The Music Box

By Phoebe Piercy and Elina Sendonaris

Overview

Our vision was to create some kind of 'black box', into which you would feed an audio file, and get out information about the parameters of the music, which would then be displayed, via VGA, on a screen. The parameters we chose were instrument, pitch and tempo, with stretch goals of more complicated things.

We will define three tiers: **Commitment Goals, Goals** and **Stretch Goals.**

COMMITMENT GOALS

- **Display**
  - A basic display that will show the information we have calculate, as described below.
- **Tempo Detection**
  - By passing the audio signal through a low pass filter, in order to isolate the low frequency drum beats from the melody, we would then transform these into a discrete signal of beats and analyse the resulting waveform in the time domain. By calculating the frequency of the peaks, we should be able to output a tempo for the music.
- **Instrument detection**
  - By feeding our system an audio file containing notes played by a single instrument, we should be able to extract some information about the timbre of the instruments, and thus broadly categorise it's timbre and thus give an idea as to the type of instrument being played. This is done through the harmonics, as instruments with more 'husky' timbres, such as saxophones, have a lot more harmonics than more 'clear' instruments, such as pianos. Below is an example of this. We will use peak detection on a short time FFT sample to count the number of harmonics present.

GOALS

- **Time Signature**
  - In specific tracks, the first beat of each bar is more pronounced. Our goal is to be able to detect this and then, from this, to calculate the time signature/number of beats in a bar. This might only work with tracks that we make and that we design to have this pronounced beat, and that will be our initial goal.
- **Instrument Detection**
  - Ideally, we will be able to distinguish between families of instruments; brass, woodwind, string, vocals etc. We are not committing to this as we are still working out whether we will be able to get down to this level of precision. It might require looking at the shape of the harmonics and how they drop off, and it might be that we don't have time to get this specific.

    - We should also be able to distinguish between pitch of instruments, so will be able to categorise it as an instrument in a bass, tenor, alto or soprano range.

- **Chord Progression Detection**
    - Ideally we will be able to feed a simple ballad into the ‘Music Box’, and tell it the time signature. This would allow it to calculate the tempo from the beat, as discussed above, and therefore calculate the number of clock cycles between theoretical chord changes, giving us a sample rate. From here, we would do an FFT calculation for each sample/chord, and use this to extract the fundamental frequencies.
- **Display:**
    - We would like to implement multiple modes for the display.
        - Basic info - Tempo, Time Sig, Detected frequencies, Notes.
        - Instrument family, range and number of harmonics.
        - Chord Progression diagram

STRETCH GOALS

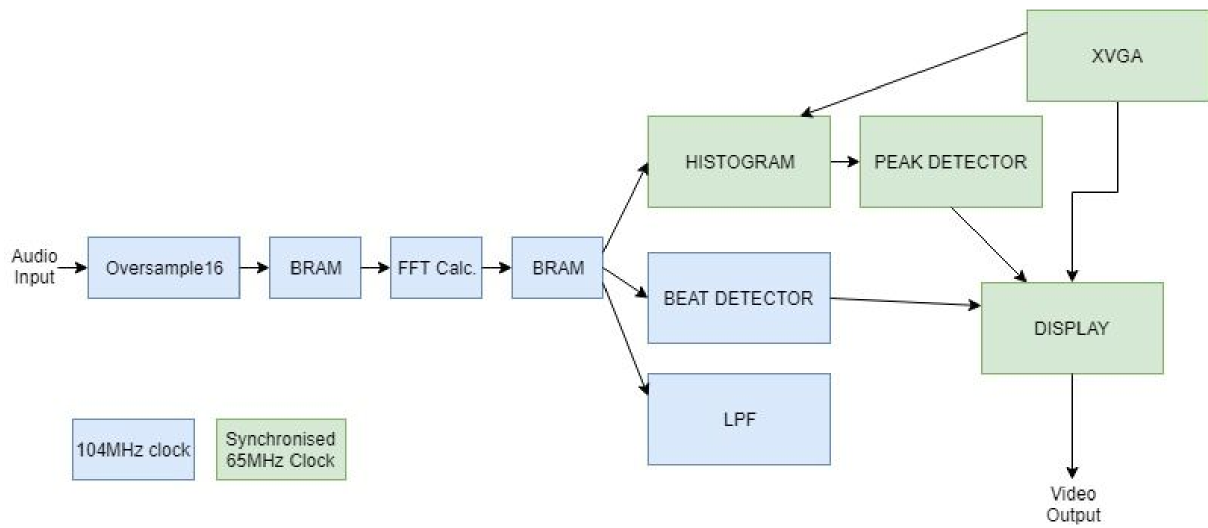
- **Key detection**
    - By passing the signal through an FFT and analysing the frequency spikes, we can map the most common frequencies to their corresponding notes (taking harmonics into account). From this, we should be able to discount accidentals, as they should occur with a much lower frequency, and so detect which sharps and flats are being used, and, from this, isolate the key.
- **Display:**
    - We would like to implement more live functionality for the display.
        - Score - Draw staves with the chord progressions detected

