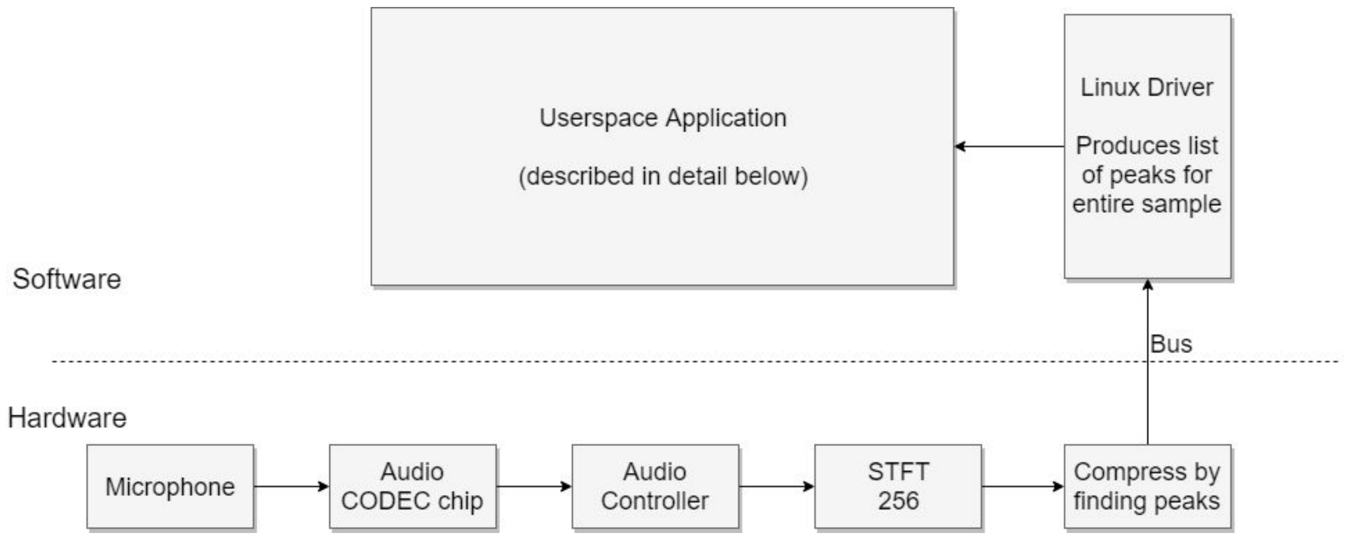
Jose Rubianes (jer2201), Tomin Perea-Chamblee (tep2116),
Eitan Kaplan (ek2928), Mikhail Belov (mb4560)
CSEEW4840 Embedded Systems
Design Document

