

Do Re Mi F(PG)a Final Report

Sriram Sethuraman
*Department of Mechanical
 Engineering
 Massachusetts Institute of
 Technology*
 Cambridge, MA, USA
 sriram_s@mit.edu

Jessica Shoemaker
*Electrical Engineering and
 Computer Science
 Massachusetts Institute of
 Technology*
 Cambridge, MA, USA
 jtshoe@mit.edu

Cole Ruehle
*Electrical Engineering and
 Computer Science
 Massachusetts Institute of
 Technology*
 Cambridge, MA, USA
 cruehle@mit.edu



Fig. 1. System Block Diagram

Abstract—Our project is a singing trainer game based on a viral TikTok filter. The primary objective is to create an interactive platform that processes .wav files to train users to sing along accurately. The system will analyze a song file, convert it into a format the FPGA can interpret, and display a flappy-bird-style game where the player’s pitch controls the height of a ball navigating through obstacles. Only singing the correct notes at the right time allows the ball to pass, enhancing the player’s musical accuracy.

Index Terms—audio processing, FPGA, pitch detection, game engine, music training

I. INTRODUCTION

We present a report on a singing trainer game, inspired by a viral TikTok filter. The system aims to process audio input, detect fundamental frequencies (notes), and integrate this data into a real-time interactive video game. The player’s sung pitch controls a ball in a visual environment, ensuring that only accurate notes lead to successful navigation of obstacles.

II. OVERALL DESIGN

Our design consists of multiple components: audio processing for real-time pitch detection, music processing for interpreting a given .wav file of a melody, video generation to visualize the gameplay, and file storage/loading from an SD card.

III. AUDIO PROCESSING

A. Big Picture

The goal of the audio processing component is to output the fundamental frequency (pitch) of a user’s voice in real time. Three main components form the processing chain:

- **Microphone controller:** Samples audio and provides the signal.
- **Bandpass filter:** Restricts the signal to 100–1000 Hz, a typical human vocal range.

- **YIN algorithm:** A fundamental frequency detection method implemented from scratch in SystemVerilog.

B. Microphone

We currently use the analog microphone and SPI ADC system from a previous lab setup. The sampling rate started out at 8 kHz, allowing ample time for the band pass filter and YIN calculations each cycle. However, after testing, the sampling rate was increased to 16 kHz to allow for higher resolution of the YIN algorithm (more discussion below).

C. Bandpass Filter

The bandpass filter ensures only frequencies of interest (100–1000 Hz) reach the YIN algorithm. We tested a 6th-order Infinite Impulse Response (IIR) filter designed in Python (*scipy.signal*) and then implemented it in SystemVerilog. An IIR filter was chosen for the limited number of taps necessary to implement it in hardware compared to an equivalent FIR filter. The filter was implemented as the following difference function:

$$a[0] \cdot y[n] = b[0] \cdot x[n] + b[1] \cdot x[n-1] + \dots + b[M] \cdot x[n-M] - a[1] \cdot y[n-1] - \dots - a[N] \cdot y[n-N]$$

The coefficients and intermediate values are stored as 32-bit fixed point numbers for accurate calculation. A state machine (IDLE, CALC, UPDATE) controls the filtering process. Simulation results closely matched Python outputs, confirming the filter’s correctness. After testing, the 6th order IIR filter was replaced with a 2nd order IIR filter to account for the 16 kHz audio change (more discussion later).

Registers named *a* and *b* were used to store coefficients, and registers *x* and *y* buffer the last few values of $x[n]$ and $y[n]$ for use in calculation. The module is controlled using a simple state machine with three states:

- **IDLE:** Do nothing until a new audio sample is received. When it is, switch to **CALC**.
- **CALC:** Every clock cycle,

$$\text{result} \leftarrow \text{result} + b[i] \cdot x[n-i] - a[i] \cdot y[n-i].$$

This lasts 7 cycles before switching to **UPDATE**.

- **UPDATE:** Shift the *x* and *y* buffers and store the latest values of $x[n]$ and $y[n]$. Set the output valid signal high and return to **IDLE**.

