Lab 3A: Decreasing Latency using LUTs and Run-time Samples

Oviya Seeniraj

Purpose and Expected Goals

The primary objective of this lab was to design and optimize a **Chromatic Tuner** by leveraging a Fast Fourier Transform (FFT) for frequency detection using an onboard microphone. Expected goals included:

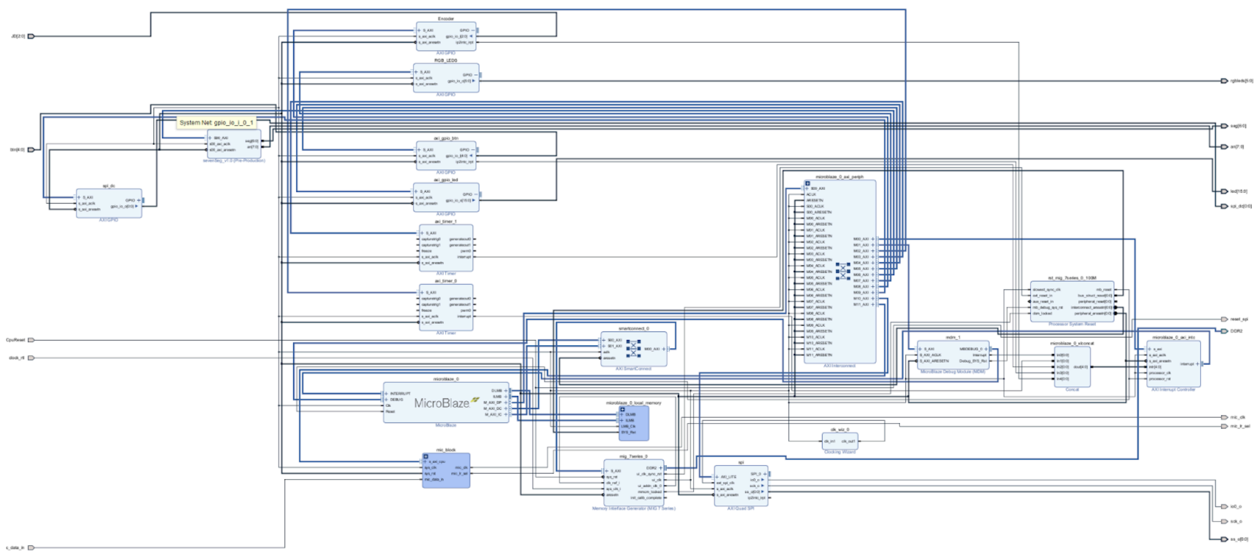
1. Optimizing FFT computation for low latency (< 25ms).
2. Configuring hardware peripherals (specifically a microphone and stream grabber).
3. Ensuring accurate frequency detection across a wide range of frequencies (210 Hz-5 kHz).

2. Methodology

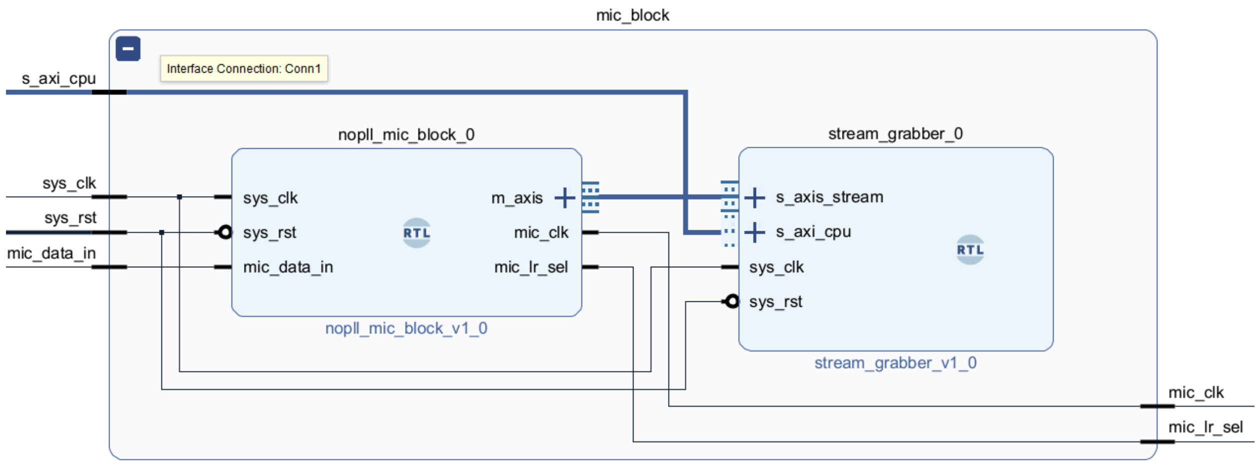
We develop the system on a Microblaze processor implemented on an Artix A7-100T FPGA board. To interface with the microphone peripheral, configure hardware in Vivado, enabling input capture via stream grabber. Perform frequency analysis using a Fast Fourier Transform (FFT) and enhance the initial implementation by precomputing trigonometric operations and streamlining repetitive sections of the code to improve performance. Optimize further by changing sampling and bin settings - though this may reduce accuracy for higher frequencies, it will significantly decrease computational latency to go below the threshold of 25 ms. By following this implementation, my final implementation meets this threshold (6 ms < 25 ms)

