ECE532 Final Project Report: Pitch Training System

April 7, 2025

Group 5:

Angus Wu, Grace Liu, Lucy Qu, Santiago Ibagon

1. Overview

1.1 Motivation

For musicians, being able to identify pitch is an important ability when playing or singing that allows them to recognize if they are in tune, but it can be a difficult skill to master. With our project, the user can train their own ear to learn how to identify pitches, as well as sing them accurately. We decided to do this project to gain experience with audio processing, and some of the techniques that are used for it in hardware. We also saw that its complexity could easily be modified depending on the difficulty we encountered over the course of working on it, by changing the peripherals and IP blocks used in the design. Additionally, it provided a balance between concepts that we were already familiar with, and those that we would have to learn more about.

1.2 Goals

The primary intended functionality of our design was to both successfully identify the pitch of a sound provided as input, and produce a tone with a given pitch as output. For this to work properly, we had to design an audio input pipeline that could take in external sound, transform it into its frequency spectrum, and consistently identify the dominant pitch present in the sound. It also required the design of an audio synthesizer that the user could easily control to play and hear the desired notes.

1.3 Project Repository

This project can be accessed via our GitHub repository. The demo video can also be accessed here (click me)!

1.4 Block Diagram

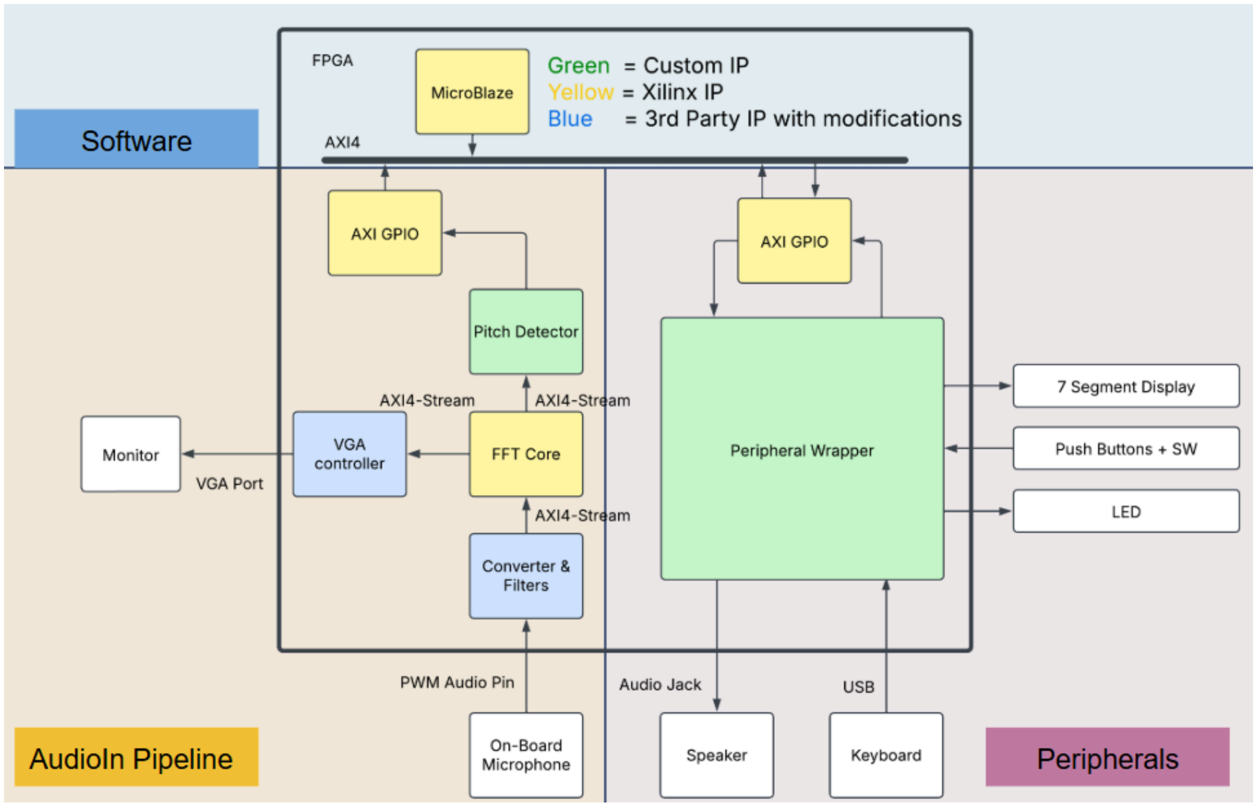

Figure 1. System Block Diagram

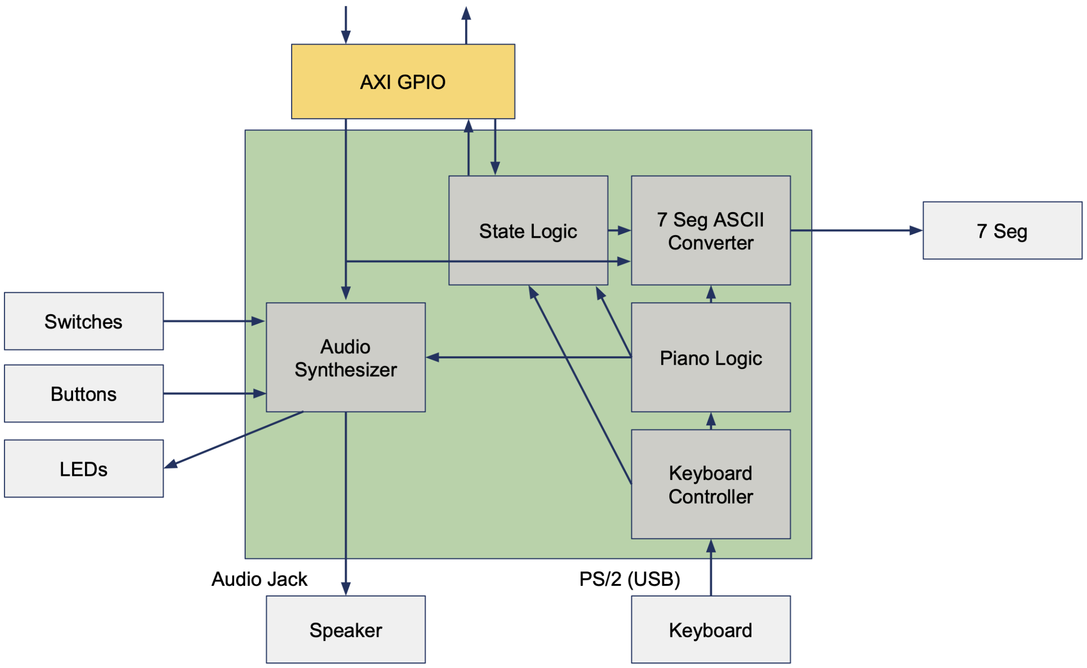

Figure 2. Peripherals Wrapper Block Diagram

1.5 Description of IP

The system is divided into three main segments: the MicroBlaze software, which manages the overall game logic; the audio input pipeline, which processes data from the onboard microphone; and the peripheral interface, which handles user interaction through peripheral display and input.

1.5.1 Microblaze

Microblaze is responsible for conducting note comparison, note generation, and state transition in progression of the system and input from hardware peripherals. There are three operating modes, home screen, ear-training, and free-play.

- Homescreen:
  - Enter this mode if “Q” was pressed on the keyboard.
- Ear-training:
  - Enter this mode if “1” was pressed on the keyboard from Homescreen.
  - Pseudo-random generator generates randomly the number of notes to include in the output chord (1-3) and the notes in the chord.
  - Output the chord to the audio jack.
  - Takes user input from the keyboard piano and compares it with the chord.
  - Outputs the result to 7 segments to showcase if user inputs match the chord.
- Free-play:
  - Enter this mode if “2” was pressed on the keyboard from Homescreen.
  - Receives bin number from the audio pipeline and keyboard piano.
  - Conduct real-time note comparison whenever the user sings.
  - Output comparison results to 7 segments display to show if they are matched.