Totally Not Shazam
Song Recognition

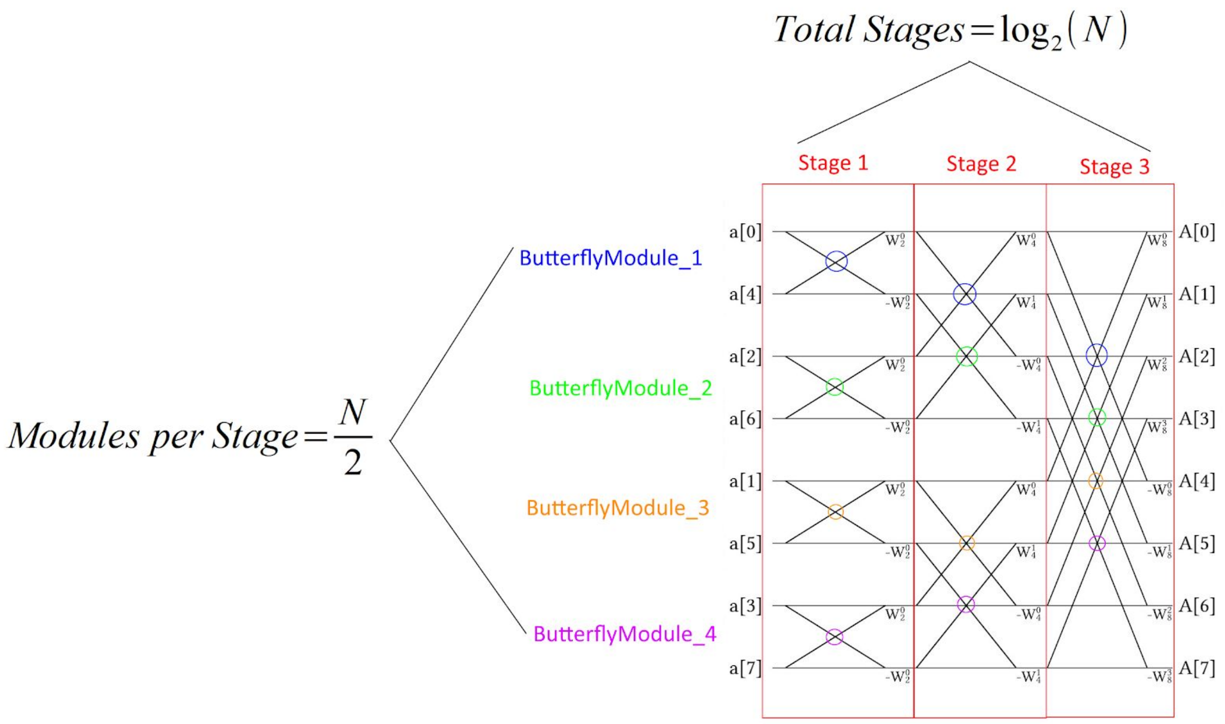
Summary:

Use an external microphone through the microphone port on the development board to recognize songs being played in the background. Song recognition will depend heavily on signal analysis, which will be accelerated by a custom FFT unit implemented on the FPGA. Please find the diagram summarizing the project below.



Hardware:

We plan to calculate the STFT by implementing the Cooley-Tukey algorithm, so we can avoid large matrix multiplications. The base hardware block needed to accomplish this would be a module (ButterflyModule) that performs a single radix-2 FFT calculation. Then, this module can be replicated and pipelined to calculate any N-point FFT as shown below.



Since the structure of the pipeline can be determined by the number of points, this design should make it easy to parameterize the accuracy of our STFT. Furthermore, pipelining our STFT should allow us to achieve higher throughput, and thus a higher maximum sampling frequency.

In the case where our accuracy requirements are too high, and our pipeline stages become too wide to be practical, we could instead have one butterfly module per stage, and calculated each stage output serialy. That would reduce the number of modules in the pipeline from $\frac{N}{2}log_2(N)$ to $log_2(N)$, while decreasing our throughput by N/2. But since most songs are sampled at a rate of only 40 kHz, this decrease in throughput shouldn't be an issue for our purposes.

Interface with Software:

Data produced by hardware module will be stored in a memory mapped buffer. Both the STFT result produced by the aforementioned pipelined circuitry and the value of a counter that tallies the number of STFT results thus far computed will be placed in the buffer. The driver will read from the buffer to retrieve the data. Once the driver begins reading, the buffer will not be overwritten until the data retrieval is finished. However, the hardware makes no guarantee that the next sample seen by the driver will be the next sample sequentially (i.e., if the driver is too slow retrieving data, some data may be dropped). Our algorithm is robust against occasional missed sample (as long as timing data is preserved -- hence the need for the sample counter).

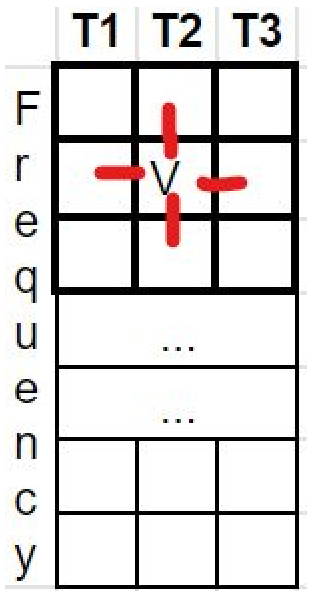
What the data is that is stored in the buffer will depend on the final decision regarding the peak finding module. It will be one of three things:

- A bit array, where a one represents a peak at a certain frequency at that time.
- 4-8 frequency-amplitude pairs, representing the locations and amplitudes of some significant peaks.
- The raw STFT data. Each sample (i.e., each filled buffer) represents one FFT from the STFT.

