Karaoke Machine

Final Report

Saturday, April 7, 2025

Name
Srinidhi Shankar
Natasha Yang
Jennifer Mah
Jimmy Velasco

Table of Contents

1.0 Overview

1.1 Background and Motivation

For ECE532: Digital System Design, the team had to create a project using a Nexys DDR or Nexys Video FPGA. Originally, our team wanted to make a karaoke machine, which would play a song and display lyrics on a screen while the user sang into the microphone. The screen would have also dynamically displayed how well the user is matching pitch, everytime an input sample from the microphone is processed. This project was ambitious, and the team eventually decided to scale it down and create a sight reading and pitch detection system, which would use similar logic and keep the user experience aspect but simplify it into individual notes, rather than complete songs. The proposed system is a useful way to learn how to sight read music and learn how to play or sing music by ear.

1.2 Goals

The goal of this project was to create a sight reading and pitch detection system in hardware on a Nexys Video FPGA board. A note should be displayed through HDMI on a music score, to allow the user the opportunity to match pitch solely through sight reading, but should also be able to be played on a speaker upon pressing a push button. The user can sing the notes into a microphone, and the expected note, as well as the note sung should be displayed on the OLEDs. This way, the user will be able to see how far off they sang from the desired note and use that to self-correct accordingly. The system is controlled using the push buttons on the FPGA and instructions, as well as the name of the note to sing are printed on a laptop's terminal. As such, through our project, there are three means for the user to be able to sing a note. Firstly, they can simply read the score on the HDMI and match pitch. If they are unable to read the note, they can read the terminal to see the name of the note they are expected to sing. Finally, if more guidance is needed, the user can click the appropriate push button to hear the notes on the speaker. They can click on a separate push button to start recording their audio, and click on a final push button to see their results on the OLEDs.

1.3 Functional Requirements

The functional requirements of the project are listed in the table below.

Table 1: Functional Requirements

Functional Requirements	**Description**
Audio In/Tone Matching	Given audio input from a user, the machine should be able to match the audio input's pitch to a note.
Display/Meaningful GUI	The desired note on a music score and its name should be displayed meaningfully, as well as the actual note that was inputted through the audio input.
Audio Out	Play the desired note out loud.

