octave_notes;
    logic [11:0] third_octave_notes;
    logic [11:0] fourth_octave_notes;

    //just for some debugging
    assign led[11:0] = first_octave_notes;

    //lowest octave decoder bank
    //clocked at 3.75kHz fs (post FIR)
    octave_filter_bank #(.DEBOUNCE_CYCLES('d80)) octave1( //about 21ms of debouncing
        .clk_in(clk_100mhz),
        //.reset(btnc),
        .sample_clk(trigger_3_75k), //From 3.75kHz Trigger
        .in_sample(oct1_samp),
        .note_detected(first_octave_notes)
    );

    //second lowest octave decoder bank (7.5kHz)
    octave_filter_bank #(.DEBOUNCE_CYCLES('d160)) octave2( //21ms debounce
        .clk_in(clk_100mhz),
        //.reset(btnc),
        .sample_clk(trigger_7_5k), //From 7.5kHz Trigger
        .in_sample(oct2_samp),
        .note_detected(second_octave_notes)
    );

    //second highest octave decoder bank (15kHz)
    octave_filter_bank #(.DEBOUNCE_CYCLES('d320)) octave3( //21ms debounce
        .clk_in(clk_100mhz),
        //.reset(btnc),
        .sample_clk(trigger_15k), //From 15kHz Trigger
        .in_sample(oct3_samp),
        .note_detected(third_octave_notes)
    );

    //highest octave decoder bank (30kHz)
    octave_filter_bank #(.DEBOUNCE_CYCLES('d640)) octave4( //21ms debounce
        .clk_in(clk_100mhz),
        //.reset(btnc),
        .sample_clk(trigger_30k), //From 30kHz Trigger
        .in_sample(oct4_samp),
        .note_detected(fourth_octave_notes)
    );

    //=========================== Note Decoder Module ===========================
    //since we're supporting single-note detection, all this does really is
    //pick out the lowest activated note from the outputs of all the octave bank decoders

    logic [45:0] decoder_in; //Input to Note Decoder
    logic [45:0] decoder_out; //To MIDI Module

    //concatenate the outputs of all the octave decoders together (but only take the bottom 10 bits of the highest octave)
    //  \--> since Ishaan's guitar can't play a D6 or Eb6, no point in trying to send those out
    assign decoder_in = {fourth_octave_notes[9:0], third_octave_notes, second_octave_notes, first_octave_notes};

    note_decoder #(.MAXINDEX('d45)) my_decoder
            (   .clk(clk_100mhz),
                .trig(trigger_3_75k),
                //.rst(btnc),   //had some synthesis error with routing the reset line, so just commenting it out
                .pitches_in(decoder_in),
                .notes_out(decoder_out)
             );

    //============================= MIDI Transmitter ============================
    //given whether a note is activated or not, send an appropriate MIDI message
    //NOTE_ONSETs will be sent if a particular note is "newly played"
    //NOTE_OFFSETs will be sent if a particular note is "stopped being played"
    logic midi_temp;

    assign midi_tx = !midi_temp; //Invert Output of MIDI Module -> has to deal with how we implemented our MIDI transmitter

    midi_tx my_tx(.clk_in(clk_100mhz),
                    .rst_in(btnc),
                    .notes_in(decoder_out), //From Note Decoder Module
                    .data_out(midi_temp)); //Invert this output


endmodule
```

MIDI Module

```
module midi_tx(     input           clk_in, //Assume 100 MHz clock passed in
                    input           rst_in,
                    input [45:0]     notes_in, //one for eah of 46 notes
                    output logic    data_out); //single bit, serial out; sequence: command nybble, channel number, key byte, velocity byte

    parameter   DIVISOR = 3200; //Baud Rate: 31250 bits/sec so 3200 cycles between bits
    parameter   NOTE_ON = 4'h9, NOTE_OFF = 4'h8; //Nybbles to start and stop notes
    parameter   CHAN = 4'd1; //Working with instrument channel 1
    parameter   KEY_NUM = 8'd40; //Lowest Note is E2 (MIDI Key number 40)
    parameter   NOTE_VEL = 8'd64; //Fixed note velocity
    parameter   START = 1'b0, STOP = 1'b1; //Start and Stop bits

    logic [29:0]        shift_buffer; //1 message
    logic [31:0]        count; // Number of clock cycles that have passed since transmitting last bit
    logic [4:0]         indexToSend; //indicates which index in our buffer we should be transmitting
    logic [7:0]         currentNote; //8 bits so that it aligns with size of midi data message