Peak Finding/Compression Algorithm:

The FFT module will produce one FFT at a time. The next step is to find the amplitude peaks of the STFT. One option is to find the peaks in software. However, each FFT will be N values (for now, we are assuming N=256), and since those values are 24 bits wide, it would be quite a lot of data to send on the bus to the processor so that the peaks could be found in software.

Therefore, we plan on on finding the peaks in hardware. While it is reasonable to find all of the peaks in hardware, the algorithm relies on using only the most “significant” peaks (which roughly means the largest valued, reasonably spaced peaks). It is not trivial to find only the most significant peaks in hardware. We are considering two possible options for doing so, one which is easier to implement, but may not succeed in finding the right number of peaks and being robust against noise; and one which is more difficult to implement and uses more resources, but is more likely to allow us to get the right peaks to produce good fingerprints. By the Milestone one, we plan on implementing both of the possible algorithms in software to see if they are effective, and make a decision from there. The algorithms are described below. If neither is effective, we will simply send the entire STFT on the bus to software, and deal with the resulting latency.



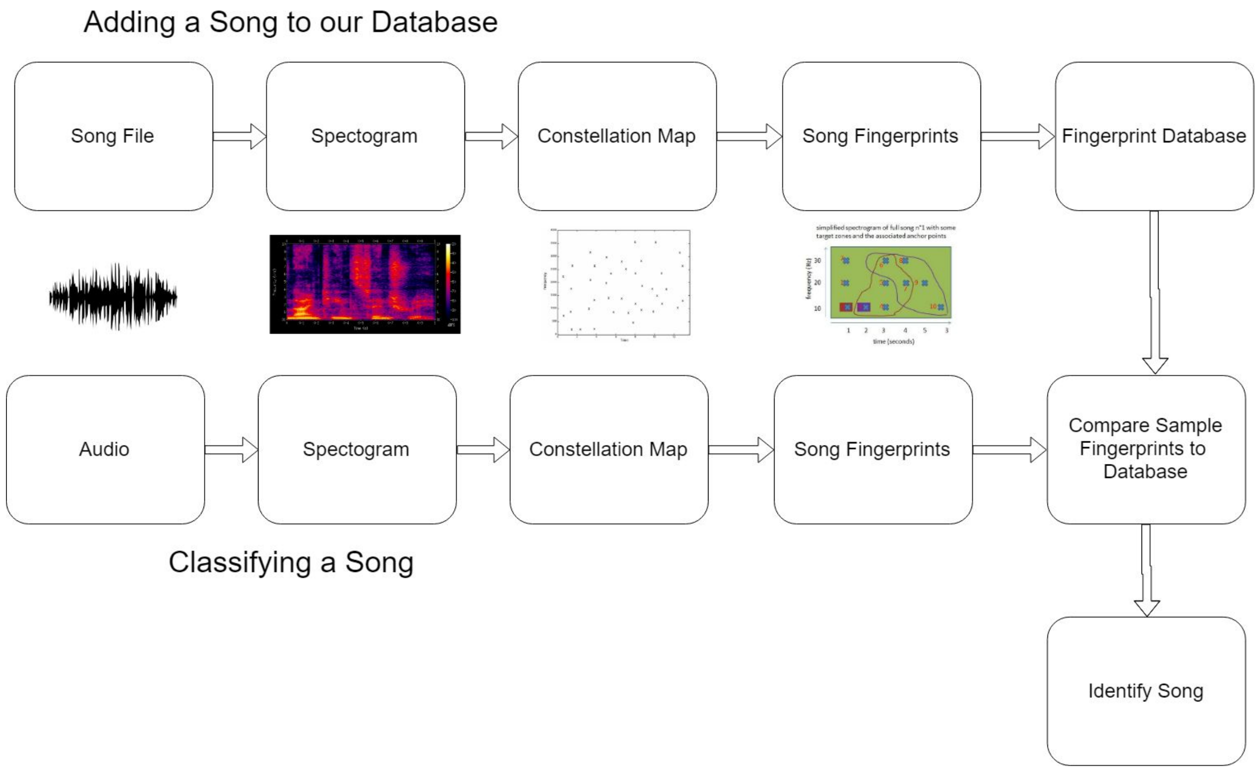
Both algorithms start the same way: at any given time we will store three FFTs, for consecutive STFT time samples. For each value in the middle FFT, we will check if it is greater than all four neighboring values (marked in red in diagram). If it is, then we will mark that value as a peak.

