ECE 532 Digital Systems Design
Final Project Group Report

Audio Pitch Analyzer

Group 14
Daniel Strapp
Matthew Stewart
Adam Pietrewicz
Peishuo Cai

April 7th 2025

Contents

1. Overview

This Audio Pitch Analyzer project is a tool used for analyzing and visualizing audio of various instruments and other sounds. It aims to detect the fundamental frequency (ie. the note) in the audio by using a pitch detection algorithm based around analysis of what is known as the "Cepstrum" of the audio. The user can play an instrument into a microphone connected to the Nexys Video FPGA development board, or receive audio from an external device via the line-in port. The system provides 3 GUI displays on an HDMI monitor when audio is being played: an input audio waveform in the time domain, a live-time spectrogram of the audio, and finally a pitch graph display that shows what notes/pitches are being detected in the audio (also in live-time). These 3 displays can be selected using an external keyboard connected to the board. In addition, the user can play back the notes detected in the pitch graph as a digitally generated sequence of notes. The system connects to speakers through which the digitally generated sequence is played back.

1.1 Background

Analyzing input audio frequencies / pitches is a field with significant commercial and industrial applications; audio analysis software is a useful tool for musicians and producers. The problem of analyzing audio pitch can be treated as a digital signal processing problem, a widely developed area with many possible solutions that could be applicable. Of particular note is the Fast Fourier Transform (FFT) algorithm, which can convert the time-domain signal of the audio input into the frequency domain - a clearly useful operation when our goal is to detect the fundamental frequency. This is leveraged as part of our solution.

1.2 Motivation

Our group was interested in tackling an audio/video related digital systems project because it is very interactive and engaging for users. We also wanted to learn about how to interact with the audio input and HDMI video interface using Verilog on the Nexys Video FPGA as these are very widely used peripherals in electronic systems. In addition, we were curious to explore different frequency/note detection algorithms and test to see which is most effective for our application. Using an FFT in Verilog was our starting point for determining the frequency of audio data, however we discovered a cepstrum-based algorithm which is a "nonlinear signal processing technique with a variety of applications in areas such as speech and image processing."[1] Therefore we did research into this and experimented to develop a note detection algorithm that works well with microphone input data and also songs.

[1] https://www.mathworks.com/help/signal/ug/cepstrum-analysis.html

1.3 Goals

The functional requirements for this project were defined as follows:

1. **Audio Input Acquisition:** Read line-in microphone input using the Nexys Video board's Audio Codec (ADAU1761)
2. **Pitch Detection Algorithm:** Implement a custom Verilog IP block that determines the pitch of the audio input
3. **Real-Time HDMI Displays:**
    a. Display time-domain waveform of audio input
    b. Display frequency-domain spectrogram of audio input
    c. Display the detected musical notes on a pitch graph where the x-axis is time, and the y-axis represents pitch, ie. musical notes (C, C#, D, etc) in ascending order of frequency from the bottom to the top
4. **Audio Playback:**
    a. Playback the audio input through speakers connected to the Nexys Video board
    b. Playback an auto generated audio output of the detected notes on the pitch graph

1.4 Block Diagram

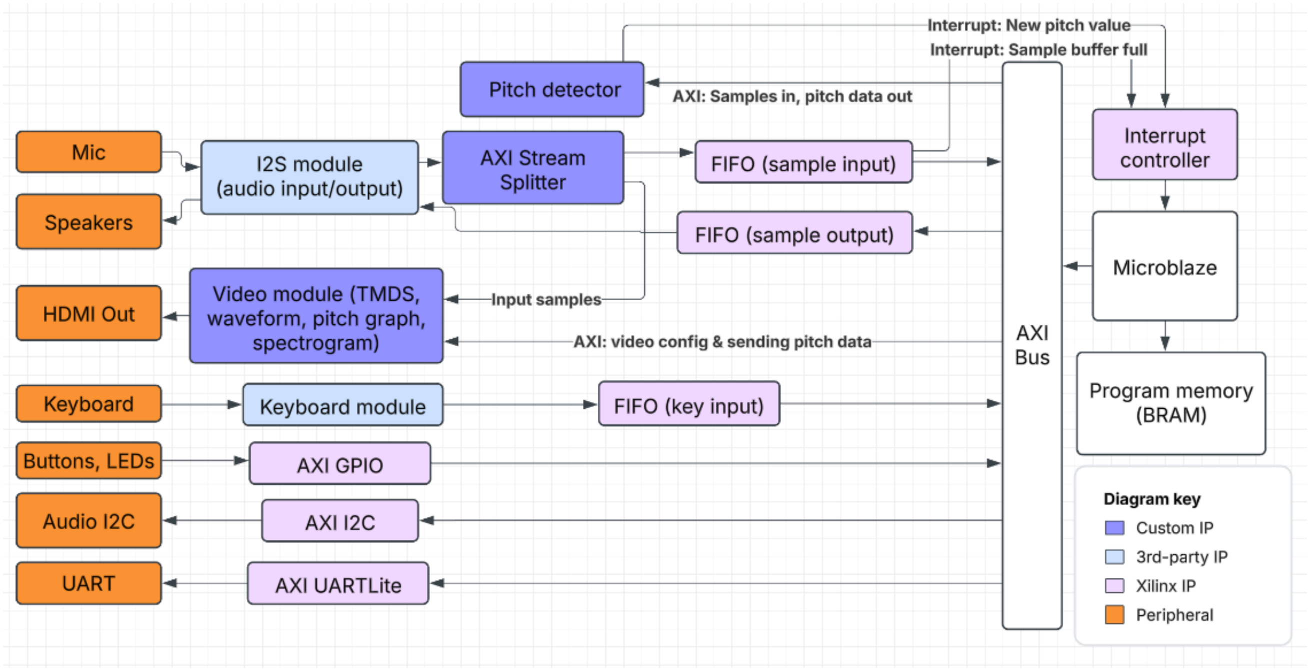

Figure 1: Final System Block Diagram

1.5 IP Overview

Component	**Description**
Video Module	Includes HDMI controller and 3 Verilog modules that produce the GUI displays
Pitch Detector	The Cepstrum-based pitch detector that processes audio input data and outputs the detected pitch
AXI Stream Splitter	Splits one AXI Stream input into two AXI Stream outputs
I2S Module	Used to read/write audio data from/to the microphone and speakers through the audio codec
Keyboard Module	Receives and reads the keycodes from the keyboard and sends them to a FIFO that is connected to the AXI bus
AXI Stream FIFO	Converts an AXI Stream interface to a memory-mapped AXI interface that can communicate with the Microblaze
Microblaze	Xilinx soft processor that controls the system. Handles keyboard presses, selects video module display, passes audio samples to pitch detection algorithm, records detected notes
AXI Uartlite	Connects microblaze program output to UART port
AXI GPIO	Memory mapped interface for GPIO signals (LEDs, onboard buttons)
AXI I2C Module	Used to interact and configure the Audio Codec using the I2C bus