1.5.2 Audio Input Pipeline

1.5.2.1 Converter and Filters

The Audio Pipeline receives Pulse-Density Modulation (PDM) audio signal from the onboard microphone. The PDM signals are first converted into Pulse Code Modulation (PCM) signals that are compatible with FFT IP core input. They are then passed through a series of filters that remove noise and reduced the sampling rate to 48 kHz. This block was developed based on a third-party IP from a reference project, which was originally designed for speech frequency detection [1]. It was developed in Vivado 2014.2, and filters IP cores were provided as netlist files (.ngc), which are no longer compatible with current Windows systems. Therefore, the filters are regenerated based on our understanding of their intended functionalities.

1.5.2.2 FFT Core

The FFT core processes the filtered and downsampled audio input in the time domain and converts it into the frequency domain. It collects 16384 time domain samples and generates 16384 frequency bins, each with a resolution of 2.9297 Hz. From these, 128 bins, ranging from 70.31 Hz (bin 24) to 444.63 Hz, are selected to cover the vocal range of the group members and are forwarded to the Pitch Detection and VGA controller modules.

1.5.2.3 VGA Controller

The VGA controller displays the system title, “Pitch Training,” in the upper half of the screen and the amplitude (power) of frequency bins dynamically in the lower half of the screen. The title is rendered using character data stored in ROM. Frequency data is displayed by continuously reading the FFT output from a dual port BRAM at 25 MHz VGA clock, while the data is written at 100 MHz. Majority of the frequency bins are displayed in green while the closest bin to the notes from D2 to A4 is shown in blue. The VGA controller is based on a third party IP from the same reference project used for the Converter and Filters. While the original IP supported dynamic frequency display, it was modified to also display the title and specific frequency bins ranges relevant to this project.

1.5.2.4 Pitch Detector

The pitch detection system receives the audio spectrum data from the FFT Core, then using a rolling average to stabilize the potentially unstable data it receives, it outputs the detected dominant frequency, which it sends to the MicroBlaze through a AXI GPIO IP Block for processing. This block is an entirely custom design and implementation.

1.5.3 Peripherals Wrapper

1.5.3.1 Keyboard Controller

The keyboard controller converts PS/2 keyboard input into a state map that shows which keys are pressed simultaneously. For the engineer’s convenience, this IP block outputs both the PS/2 keycode and the ASCII code equivalent of the most recent key press. To integrate into our device, this controller is extended with a “piano logic” that maps a piano-like key layout onto a computer keyboard. The keyboard controller also interfaces with FPGA state logic to convert key presses of non-piano keys into the preset UI control signals, which are sent back to the microblaze for device control and operation.

1.5.3.2 Audio Synthesizer

The audio synthesizer generates PWM audio output required for the audio jack. As the scope of this project is constrained to the notes playable on a piano, the custom IP block is responsible for generating sound of frequencies provided by the user’s piano input (a piano layout simulated on a computer keyboard) in free play mode, and notes generated by the microblaze in ear training mode. During free play mode, the user can play multiple notes simultaneously to sound out a chord. Augmenting the key functionality, additional inputs can be configured to change the audio volume, as well as the octave at which the piano notes are played.

1.5.3.3 7 Segment Display Controller

The 7 segment display controller facilitates the display of the key user interface (UI) components on the array of 8 displays available on the Nexys 4 DDR. An upstream device would input its registered ASCII code signals and character positioning, then the controller converts the ASCII codes into a displayable 7 segment representation. The 7 segment display UI contains 4 main elements crucial for device operation, from left to right they consist of: (i) the name of the note sent from the Microblaze, (ii) the name of the note sent from the piano, (iii) the correctly of note comparison from the Microblaze, and (iv) the operational mode the user is currently using.

1.6 Project Complexity

Description	Complexity Points
Software State Machine and Note Comparison Logic	0.2
Software Pseudo random note generation	0.25
Custom IP (peripherals, audio in pipeline, and pitch detection)	0.50
Onboard Microphone	0.50
On-board audio output port	0.50
7 Segment Display	0.20
VGA output without MicroBlaze involvement	1.00
Visualize meaningful results with a GUI	0.75
USB keyboard implementation	0.50
Total	4.4

Table 1: Complexity Point Table

2. Outcome

2.1 Results

Both the audio input and output pipelines worked as expected, allowing us to integrate them into the free play and ear training modes, as we originally set out to do.

The system's audio processing was able to analyze the audio input and produce a resulting dominant pitch, although the output could be quite inconsistent at times. The quality of pitch detection was largely dependent on the background noise present during use, as the user's voice could be lost in the sound of other voices, making it difficult to identify a single highest peak.

The team initially considered the negative effects of electromagnetic interference from other onboard working peripherals on the onboard microphone's sound quality. As a result, the original system design incorporated the Pmod MIC3 as the microphone. However, this choice was later reconsidered after a third party reference project was integrated into the audio analysis pipeline. The reference project showcases the functionality of the onboard microphone as it provides satisfactory outputs. Given the project's tight time constraints, the Pmod MIC3 was ultimately excluded from the final design.

Despite this change of audio input source, the audio pipeline outputs a suitable result. When provided with a pure sine wave audio input played near the microphone, the system could consistently identify the frequency and maintain a stable output on the seven segment display. For the audio output, the keyboard–controlled synthesizer performed as we had set out, allowing the user to select a note, and playing sounds on any output connected to the onboard audio jack.

With both audio components working, we made use of them for the freeplay and ear training modes by having them communicate with the MicroBlaze processor, which was running the program control logic. The audio processing blocks also provided output on the GUI, both through a VGA display to show the input audio frequency spectrum, and the seven–segment display to show information about the detected note and the note being played on the keyboard.

2.2 Next Steps

Because of time constraints and the complexity of implementation, our system uses a fairly simple pitch detection algorithm and a potential starting point for any future steps could be to test a variety of other algorithms to determine which provides the best performance. As this is an active area of research, there are many candidates to be tested. Some involve frequency domain analysis such as Harmonic Product Spectrum [2], and cepstrum analysis [3], as well as some involve time domain analysis, such as zero-cross rate and autocorrelation [4].

With a new pitch detection system, there is also the possibility of implementing functionality to allow the system to perform pitch detection on instruments, allowing the system to be used as a guitar tuner, for instance. Along with this, an algorithm could be added to the system to distinguish between human voice and instrument sounds based on different timber, so that the system can switch modes accordingly.

Additionally, the main visual interface is implemented using the 7 segment display, while the VGA output is not interactive and does not reflect the state of the game. Currently, the VGA controller is only driven by the audio input pipeline. To build an interactive interface on VGA, the VGA controller also needs to be driven by the Microblaze or peripheral path. The 7 segment display is sufficient for the current modes and functionality supported by the system. However, if additional features mentioned earlier are implemented, we will also develop a more comprehensive visual interface on VGA display.

3. Project Schedule

3.1 Milestones & Discussion

This section presents a comparison between the originally proposed milestones and the actual progress achieved during the project. These comparisons highlight the unforeseen challenges encountered by the team, which will also be discussed in detail.

3.1.1. Proposed Milestone

Milestone	Objective	Outcome
1	Confirmed the scope and goal of the project. Detailed block diagram drawn	Confirmed choices on which existing/customized IP blocks should be used/created, communication protocols between each block, and which hardware peripherals on/off-board the project will be used.
2	Verify existing IP, prototype customized IP, and configuration of hardware peripherals	On the software side, FFT LogiCORE™ IP usability is tested. A suitable algorithm for pitch detection and audio synthesizer is selected. On the hardware side, Pmod MIC3 and audio jack should be configured.
3	Verify the functionality of software-customized IPs and configure hardware peripherals.	On the software side, subsystem integration occurs between FFT IP, Pmod MIC3, and audio jack. Customized algorithms, the pitch detector, and the audio synthesizer, should be tested in simulation. On the hardware side, the VGA display, 7-segment display, and the keyboard are configured.
4	Subsystem integration and finalization of customized IPs	On the software side, the customized algorithm block's design should be finalized after reflecting on the simulation results. On the hardware side: subsystem integration should be attempted between VGA, keyboard, and 7-segment display. The GUI to display on VGA should also be drawn.
5	Full system integration	The team would integrate the IP blocks in software with the working hardware pipeline, with user input fed into the system via microphone and keyboard and result of note comparison and mode selection output from

		VGA and 7-segment display.
6	Modify the prototype if needed	Reflecting on the progress of the last milestone, the prototype design may be modified for optimization or add extra features to the baseline model.

