The DiGuitar

Pronounced "Digi-Tar" (Formerly Note-ator)
6.111 Final Project Proposal
Eric Pence & Ishaan Govindarajan

Project Overview

The DiGuitar aims to receive 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 will take advantage of the Artix-7's onboard ADC to sample the incoming audio waveform. Further digital signal processing (DSP) techniques such as digital Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filtering, and Fast Fourier Transforms (FFTs) may be used to detect the frequency composition of the guitar input waveform. Further processing of the spectral content will take place to detect the particular notes being played. Note information will be transmitted in a MIDI compliant format to be received by other devices for recording and playback.

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.

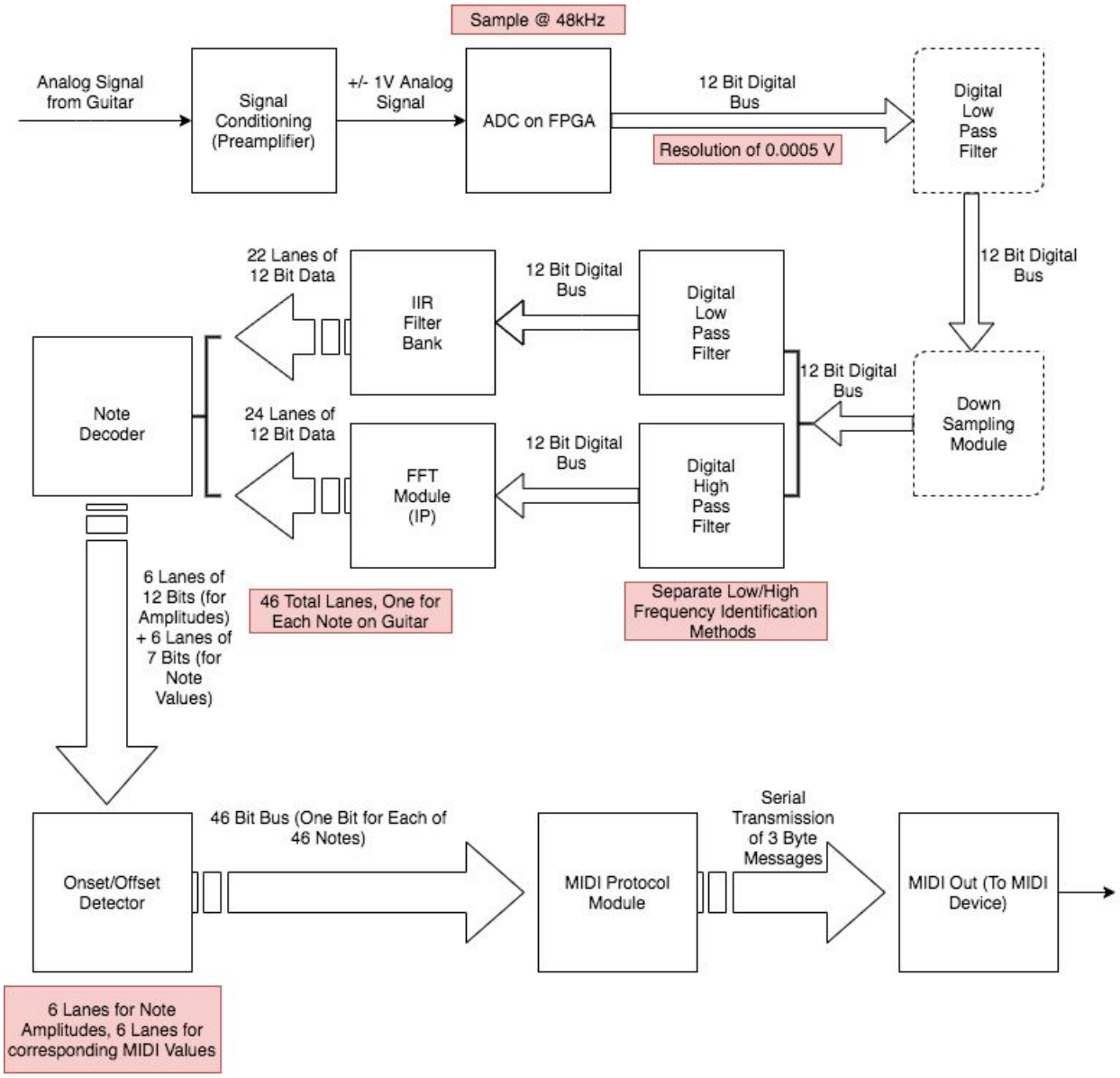
Requirements driven by Guitar Audio Input
Using a guitar as an input source primarily dictates the dynamic range of frequencies the DiGuitar must be able to process. The lowest note that a guitar can produce is an E2[1] (low E-string, MIDI code 40, 82.41 Hz[2]). The second lowest note that it can produce 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 proposal.
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 should be ~2250Hz or higher.** Since high frequency harmonics will be present in the audio input and could alias at low sampling frequencies, it would be wise to sample at a higher frequency e.g. 25-50kHz and downsample after using a low-pass FIR or IIR filter if a low bitrate is necessary.