Table 1: Description of IP components of the Audio Pitch Analyzer

2. Outcome

Our initial project proposal was to design an audio pitch analyzer with different modes of output so that a user can tune their instrument. We wanted to create a tool that is capable of real-time analysis of audio feed, instead of the widely available single note detection. The original GUI display proposed was just the detected note/frequency to be printed on the HDMI monitor as the user tunes their instrument. The spectrogram and waveform of the audio input were proposed as additional GUI features. Other stretch goals included creating a GUI guitar tuner display where a moving needle shows how close or far off the strummed string is from the nearest note, and also saving recorded audio to a file.

For our final project results, we were able to meet the MVP requirements which consist of:

- Use the audio codec to stream in audio samples
- Have a note detection module that analyzes the audio sample and calculate the dominant note
- Display detected note on a HDMI monitor
- Playback the detected notes as a digitally generated audio

On top of these requirements, we also implemented our GUI mode stretch goals:

- A real time spectrogram and waveform display of the audio sample
- Have a note detection display (pitch graph) with playback

However, for our final product, we pivoted away from the guitar tuner idea, so that we could focus on these more interesting graphical display modes. Moreover, we added a USB keyboard to interact with the microblaze and switch between operation and display modes.

Overall, our project works well with clean input audio from controlled sources (direct input from computer audio line-out), smooth HDMI live waveform, spectrogram and pitch graph display of the audio input, and is able to successfully detect notes within a reasonable range. There are a couple of improvements that could make the project even better. Firstly, the mic-in audio, although supported by our system, is too quiet for our note detection algorithm and has to be amplified in software. This could be worked on further and improved upon. Secondly, our different operation modes function best with different amplitudes of audio. For example, the note detector and playback function requires a very high amplitude, while the spectrogram requires a lower amplitude to properly display the notes within the scale on the screen. This could be improved by adding a fixed coefficient on each detection/display mode to better fit the input audio to the model being used, since the current solution is manually tuning the amplitude gain through our keyboard interface. Lastly, due to the nature of our note detection algorithm (Cepstrum), we are more accurate when detecting notes from a string instrument, since the algorithm is focused on harmonic octaves.

If we could start over, one thing we would want to do is utilize more online resources earlier on during our project. Our initial goals of having the audio I2C completely written in Verilog was not a good use of our time, given the scope of our project. Had we started with a microblaze approach by following the demos and tutorials online, we would have 1-2 extra weeks to work on polishing our system and working towards our stretch goals. Another aspect is to not use the Zed board for our development process. We tried to split up the work between the Nexys Video board and the Zed board, however, due to the inherent differences between the two boards, implementations that worked on the Video board would fail on the Zed board, and a lot of time was dedicated to figuring out the differences between the two hardware platforms. If we were to start over, it would be beneficial to have two of the same boards, or to have had more in-person work sessions towards the start of the term, where each person's development can be tested on the same hardware platform.

If someone were to take over our project from the current stage, we would suggest the next step is to work on the microphone input mode. With satisfactory analysis and display features, our project can be functionally complete as a product if the microphone input mode was working as well as the line-in mode. We would also suggest working on tuning the different analysis modes with preset amplitude gains to make the product functional without manual tuning between mode switches.

3. Project Schedule

Below is a table that compares our initial weekly milestone goals to what was actually worked on and completed.

Original Projected Milestones	**Actual Accomplished Work**
Milestone 1: ● Research pitch detection algorithms ● Research HDMI output ● Research ADC for audio line-in	● Researched Fast Fourier Transform (FFT) and hardware implementations ● Created python script used for generating .wav files of single music notes for later testing ● Researched HDMI technical specifications and ensured that FPGA is capable of meeting video data processing requirement ● Looked into how to read audio data from Codec. Ran an existing Vivado audio project to verify correctness of data flow, audio output and LEDs did not work
Milestone 2: ● Analyzer: Pitch detect basic sine waves (FFT IP) ● HDMI: Generate a blank 640x480 hdmi output ● ADC: Output sampled audio at desired rate	● Wrote python scripts testing basic FFT and AMDF-based pitch detection ● Researched how to use Xilinx FFT IP in Vivado ● Tried to get an HDMI demo project working on the Zedboard, faced issues due to board compatibility (Video board vs ZedBoard) ● Further debugging and testing the online audio demo for the Nexys Video board
Milestone 3: ● Analyzer: Detect more complex sounds, ie. musical instruments (still a single note) ● UI: Display the note as text	● Implemented Verilog modules for two pitch detector candidates (Cepstrum-based and FFT-based), wrote testbenches ● Created semi-custom IP that displays graphic images using HDMI by displaying image stored in BRAM ● Got audio recording and playback demo project to