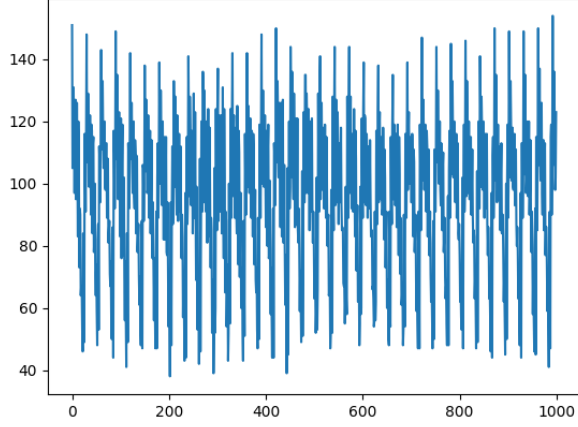


Fig. 2. Unfiltered audio sample

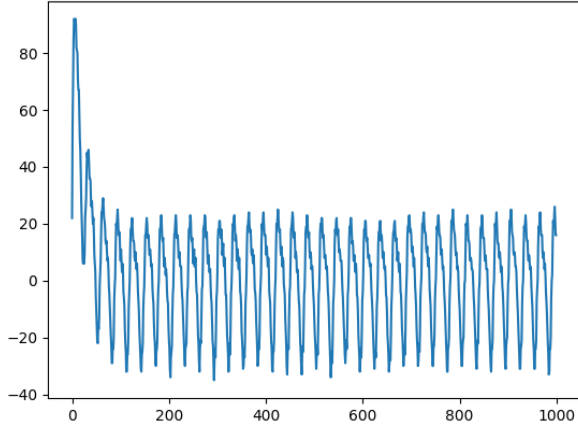


Fig. 3. Filtered audio sample (simulation output)

D. YIN Algorithm

YIN is a fundamental frequency detection algorithm created in the early 2000's specifically for real time low-latency music and voice processing. It is a variation on autocorrelation, where the difference between a signal and a shifted version of it is calculated for many different shifts, and the lowest shift with lowest difference gives the fundamental period (and inverting that gives you frequency). Other than the benefits listed above, the main reason this algorithm was chosen was that it can be implemented completely from scratch instead of relying upon IP to carry out a FFT. The input to YIN is a signal "window," and after lots of testing I found a window size of 500 samples to yield the best results. The steps to YIN are fairly simple, but lead to very accurate results.

- First, the difference function is calculated on the sample window. It is a function of tau (shift amount), and W is

the window size.

$$d_{\tau}(\tau) = \sum_{j=1}^W (x_j - x_{j+\tau})^2$$

- Next, the Cumulative Mean Normalized Difference Function (CMNDF) is calculated. This is also a function of tau and divides the difference function by the average of the difference function of all shifts below the input.

$$d'_{\tau}(\tau) = \begin{cases} 1, & \text{if } \tau = 0, \\ \frac{d_{\tau}(\tau)}{\frac{1}{\tau} \sum_{j=1}^{\tau} d_{\tau}(j)}, & \text{otherwise.} \end{cases}$$

- The candidate output shift is the lowest τ for which the CMNDF is below a certain threshold (usually set to 0.1 according to the paper).
- Lastly, we use parabolic interpolation to estimate the "true" vertex of the CMNDF.

$$\tau \approx \tau + \frac{d'_{\tau}(\tau - 1) - d'_{\tau}(\tau + 1)}{2(d'_{\tau}(\tau - 1) - 2d'_{\tau}(\tau) + d'_{\tau}(\tau + 1))}$$

- And the final output just requires converting our best shift into the frequency.

$$f_o = \frac{\text{sample rate}}{\tau}$$

Implementing this in Verilog is slightly more challenging. The module contains two register arrays (hopefully both synthesized as BRAM), one of length WINDOW_SIZE to store the audio samples and another of length MAX_TAU - MIN_TAU to store the difference function outputs. (On a side note, we know MAX_TAU and MIN_TAU since we know the lowest and highest frequencies we are searching for). This is the only memory really needed for this computation as the rest can be computed on the fly.

This module is a state machine very similar in nature to the bandpass filter state machine, with each state being a step in the process. In testing, it was found that the parabolic interpolation step contributed very insignificantly to the accuracy of the algorithm and was therefore dropped from the implementation to save resources and compute time.

- **IDLE**: When a new filtered audio sample comes in, update the buffer. If a new computation is triggered, go to **DIFF_FUNCTION**.
- **DIFF_FUNCTION**: Iterate through all τ to calculate the difference function for each one. This step is by far the most time consuming since 500 subtractions and multiplications have to be done for each τ . Once we have reached MAX_TAU, go to **CMNDF**.
- **CMNDF**: Calculate the CMNDF starting from MIN_TAU by using an accumulator to store the partial sum and a custom divider. When we've found a τ that has a CMNDF less than 0.1, switch to **PI**.
- **TD (trigger divide)**: Plug in the value of τ we found into a divider computing sample rate / τ . Then, switch to **DIVIDE**.
- **DIVIDE**: When the divider is done, output the fundamental frequency.