    logic               transmitting; //boolean value to indicate if we are transmitting message

    logic [45:0]        notes_old; //to store former values of notes

    always_ff @(posedge clk_in)begin

        if(rst_in)begin
            count <= 32'b0; //counts clock cycles
            indexToSend <= 5'b0; //tracks which index in 30 bit message to send
            transmitting <= 1'b0; // indicates if we are sending message or not
            currentNote <= 8'b0; //indicates which note we are checking for ON/OFF
            notes_old <= 46'b0; //Stores previous values of notes (since they were last updated)
            data_out <= 1'b1; //Keep transmit line high

        end else if (!transmitting) begin
            currentNote <= (currentNote < 8'd45) ? currentNote + 8'b1 : 8'b0; //iterate through 46 notes
            notes_old[currentNote] <= notes_in[currentNote]; // UPDATE

            if (notes_old[currentNote] != notes_in[currentNote]) begin //If the value changes, the note has either come on or come off.
                shift_buffer[9 : 0] <= (notes_in[currentNote] == 1'b1) ?  {STOP, NOTE_ON, CHAN, START} : {STOP, NOTE_OFF, CHAN,
START}; //Send note on or off message
                shift_buffer[19 : 10] <= {STOP, (KEY_NUM + currentNote), START}; //Indicate which note is coming on or off
                shift_buffer[29 : 20] <= {STOP, NOTE_VEL, START}; //indicate note velocity
                transmitting <= 1; //now transmitting
            end else data_out <= 1; //Keep transmit line high

        end else if (transmitting) begin
           if(count>=DIVISOR) begin //Check if 3200 clk cycles have passed
                data_out <= shift_buffer[indexToSend]; //Send next bit every 3200 cycles
                indexToSend <= (indexToSend >= 8'd29) ? 8'b0 : indexToSend + 1; //Increment which bit we are sending
                transmitting <= (indexToSend >= 8'd29) ? 0 : 1; //If we have sent everything, stop transmitting
                count <= 0; //reset count
           end else count <= count + 1; //Increment clock cycle count to mainting 31250 baud

        end
    end
endmodule
```

Note Decoder

```
module note_decoder
     # (parameter MAXINDEX) //parameterize by number of (notes - 1)
      (   input clk,
        input trig, //register the output synchronous to this trigger
        input rst,
        input logic [MAXINDEX:0] pitches_in,
        output logic [MAXINDEX:0] notes_out
       );


  logic lowest_found; //Boolean to indicate whether we have found the lowest frequency on given For Loop iteration

  logic [MAXINDEX:0] in; //Register input
  logic [MAXINDEX:0] out; //Register output


  always_comb begin
    lowest_found = 0;
    for (int i = 0; i <= MAXINDEX; i++) begin
      out[i] = in[i] && !lowest_found; //will deassert anything after lowest note has been found
      lowest_found = lowest_found || in[i]; //lowest found will be 0 until a 1 is found
    end
  end

  always_ff @ (posedge clk)begin
    if (rst) begin
      notes_out <= 0;
      in <= 0;
    end
    else begin
      if (trig) begin
        in <= pitches_in;
        notes_out <= out; //To Registered Output
      end
    end
  end
endmodule
```

Harmonic FIR Filter, Autocorrelator, and Debouncer

```
module filter_block
  #(  parameter SC_LEN,                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 31
    parameter PEAK_MASK = 64'hFFFFFFFF, //if theres a 1 in a corresponding bit, that means a correlation peak in that particular
spot is valid
    parameter DB_THRESHOLD = 13'sd30,    //autocorrelation threshold where the output is considered "activated"
    parameter DB_LENGTH
  )

  (   input clk_in,
    input sample_clock,
    input reset,

    output logic signed [7:0] fir_sample,
    output logic fir_start,
    input logic signed [7:0] fir_result,
    input logic fir_done,
    input logic signed [7:0] in_sample,    //the sample that we've been passed
    output logic note_detected
  );

  //========================= STATE RELEVANT VARIABLES/PARAMS ==========================
  logic [1:0] state;
  parameter ST_IDLE = 2'd0;
  parameter ST_ADD_AC_QUEUE = 2'd1;
  parameter ST_RUN_AC = 2'd2;
  parameter ST_OUT_DB = 2'd3;