● Low-pass filter (if needed) ● Microblaze: Record a sequence of notes from the Analyzer ● Integration: Audio capture, analyzer, text showing note on HDMI display	work on the Zedboard after debugging with ILA. Added I2C module to configure audio codec
Milestone 4: ● Analyzer: Detect chords ● UI: Additional display modes (toggled by buttons or keyboard) ● UI: Show current waveform ● UI: Show history of played notes	● Added AXI ports to Cepstrum pitch analyzer module, integrated into project ● Began integration of pitch detection, microphone input and HDMI module ● Began research and work on integrating the keyboard using the Zedboard ● Added framebuffer to HDMI module ● Further testing and integration of audio input and output on the Zedboard. Able to record and playback 0.1 second of audio using pushbuttons
Milestone 5: ● UI: Spectrogram (FFT output) ● Audio playback feature ● Save recorded note sequence to file on SD card	● Added interrupt signal to Cepstrum module when output is ready ● Further work on getting Zedboard to read and detect keyboard presses ● Began working on pitch graph display Verilog module and testbench that simulates the HDMI controller and outputs frames to file ● Development of audio playback of notes detected and displayed on pitch graph (basic sine wave audio)
Milestone 6: ● Integration: All features working on the board (analyzer improvements, UI modes, playback, recording)	● Audio waveform and pitch graph display working, and can toggle between the two ● Microphone and line-in audio input woking, added ability to increase/decrease volume of audio input ● Successfully integrated keyboard to Microblaze system using existing USB Host Verilog module. Customized so it sends keycodes to AXI FIFO
Final Presentation:	● Got audio input waveform, spectrogram and pitch graph working smoother. For the pitch graph we removed any outlier data outputs from the pitch detection algorithm to clean up the display

Table 2: Projected Milestones versus Accomplished Tasks

The milestones developed for this project divided the work into three main sections: the audio pitch detection algorithm, the HDMI module and audio interface module. Each member of the team mainly focused on one of these components of the project, and near the end during the integration phase the roles became more hybrid and were delegated based on what was left to implement. Compartmentalizing the work like this was beneficial for the project development because it could be done in parallel to some extent, but the integration phase proved to be challenging and time consuming.

The initial defined milestones were ambitious in getting a basic version of the project components working (pitch algorithm, HDMI, audio) as quickly as possible in order to then have time to integrate and iteratively improve them. The team also planned to have a final version of the project working by milestone 6, however due to many unforeseen challenges the team lost a buffer week and had to finish integration on the week before the presentation. This was due to the challenge of interfacing with the HDMI monitor and audio Codec. The team did research and explored online sources and projects to see how to interface with these modules but getting them to work was not easy. Design changes had to be made to existing HDMI modules as we initially attempted to make them read a framebuffer from BRAM, only to pivot to fully verilog-generated RGB output later, which cost us time. Next, getting the microphone to work was an early technical hurdle as the microphone audio in and speaker audio out was not functioning on the Nexys Video board based on an online Diligent demo project, resulting in significant levels of debugging for the audio demo Vivado project.

The milestones to get the audio and video working were clearly too ambitious to do in only a week, so the team kept working on each component and didn't lose focus. However, project integration was postponed due to this. The team found that a big challenge in development of the audio and video component of the project was the fact that we only had one Nexys Video board to work with. It was difficult to share the board between members because we didn't know when something would finally be working or how much more time was needed to debug. The addition of having the Zedboard as a second board was in the end not helpful for developing the project; the team met even more challenges in using it. For example, the constraints file and pin mappings were different from the Nexys Video and resulted in unique errors when synthesizing, implementing and generating the bitstream for the project.

In the end, focusing on developing modules only on the Nexys Video proved to be the correct choice as we could focus on integration rather than debugging the Zedboard. For example, we were not able to get the Zedboard to read keyboard keycodes due to errors in the provided constraints file, and instead scheduled extra team meetings to work on this together on the Nexys Video board. After the mid project demo (Milestone 4) the team focused on polishing core functionalities and working on some of our stretch goals, such as USB keyboard input for UI control and new HDMI displays such as the pitch graph and spectrogram.

Regarding the pitch detection algorithm work, significant research was done and multiple algorithms were tested in Python to learn and see which is most effective for our application. This testing and research phase was effective in deciding which algorithm we would use for the project (Cepstrum or basic FFT), and using python scripts to compare a software FFT versus the hardware Xilinx FFT IP helped us understand how to debug and develop the algorithms in Verilog.

Ultimately, the actual accomplishments reflect a shift toward stability and integration. While some proposal milestones were deferred or replaced, the iterative development process allowed the team to focus on integration, performance tuning, and UI refinement. The adaptive milestones prioritized project viability and integration, leading to a strong and functional final product.

4. Description of IP Blocks

4.1 Video Module (Custom IP)

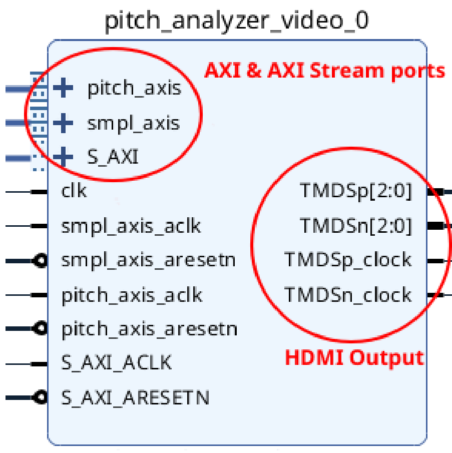

Figure 2: Custom Video Module IP Block that can switch between different GUI displays

This is imported as a single module in the block design, but it can be roughly divided into four submodules: the HDMI controller, and the three video modes. Also, it has an AXI slave interface which is used to configure the three video modes, as well as two AXI Stream interfaces (for audio sample data and pitch data, respectively) which are passed to the video submodules as needed.