E. Optimizations

Firstly, in the implementation of YIN, many divisions were saved by manipulating the CMNDF equation. Instead of finding τ such that

$$\frac{d_\tau(\tau)}{\left(\frac{1}{\tau} \sum_{j=1}^{\tau} d_\tau(j)\right)} < 0.1,$$

we can find τ such that

$$\tau \cdot d_\tau(\tau) < 0.1 \cdot \sum_{j=1}^{\tau} d_\tau(j).$$

This approach avoids any division and only requires two multiplications. Here is a graphical representation of the CMNDF portion of the YIN algorithm working to find the point where $\tau \cdot d_\tau(\tau)$ and $0.1 \cdot \sum_{j=1}^{\tau} d_\tau(j)$ meet.

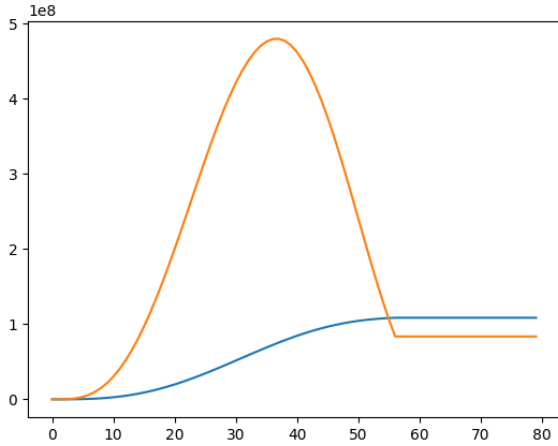


Fig. 4. CMNDF finding the correct tau (simulation output)

Another set of optimizations that had to be made was pipelining a significant portion of the computations executed by both YIN and the bandpass filter since the design did not meeting timing. There isn't much detail to go into with this, just that a lot of operations that consisted of indexing into an array, carrying out a multiplication, then adding that result to a running sum were split into three clock cycles instead of one.

The last optimization that helped improve the YIN algorithm was rewriting all of it to work with 16 kHz audio instead of 8 kHz audio. This is because the final fundamental frequency is computed as

$$f_0 = \frac{F_s}{\tau_{\text{final}}},$$

so doubling F_s firstly increases the "resolution" of τ (each shift increment shifts the signal half as much compared to its period as before). Secondly, since the value of τ is an integer, doubling F_s gives us more resolution in the set of outputs we can return.

F. Evaluating YIN

After running many simulations of the YIN module I've calculated that it takes approximately 30,000-40,000 clock cycles after pipelining, which could be significantly improved by performing multiplications in parallel since only 14 DSP's are used in this design. However, this isn't necessary at all since using the 100 MHz clock this runs 2,500 times per second, which is already overkill, since we only need a frequency update at 60 Hz, the screen refresh rate.

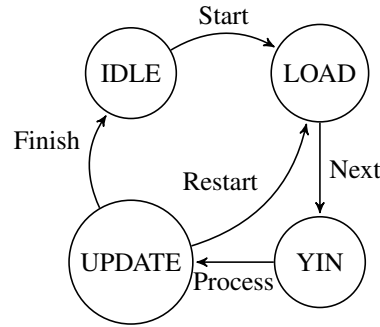
The main timing requirement for this component was to be able to keep up with the screen refresh rate, which it does. And after all the pipelining optimizations, the max delay path in these modules has a WNS of 0.611 ns, adhering to proper timing.

The minimum goal for this part of the system was to be able to accurately detect frequency of singing live from scratch using the lab microphones, which has been accomplished here. The I2S microphone was to be utilized if YIN was finished before Thanksgiving, however it was a very difficult system to both have simulating correctly and working correctly on hardware so that goal was not achieved.

IV. MUSIC PROCESSING

A. NoteLoader Module

This module is a finite state machine, with the following states IDLE, LOAD, YIN, and UPDATE. These states progress in order and after update, if the limit for total notes is not reached, it goes back into load state, otherwise it goes to the idle state. The note loader uses 512x16 BRAM storage to keep track of the audio sample information.



B. Frequency to Note Module

The Frequency to Note module takes a frequency output of YIN and identifies each note by letter (A–G), accidental (natural or flat), and octave. Each note is encoded into an 8-bit binary sequence with bits allocated as follows: letters (3 bits), accidental (2 bits), octave (3 bits).

The system can store the first 32 notes, transferring them to BRAM for integration with the graphics engine. This allows for synchronized note display and gameplay interaction.

C. Note Processor Module

This is the main module of the music processing component, tying the pieces together and making sure the timing is consistent between modules.