  //========================= AUTOCORRELATION RELEVANT VARIABLES/PARAMS
==========================
  logic signed [7:0] ac_samples [(SC_LEN << 1) + (SC_LEN >> 1) - 1 :0];   //buffer to hold our autocorrelation samples (2.5
CYCLES)

  logic signed [31:0] ac_accumulator; //a sample is 8 bits, we're multiplying samples together so need 16 bits to store the fixed point
result
                                        //adding like 5 bits since we accumulate across an entire wave cycle (max like 30 samples per cycle)

  logic [5:0] cycle_index; //pointer which part of the reference cycle/autocorrelation window
  logic [5:0] cycle_offset; //an offset in our buffer for the second waveform to convolve for the autocorrelation if that makes sense

  logic signed [31:0] peak_correlation;  //stores the value of the peak autocorrelation so far
  logic [5:0] peak_offset;               //stores the cycle_offset of the peak autocorrelation value so far

  //======================= DEBOUNCE RELEVANT VARIABLES/PARAMS ======================
  logic [DB_LENGTH-1:0] db_buff; //how many output samples have to be true before we output a "true" from the frequency
detector

  //=============================================================

  //note detection is the logical AND of every value in the debounce buffer
  assign note_detected = &db_buff; //logical AND every element

  always_ff @(posedge clk_in) begin

    if(reset) begin
        db_buff <= 0; //reset the output debounce buffer
        for (int i = 0; i < ((SC_LEN << 1) + (SC_LEN >> 1)); i++) ac_samples[i] <= 'sd0;    // clear out autocorrelation samples

        ac_accumulator <= 'sd0;
        cycle_index <= 0;
        cycle_offset <= 0;
        peak_correlation <= 'sd0;
        peak_offset <= 0;

        fir_start <= 0; //deassert this output to the FIR module
        state <= ST_IDLE;
    end

    else if (sample_clock) begin
        fir_sample <= in_sample;
        fir_start <= 1;            //start the FIR filter convolution next cycle
        state <= ST_ADD_AC_QUEUE;   //go into the state where we're start to enqueue the FIR result
    end
    else if (state == ST_ADD_AC_QUEUE) begin //if the FIR computation is done and we're waiting to enqueue it
      fir_start <= 0; //deassert start

      if(fir_done) begin
        ac_samples[(SC_LEN << 1) + (SC_LEN >> 1) - 1:1] <= ac_samples[(SC_LEN << 1) + (SC_LEN >> 1)-2:0]; //shift our
autocorrelation buffer over
        ac_samples[0] <= fir_result; //add the new sample at the beginning of our autocorrelation buffer
        //reset our cycle index and offset for the autocorrelation convolutions
        cycle_index <= 0;
        cycle_offset <= 0;
        ac_accumulator <= 'sd0;
        state <= ST_RUN_AC; //run the autocorrelation

        peak_correlation <= -'sd2_147_483_648; //reset the peak index to the smallest 32 bit number

      end
    end

    else if (state == ST_RUN_AC) begin
      //calculate the autocorrelation one multiplication at a time (if it's in the valid range)
      //on the last cycle of a particular convolution, compare the peak correlation values and update as necessary
      if(cycle_index < SC_LEN) begin
        //compute the autocorrelation starting at a half-cycle shift
        ac_accumulator <= ac_accumulator + (ac_samples[cycle_index] * ac_samples[(SC_LEN >> 1) + cycle_offset +
cycle_index]);
        cycle_index <= cycle_index + 1;
      end else begin //we're gonna update our comparison here and reset our counter
        peak_correlation <= (ac_accumulator > peak_correlation) ? ac_accumulator : peak_correlation;
        peak_offset <= (ac_accumulator > peak_correlation) ? cycle_offset : peak_offset;
        cycle_index <= 0;
        cycle_offset <= cycle_offset + 1;
        ac_accumulator <= 'sd0;
      end

      //if we've computed the entire autocorrelation, move to the next state, the output computation
      if((cycle_index >= SC_LEN) && (cycle_offset >= (SC_LEN - 1))) state <= ST_OUT_DB;
    end

    //compute the output going to the debouncer array
    else if (state == ST_OUT_DB) begin
      db_buff[DB_LENGTH-1:1] <= db_buff[DB_LENGTH-2:0];
      db_buff[0] <= PEAK_MASK[peak_offset] && (peak_correlation >= DB_THRESHOLD);
      state <= ST_IDLE;
    end

  end


endmodule
```

E-Flat FIR Wrapper (For each FIR module in the autocorrelator there is a wrapper.)