4.1.1 HDMI Controller

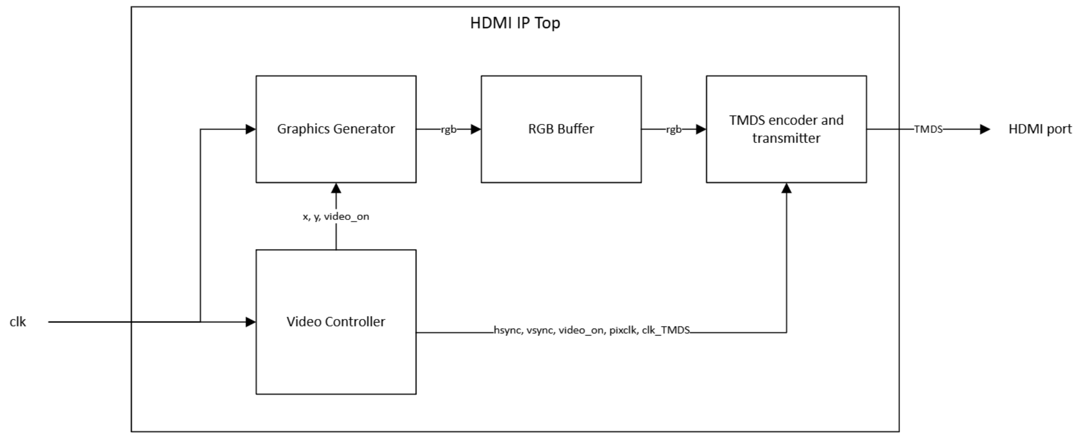

Figure 3: HDMI IP Block Diagram

The HDMI controller is a semi-custom IP adapted from Github User *Dom*'s port[2] of an HDMI guide from *fpga4fun.com*[3] into Vivado 2020.1, as well as Github User *FPGADude*'s VGA text generation project[4]. It is capable of outputting HDMI signals at 640x480p resolution at 60 Hz, with an 8bpp color depth.

The controller has three main components: a video controller, graphics generator, and TMDS encoder. The video controller works by taking in a 100 MHz clock input and generating two clocks using the Base Mixed Mode Clock Manager from Xilinx; a pixel clock and TMDS clock of 25 MHz and 250 MHz respectively. These clock frequencies are responsible for the final resolution and framerate of the HDMI output. The video controller additionally uses two counters to track the coordinates of the current pixel being drawn, counting up to 800 in X and 525 in Y. Required hsync and vsync signals are generated from the counters, as well as a video_on signal indicating the counters are within the visible range.

To determine the color for each pixel, the X, Y, and video_on signals are fed into the graphics generator block. The graphics generator block has three output modes: Waveform View, Pitch Graph, and Spectrogram. Depending on which mode is selected via a multiplexer, the block determines the red, green, and blue (RGB) values for the current pixel. These details are covered in the subsequent sections.

After passing the RGB signals through a simple flip-flop buffer, the final step is to encode the data in HDMI format. The TMDS encoder block receives the buffered RGB values, HDMI clocks, and sync signals. It encodes each RGB signal in 8b10b format using three encoder instances, then passes the data into three TMDS shift registers clocked at 250 MHz. The resulting signals are connected to the HDMI ports and sent outside the FPGA.

The code for the graphics generator is all custom, while the HDMI guide source project was modified to alter the clock frequencies for the desired resolution, and spliced in half to support adjustable graphics generation with custom RGB values. The VGA text generation project was transformed to support HDMI by splitting the RGB signals and the ASCII text generator was removed in favor of the graphics generator modules.

[2] https://github.com/dominic-meads/HDMI_FPGA/tree/master/HDMI_FPGA4fun
[3] https://www.fpga4fun.com/HDMI.html
[4] https://github.com/FPGADude/Digital-Design/tree/main/FPGA%20Projects/VGA%20Projects/VGA%20Text%20Generation

4.1.2 Video Mode: Waveform View

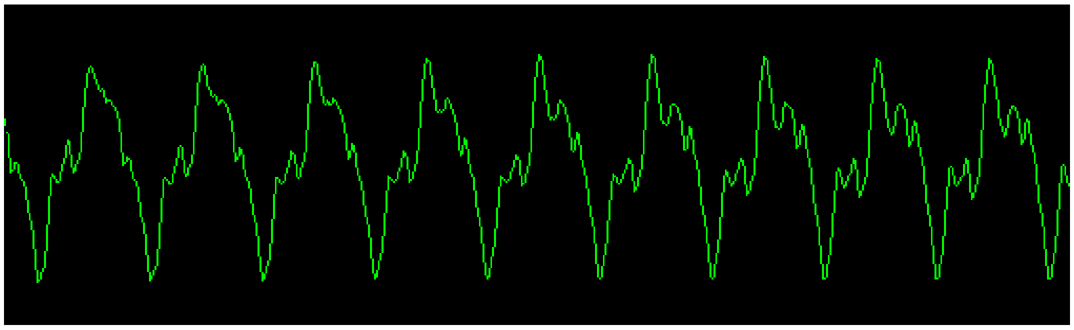

Figure 4: Input Audio Waveform Display

This module takes the 16-bit sample data from the AXI Stream port and displays the resulting waveform on the screen. Each sample is converted to a y-coordinate and stored in a buffer, which is referenced to determine which pixels to draw as green. It uses a ping-pong buffer (aka double-buffering) so that the incoming sample data does not distort the image while it is being rendered.

Since each buffer swap breaks the continuity of the waveform, it requires at least 640 samples of input (the horizontal resolution) each frame in order to display a continuous waveform. Fortunately, 39KHz / 60 = 650 samples/frame, which is just enough, as long as the data passes directly to the video module. (Originally, the microblaze would forward the sample data to the video module upon receiving it, but this was not fast enough to ensure 640 samples were received every frame.)