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 would be 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. This latency target will directly affect note detection strategies.

System Architecture/Block Diagram



Module Descriptions

Preamplifier (EXTERNAL)
Buffers and amplifies the guitar voltage output to something that can be read by the FPGA ADC (+/- 1V range). Amplification ratio will be determined after measuring guitar pickup output voltage, but is forecasted to be in the 5-10x range. Gain and offset requirements of this application should easily be met by Op-amps available in EDS or lab.

ADC
Xilinx IP block to digitize the input audio waveform; will likely be 10-12 bits deep.

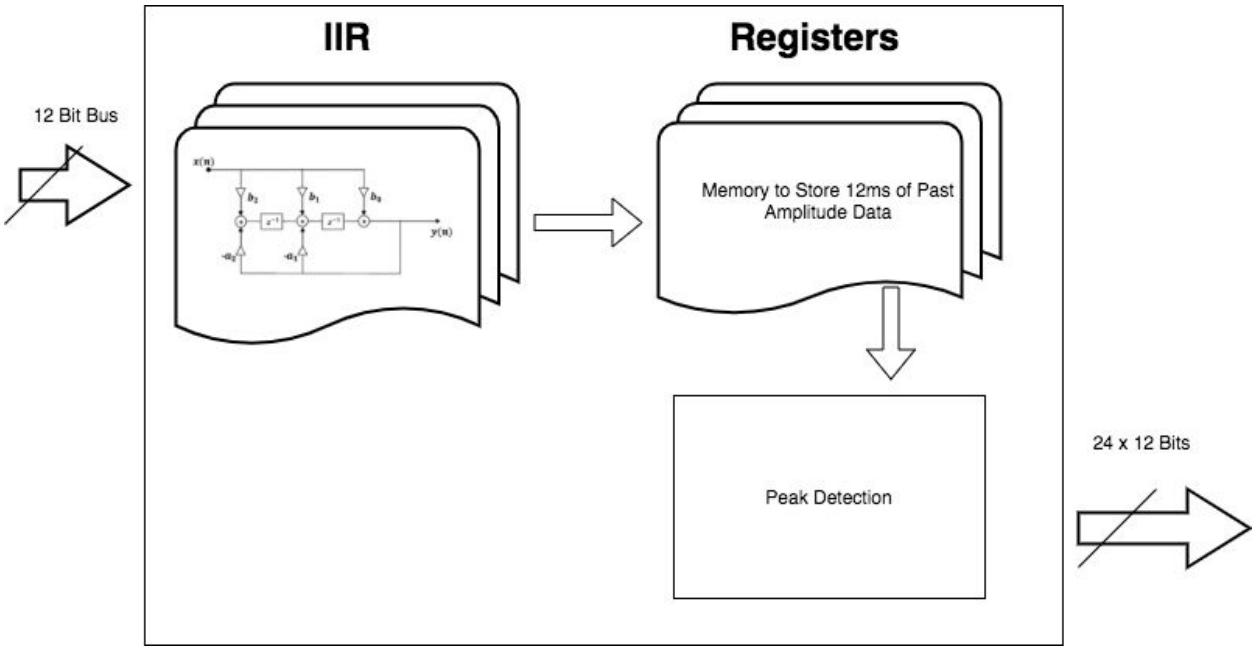
Digital LP Filter / Downsampler (TBD) [Eric]
A module that digitally filters (FIR or IIR TBD) high frequency harmonics of the input audio and down samples to decrease the data throughput to the rest of the note detector (TBD). Output is the same bus width as the module input (likely 12 bits) for easy integration/removal.

High-Frequency Decoder Block (Digital High-Pass Filter [Eric] and FFT Module [Ishaan])
A section that high-passes the audio input and detects the high frequencies in the audio stream. At the moment, the cutoff frequency of this submodule is forecasted to be around the E4 (329Hz, high E-string on guitar). The output of this module will be 22 8-12 bit bit buses, one bus for each pitch between the E4 to the $C^{\#}6$ (inclusive).
Note: Currently an FFT module is proposed to do the frequency decomposition, but more research will be done to select an optimized pitch detection strategy that is accurate and minimizes latency.