Overall Structure

Main module: nexys4_fft_demo

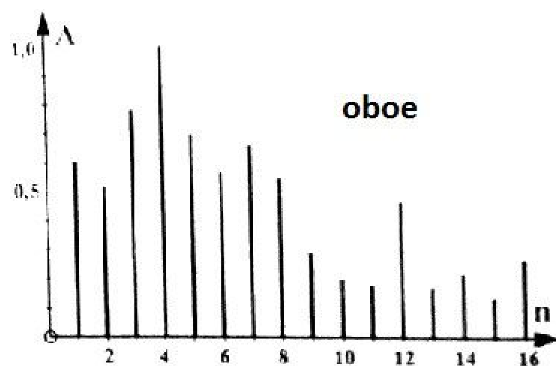
- Utils.v
    - debounce
    - Level_to_pulse converter
    - Display_8hex
    - Pwm11
    - Synchronize
    - Xvga - instantiate VGA signals
- Oversample16.v - to oversample the input audio by 16x
- Bram_to_fft.v - extract audio info from BRAM and put it through fft calculations
- Fft_mag.v - calculate fft magnitudes
- Bram_fft.v - store magnitudes once calculated
- Histogram.v - displays and extracts info from fft data
- Peak_detector.v - processes information from histogram.v
- Display.v - provides all the display text and processing
- Lpf.v and fir31.v - low pass filters
- Beat_detector.v - computes tempo

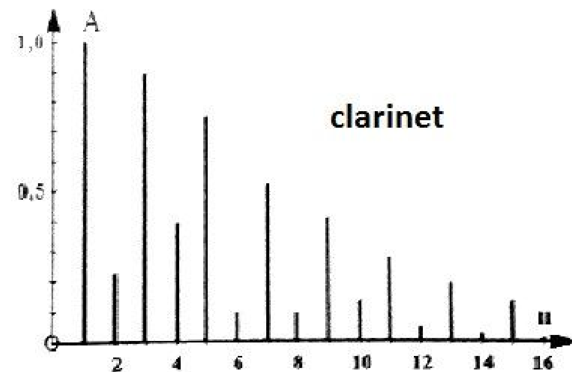
Overall Block diagram

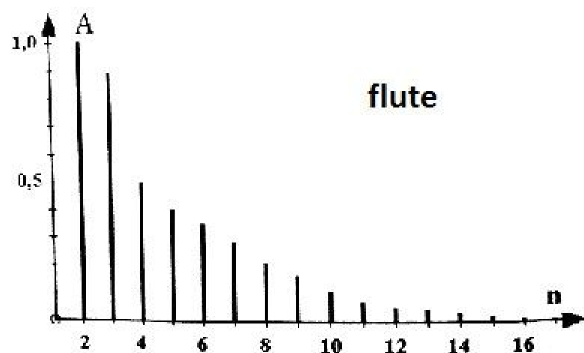


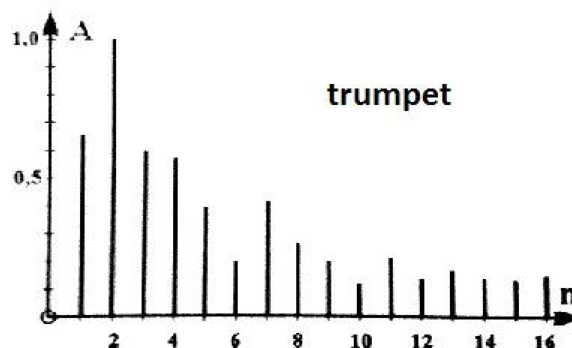
Frequency & Instrument detection
Modules: peak_detector and histogram
Author: Phoebe Piercy, with the histogram code taking its base from Mitchell Gu

The Idea
Once we had decided on the concept of a music analyser, we started to look at what information we could discern from an FFT. The fundamental frequency (and stretch goals of keys and chord progressions as stretch goals) seemed the obvious ones, but we wanted to go further and find something a little more unusual. I remembered something my music theory teacher in high school said about instrument timbres, and how differences in the overtones/harmonics are what gave each instrument its distinct sound. We did a little research to determine whether this was something we would actually be able to distinguish between. It appeared that the two main factors in determining timbre were the number of overtones, and the shape of the harmonic series/the intensity of each of the overtones, as shown by the examples below.