Results

1. Vivado Block schematic



2. Mic Block Internal Wiring



3. Steps taken to optimize original FFT code

In order to optimize the original FFT code, I implemented a lookup table (LUT) to pre-compute sine and cosine functions, lowered the sampling rate, and made minor changes to the code in terms of placement and factoring out reused code.

In terms of the lookup table, I replaced the sine and cosine call in the fft file with a lookup in a table I created before I call the grand loop in main. In order to ensure accessibility, I made the lookup tables that I declared in fft.h externs so I could initialize them in main. I implemented the lookup table as a two-dimensinal array, since sine and cosine were called using two variables: k and b, each of which was one of my dimensions. Thus, we replace each sine/cosine computation with an array lookup, speeding up the process thoroughly. After a few iterations, this alone brought my program time down from ~1300ms to ~92 ms.

Then, I lowered the sampling rate in order to increase latency. I downsampled my signal by taking every 4th signal from the stream grabber. This made my sampling rate 48/4 → 12kHz, which is lower than the initial FFT. I then changed the number of bins to 128, so that I could ensure the same bin spacing and frequency distinguishing abilities as the original FFT. (48000/516 = 12000/128). However, the one major change of this is that according tot he Nyquist Theorem, we can now only detect up to 12kHz/2 = 6kHz rather than 48kHz/2 = 24kHz. However, for our purposes of music and the testing data given, this is enough. Implementing this resulted in a latency of ~55 ms.

Lastly, I made small adjustments to my code, removing all possible unnecessary statements. For example, I moved the sample_f calculation outside of the while loop since it was a constant and replaced repeated multiplicative statements with a variable. I also moved the frequency print statement from above the timer stop to after it: the latency of the print was slowing down my calculations. These and some more minor adjustments were the pushes needed to reduce my latency: I was able to run my program with a program time of 6 ms after this.

4. Was there a difference in the FFT computation for low and high frequencies?

In terms of the data tested, both could be found relatively accurately. My implementation method, while it had some error, did not output the constant 170-190 values instead of 0 Hz that I had originally mitigated with an if statement, which also reduced my latency. My error was higher for lower frequencies, but higher frequencies past 6 kHz could not be accurately detected due to my sampling rate of 12kHz. Thus, there are differences, but overall, the tested data reacted similarly.

5. Performance comparison

My FFT implementation is by far superior to the given default code. As described in question 2, I brought the latency down from 1300 ms to 6 ms, which is a 99.6% reduction in program runtime.

6. Performance analysis

According to the performance analyzer, I spent this much time in each function

- Fft: 53%, code segment length = 2376, goodness metric = 0.53 * 2376=1259.28
- Fsl_read_values: 29%, code segment length = 916, goodness metric = 0.29 * 916 = 265.64
- Main: 21%, code segment length = 1673, goodness metric = 0.21 * 1673 = 351.33

Video:

IMG_8084.MOV

https://drive.google.com/file/d/1_4shLELWDd1bd6YqYdkmHwFS5HIUFJRl/view?usp=drivesdk