4.1.3 Video Mode: Pitch Graph

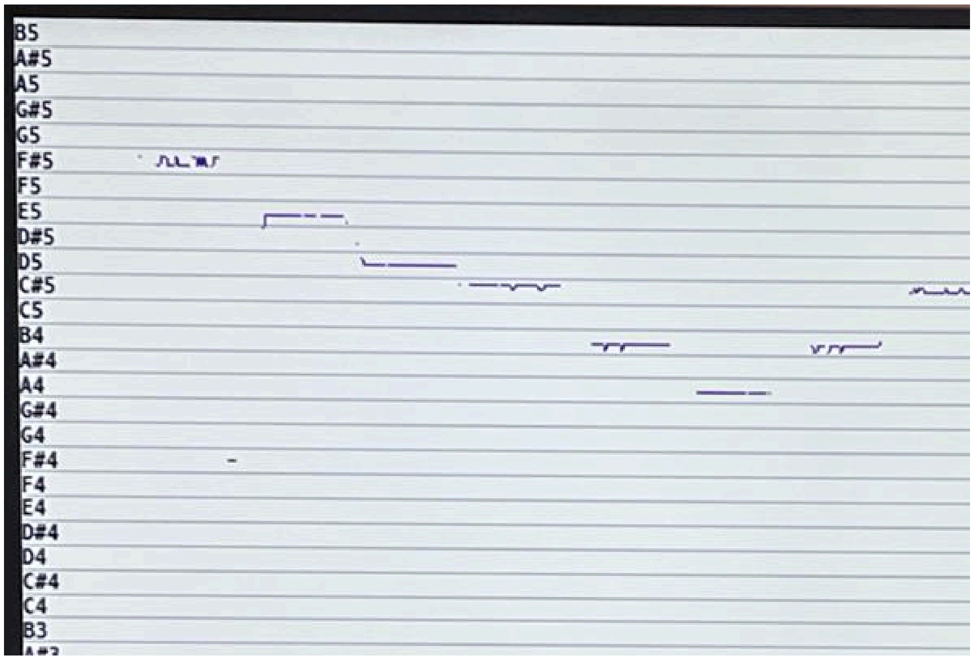

Figure 5: Pitch Graph Display with notes on the y-axis and time on the x-axis

The pitch graph demonstrates the accuracy (and sometimes lack thereof) of the pitch detector module. This mode is interactive - through AXI registers, the microblaze can start or reset pitch tracking, and update the playback cursor position. Since this does not handle audio playback, the microblaze must keep a separate copy of the pitch history in program memory.

The pitch data itself is received through a separate AXI Stream port as integer values (ie. 440 for 440Hz), which is also sent from the microblaze through an AXI Stream FIFO block. Pitches are plotted on a logarithmic scale, because otherwise each octave would be double the value of the previous; it is more intuitive and useful for octaves (and therefore semitones) to be a fixed distance apart instead. The logarithm function is implemented through a lookup table which produces 24.8 fixed-point numbers; this is then converted into a y-coordinate and stored in memory.

The note labels and the lines separating them are part of a background image which was generated using a python script. Each bar encapsulates the range of frequencies corresponding to that note. For example, a perfect "A4" would be 440Hz in the centre of the bar, but can range from ~428-453 Hz in this graph.

4.1.4 Video Mode: Spectrogram

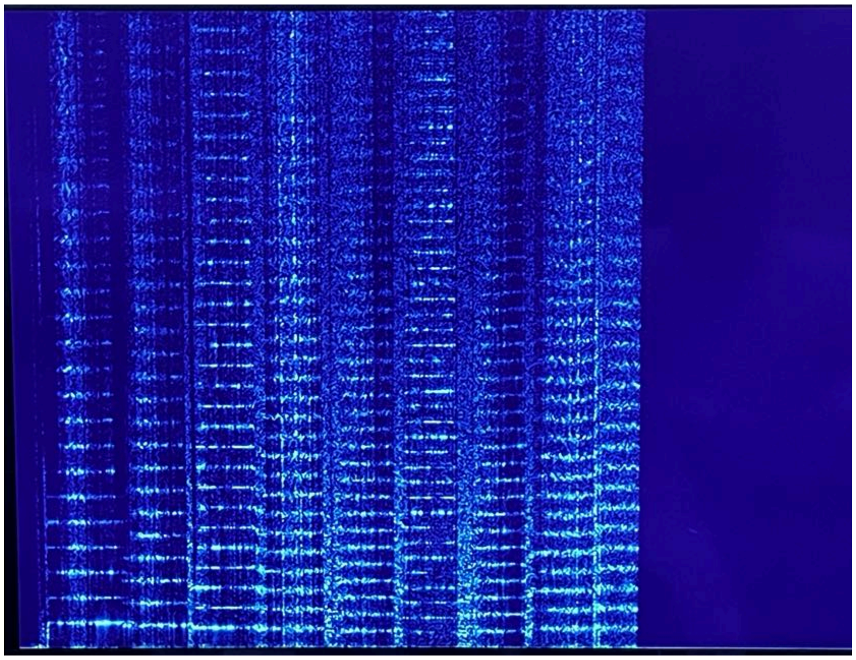

Figure 6: Spectrogram Display with frequency on the y-axis and time on the x-axis

The spectrogram shows the breakdown of the frequency spectrum of the audio. The x-axis represents time, the y-axis represents frequency, and the intensity of a pixel represents the strength of that frequency at that moment of time. In essence, each vertical column of pixels represents the output of an FFT. It allows us to see the harmonics of a sound (multiples of the fundamental frequency) very clearly.

This uses an xFFT IP block to calculate the FFT of blocks of 1024 samples at a time. This xFFT is not shared with the pitch detector module, which is somewhat wasteful, considering that they operate on the same data. However, they are configured differently; unlike the pitch detector's xFFT module which accepts floating point values - useful for the multi-stage processing involved - this version accepts fixed point values. This does save us the hassle of needing to convert floating point numbers back to integers in order to calculate a pixel intensity value.

The pixel "intensity" values are mapped to colors using the "viridis" color map, as this seems easier for the human eye to interpret than simply converting the data linearly to RGB values. To keep BRAM usage manageable, each pixel is stored as a 4-bit intensity value which indexes the color map.

4.1.5 Video Testbench

To help develop the above video modes, we created a testbench which simulates the HDMI controller at a high level and outputs the result of a single frame to a file. On each cycle where a pixel should be rendered, it writes the RGB value of that pixel to the file. The resulting ".rgb" file has no metadata, but can be converted to a PNG file using ImageMagick:

```
convert -size 640x480 -depth 8 gfx-00.rgb gfx-00.png
```

This saved us an enormous amount of time by sparing us the need to run synthesis and implementation when testing the video code, and also freed up the board for other team members to use.

After adding the spectrogram with its xFFT module, the speed of the testbench slowed to a crawl - problematic when waiting for a full frame to be rendered. To counteract this, after the FFT finished working, we would disable the FFT clock by using the "force constant" feature to set it to 0, which brought the simulation back to a tolerable speed.