```
module w_fir_eflat (
        input logic clkin, start,
        output logic done,
        input logic [7:0] data_in,
        output logic [7:0] data_out
    );

    logic [15:0] ext_data;

    assign data_out = ext_data[7:0];

    eflat_fir eflat (
        .aclk(clkin),
        .s_axis_data_tvalid(start),
        .s_axis_data_tdata(data_in),
        .m_axis_data_tvalid(done),
        .m_axis_data_tdata(ext_data)
    );

endmodule
```

Octave Frequency Decoder Bank

```
module octave_filter_bank
    # (parameter DEBOUNCE_CYCLES = 'd83)    //how many samples we should debounce the output for
    (
        input clk_in, reset, sample_clk,
        input logic signed [7:0] in_sample,
        output logic [11:0] note_detected
    );

    assign note_detected = {    detect_eflat,
                                detect_d,
                                detect_dflat,
                                detect_c,
                                detect_b,
                                detect_bflat,
                                detect_a,
                                detect_gsharp,
                                detect_g,
                                detect_fsharp,
                                detect_f,
                                detect_e };

    //============================ E FILTER =================================

    logic signed [7:0] f_e_sample;
    logic fir_e_start;
    logic signed [7:0] f_e_result;
    logic fir_e_done;
    logic detect_e;

    filter_block #(
            .SC_LEN('d46),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 31
            .PEAK_MASK(64'd58720256),       //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot
is valid
            .DB_THRESHOLD(32'sd0),//3768),   //autocorrelation threshold where the output is considered "activated"
            .DB_LENGTH(DEBOUNCE_CYCLES)
        ) e_block (
            .clk_in(clk_in),
            .sample_clock(sample_clk),
            .reset(reset),

            .fir_sample(f_e_sample),
            .fir_start(fir_e_start),
            .fir_result(f_e_result),
            .fir_done(fir_e_done),

            .in_sample(in_sample),    //the sample that we've been passed
            .note_detected(detect_e)
        );

    //125 tap FIR filter to isolate the particular note (kills all harmonics)
    //controlled by the note detector block
    w_fir_e e_fir (
            .clkin(clk_in), .start(fir_e_start),
            .done(fir_e_done),
            .data_in(f_e_sample),
            .data_out(f_e_result)
        );

    //========================== F FILTER ===========================

    logic signed [7:0] f_f_sample;
    logic fir_f_start;
    logic signed [7:0] f_f_result;
    logic fir_f_done;
    logic detect_f;

    filter_block #(
            .SC_LEN('d43),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 63
            .PEAK_MASK(64'd31457280),       //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot is valid
            .DB_THRESHOLD(32'sd0),//3522),  //autocorrelation threshold where the output is considered "activated"
            .DB_LENGTH(DEBOUNCE_CYCLES)
        ) f_block (
            .clk_in(clk_in),
            .sample_clock(sample_clk),
            .reset(reset),

            .fir_sample(f_f_sample),
            .fir_start(fir_f_start),
            .fir_result(f_f_result),
            .fir_done(fir_f_done),

            .in_sample(in_sample),    //the sample that we've been passed
            .note_detected(detect_f)
        );

    //125 tap FIR filter to isolate the particular note (kills all harmonics)
    //controlled by the note detector block
    w_fir_f f_fir (
            .clkin(clk_in), .start(fir_f_start),
            .done(fir_f_done),
            .data_in(f_f_sample),
            .data_out(f_f_result)
        );

    //========================== F# FILTER ===========================

    logic signed [7:0] f_fsharp_sample;
    logic fir_fsharp_start;
    logic signed [7:0] f_fsharp_result;
    logic fir_fsharp_done;
    logic detect_fsharp;

    filter_block #(
            .SC_LEN('d41),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 63
            .PEAK_MASK(64'd7340032),        //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot
is valid
        .DB_THRESHOLD(32'sd0),//3358),  //autocorrelation threshold where the output is considered "activated"
        .DB_LENGTH(DEBOUNCE_CYCLES)
      ) f_sharp_block (
        .clk_in(clk_in),
        .sample_clock(sample_clk),
        .reset(reset),

        .fir_sample(f_fsharp_sample),
        .fir_start(fir_fsharp_start),
        .fir_result(f_fsharp_result),
        .fir_done(fir_fsharp_done),