1.4 Block Diagram and Description of IPs and Peripherals

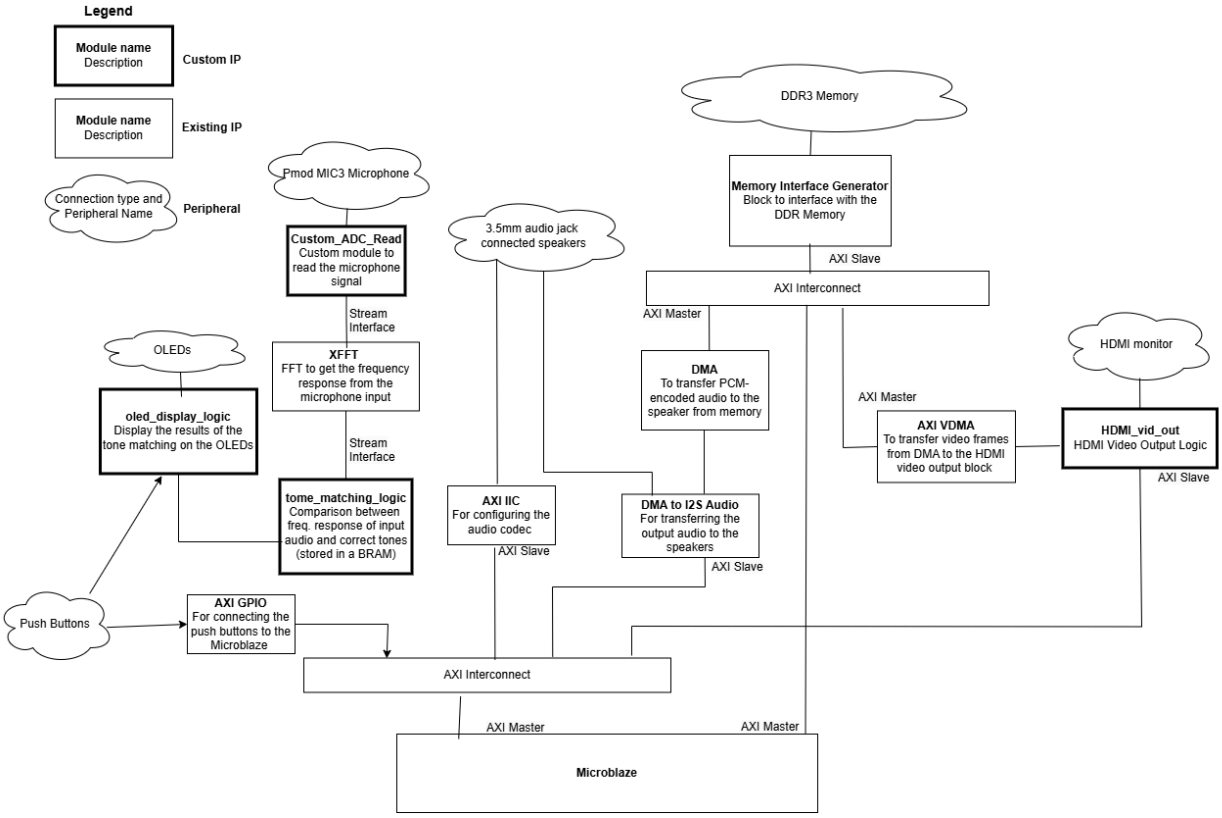

Figure 1: Block Diagram

2.0 Outcome

2.1 Results

Proposed Features	Final Features

2.2 Future Improvements

Given more time, certain improvements could be made to the system. One improvement could be to implement more effective filtering so the note sung into the microphone can be better detected. Previous Diligent demos that were employed and voided implemented various high and low pass filters that could be extended for this use case with the PMOD microphone. This addition would limit the amount of noise from the input of the microphone, allowing for more accurate FFT samples. We would also be able to limit the noise margin within our tone matching algorithm to collect more precise data, enabling the detection of whether the user is flat or sharp. The project currently has a bug where it has difficulty detecting E4, and misrepresents it as either a D# or an F, which can be explained due to the lack of filtering and high noise margins in our pitch detection algorithm.

Currently, our system can detect pitches from C4 to B4. Another improvement includes expanding the pitch detection algorithm to include more notes. The pitch detection is currently hard coded to be able to apply a noise margin to the notes as previously discussed. However, with a mathematical algorithm rather than inequality statements, this algorithm can become further scalable.

Finally, this project has the ability to scale into a more complex karaoke machine, by using an SD card to store videos of karaoke songs, and displaying the video on the HDMI. If this is done as an extension to our project, the tone matching can be extended to create a scoring algorithm, in which an average of all correct and incorrect notes are taken to provide the user with an overall score on their performance.

3.0 Project Schedule

The project was divided into 6 weekly milestones. There was also an extra week after milestone 6 to implement the project before the final demo that allowed us to complete the system. The table below shows the original milestones we planned, as well as the actual achievements that were accomplished that week.

Table 1: Original and Actual Milestones

M#	Original Milestone	Actual Milestone
M1	**Research & Development:** This milestone includes the research of tone matching algorithms, peripheral and IP implementations, specifically HDMI, I2S, microphone, speaker, and SD card, a project block diagram, and a proposal.	We were able to accomplish this milestone as expected. We researched what microphone we were going to use, how to implement HDMI, how to implement the tone matching algorithm and the FFT IP block provided by Vivado.
M2	**Algorithms:** The results gathered from M1 research on algorithms are to be prototyped. This milestone focuses on performance profiling and feasibility assessment. Current methods include FFT with bandpass and Autocorrelation.	We attempted to display an image on the HDMI, get output on the microphone and get output from the FFT IP. Unfortunately, we were unable to get the peripherals or the FFT IP to work. We also did some additional modeling of the FFT and tone matching algorithms on Python.
M3	**HDMI and Sound in Hardware:** Both HDMI and sound peripherals should be programmed into the FPGA. Systems do not have to be interconnected.	The HDMI, microphone and FFT IP were all implemented separately on the FPGA board during this milestone. The HDMI outputted two images, we got output from the PMOD microphone that was unfiltered, and the FFT IP was giving an output but we had not interpreted it yet. This week, we also lost our PMOD