D. Evaluation of Design

This portion of the project takes up sizable amount of BRAM, but we made sure to budget for this and choose to preprocess song information such that we would have enough resources for all parts of the game. The timing passes for all these modules, although hooking notelocator up to future parts of the code proved to be challenging due to the different clocks in various modules.

V. VIDEO GENERATION

A. Sprite Generation

We generate three main sprites:

- **Walls:** Represent notes as block sprites with variable

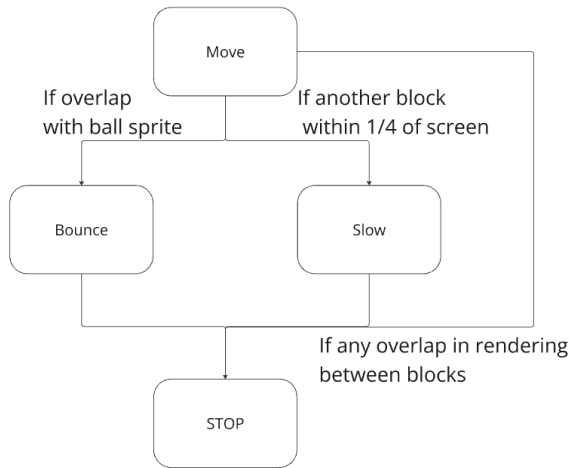


Fig. 5. Finite state machine for wall behavior.

gaps. The gap position corresponds to the correct pitch. Walls move horizontally, challenging the player to match the pitch. The wall sprite takes in the note as a frequency and uses a simple formula to display frequencies within Jessica's vocal range on the screen.

Each of the 3 walls also interacts with the other 2 to determine their movement speed. This is a constant set in the game code, and when walls are unblocked they move from left to right at this speed. When the sprite reaches $x = 0$ it requests a new note from the note selector, and re initializes its internal variables to render a new gap at the correct height.

Secondly there is an interaction with the ball, each sprite functions as a FSM, and in the case that the ball sprite intersects the ball in a location other than the gap, it enters the bounce state, in which it quickly moves away from the ball then returns to its normal movement speed. The bounce is not constant to create a fun feeling imitating the behavior of a rubber ball, or spring. It is this bouncing behavior that will cause 2 blocks to overlap. After the

bounce completes the movement speed returns to normal. Finally there is interaction between the blocks. When a block is within 1/4 of the screen of any other block sprite it slows its speed to 1/2 of the previous speed. This condition is ignored if the blocks are past the ball as they cannot intersect and are no longer of interest to the player. Finally, if 2 blocks intersect, they freeze ending the game with a loss giving players a limited time to master the note, and making it a challenge.

- **Ball:** Controlled by the player's pitch input from YIN. This updates every time Yin is called and again takes in a frequency and outputs a hight using the matching formula from the wall section. All of the logic of ball interaction is calculated by the walls, so the ball behaves as a naive actor.
- **Note Labels:** Small images representing letters, accidentals (flat), or blanks. This is where the True Note implementation is most useful. Using a simple set math in Verilog one can map the true note representation into a set of memory locations that store the sprites for letters. This allows you to avoid creating a secondary table that converts from frequency to this representation just for this purpose and removes ambiguity with slight differences in pitch that may be identified by the note decoder and what the sprites were expecting.

VI. PHYSICS/GAME ENGINE

The physics and overall game engine is very simple and largely controlled by the behavior of the wall sprites simplifying the process by avoiding many interacting state machines.

VII. FILE STORAGE AND ACCESS

A. Big Picture

The File Access component retrieves binary data from an SD card over SPI and stores it in BRAM for processing. We use a simple custom file system to locate file start sectors and lengths from metadata in sector 0.

B. File Generation

Files are raw binary, and a Python script prepares the SD card structure:

- Sector 0: Header with start sector and length of each file.
- Subsequent sectors: Continuous blocks of file data.

C. File Access and Loading

The FPGA uses SPI commands (CMD0, CMD1, CMD17) to initialize and read SD card sectors. A state machine manages initialization, reading the header, and loading the file into BRAM. Data is transferred in 512-byte sectors and stored for downstream processing.