        .in_sample(in_sample),    //the sample that we've been passed
        .note_detected(detect_fsharp)
      );

    //125 tap FIR filter to isolate the particular note (kills all harmonics)
    //controlled by the note detector block
    w_fir_fsharp fsharp_fir (
        .clkin(clk_in), .start(fir_fsharp_start),
        .done(fir_fsharp_done),
        .data_in(f_fsharp_sample),
        .data_out(f_fsharp_result)
      );

    //========================== G FILTER ===========================

    logic signed [7:0] f_g_sample;
    logic fir_g_start;
    logic signed [7:0] f_g_result;
    logic fir_g_done;
    logic detect_g;

    filter_block #(
        .SC_LEN('d38),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 63
        .PEAK_MASK(64'd3670016),        //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot
is valid
        .DB_THRESHOLD(32'sd0),//3112),  //autocorrelation threshold where the output is considered "activated"
        .DB_LENGTH(DEBOUNCE_CYCLES)
      ) g_block (
        .clk_in(clk_in),
        .sample_clock(sample_clk),
        .reset(reset),

        .fir_sample(f_g_sample),
        .fir_start(fir_g_start),
        .fir_result(f_g_result),
        .fir_done(fir_g_done),

        .in_sample(in_sample),    //the sample that we've been passed
        .note_detected(detect_g)
      );

    //125 tap FIR filter to isolate the particular note (kills all harmonics)
    //controlled by the note detector block
    w_fir_g g_fir (
        .clkin(clk_in), .start(fir_g_start),
        .done(fir_g_done),
        .data_in(f_g_sample),
        .data_out(f_g_result)
      );

    //========================== G# FILTER ============================

    logic signed [7:0] f_gsharp_sample;
    logic fir_gsharp_start;
    logic signed [7:0] f_gsharp_result;
    logic fir_gsharp_done;
    logic detect_gsharp;

    filter_block #(
        .SC_LEN('d36),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 63
        .PEAK_MASK(64'd1835008),        //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot
is valid
        .DB_THRESHOLD(32'sd0),//2949),  //autocorrelation threshold where the output is considered "activated"
        .DB_LENGTH(DEBOUNCE_CYCLES)
      ) gsharp_block (
        .clk_in(clk_in),
        .sample_clock(sample_clk),
        .reset(reset),

        .fir_sample(f_gsharp_sample),
        .fir_start(fir_gsharp_start),
        .fir_result(f_gsharp_result),
        .fir_done(fir_gsharp_done),

        .in_sample(in_sample),    //the sample that we've been passed
        .note_detected(detect_gsharp)
      );

    w_fir_gsharp gsharp_fir (
        .clkin(clk_in), .start(fir_gsharp_start),
        .done(fir_gsharp_done),
        .data_in(f_gsharp_sample),
        .data_out(f_gsharp_result)
      );
    //========================== A FILTER ===========================

    logic signed [7:0] f_a_sample;
    logic fir_a_start;
    logic signed [7:0] f_a_result;
    logic fir_a_done;
    logic detect_a;

    filter_block #(
        .SC_LEN('d34),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 63
        .PEAK_MASK(64'd917504),         //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot is valid
        .DB_THRESHOLD(32'sd0),//2785),  //autocorrelation threshold where the output is considered "activated"
        .DB_LENGTH(DEBOUNCE_CYCLES)
      ) a_block (
        .clk_in(clk_in),
        .sample_clock(sample_clk),
        .reset(reset),

        .fir_sample(f_a_sample),
        .fir_start(fir_a_start),
        .fir_result(f_a_result),
        .fir_done(fir_a_done),

        .in_sample(in_sample),    //the sample that we've been passed
        .note_detected(detect_a)
      );

    w_fir_a a_fir (
        .clkin(clk_in), .start(fir_a_start),
        .done(fir_a_done),
        .data_in(f_a_sample),
        .data_out(f_a_result)
      );

    //========================== Bb FILTER ===========================

    logic signed [7:0] f_bflat_sample;
    logic fir_bflat_start;
    logic signed [7:0] f_bflat_result;
    logic fir_bflat_done;
    logic detect_bflat;

    filter_block #(
        .SC_LEN('d32),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 63
        .PEAK_MASK(64'd458752),         //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot is valid
        .DB_THRESHOLD(32'sd0),//2621),  //autocorrelation threshold where the output is considered "activated"
        .DB_LENGTH(DEBOUNCE_CYCLES)
      ) bflat_block (
        .clk_in(clk_in),
        .sample_clock(sample_clk),
        .reset(reset),