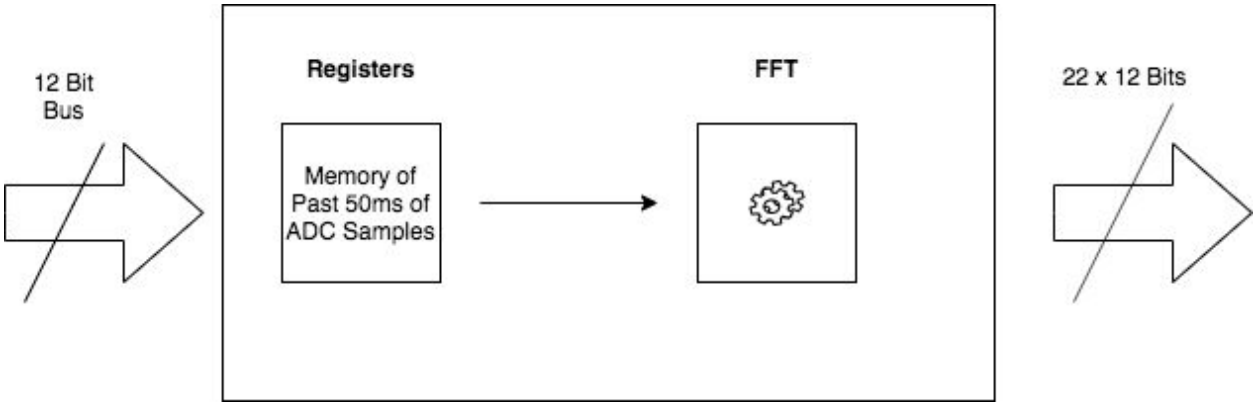
Low-Frequency Decoder Block (Digital Low-Pass Filter [Eric] and IIR Filter Bank [Ishaan])
STRETCH GOAL 1
This module intends to decode notes in the E2 to E4 range (82.4 - 329Hz). However, this is more challenging than the high frequency decoder block due to the real-time requirement along with the tight spacing between neighboring pitches (see *Constraints and Requirements Exploration)*. FFT would not be practical in this application due to the 5Hz difference in the low frequency pitches, requiring a sample length of 200ms to confidently differentiate between those pitches.
As of now, the proposed strategy for low frequency note decoding is the use of a high-Q second-order IIR band-pass filter bank to individually pick out frequencies in the audio waveform. An IIR filter bank is preferred due to their “snappy” response (i.e. low latency) over FIR filters.
IIR coefficient generation could be scripted in python using the scipy.signal.iirpeak function in Python, though coefficient values may be converted from floating-point to fixed-point to ease computation in FPGA hardware.

Amplitudes of the various peaks could be estimated in a single cycle of the note frequency and be output to the pitch decoder in a similar format to the high-frequency decoder block (i.e. 24 8-12 bit buses). If the IIR detection strategy is successful, the design may be ported over to the high-frequency decoder block as well.

Low Frequency Decoder Block Diagram:



High Frequency Decoder Block Diagram:



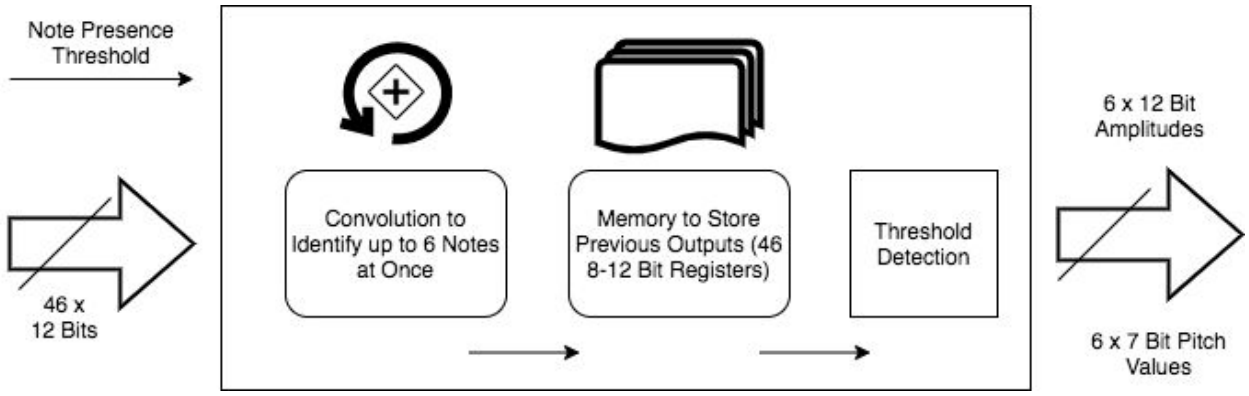
Pitch Decoder [Eric]