Table 2: Proposed Milestones Table

3.1.2 Actual Progression of Project

Below lists the team's actual progression in the project with achievements accomplished in the corresponding milestone.

Milestone	Objective	Accomplishment
1	Confirmed the scope and goal of the project. Detailed block diagram drawn	Discussed with our TA regarding the scope and potential challenges of the project. His suggestions helped the team to define the most urgent goals for next week.
2	Verify existing IP, prototype customized IP and configuration of hardware peripherals	Verification of existing IP: FFT LogiCORE™ IP functionality was verified via simulation (Lucy). Prototyping customized IP: Successful simulation of the harmonic product spectrum pitch detection algorithm(Santiago). Hardware configuration: ● Created a simple chord-playing design, output via the PWM audio jack (Angus). ● Attempted configuration of Pmod MIC3, however unsuccessful due to unknown library compilation error (Grace).
3	Software development and hardware configuration	Audio pipeline: ● Investigation of Microblaze, DMA, audio input and the FFT IP block integration and if integration of reference design is possible (Lucy). ● Verilog prototype of harmonic product spectrum pitch detection (Santiago). Hardware: ● Configure and integrate of keyboard and 7-segment display (Angus). ● Configuration of Pmod MIC3 and onboard microphone and validated functionality (Grace).

4	Software development and hardware configuration	Audio pipeline: ● Reconstruction of reference design to adapt to Vivado 2018.3 and modified the design for customized use of the project(Lucy). ● Verification of pitch detection via the 7-segment display (Santiago). Hardware: ● Subsystem integration between the keyboard, 7 segments, and audio output with different keys pressed corresponds to a unique sound output (Angus). ● Research data conversion from 12-bit ADC Pmod Mic3 to 1-bit PDM and clock domain crossing to enable Pmod MIC3 integration into the audio pipeline and create prototype referencing delta sigma modulation(Grace).
5	Subsystem integration - software and hardware	Audio pipeline: ● Integration of pitch detector into the working pipeline with skeleton-based on the reference design (Santiago). ● Further modification of the working audio pipeline to ensure smoother integration with the overall system (Lucy). ● Prototype MicroBlaze Finite State Machine(FSM) (Grace). Hardware: ● Package the audio synthesizer and keyboard piano as a module for integration and test the functionality via baseline FSM (Angus).
6	Full System Integration	Working on system integration.

Table 3: Actual Milestones Table

3.1.3 Evaluation of the Milestones

Throughout the progression of the project, milestones were frequently adjusted due to new discoveries regarding various aspects of the design, as well as challenges that emerged during exploration.

3.1.3.1 Integration of a Third Party Reference Project

Initially, the team assumed that the FFT IP block would produce clean and reliable frequency outputs, and the project milestones were planned around that assumption. However, both simulation and real-life testing revealed that the output was noisy and inconsistent.

This led the team to research how other projects addressed similar issues. A reference project [1] provided valuable insights, particularly in the use of various filters and the configuration of multiple clock domains across different filtering IP blocks. Understanding and integrating the strategies from this reference project became a major task in our workflow—representing a significant shift from our originally planned milestones.

3.1.3.2 Re-evaluating the Use of AXI Protocol

In the initial system design, the team planned to have nearly all IP blocks communicate with the MicroBlaze processor via the AXI protocol. However, as the project progressed and taking in insights from our TA, we realized that such extensive use of AXI was unnecessary and overly complex for our application.

As a result, we revised our architecture. Currently, the audio pipeline and the hardware peripheral pipeline were restructured to operate more independently. Each pipeline now performs internal processing autonomously and only transmits final results to the MicroBlaze, which is responsible for comparing the outputs and generating the final response. This shift significantly simplified system integration and improved performance.

4. Block Description

4.1 Microblaze FSM

The following section details the implementation of software finite state machines (FSMs) in the project.
The system uses a total of three FSMs: one high-level FSM that controls the current operating mode of the system, and two lower-level FSMs that manage the detailed procedures within specific modes.

4.1.1 Mode FSM

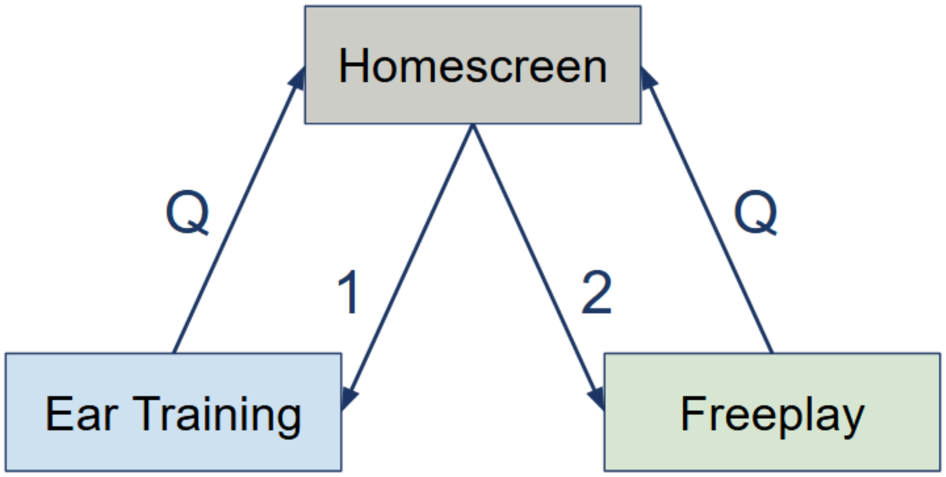

Figure 3. The Mode FSM

As shown in Figure 3, the Mode FSM controls the current operating mode of the system. There are three main modes in this project: *Homescreen*, *Ear Training*, and *Free Play*. The system starts in the Homescreen mode by default. Users can switch to other modes using keyboard inputs: pressing "1" enters Ear Training mode, and "2" enters Free Play mode. To change modes again, the user must first return to the Homescreen by pressing "Q", and then select the desired mode using "1" or "2".