        .fir_sample(f_bflat_sample),
        .fir_start(fir_bflat_start),
        .fir_result(f_bflat_result),
            .fir_done(fir_bflat_done),

            .in_sample(in_sample),    //the sample that we've been passed
            .note_detected(detect_bflat)
        );

    w_fir_bflat bflat_fir (
            .clkin(clk_in), .start(fir_bflat_start),
            .done(fir_bflat_done),
            .data_in(f_bflat_sample),
            .data_out(f_bflat_result)
        );

    //========================== B FILTER ============================

    logic signed [7:0] f_b_sample;
    logic fir_b_start;
    logic signed [7:0] f_b_result;
    logic fir_b_done;
    logic detect_b;

    filter_block #(
            .SC_LEN('d30),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 63
            .PEAK_MASK(64'd229376),         //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot is
valid
            .DB_THRESHOLD(32'sd0),//2457),  //autocorrelation threshold where the output is considered "activated"
            .DB_LENGTH(DEBOUNCE_CYCLES)
        ) b_block (
            .clk_in(clk_in),
            .sample_clock(sample_clk),
            .reset(reset),

            .fir_sample(f_b_sample),
            .fir_start(fir_b_start),
            .fir_result(f_b_result),
            .fir_done(fir_b_done),

            .in_sample(in_sample),    //the sample that we've been passed
            .note_detected(detect_b)
        );

    w_fir_b b_fir (
            .clkin(clk_in), .start(fir_b_start),
            .done(fir_b_done),
            .data_in(f_b_sample),
            .data_out(f_b_result)
        );

    //========================== C FILTER ============================

    logic signed [7:0] f_c_sample;
    logic fir_c_start;
    logic signed [7:0] f_c_result;
    logic fir_c_done;
    logic detect_c;

    filter_block #(
            .SC_LEN('d29),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 63
            .PEAK_MASK(64'd114688),         //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot is valid
            .DB_THRESHOLD(32'sd0),//2375),  //autocorrelation threshold where the output is considered "activated"
            .DB_LENGTH(DEBOUNCE_CYCLES)
        ) c_block (
            .clk_in(clk_in),
            .sample_clock(sample_clk),
            .reset(reset),

            .fir_sample(f_c_sample),
            .fir_start(fir_c_start),
            .fir_result(f_c_result),
            .fir_done(fir_c_done),

            .in_sample(in_sample),    //the sample that we've been passed
            .note_detected(detect_c)
        );

    w_fir_c c_fir (
            .clkin(clk_in), .start(fir_c_start),
            .done(fir_c_done),
            .data_in(f_c_sample),
            .data_out(f_c_result)
        );

    //========================= Db FILTER ===========================

    logic signed [7:0] f_dflat_sample;
    logic fir_dflat_start;
    logic signed [7:0] f_dflat_result;
    logic fir_dflat_done;
    logic detect_dflat;

    filter_block #(
            .SC_LEN('d27),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 63
            .PEAK_MASK(64'd49152),          //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot is valid
            .DB_THRESHOLD(32'sd0),//2211),  //autocorrelation threshold where the output is considered "activated"
            .DB_LENGTH(DEBOUNCE_CYCLES)
        ) dflat_block (
            .clk_in(clk_in),
            .sample_clock(sample_clk),
            .reset(reset),

            .fir_sample(f_dflat_sample),
            .fir_start(fir_dflat_start),
            .fir_result(f_dflat_result),
            .fir_done(fir_dflat_done),

            .in_sample(in_sample),    //the sample that we've been passed
            .note_detected(detect_dflat)
        );

    w_fir_dflat dflat_fir (
            .clkin(clk_in), .start(fir_dflat_start),
            .done(fir_dflat_done),
            .data_in(f_dflat_sample),
            .data_out(f_dflat_result)
        );

    //========================== D FILTER ============================

    logic signed [7:0] f_d_sample;
    logic fir_d_start;
    logic signed [7:0] f_d_result;
    logic fir_d_done;
    logic detect_d;

    filter_block #(
            .SC_LEN('d26),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 63
            .PEAK_MASK(64'd57344),          //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot is
valid
            .DB_THRESHOLD(32'sd0),//2129),  //autocorrelation threshold where the output is considered "activated"
            .DB_LENGTH(DEBOUNCE_CYCLES)
        ) d_block (
            .clk_in(clk_in),
            .sample_clock(sample_clk),
            .reset(reset),