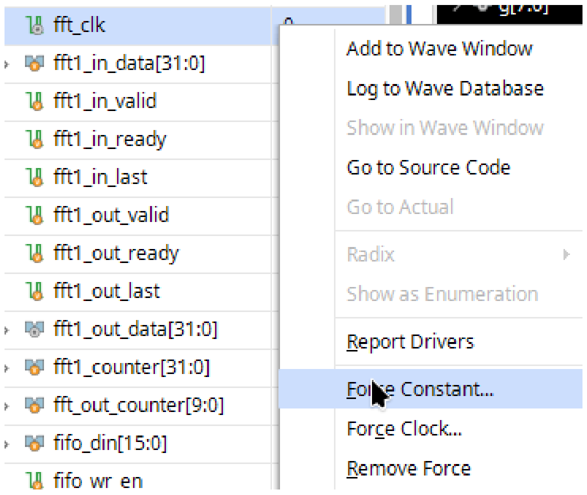

Figure 7: Setting the FFT clock signal to 0 using Force Constant in Vivado

We've made a simplified, standalone version of this testbench available in our repository, which could be helpful for anyone who wants to implement sophisticated graphics in Verilog.[5]

[5] https://github.com/Stewmath/ECE532-Audio-Analyzer/blob/master/examples/video_testbench.sv

4.2 Pitch Detector (Custom IP)

The pitch detector module has an AXI Stream port (1) for receiving audio samples (sent from the microblaze), and an AXI port (2) for configuration and retrieving the pitch data. It also triggers an interrupt signal (4) when it has finished calculating a pitch, which can be de-asserted with an AXI register write. There is also an AXI stream output port (3) which used to help with debugging but is currently unused.

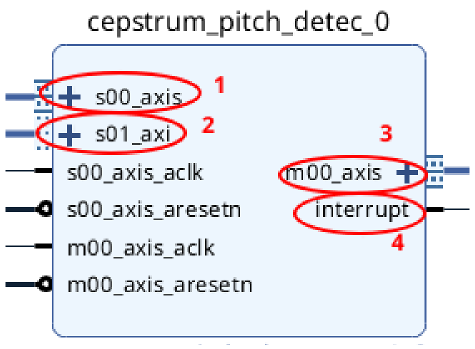

Figure 8: Cepstrum Audio Pitch Detection IP Block

It works by computing the "cepstrum" of each batch of 1024 samples. The cepstrum is computed in three steps:

1. Compute the FFT (discard the imaginary part)
2. Compute the log-magnitude
3. Compute the FFT again

The end result is a modified time-domain signal with peaks at points that have strong harmonics (bottom-left graph in Figure 9).

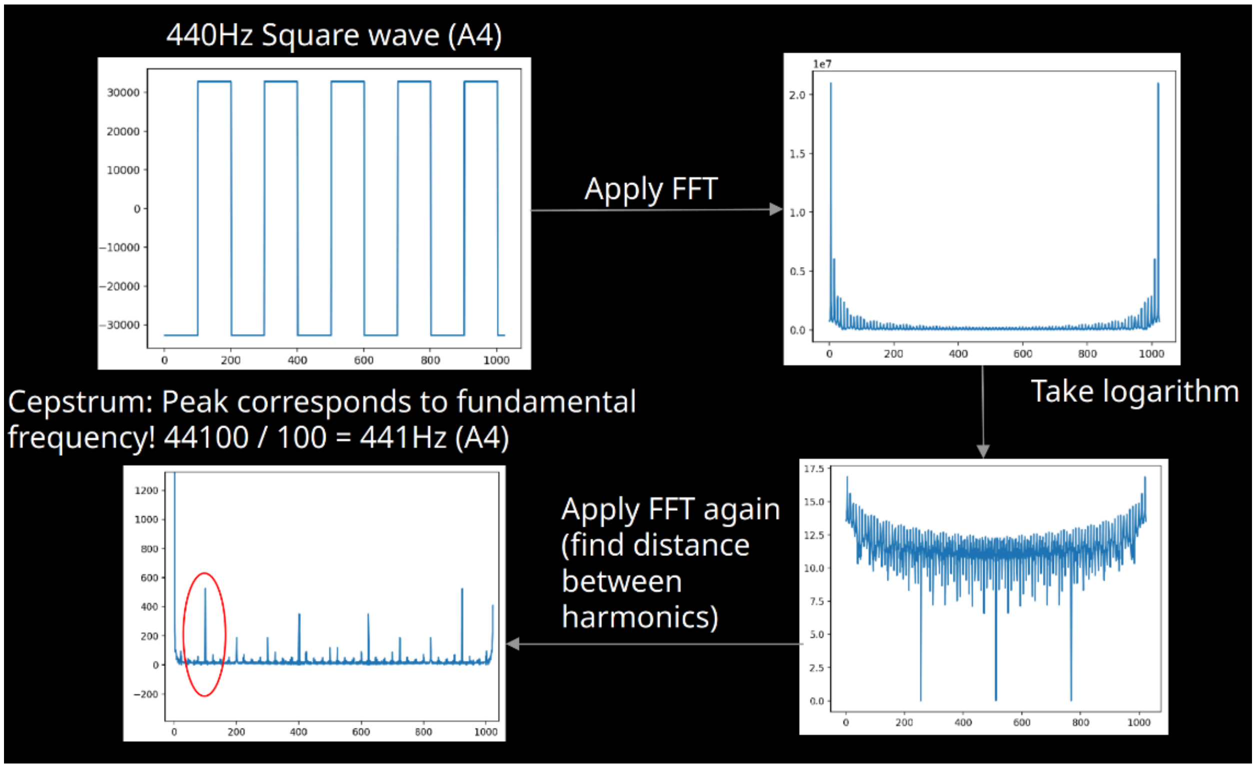

Figure 9: Cepstrum Algorithm steps in detail

If the fundamental frequency is 440 Hz, we would expect there to also be harmonics at 880 Hz, 1320 Hz, 1760 Hz, etc; so long as the sound is not a basic sine wave, these harmonics should exist to some extent. We can see them in the top-right figure (FFT output), but the harmonics are quite weak, which is why we take the logarithm to "normalize" these harmonics to have magnitudes which are close to one another (bottom-right figure). Finally, taking the FFT again produces peaks corresponding to the distance between these harmonics, ie. the fundamental frequency (or rather the period corresponding to the fundamental frequency).[6]

We found that this algorithm works quite well to detect the pitch of violins, but struggles with other sounds such as pianos and singing. We observed with the spectrogram viewer that violins had very strong harmonics compared to other instruments, which may explain why they worked well. It also depends on having a very clean input sound - mic input never produced reliable results, so we relied on the line-in port to provide very clean audio, such as MIDI keyboard audio[7] or concert recordings[8].