		microphone. The OLEDs were implemented as well.
M4	**Logic in Hardware:** The algorithms developed during M2 should be converted to HDL or C and loaded on the FPGA. Testbenches should assert the functionality.	Our team had many midterms during this week, so we accomplished smaller goals for this milestone. We ordered a new PMOD microphone to replace the lost one, and were able to plot the output of the FFT IP on the VGA.
M5	**Implementation:** This milestone focuses on integrating all the blocks together. The addition of buffers and interconnection using a bus. All elements should be able to work together and produce an MVP.	This week was also very busy for our team and some of us caught ill. The HDMI display and OLEDs were changed to show a more meaningful display (notes on a music score and note names) and were integrated with the microphone and speaker. We also soldered the new PMOD microphone and started working on the tone matching algorithm.
M6	**Bug fix/ Buffer:** Although M1 to M5 milestones lay out the ideal timeline for the project, milestone 6 is considered as buffer time for either bug fixes, offset for pushback due to unexpected drawbacks or fine-tuning features. This milestone also sets time to develop a presentation for the final demo.	We debugged the FFT IP output and created the IP to find the maximum FFT frequency. We also wrote and simulated the rest of the tone matching algorithm.

During the week after milestone 6 and before the demo, we were able to get the tone matching algorithm working and we integrated it into our project, along with the other components.

Between the first and the third milestone, the team was able to accomplish the original goals set by the planned milestones. However, as we faced certain challenges, such as getting the FFT IP to work, losing the PMOD microphone, and managing the workload with other courses, we did not meet our original goals for the milestones after the fourth. However, our team had anticipated that certain aspects of the project may take longer, which is why our Milestone 6 was originally a buffer milestone. Giving ourselves a buffer allowed us to catch up and finalize the project in time for the final demo.

4.0 Block Descriptions

4.1 ADC and Decimator

This project required the use of the PMOD microphone to input audio data from the user for further processing. To be able to achieve this, we needed an SPI controller to communicate with the microphone. This was adapted from a third party IP [A], and modified to suit our needs. The PMOD microphone transforms audio input to a digital 12-bit value, transmitted serially. The ADC controller essentially reads the 12-bit serial data from the microphone at a rate of one bit per clock cycle, then allows the full data to be available when the entire 12 bits are ready. As this is primarily a third party IP, the full details on implementation will not be discussed in this report.

The chip select (*adc_cs*) will control whether the ADC is active, and this select is an active low signal. The serial data input is stored in *adc_sd* which reads bit by bit and shifts them into a shift register. The ready signal goes high for one clock cycle when the full 12 bit data has been captured, and full 12 bit data is stored in the *data* register.

This *data* signal is then sent into a **decimator** to reduce the sampling rate, such that frequencies within the range of human hearing were distinguishable by the FFT. The module takes *data_in*, sampled at a higher date, and outputs the data at a lower sampling rate controlled by the parameter *TIMES*. To set the Nyquist frequency of the FFT IP, the *TIMES* parameter was set to 64, such that for every 64 input samples, there is only one output (1MSamples per second/ 64 samples = 15.625 kHz). This math is further explained in Section 4.3. There is an internal

counter (*count*) that waits for all 64 samples to be processed and asserts the *new_sample* flag. This tells the BRAM controller to store the new sample. Once all 1024 samples for the FFT have been written into the BRAM, they are sent to the FFT to begin processing.

4.2 FFT IP

The FFT has many configurable parameters, which is beneficial for a user who knows exactly how the FFTs are implemented and how they should work. However, for the casual FFT user, the number of choices becomes overwhelming, and the amount of documentation one must read to understand these possible choices is daunting. As examples of the latter group of users, the following configurations to Vivado's xFFT IP block are based on what could be a misguided understanding of the FFT's operation. It may not be the optimal way of configuring the FFT, but it worked well enough for our use case.

1. Transform Length of 1024: the decision behind this length was made after we decided to limit our detectable note range from C4 to B4, and after we'd made the decimator. The difference in frequency between notes change, where lower notes have much closer frequencies to each other than higher notes (e.g., the distance from C0 to C#0 is 0.97Hz, which the distance from C4 to C#4 is 15.55Hz). Since the decimator set the sampling rate to 15.625kHz, 1024 samples is the smallest number of samples the FFT could have in order to detect any notes higher than and including C4. (The exact calculations will follow in Section 4.3.) Therefore, we set the number of samples to 1024, as more samples would have increased resource usage.
2. Input Data Width of 16: the input data width had to be larger than 12 (width of microphone data). Note that we did not have enough time to experiment with changing this width, and the only reason it was set to 16 was because 16 is the closest power of 2 to 12, and powers of 2 are *typically* good for resource usage. If we had more time, we would collect empirical data to support this hypothesis.
3. Scaling Option set to Block Floating Point: Because of our unfamiliarity with FFTs, we weren't sure what an appropriate scaling factor would be. We found a Xilinx support article [:)] that stated that setting the scaling to Block Floating Point would take care of scaling for us. Additionally, other students confirmed that Block Floating Point worked for them, so we figured we'd use it too. Out of all potential scaling options, Block

Floating Point is said to increase resource usage the most. If we had more time, we would try experimenting with the Scaled option to save on resources.