At this point, we can flag all peaks. The simplest thing to do, would be to create a bit array representing where the peaks are within that particular STFT time sample. However, as mentioned above these peaks must be narrowed down to only the significant peaks. One of two strategies will be used to do so:

A) In addition to being larger than its immediate neighbors, add an additional requirement that the amplitude must be greater than some dynamically determined value (perhaps determined using exponential smoothing for each frequency band). Then, the resulting peak locations will be represented in a bit array. That bit array will be what is finally sent to the software portion of the system. Note that in this approach all amplitude data is lost, so the software cannot rely on amplitude data to further prune the peaks if necessary. Assuming 24 bit amplitude data (what the CODEC provides), this results in an 24x reduction in necessary bus throughput.

B) We will divide the N frequencies into a fixed number L of logarithmic bands (L will likely be between 4-8). In each band, we will find the peak candidate with the max amplitude (the max function will be implemented with a comparison tree). Then, for each of these L max peaks, their frequency and amplitude information will be sent to the software. The advantage of this approach is that a fixed (maximum) number of peaks is sent for each STFT sample, which allows us to keep the interface simple and preserve the amplitude data, while limiting the necessary bus throughput (this approach may even require lower throughput than approach A, depending amplitude data width necessary, and on the number of bins). The main advantage of this approach is that the software has more to work with when it comes to further pruning the number of peaks. However, the max operation required is expensive.

Software:



The algorithm for classifying songs involves finding peak frequencies over time, then using those peaks to generate fingerprints that act as keys (hashes) for a large database (hash table) of cataloged songs. This is accomplished by identifying the most important frequency ranges that humans can perceive (bins), and, then, finding the most significant frequencies that occur in those ranges in any song (local extrema within the selected bins of frequencies in the Fourier transform of the signal). Afterwards, the identified frequency peaks form a unique characterization of a song that is called a fingerprint which is stored in the hash table (a more descriptive and detailed explanation of hash table generation that can be found here. It is noise intolerant and time invariant. Also, it ensures consistency over recording hardware, volume, levels of background noise, etc). Generating these hashes for incoming audio allows for quick searching of possible song matches, rather than sequentially searching for a match across the entire catalog (songs which have already been fingerprinted according to the same algorithm, and so the catalog is generated or retrieved ahead of time).

We already have a functioning Python prototype that is able to classify songs with high accuracy. We need to optimize our existing implementation, then translate it to C++.

Projected Milestones:

Milestone 1 - 04/05

- Get input from the mic-in port to our FPGA module. Verify that signal is being received by using LEDs to display level.
- Using software model (in Python), determine which of the potential peak finding algorithms work. Assess which working algorithm will be most practical to implement in hardware.

Milestone 2 - 04/19

- Implement STFT logic in hardware.

Milestone 3 - 05/03

- Implement peak finding hardware (if practical).
- Translate Python software model to C++
- Write driver.
- Pass one song through system, and generate fingerprints for that song.

Final Project

- Finish userspace application
- Add songs to our hashtable (produce the data on a workstation and just copy onto board SD card)
- Tweak parameters to increase accuracy.