The Verilog implementation of this algorithm uses two xFFT IP blocks with floating-point inputs (the conversion to floating point is done on the microblaze before sending the samples). Vivado's "Floating Point" IP block is used three times - to perform "abs" and "log" calculations for computing the cepstrum, and "greater than" comparisons for finding the peak. When finished, the final computed pitch value is stored in a register that can be accessed over AXI, and the interrupt wire is set to high.

[6] For more information, this video contains a good introduction to cepstral analysis.
[7] We used https://virtualpiano.eu/; Works best with the "violin" instrument.
[8] We tested it with this recording of Pachelbel's Canon (violin solo).

We had a testbench (cepstrum_tb.v) which we used during development, but it is currently non-functional as the pitch detector is no longer sending anything over its output AXI Stream port for debugging. The testbench would output the cepstrum to a file to be graphed using matplotlib.

4.3 AXI Stream Splitter (Custom IP)

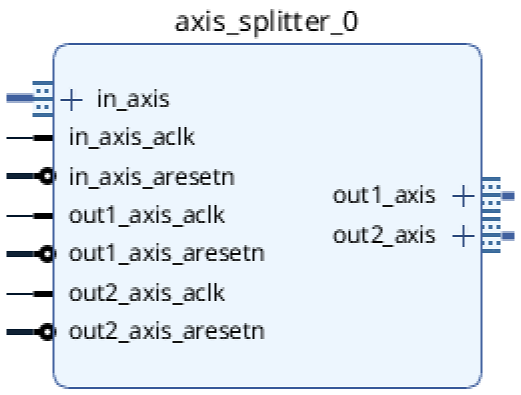

Figure 10: Custom AXI Stream Splitter IP Block

This module simply takes an AXI Stream as an input, and sends it to two AXI Stream outputs. It will stall if either of the two outputs are not ready. This is used to pass the mic/line-in data from the I2S module to both the video module and the Microblaze. Despite having separate clock inputs for each AXI Stream port, all ports must be driven by the same clock.

4.4 I2S Module (3rd-Party IP, Modified)

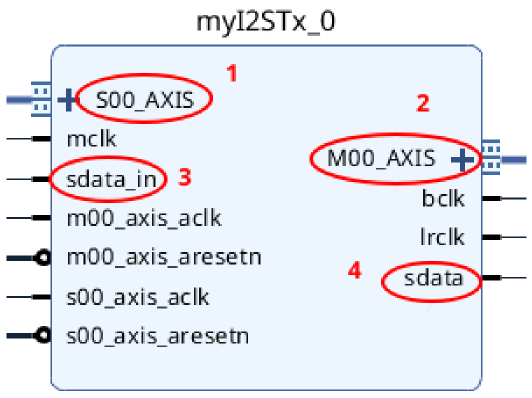

Figure 11: I2S IP Block

The I2S module converts 16-bit audio samples into an I2S stream, and vice-versa, interfacing with the Nexys Video's ADAU1761 codec. It was based on a tutorial found online originally meant for the ZedBoard[9]. The initial version supported only audio output by receiving samples over an AXI Stream

[9] Currently down but archived on the wayback machine. Original code here.

interface (1) and sending them over the sdata pin (4). We needed to modify this code to also process audio input from the sdata_in pin (3) and send it to another AXI Stream interface (2). The AXI Stream interfaces are 32 bits wide, with 16 bits each for left and right channels.

Assuming that the input "mclk" runs at 10MHz, this module drives "bclk" and "lrclk" such that the sample rate for both input and output audio is 39.06KHz. The sample rate could be coarsely adjusted by modifying the input clock speeds, but we did not attempt this.

The input audio sample stream (from the mic / line-in) is a bit buggy - the 16-bit audio sample must be shifted left by one, with the upper bit being discarded. We never got around to fixing the module, instead the consumers need to work around this problem. This does not apply to output samples.

4.5 Keyboard (3rd-Party IP, Modified)

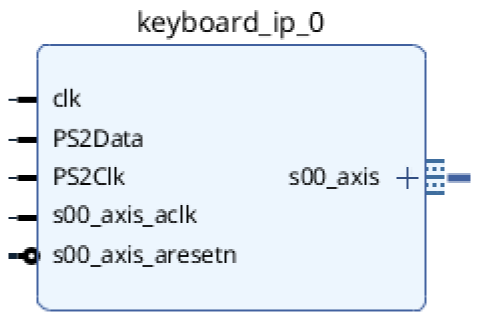

Figure 12: Modified Keyboard IP Block with AXI Interface

This module was based on Diligent's Nexys Video keyboard demo[10]. Since this module was purely verilog-based, there was initially no way to connect it to the Microblaze. So, we added an AXI Stream output port, which we then connected to the microblaze (again, through an AXI Stream FIFO IP block).

4.6 AXI Stream FIFO (Vivado IP)

This module effectively converts an AXI Stream interface to a memory-mapped AXI interface. We found this module very useful, as it allowed us to design modules with a very simple AXI Stream interface that can communicate with the Microblaze, rather than having to write a full memory-mapped AXI interface. Our project uses five instances of this module.

4.7 Microblaze (Vivado IP)

We use a microblaze soft-core processor for general control of the system - handling keyboard inputs, selecting which video mode to display, passing audio samples to the pitch detector, and recording the

[10] https://github.com/Digilent/Nexys-Video-Keyboard

history of pitches detected. It is configured with 1MB of BRAM for its memory, which is sufficient for this project; we did not need to use DDR memory.