4. XF_INDEX and Output Ordering set to Natural Order: After the frequency response of the audio signal was calculated by the FFT, we added some logic to find the magnitude squared of each output sample (details follow in Section 4.3), and store it in a BRAM for further processes. We used the XF_INDEX number of each output data sample as the address in which the sample would be stored in the BRAM. Natural Order was set to make debugging easier (it may not be necessary to set this, but we have not tried without. Removing Natural Ordering saves on resource usage.)

4.3 Tone Matching IP

The tone matching IP is an integral part of our project, without which the entire basis of the project would not have been accomplished. This was also a custom IP created for this project to match the expected notes to the actual notes sung by the user. To construct the IP, the following background information is required:

The sampling frequency (F_s) of the FFT is equal to 15.625kHz. The transform length (N) of the IP is equal to 1024. As such, the smallest frequency difference that the FFT can detect is:

$$\Delta f \ = F_s \ / \ N \ = \ 15.625 \ kHz \ / \ 1024 \ \simeq \ 15.26 Hz \ per \ bin$$

The highest frequency that the FFT can detect is known as the *Nyquist frequency* which is equal to:

$$F_s \ / \ 2 \ = \ 15.625 kHz \ / 2 \ = \ 7.63 kHz$$

The FFT provides N complex numbers, of which only the first $N/2 \ + \ 1$ bins are unique for real-valued signals. The input from the PMOD microphone is an ADC signal, so it samples real voltages, so we can conclude that the FFT signal is symmetric. Bin 0 is the DC value of the signal, bin 1 is very low bass, bin 2 to bin 511 would correspond to positive frequencies, and bin 512 would be the Nyquist frequency:

$$512 \ \times \ 30.52 \ = \ 7.63 \text{ kHz}$$

The bin number in the design corresponds to the current bin of the FFT being processed, which can be incremented from 0 to 512, as to only access the values in the BRAM for those addresses, and drop the latter 512. To calculate the frequency per bin in our IP, we can take *in_addr* and

multiply by the constant we have solved for mathematically as 15.26Hz / bin. Although the mathematically derived value to multiply the bins by to recover the original frequency was 15.26, through empirical testing and calibration, it was discovered that 23 was a closer scaling factor.

The above will solve the frequency for every bin. However, to be able to detect the note being sung, the IP must be able to find the note for the frequency of highest magnitude. The signal from the FFT IP gets squared rooted and added together:

$$Magnitude^2 = Re(FFT)^2 + Im(FFT)^2$$

For the purposes of finding the greatest magnitude, square rooting this value is unnecessary, so this value is directly passed into the tone matching IP.

The tone matching IP takes the following inputs:
clk, // synchronous clock
in_signal, // magnitude of the FFT for the current bin
and returns the following outputs:
in_addr, // BRAM address, corresponding to the bin number
note_name, // three ASCII characters corresponding to the note that was sung

With all of the above information, the algorithm can now be defined:

1. Declare an internal maximum magnitude (*max_mag*) and a maximum address.
2. If *in_signal* > *max_mag*, *in_addr* >= *11*, and *in_addr* <= *27,* then set *max_mag* = *in_signal,* and set *max_addr* = *in_addr*. Bin 11 and bin 27 correspond to the bin C4 and B4 would be in, respectively (discussion regarding the choice to only search this limited space in Section 2.2).
3. If *in_addr* > *27,* we can stop processing this signal, and set *in_addr* = *11*. We can then start the note lookup.
4. In the note_lookup, compare each frequency with the range of values corresponding to a note with a +/- 6% error margin to account for variations in tone. For example, frequencies between 245 and 276 should map to a C4, even though C4 = 261.63Hz.

This 24 bit (3 ASCII) *note_name* signal gets set as an input to the OLED IP for further processing.