4.1.2 Ear Training FSM

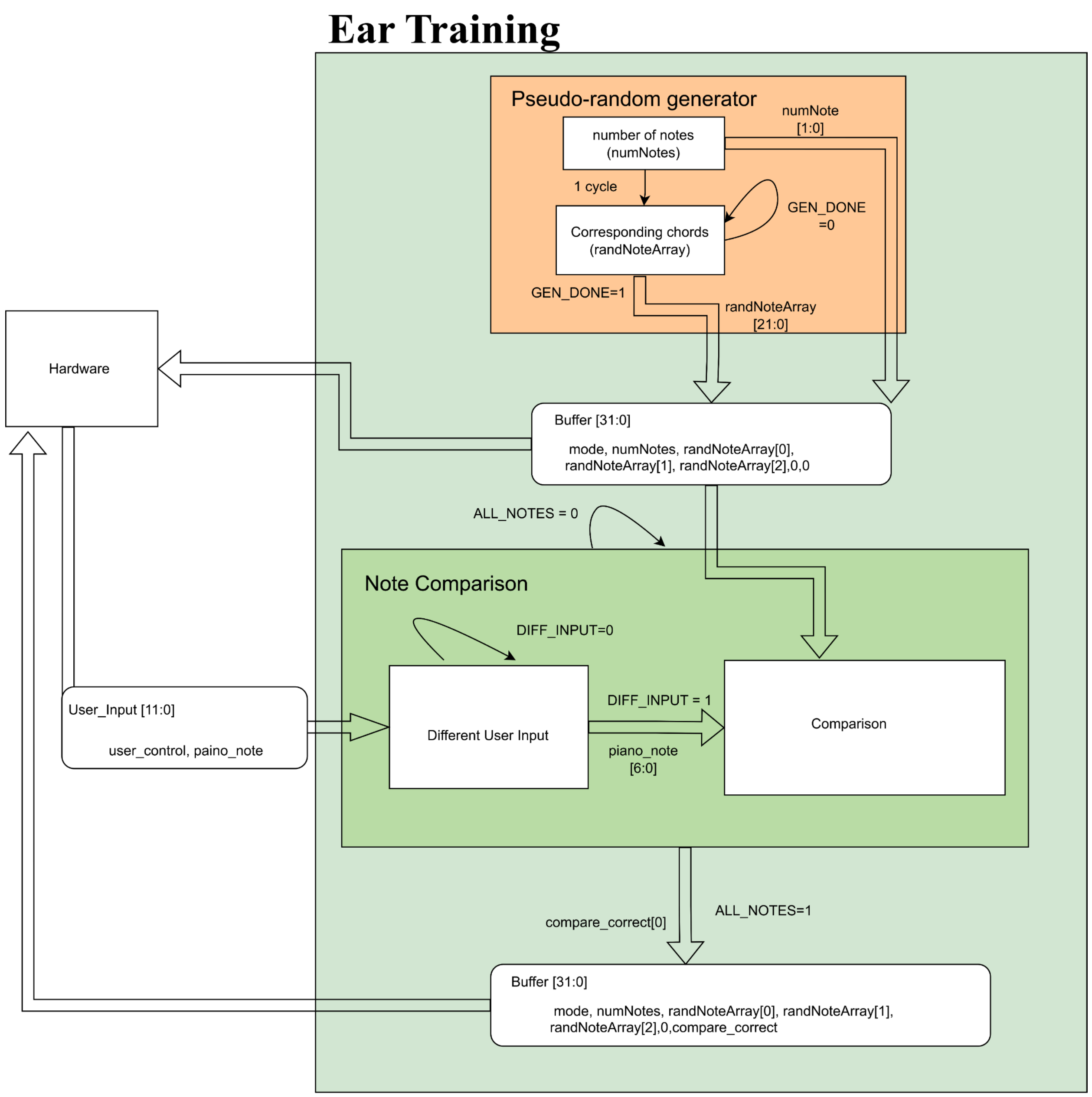

Figure 4. Ear Training FSM

After the user enters Ear Training mode, the system transitions into the Ear Training FSM.

This FSM is composed of several key components:

- **Pseudo-Random Number Generator (PRNG):**
  The PRNG is implemented using an XOR-shift algorithm [5]. It uses the user's previous input as a seed to generate random values. The generator determines both the number of notes in the chord and the specific notes to be played.

    - GEN_DONE**:** This signal indicates when the pseudo-random number generator should continue generating non-repeating notes, ensuring the total number of notes matches the target specified for the chord.

- **32-bit Buffer (Encoder for AXI GPIO):**
  The outputs from the PRNG (i.e., the number of notes and the specific notes) are loaded into a 31-bit buffer at designated positions. This buffer then forwards the data to the hardware peripherals. The precise placement of bits within the buffer allows the peripheral to decode the input correctly, display the mode on the 7 segment display and synthesize the corresponding sound via audio output.
    - The 32-bit input data buffer transmitting from the MicroBlaze processor to the hardware peripheral via AXI GPIO is structured as follows:
        - Bits 1–0: mode_sel – 2-bit value indicating the operation mode:
            - 00: Home
            - 01: Ear Training
            - 10: Free Play
        - Bits 3–2: play_note_num – 2-bit value representing the number of notes in the generated chord (maximum of 3) (used only in Ear Training).
        - Bits 9–3: play_note_id_0 – 7-bit value indicating the ID of the first note in the chord (used only in Ear Training).
        - Bits 16–10: play_note_id_1 – 7-bit value indicating the ID of the second note in the chord (used only in Ear Training).
        - Bits 23–17: play_note_id_2 – 7-bit value indicating the ID of the third note in the chord (used only in Ear Training).
        - Bits 29–24: sung_note_id – 6-bit value representing the ID of the note sung by the user (used only in Free Play).
        - Bit 31: compare_correct – 1-bit flag indicating whether the sung note matches all generated notes (used in both Ear Training and Free Play).
- **Note Comparison Block:**
  After hearing the synthesized chord, the user inputs their answer using the keyboard. These inputs are sent to the Note Comparison Block, which waits for the expected number of inputs and compares them against the originally generated chord. This determines whether the user has answered correctly.
    - DIFF_INPUT: This signal indicates whether the current user_input differs from the previous one. If the input is the same, it means the user has not entered a new note and the system should wait until a distinct input is received before sending it to the Comparison Block.
    - ALL_NOTES: This signal indicates whether all necessary comparisons between the user's inputs and the generated notes have been completed. Once all comparisons are done, the system outputs the result to the buffer.

  - The 12-bit output data sent from the hardware peripheral module to the MicroBlaze processor via AXI GPIO is structured as follows:
    - Bits 3–0: user_controls – 4-bit value representing user input controls via keyboard, indicating mode switching.

    - Bits 11–4: piano_note_id – 7-bit value indicating the ID of the most recently played piano note pressed on the keyboard.
- **32-bit Buffer (Encoder for AXI GPIO) :**
  When the system exits the Note Comparison stage, the results of the comparison—along with the previously stored input data—are written into the 32-bit buffer. This complete data package is then transmitted to the hardware peripheral, which visualizes the comparison results.

4.1.3 Free Play Mode

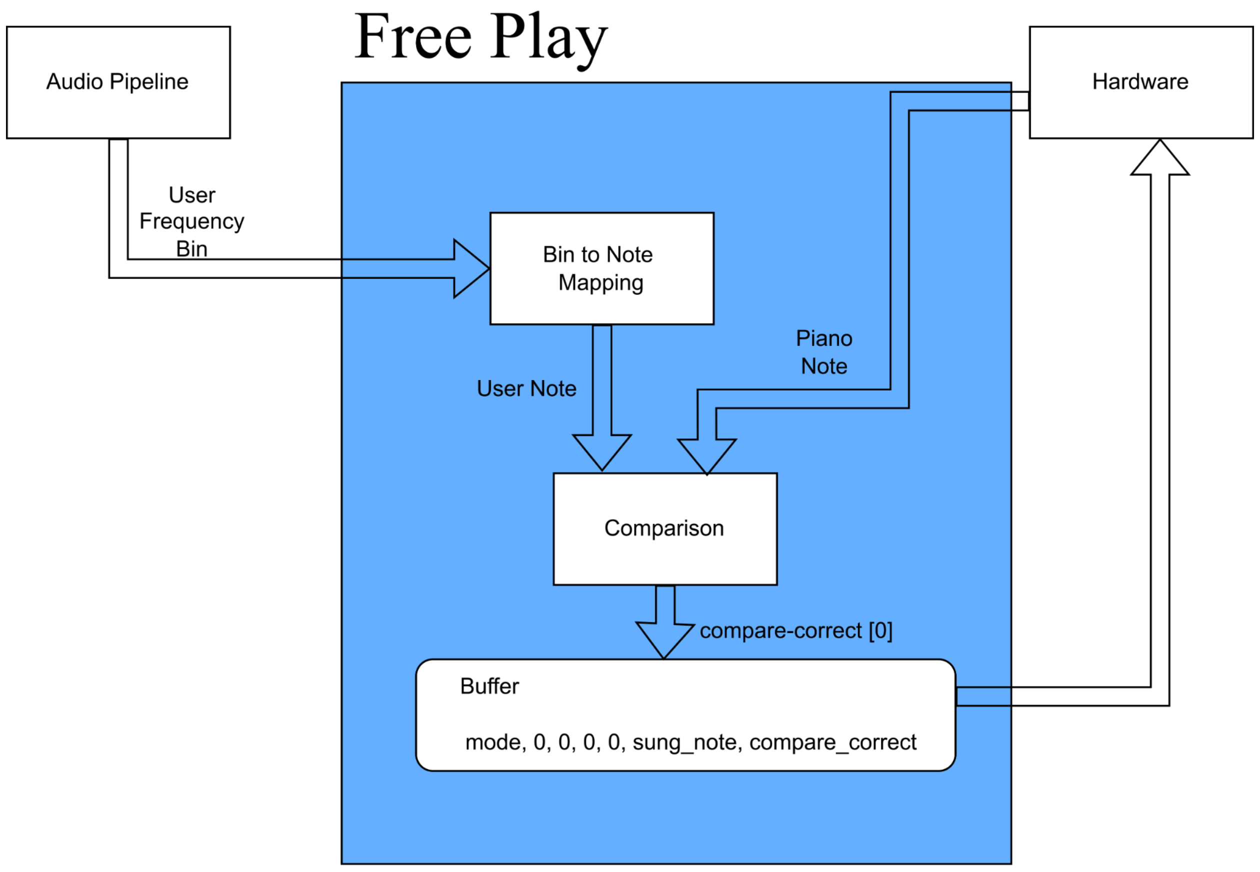

Figure 5. Free Play Mode's FSM

Once the system enters Free Play mode, this FSM takes over control. It processes piano note inputs from the hardware peripherals and real-time audio input from the audio pipeline.