Since an individual guitar note is composed of a fundamental frequency and several harmonics, the output from the frequency decoders must be further processed. The pitch decoder in its absolute simplest form will select out the lowest frequency from its 46 8-12bit inputs above its

noise floor (threshold tbd). After identification, the pitch (represented as a MIDI code) along with its intensity (both represented as 8-bit buses) will be output to the note decoder.

STRETCH GOAL 2

A significant shortcoming of the minimum viable pitch decoder is that it only supports a single note output at a time. With 6 strings, a guitar can output up to 6 different pitches at the same time. A more robust note detector would be capable of multi pitch decoding. To do this correctly, the detector must have knowledge of the harmonic structure of a guitar note, i.e. what the relative amplitudes of the harmonics of a guitar note are. Knowing this structure, a "compensation" kernel can be generated and convolved with all the outputs from the frequency decoder blocks, starting with the low frequencies first. This will have an effect of "subtracting out" harmonic overtones from all the detected frequencies. The decoder can then output the 6 pitches with the largest corresponding amplitudes above a threshold value (TBD) in a similar format as before (2 8-bit buses per pitch representing MIDI value and intensity).



Onset/Offset Detector [Ishaan]

The onset/offset detector takes the input the pitch decoder and determines whether to assert or de-assert 46 single-bit output lines to the MIDI protocol layer. A note onset is detected if:

a) It is outputted by the pitch decoder and wasn't outputted in previous cycles or

b) Its amplitude is somewhat higher (threshold TBD) than its previous amplitude

A note offset is detected if its MIDI value is not present in the output of the pitch decoder when it had been in previous cycles.

The outputs of the Onset/Offset Detector will likely need to be de-glitched due to potential wide band frequency content of string strikes. This may be accomplished in a similar way to our button debouncing strategy.

This onset detection strategy is somewhat primitive and may not be as effective as anticipated. If more robust detection strategies are required, an Energy Novelty based approach[7] may be considered for use.

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 will be 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 will not be supported. The output will be 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 will closely resemble the following (from this source[8]).

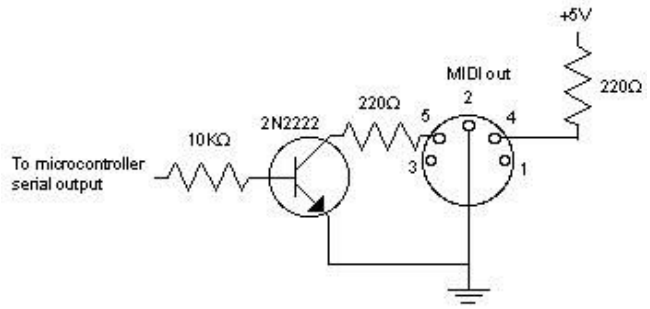


Fig x: MIDI Line Driver Schematic

This circuit will require inverted logic from the FPGA, which can easily be accomplished by inverting the output from the MIDI Protocol Module.

Testing

Hardware Conditioning: We will test our preamplifier using a scope from the lab, verifying our output is in the +/- 1V range and does not introduce significant noise.
Digital Low Pass and High Pass Filters: We will create test benches for these, referencing what we learned in Lab 5A.
IIR Verilog: The IIR Filter Block will first be prototyped in Python, using scipy.signal.iirpeak. Once we have a working Python implementation, we will use it to design our test bench in Verilog. After creating a Verilog test bench, we will proceed to design the IIR Filter Block in Verilog.
FFT Verilog: This is an IP block, so we will not test it in isolation.
Pitch Decoder: This will be prototyped in Python to verify compatibility with the IIR Filter Block (and potentially an FFT module). We will create a test bench based on this Python module.
Onset/Offset Detector: This will also be prototypes in Python and a test bench will be made in accordance with the proper behavior. Once we have modeled our testbench after the behavior we get from a fully functional Python implementation, we will begin to write the Verilog and debug.

MIDI: We will use an Arduino Microcontoller to prototype our MIDI communication. Once we are able to communicate with a MIDI compliant device using our Arduino, we will build a test bench based on this correct behavior. Once we have the test bench created, we will begin to write the Verilog and debug the MIDI module. We will need a MIDI to USB cable for this.

Required 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/