4.4 OLED IP

4.5 HDMI IPs

The HDMI implementation was adapted from the Diligent HDMI demo [>]. We made a few changes to suit our use case:

1. Removed the HDMI input: Because we don't need HDMI pass-through, we found we could save resources by removing the IP blocks that controlled the HDMI input data path from the original demo. The following blocks remained in our design: the AXI Video DMA, the AXI4-Stream to Video Out, the RGB to DVI Video Encoder, and the Video Timing Controller (all are pre-existing IPs). The AXI Video DMA block is responsible for taking video frames stored in memory and sending them to the AXI4-Stream to Video Out block (whose timing is controlled by the Video Timing Controller), which sends the video to the RGB to DVI Video Encoder, which controls the TMDS_OUT pins to tell the HDMI monitor what to display.
2. We added Microblaze code to write video frames stored as C arrays in header files to DDR memory, such that the VDMA could access them. A different video frame would be shown whenever the user pressed the Down button.

4.6 Speaker

The speaker implementation was adapted from the Diligent DMA demo [+]. The speaker IP blocks were kept the same as in the demo. These include Diligent's d_axi_i2s_audio IP, which is used for sending an audio signal to the audio codec, and the AXI IIC block, used to configure the audio codec. The Microblaze code to configure the audio codec was not changed from the Diligent demo. To control the speaker, however, we wrote a simple Microblaze function that would write a note (in the form of a PCM signal) into DDR memory. When the user pressed the Right button, the Microblaze would tell a DMA to the note to the d_axi_i2s_audio block to be played through the on-board headphone jack.

4.7 Microblaze

The major functionality of the Microblaze has been discussed in Sections 4.5 and 4.6. This section will further explain what the Microblaze was used for. First note that the Microblaze was connected to a Microblaze local memory (BRAM), a Microblaze Debug Module, an AXI Interrupt Controller, and two AXI Interconnects. The AXI Interrupt Controller was necessary to interface with the various peripherals the MIcroblaze controlled. The Microblaze was connected as the only master to one of the AXI Interconnects, such that it could configure the peripherals (write to their configuration registers) through software. The Microblaze was connected as one of many masters to the second interconnect, where the only slave was the MIG. This connection was made so that the Microblaze could write data into memory to be read by other peripherals.

To be specific, the Microblaze was in charge of controlling the HDMI and speaker (the Pmod mic and OLEDs had their own data path.) The demo.c file to be run on the Microblaze contains the peripheral set-up code and the main event loop. The set-up code includes code to initialize blocks such as the VDMA, GPIO Interrupt Controller, and the Audio Codec. Most of the set-up code came from the Diligent demos mentioned in Sections 4.5 and 4.6.

After running the peripheral set-up code, the main event loop would start. The Microblaze would wait for an interrupt to trigger, upon which it would respond. The most important interrupts were the push button interrupts, which would trigger the HDMI and speaker functionality (i.e., play the notes over the speaker, or to change the image on the HDMI).

5.0 Design Tree Description

k

6.0 Tips and Tricks

An important piece of advice that we would like to extend to future students taking this course is the importance of prioritization and developing a minimum viable product. Do not spend time making and committing to features that are not a part of your minimum viable product until you have one fully fleshed out which you now have time to update. Our team struggled with being

too ambitious right from the start and scrambling towards the end to get anything working, which might have been mitigated if we had simplified our design from the start.

Another key piece of advice is the importance of delegation and time management. If you would like to develop your skill in an area of complete unfamiliarity, feel free to do so, but make sure your team knows that you will take more time and will potentially struggle and need their assistance. If you want the project to go as smoothly as possible, highlight each member's strengths and weaknesses, and delegate tasks such that each member has something that they are comfortable working with. Especially because hardware is already so difficult, this is an important choice as it can make or break the entire project.

7.0 Video

A video of our project has been uploaded to YouTube and can be found with the following link: https://youtu.be/eKvLNfcHwyM.

8.0 References

[A]https://github.com/npalladino100/Artix-7-FFT

[:)] https://adaptivesupport.amd.com/s/article/1160838?language=en_US

[>] https://digilent.com/reference/learn/programmable-logic/tutorials/nexys-video-hdmi-demo/start?srsltid=AfmBOoqzUHSgasA8s23wbODopeVNx3s3QewHb5kgLYkh-hSfDQDScJiI

[+] https://digilent.com/reference/programmable-logic/nexys-video/demos/dma-audio