This FSM consists of the following major components:

- **Bin to Note Mapping:**
  The input from the audio pipeline is a frequency bin number ranging from 34 to 94. This block maps the bin number to the closest corresponding piano note in the range of 33 to 44, representing the playable note range in the system. Bin numbers and piano notes outside such ranges are out of the scope of this project.

- **Comparison Block:**
  Taking both the hardware input note and the mapped audio note as inputs, the Comparison Block compares them and outputs a 1-bit signal, compare_correct, indicating whether the notes match.

- **32-bit Buffer (Encoder for AXI GPIO) :**
  After receiving the comparison result, the buffer packages the data, along with the user's sung note and hardware input, and transmits it to the hardware peripheral for decoding and visualization of the input and comparison result.

4.2 Audio Input Pipeline

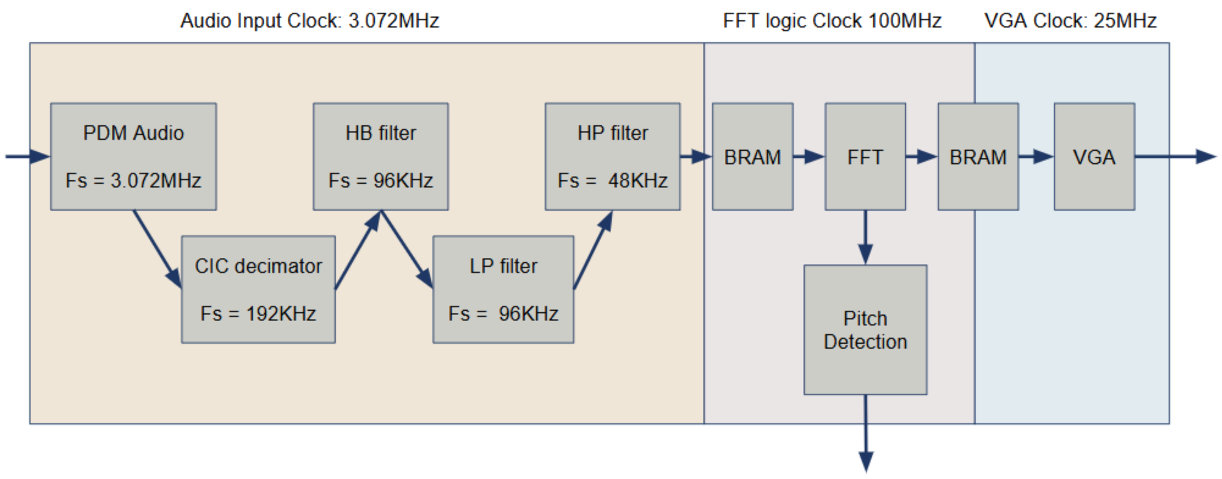

Figure 6. Block Diagram for Audio Input Pipeline

4.2.1 Converter and Filters

This module receives raw PDM signals from the onboard microphone and processes them through a series of filters to reduce noise and downsample the audio signal before passing it to the FFT core. All filter IP cores use the AXI4 Stream interface for both input and output, allowing them to be efficiently connected in sequence.

The raw PDM signal, sampled at 3.072 MHz, is first passed through the Vivado Cascaded Integrator Comb (CIC) Compiler IP. The CIC filter, commonly used for efficient multirate signal conversion and downsample [6], converts the PDM signal to a PCM format while downsampling by a factor of 16, resulting in a 192 kHz sampling rate.

Next, the PCM signal is fed into a half band filter, which further downsamples the audio to 96 kHz. This half band filter, implemented using the Vivado FIR Compiler IP, helps prevent aliasing during the downsampling process [7]. The corresponding frequency response is shown below.

Following the half band filter with decimation, the data are passed through an additional low pass filter to further eliminate high frequency noises. This filter is also implemented using Vivado FIR Compiler IP [7].