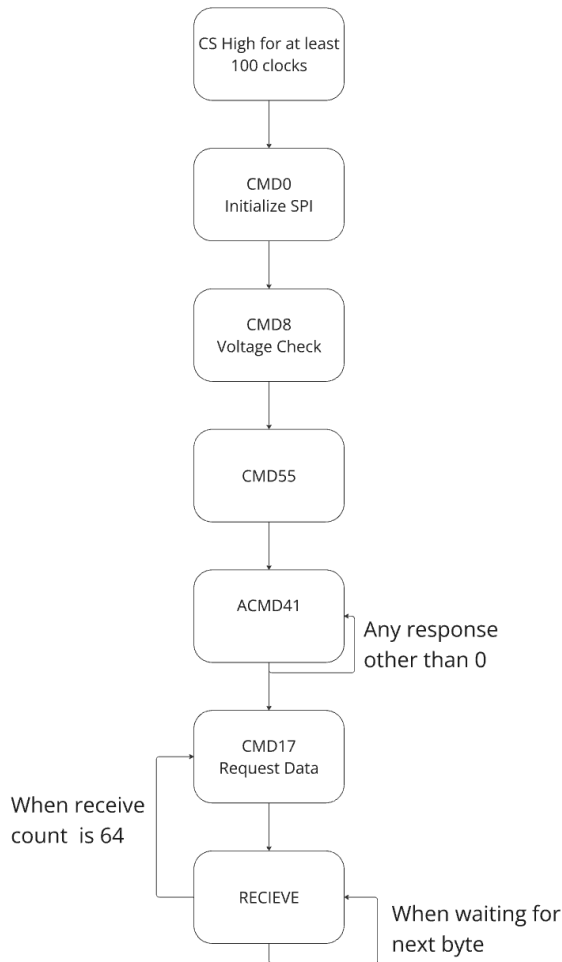


Fig. 6. Finite State Machine for SD Card Access

VIII. EVALUATION/GOAL EVALUATIONS

We reached our commitment goal for audio, which was to use an existing microphone to output pitch in real time. We did proceed to use a different microphone since the one we had already worked with proved to be adequate for the purposes of our game. For graphics our goal was to display past and future notes as well as sprite representations of these notes. We met this goal and even got to our stretch goal of retro graphics, implementing a "Flappy Bird" feel to the game design. For music, the goal was to dynamically select the first 32 notes of a song and output frequencies and this goal was met. We decided not to pursue the stretch goal of synchronously processing the audio at the same time as the singer because we determined this was not a good allocation of BRAM and other FPGA resources. Lastly, we hit our commitment for song loading, successfully loading from a mem file. Although we saw some success with the reading of SD cards, we ultimately ran out of time to implement this song selection feature into the game.

Vivado Log Analysis

- **Timing** The process of running YIN is challenging from


a timing perspective as it involves a number of multiplications and most importantly a divide. Therefore the largest goal in implementing it was to try and reduce the negative slack in order to make the design functional. Our final negative slack was -9 ns which though negative was still functionally enough to produce a functional output for both use cases.

- **Resource Usage** The Overall resource usage of the system is within the bounds of what the FPGA can provide. As both YIN calculations are done at different times, there is no competition for resources between them. Additionally our greatest worry, BRAM saturation, did not occur despite the music files being rather long when loaded onto the system. With the total amount of allocated BRAM being 73 kilobytes, that was more than enough to store our compressed sprites and segments of simplified song
- **Latency** While the latency of YIN is low enough for live input as discussed above. So is the case with the note loading process which is likely the most intense action that takes place during the process of playing the game. We have been able to limit that to the process of running YIN 32 times sequentially, which though it may seem time inefficient is more than enough to produce a seamless user experience as YIN has been optimized to be run many times per second to update ball position.

IX. CONCLUSION AND FUTURE WORK

This report outlines the full pipeline: from audio input, filtering, and pitch detection (YIN), to note interpretation from a .wav file, real-time sprite generation, and a physics-based game engine, all supported by file storage on an SD card. This project could be augmented by building on a bigger FPGA, allowing for longer songs and bigger files in addition to the integration of the SD card.

X. THE CODE

You can find the project repository on 

XI. CONTRIBUTION

All members of the group worked diligently throughout the semester to bring this game to life. Sethuraman focused on the YIN research and implementation as well as mic testing and sim data generation. He wrote the mic connection and tested various sampling frequencies to decide on the most accurate. Shoemaker generated audio files for testing and built song processing aspects. She researched making smooth sine waves and provided vocals testing for the project. Ruehle worked on graphics and researched/implemented the beginnings of the SD card connection to the game design. He tested different visual styles and font choices to make the game look like it does in its current iteration. All worked together on integration and testing of modules.

REFERENCES

- [1] A. de Cheveigné and H. Kawahara, “YIN, a fundamental frequency estimator for speech and music,” *The Journal of the Acoustical Society of America*, vol. 111, no. 4, pp. 1917–1930, 2002.