            .fir_sample(f_d_sample),
            .fir_start(fir_d_start),
            .fir_result(f_d_result),
            .fir_done(fir_d_done),

            .in_sample(in_sample),    //the sample that we've been passed
            .note_detected(detect_d)
        );

    w_fir_d d_fir (
            .clkin(clk_in), .start(fir_d_start),
            .done(fir_d_done),
            .data_in(f_d_sample),
            .data_out(f_d_result)
        );

    //========================== Eb FILTER ============================

    logic signed [7:0] f_eflat_sample;
    logic fir_eflat_start;
    logic signed [7:0] f_eflat_result;
    logic fir_eflat_done;
    logic detect_eflat;

    filter_block #(
            .SC_LEN('d24),                  //length of a single cycle in the autocorrelation buffer (samples), SHOULD BE LESS THAN 63
            .PEAK_MASK(64'd12288),          //if theres a 1 in a corresponding bit, that means a correlation peak in that particular spot is
valid
            .DB_THRESHOLD(32'sd0),//1966),  //autocorrelation threshold where the output is considered "activated"
            .DB_LENGTH(DEBOUNCE_CYCLES)
        ) eflat_block (
            .clk_in(clk_in),
            .sample_clock(sample_clk),
            .reset(reset),
            .fir_sample(f_eflat_sample),
            .fir_start(fir_eflat_start),
            .fir_result(f_eflat_result),
            .fir_done(fir_eflat_done),

            .in_sample(in_sample),    //the sample that we've been passed
            .note_detected(detect_eflat)
        );

    w_fir_eflat eflat_fir (
            .clkin(clk_in), .start(fir_eflat_start),
            .done(fir_eflat_done),
            .data_in(f_eflat_sample),
            .data_out(f_eflat_result)
        );

endmodule
```

Sample Clock Generators

```
module sample_clock_gen #(parameter COUNT) ( //COUNT shouldn't be more than 16 bits (65535)
            input clk,
            input rst,
            output logic trigger );

  logic [15:0] counter;

  assign trigger = (counter == 0);

  always_ff @(posedge clk)begin

    if (rst) begin
      counter <= 0;
    end else begin
      if (counter >= COUNT - 1) counter <= 0;
      else counter <= counter + 1;
   end
 end
endmodule
```

Purchased Components

USB to MIDI:
https://www.amazon.com/gp/product/B0719V8MX1/ref=ox_sc_act_title_1?smid=A2MHNFOYKJHIX1&psc=1

MIDI Connector:
https://www.amazon.com/gp/product/B00OE7JU88/ref=ox_sc_act_title_2?smid=A1U6PSXYZS4A87&psc=1

¼” to 3.5mm Cable:
https://www.amazon.com/gp/product/B000068O3D/ref=ox_sc_act_title_3?smid=ATVPDKIKX0DER&th=1

3.5mm jack breakout:
https://www.amazon.com/gp/product/B01KFP0HBG/ref=ox_sc_act_title_2?smid=A34K5WF5Z9R33P&psc=1

Single AA Holders:
https://www.amazon.com/gp/product/B07BXX62JF/ref=ox_sc_act_title_1?smid=A2UIWYS7E6PLOL&psc=1

Sources

[1] https://en.wikipedia.org/wiki/Guitar
[2] https://newt.phys.unsw.edu.au/jw/notes.html
[3] https://www.highfidelity.com/blog/how-much-latency-can-live-musicians-tolerate-da8e2ebe587a
[4] https://www.soundonsound.com/techniques/optimising-latency-pc-audio-interface#7
[5] https://www.highfidelity.com/blog/how-much-latency-can-live-musicians-tolerate-da8e2ebe587a
[6] https://en.wikipedia.org/wiki/Audio-to-video_synchronization
[7] https://musicinformationretrieval.com/novelty_functions.html
[8] http://www.tigoe.com/pcomp/code/communication/midi/
[9] https://dspguru.com/dsp/faqs/fir/properties/
[10] https://www.instructables.com/id/Reliable-Frequency-Detection-Using-DSP-Techniques/
[11] https://en.wikipedia.org/wiki/Autocorrelation
[12] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.iirpeak.html
[13] https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.firwin.html