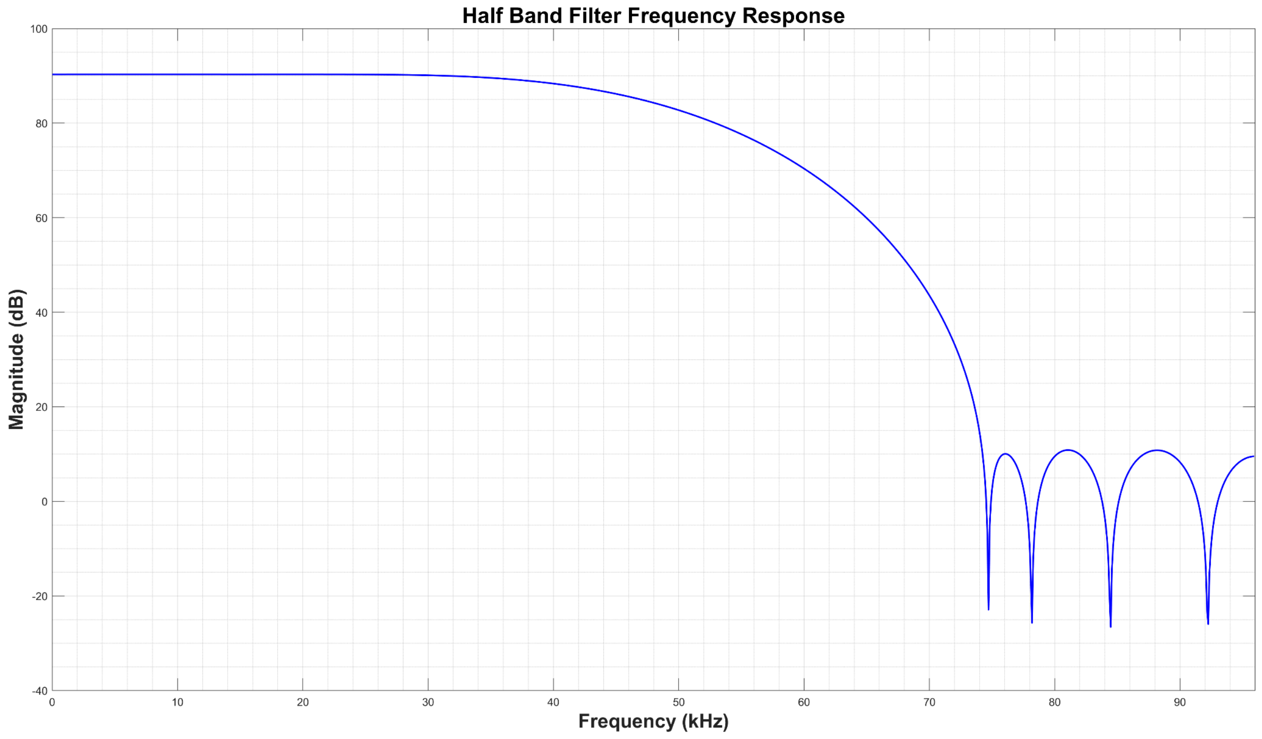

Figure 7. Frequency Response of Half Band Filter

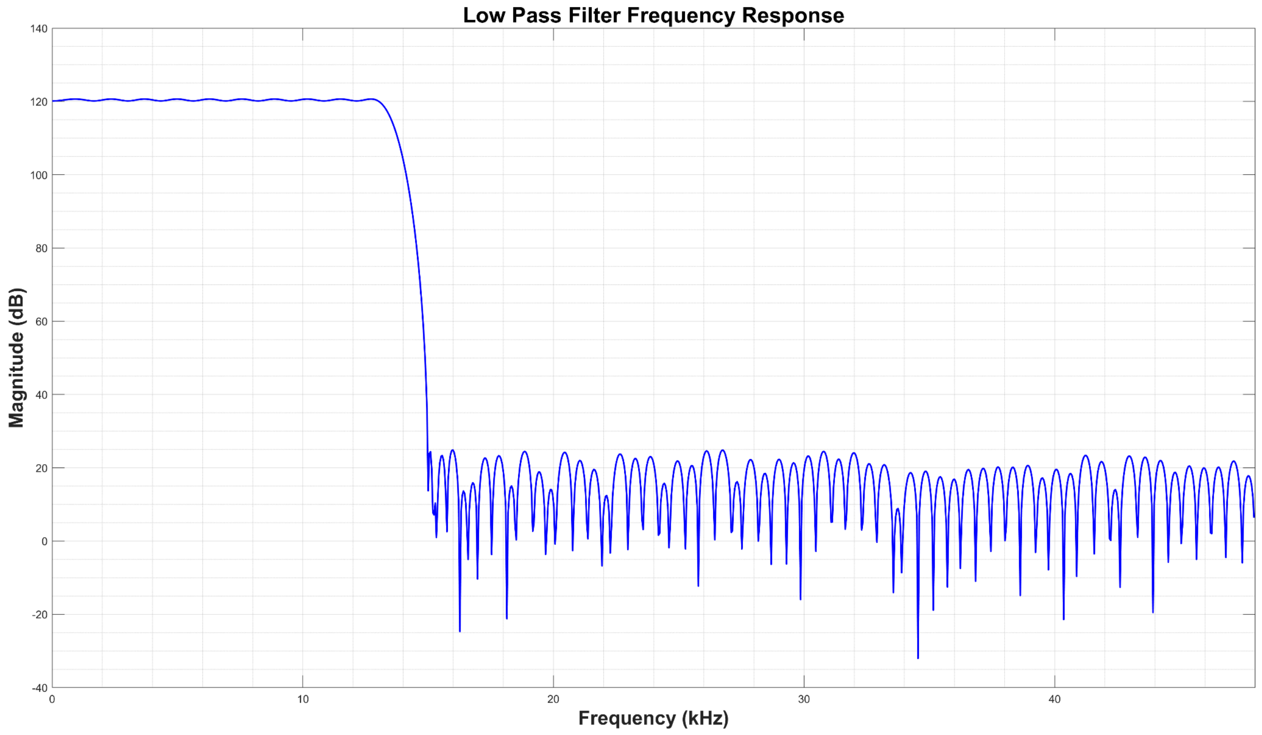

Figure 8. Frequency Response of Low Pass Filter

Finally, the signal is processed by a first order high pass RC filter, implemented as a custom VHDL module [1], to remove low frequency DC components. The module implements this function. This filter has a cutoff frequency of approximately 3.73 Hz. During this stage, the data is also downsampled once more, bringing the final sampling rate to 48 kHz.

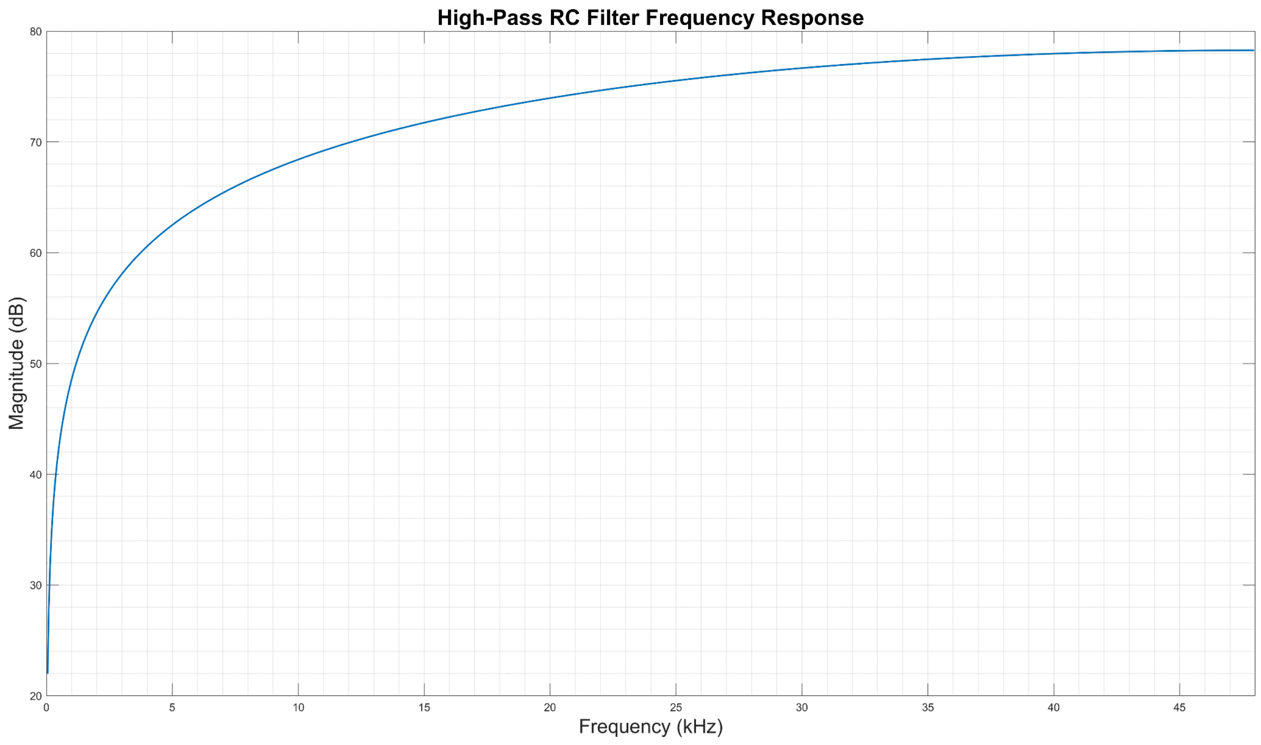

Figure 9. Frequency Response of High Pass Filter

4.2.2 FFT Core

The FFT core consists of the Xilinx FFT IP core [8], along with additional FSM logic and BRAMs wrapped around it to manage data input and output. FFT IP is configured to take in 16384 samples in the time domain and produce 16384 frequency domain bins. With an input sampling rate of 48 kHz, each bin has a frequency resolution of 2.9297 Hz. This resolution was chosen to detect notes within the F3 to E4 octave, where the smallest interval (e.g., F3 to F♯3) is approximately 11 Hz. The resolution of 2.9Hz allows us to differentiate between all adjacent notes that we need.

This block takes in filtered time domain audio data and buffers the samples into BRAM. After a frame of 16384 samples are stored in the BRAM, the module then feeds the stored time domain data into the FFT IP core with the imaginary part of the data padded to 0.

After transformation, the power at each frequency bin is computed using the formula $power = real^2 + imag^2$. Only bins corresponding to frequencies between approximately 70 Hz (bin 24) and 445 Hz (bin 128), which matches with the vocal range of team members, are retained and sent to downstream VGA controller and Pitch detection module. By limiting the number of frequency bins, we were also able to eliminate most of the noise and harmonic components that would have appeared across a broader frequency spectrum.