The microblaze is also responsible for sending audio output samples to the I2S module through an AXI Stream FIFO block. Normally, it just forwards audio input samples unmodified to the output. However, when playing back recorded audio in the pitch graph, it instead generates square waves corresponding to the frequencies of the notes. This allows us to audibly confirm that the pitch graph is accurate.

The following Vivado IPs support the operation of the Microblaze:

- AXI Interrupt Controller
    - Concat (for multiple interrupts)
- Processor System Reset
- Microblaze Debug Module

4.8 AXI Uartlite (Vivado IP)

This module provides a simple UART interface over an AXI memory-mapped bus, enabling serial communication with minimal overhead. In our project, we instantiated this module once for communication with a host computer to implement debugging and data logging features via a terminal connection to the MicroBlaze. Its lightweight implementation and ease of integration made it an ideal choice for basic serial I/O.

4.9 AXI GPIO (Vivado IP)

This module offers a memory mapped interface for the General-Purpose input/output (GPIO) signals, allowing the microblaze to interact with onboard components. We used one of these blocks to control the LEDs to display the current amplification level for the audio input sample.

4.10 AXI IIC (Vivado IP)

This module provides us with a memory-mapped interface for I2C connection for the microblaze. This allows us to interact with the audio codec on the I2C bus and configure it to our desired operation mode. We use this module during the initial setup and when switching between mic-in and line-in audio. We found this module very convenient as it handles all the I2C protocol semantics. We attempted to implement this IP from scratch in Verilog but eventually landed on the existing Vivado IP version for its simplicity of use.

5. Design Tree

```
audio_analyzer
├── 3rd_party_files
│   └── myI2STx_v1_0.v
├── audio_analyzer.sdk
│   ├── design_1_wrapper.hdf
│   ├── design_1_wrapper_hw_platform_1 // Hardware platform
│   ├── hdmi_bsp_2 // Board support package (auto-generated)
│   └── main
│        ├── Debug
│        └── src  // C Source files are here
├── audio_analyzer.srcs
│   ├── constrs_1
│   │   └── imports
│   │       └── new
│   │       └── hdmi.xdc // Constraints file
│   └── sources_1
│     ├── bd
│     │   └── design_1 // Top-level block diagram
│     ├── imports
│     └── ip_repo // Generated files from IP blocks
├── audio_analyzer.xpr
├── examples
│   └── video_testbench.sv
├── ip_repo
│   ├── axis_splitter // AXI Stream splitter module
│   ├── cepstrum_ip   // Cepstrum pitch detector module
│   ├── keyboard_ip   // Keyboard input module
│   ├── myI2STx_1.0   // I2S module
│   └── video_system  // Video module
├── python_tools  // Miscellaneous python scripts
└── README.md
```

Top-level folders:

- `3rd_party_files`: Contains the original version of the I2S module that we found online.
- `audio_analyzer.sdk`: Contains C source files for the microblaze.
- `audio_analyzer.srcs`: Contains the Vivado project, consisting mostly of one large block design which uses custom IP blocks from the ip_repo directory.
- `examples`: Contains a standalone video testbench, based on the one in our "video_system" submodule, that can be used to help develop verilog-based graphics.

- `ip_repo`: Sources for the IP blocks that are imported into the main project. All of our custom Verilog code is here, along with the 3rd-party code we used.
- `python_tools`: Miscellaneous python scripts used for research and to support development of the Verilog modules.

5.1 C Source Files

The SDK directory at “audio_analyzer.sdk/main/src” contains a number of C files that were originally taken from the Nexys Video audio demo[11]:

- iic.c, iic.h: Controlling the IIC module.
- audio.c, audio.h: Using the IIC module to configure the ADAU1761 audio codec.
- intc.c, intc.h: Interrupt handling.

The following C files were written by us:

- main.c, main.h: Main program functionality is here.
- test_input.h: An array of audio samples used for testing the cepstrum pitch detector.

5.2 Python Scripts

- Pitch detection algorithms in software:
  - `detect_pitch_amdf.py`: AMDF-based pitch detector.
  - `detect_pitch_cepstrum.py`: Cepstrum-based pitch detector.
  - `detect_pitch_fft.py`: FFT-based pitch detector.
  - `pitch_tester.py`: Tries to guess the pitch of a .wav file using one of the above 3 algorithms.
  - `pitch_over_time.py`: Plots the pitch over time of a .wav file using one of the algorithms.
- For using the cepstrum module testbench:
  - `cepstrum_tb_gen_input.py`: Read a .wav file, generate a file for the cepstrum testbench to read as input (default: "/home/matthew/tb_input.txt")
  - `cepstrum_tb_plot.py`: Plots the output of the cepstrum testbench (default: "/home/matthew/tb_output.txt"), compares against a software-generated calculation.
- .mem file generators:
  - `gen_ln_table_mem.py`: Generates the ln table used by the pitch graph Verilog module.
  - `pitch_graph_background_gen.py`: Generates the background image used by the pitch graph Verilog module.
  - `wav_to_mem.py`: Converts a .wav file to a .mem file (16-bit signed samples) that could be imported for use in a testbench.
- Miscellaneous:

[11] https://github.com/Digilent/Nexys-Video-DMA

- `common.py`: Common stuff used by scripts.
- `filter.py`: Script used for testing the effect of filters on a .wav file.
- `sound_gen.py`: Generates .wav files of square or sine waves at a particular pitch.

6 Tips & Tricks

Check if there are existing IPs that can do what you need. Writing custom Verilog modules for well-known protocols is probably not the best use of your time (educational though it may be).

Make use of the AXI Stream FIFO IP. If your module needs to communicate with the Microblaze but doesn't need memory mapping, write a simple AXI Stream interface instead and use this module to connect it to the Microblaze. AXI Stream interfaces are also much easier to write testbenches for. (Of course, this works best for streams of data.)

7 Github & Video Links

Github: https://github.com/Stewmath/ECE532-Audio-Analyzer
Demo video: https://www.youtube.com/watch?v=KJVU2bFoxcI