4.2.2 VGA Controller

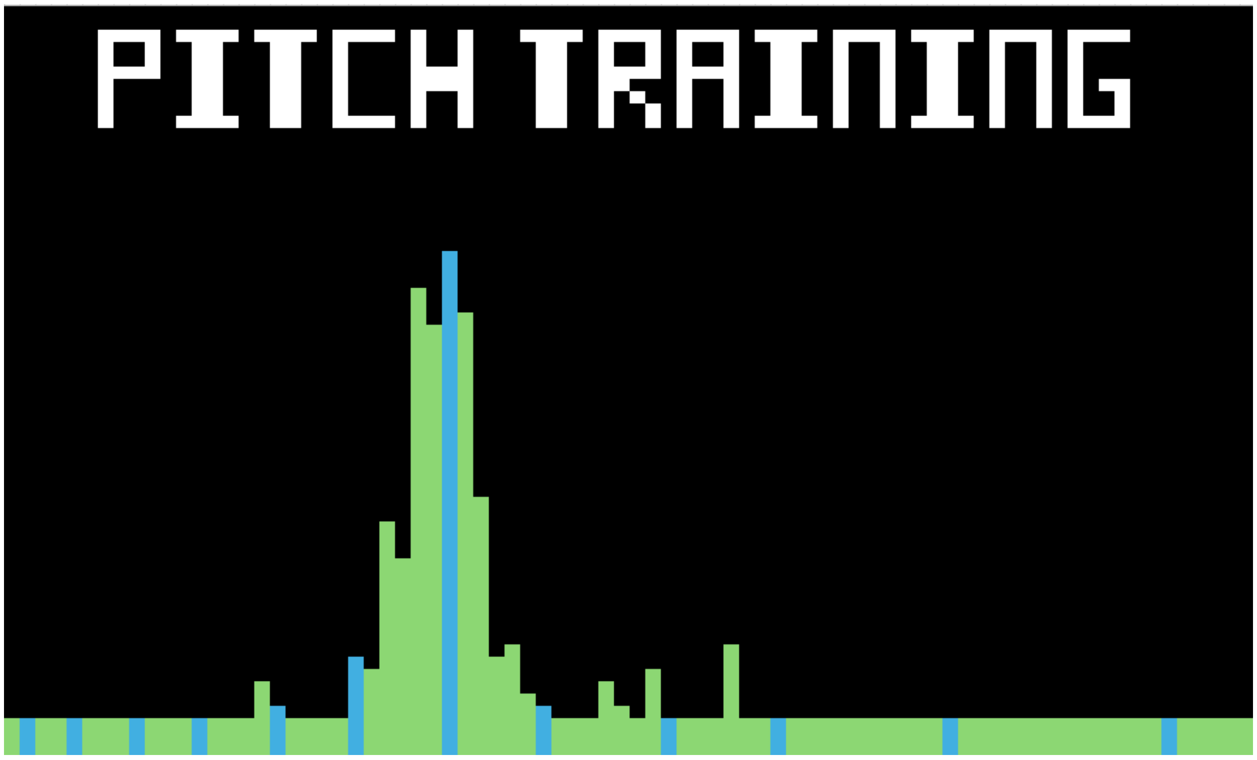

Figure 10. Illustration of VGA display design

The VGA controller is responsible for displaying two key elements: the system title and the real time power of each frequency bin as the user speaks. The majority of the frequency bins are displayed in green while the closest bin to the notes from D2 to A4 is shown in blue. The title is stored in a ROM and is continuously shown in the upper half of the screen. On the other hand, frequency bin data is dynamically updated. The FFT core writes both the power values at corresponding addresses (bin numbers) to a dual port BRAM at 100 MHz. The VGA controller reads this data out to 25 MHz. The clock domain crossing is handled by dual port BRAM, allowing asynchronous access from the FFT and VGA sides. This setup enables the VGA to render frequency data in real time without requiring synchronization with complete frame updates from the FFT core. Since both the FFT updates and VGA rendering occur rapidly, minor frame mismatches are not noticeable to the human eye.

4.5 Pitch Detection

The pitch detection system receives the audio spectrum data from the FFT Core, which it analyses to determine the dominant pitch. Originally, we had planned to make use of the Harmonic Product Spectrum algorithm [2], but it turned out to be too computationally intensive for practical use. In particular, it required a greater number of FFT outputs, which would decrease the resolution, and make it difficult to distinguish between different frequencies. The final design uses a rolling average of the highest magnitude frequency component from the spectrum data it receives. This approach ensures that we do not need to retain the entire FFT frame when identifying the peak frequency. Instead, the system maintains the last eight detected maximum bin numbers and outputs their average as the final result. This allows it to stabilize the rapidly changing input data, which it then sends to the Microblaze through the AXI GPIO IP core for processing.

4.6 Keyboard Controller & ASCII Converter

The keyboard controller converts PS/2 signal inputs into two types of output codes, PS/2 code and ASCII code. It additionally outputs a “key_released” signal and a “new_key” signal to notify downstream of any changes to the keyboard input. As the computer keyboard is actuated by a physical key press, a debouncer module is used to filter out keyboard signal noise and unintended repeated key presses. Both the PS/2 logic and rebounder logic are referenced from GitHub user Jovan Vukic [9]. However, to suit the needs of this project, the keyboard controller IP is extended with PS/2 to ASCII conversion “ps2_to_ascii.v ”to aid readability and FPGA development. This additional converter is a standalone module, implemented as a long case statement; it is useful to note that there is not a one-to-one key mapping for all keys. Furthermore, as our system mimics a piano keyboard using a computer keyboard, there is a conversion module “kb2piano_octave.v” that translates a pressed key to a piano note ID (an abstraction for the audio frequency that is used in our device system-wide).

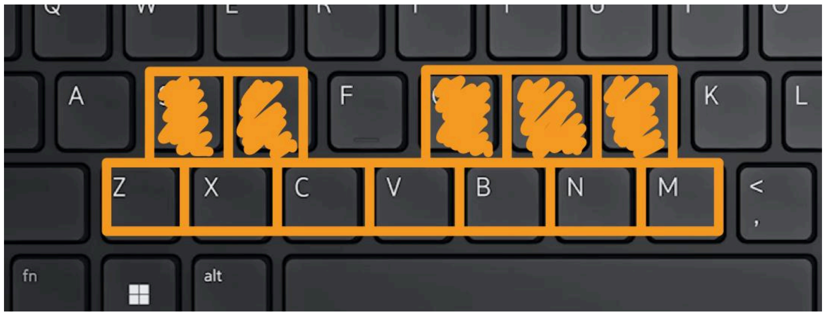

Figure 11. Simulating a piano layout on a computer keyboard.

4.7 Audio Synthesizer

The audio synthesizer generates PWM audio output from two input signals: Microblaze Note ID, and Keyboard piano Note ID. The PWM signal is output at the mono audio jack. As our device application focuses on the musical notes producible by a piano, we opted to use “note ID” - an integer index corresponding to each note on a piano, instead of sound frequency. One key insight we leveraged is the fact that piano keys repeat in pitch every octave (12 notes), where pitches that are one octave higher correspond to a doubling in frequency (or halving in period). Since our piano interface only fits one octave at a time, we include a table of periods used to produce each note in the lowest piano octave (octave 1), represented as a 32-bit unsigned integer in terms of clock periods assuming a 100MHz input clock. To obtain the period for a note in a higher octave, one can look up the period for the equivalent note in octave 1, then right shift by the desired octave number (repeatedly dividing by 2 to obtain the correct period).

The PWM audio output is generated from a module named “freq_pwm.v”, where it takes input of a 32-bit period (expressed in terms of the number of clock cycles required by a 100MHz input clock). To keep the design simple, we use a counter to generate a square wave by toggling out_pwm every “period/2” number of cycles, which creates a square wave with 50% duty cycle. One trade off is that the output audio is not a pure sine wave, and that the generated sound has overtones. However, this is not a concern, as the target tone remains the dominant tone, and all physical instruments have some amount of overtones due to resonance. To decrease volume, we right-shift the counter reset threshold because a reduced duty cycle corresponds to less average power of the speaker, thereby reducing its volume. Freq_pwm also supports dynamically changing the desired period to reuse one module for playing different notes (although not simultaneously). This feature is implemented such that the current square wave will complete its run, and only the subsequent cycles would the device output the new tone. Although the update is not instantaneous, it is a simple implementation with no perceptible delay in human time.

To support playing several notes simultaneously, we instantiate 12 freq_pwm modules, one for each piano key in an octave. The PWM signals from each module are bitwise-OR’d together. One limitation of this implementation is that one cannot play many notes while having the volume set high because the overlapping of high-duty cycle square waves would produce an approximately constant “high” PWM signal, which does not generate sound due to the lack of oscillations. The 12 freq_pwm modules are packaged into one Verilog file, “piano_octave.v”.

4.8 7 Segment Display Controller

The custom 7 Segment Display Controller functions by displaying up to eight ASCII characters at once, resulting in a total of a 64-bit input. This module, named “seg7x8”, outputs to the “seg” (segment), “dp” (decimal point), and “an” (anode) signals defined in the constraint file. In order to use this module, the upstream logic must register the characters to be printed on the display, and hold their values during the entirety of display usage.

The array of eight 7-segment displays on the Nexys 4 DDR functions by displaying only one character at a time, in a round-robin fashion in rapid succession. The round robin logic is produced by a clock divider implemented as the extracted bits [19:17] from an ever-increasing counter clocked by a fast clock input.

Every time one display is given power, the input ASCII code at the corresponding indices first gets converted into an 8-bit value that represents the segments to turn on, then this value is connected to the segment and decimal point pins, printing the character on the screen.

Figures 12, 13, 14 below shows how the 7-segment display represents the major UI elements used in this project

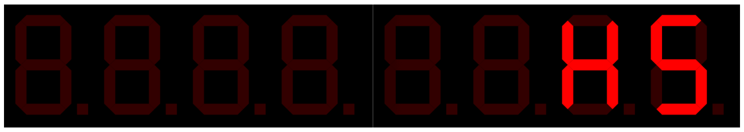
Figure 12. 7 segment display showing Home Screen UI

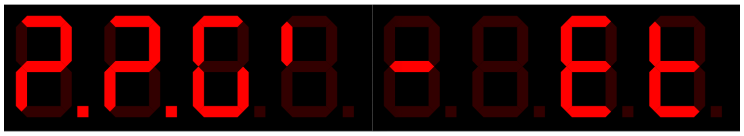
Figure 13. 7 segment display showing Ear Training Mode UI. Left to right: microblaze generated random note (represented as question marks), user answers G# (represented as G'), the notes do not match (expressed as "minus sign"), mode is ear training (ET)

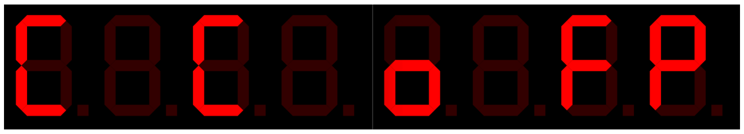
Figure 14. 7 segment display showing Free Play Mode UI. Left to right: user is singing a C note, user is playing a C note on the keyboard piano, the notes match (represented as "o"), mode is free play (FP)

5. Design Tree Description

- Top level project folder (`./MEMES_mic`)
- Top level FPGA component as Custom IP (`./PitchTrainingSystem_ip`)
- Top level FPGA component bitstream (`./PitchTrainingSystem_bitstream`)
- Audio Input Pipeline IP (`./ip/ip`)
- Top Level Peripheral (`./integration/piano_with_software/swctrl_piano.v`)
    - Piano (`./integration/piano_with_software/piano.v`)
        - Keyboard input (`./keyboard/keyboard.v`), with passthrough to module output
        - Custom PS2/ASCII library (`./keyboard/ps2_to_ascii.v`) (global include)
        - Mapping keyboard to Piano Logic (`./keyboard/kb2piano_octave.v`)
        - Wrapper for multiple Piano Notes (`./audio/piano/piano_octave.v`)
            - Piano Note (`./audio/piano/piano_note.v`)
            - Audio Jack PWM output (`./audio/freq_pwm/freq_pwm.v`)
    - 7 Segment Logic (`./7seg/seg7x8.v`)

6. Tips and Tricks

- When processing audio input from a microphone, it's helpful to focus on a narrow frequency range (e.g., vocal range) to simplify filtering and improve accuracy.
- Building on existing, successful projects can save time and effort. It's important to thoroughly understand how the reference design works before adapting it to your needs.
- Choose a project structure where components can be developed and verified independently. This allows for successful projects even if system level optimizations are not feasible within the project timeline.
- Scope the project to something you know you can do. This may result in a project that is too dull, but challenges will pop up, even in the fields you thought you are familiar with.
- Do not attempt to resolve the Linker error in Xilinx SDK. Triggers for this error are not explicit and the compiler's configurations are very complicated. Whenever this error pops up, try other methods to achieve the initial goal.
- Do not include a space in the path that points toward your Vivado Project. The project or SDK will not open. It will say "access denied", but it is a space problem.
- Avoid direct downsampling, as it introduces aliasing. Instead, use IP cores like CIC or FIR filters to perform downsampling correctly.
- Audio jack PWM output functions with 0/1 logic, despite the user guide recommending 0/z. In fact from our experience, 0/z logic is discouraged, because bitwise operations do not function as expected when z-level is involved. Much time was wasted in debugging the culprit of why our audio outputs were not propagating correctly to the top level.
- Unless there is a need for sin waves or sound waves of a specific shape, generate a square wave with a counter (and the methodology detailed in this document). It is much simpler than having to initialize a table of wave values and potentially having to interpolate wave values for higher resolution.
- Not all USB keyboards are equal. In fact for the PS/2 keyboard (via USB interface), the dumber the keyboard the better! Chances are that a "wired keyboard" with a detachable keyboard would not work. Any wireless keyboard with a USB dongle/receivers also would not work despite being able to plug it into the FPGA.

7. References

[1] Digilent Inc., Nexys 4 DDR Spectral Sources Demo, [Online]. Available: digilent.com/reference/learn/programmable-logic/tutorials/nexys-4-ddr-spectral-sources-demo/start. [Accessed: Apr. 7, 2025].

[2] M. R. Schroeder, 'Period Histogram and Product Spectrum: New Methods for Fundamental-Frequency Measurement', The Journal of the Acoustical Society of America, vol. 43, no. 4, pp. 829–834, Apr. 1968, doi: 10.1121/1.1910902.

[3] A. M. Noll and M. R. Schroeder, 'Short-Time "Cepstrum" Pitch Detection', The Journal of the Acoustical Society of America, vol. 36, no. 5_Supplement, pp. 1030–1030, May 1964, doi: 10.1121/1.2143271.

[4] R. G. Amado and J. V. Filho, "Pitch detection algorithms based on zero-cross rate and autocorrelation function for musical notes," 2008 International Conference on Audio, Language and Image Processing, Shanghai, China, 2008, pp. 449-454, doi: 10.1109/ICALIP.2008.4590188.

[5] G. Marsaglia, "Xorshift RNGs," Journal of Statistical Software, vol. 8, no. 14, 2003, doi: doi.org/10.18637/jss.v008.i14. [Accessed: Apr. 7, 2025].

[6] Xilinx, CIC Compiler v4.0 Product Guide (PG140), Version 4.0, Feb. 4, 2021. [Online]. Available: docs.amd.com/v/u/en-US/pg140-cic-compiler. [Accessed: Apr. 7, 2025].

[7] AMD Xilinx, Fast Fourier Transform LogiCORE IP Product Guide (PG109), Version 9.1, Nov. 13, 2024. [Online]. Available: docs.amd.com/r/en-US/pg109-xfft. [Accessed: Apr. 7, 2025].

[8] AMD Xilinx, Fast Fourier Transform LogiCORE IP Product Guide (PG109), Version 9.1, Nov. 13, 2024. [Online]. Available: docs.amd.com/r/en-US/pg109-xfft. [Accessed: Apr. 7, 2025].

[9] J. Vukic, "PS2-keyboard-interface/SRC/simulation/modules at main," GitHub, github.com/jovan-vukic/ps2-keyboard-interface/blob/main/src/simulation/modules. [Accessed: Apr. 7, 2025].