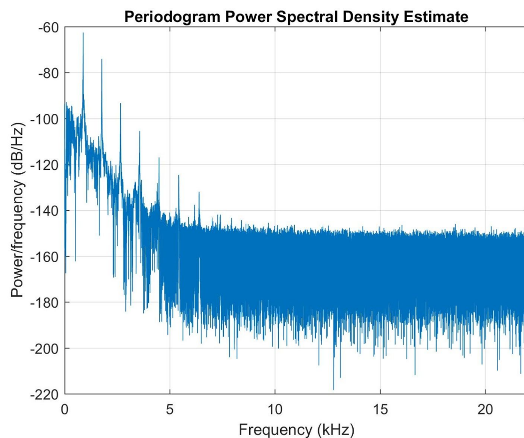




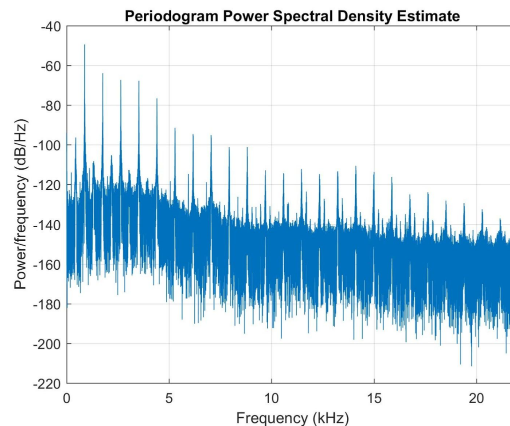


This is still an emerging field, and so I had no illusions about being able to distinguish between specific instruments (except perhaps in very distinctive cases, such as the clarinet's weaker even overtones), however we decided there was potential here to distinguish between instrument families.
The next step was to verify this ourselves. We ran a simple MATLAB FFT generation script on .wav files of instruments playing single notes, that we generated from our music notation software, and saw very clear, and most importantly, distinguishable via verilog, differences in the harmonic series. A couple of example of our resulting graphs are shown below:

PIANO A4



ALTO SAX A4



Quite honestly, we should have done more research before jumping headfirst into this as a commitment goal, but I honestly believe if we had, we would never have commited to it and, if we hadn't committed to it, we would never have got it working.
The next step for me was to understand the FFT demo module that was posted on piazza, so I could edit and add to it in a way that worked with the rest of the module and didn't mess anything up. I went through and wrote out pseudocode for it and, once I felt as though I had a fairly solid understanding of it, I started to add to it.

Peak detection

My first challenge was peak detection in order to count the number of harmonics in the fft. This needed to be done as soon as possible, so we could get a better understanding of whether this information alone would be enough to discriminate between basic instrument families. It was initially slow going, as I tried to manipulate magnitude_tdata and magnitude_tuser to get the information I needed for the top 1023 pieces of information, and had not realised the functionality of the switches in terms of filtering out noise. After realising that the magnitude data was not outputted in order, I decided to simply emulate the histogram function, which had a neat way of extracting the desired information and pipelining the process. I ended up taking over this module and revamping it for my own uses It took in a 65MHz clock and outputted an address (haddr) as needed to access the desired data in the BRAM. I didn't need anything more detailed than the histogram granularity so I used this module as my main 'information gathering' module.

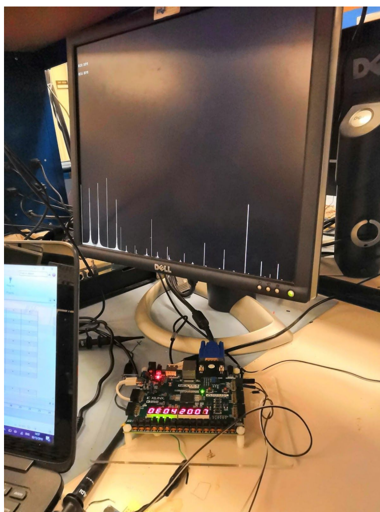
My first implementation was very simple, with the idea of getting the basic data flow in place, and then building up the actual algorithm from there. I simply checked for a peak (defined as a pixel with no pixel before it, above a certain amplitude threshold defined by a hard coded value of vcount, as the VGA scans from left to right), and incremented a value stored in a register called hpeaks, which I would reset to zero every frame. I initially had trouble with this, as it would simply keep counting upwards, something I realised was because I was only resetting every frame, whereas I should be resetting every vsync. However, this still wasn't

correct as it was now resetting to zero before I had time to catch the number of peaks, as it would stay at zero for any vcount less than my limit. On top of this, the count was so quick as to be indecipherable on the hex display I was using for testing.

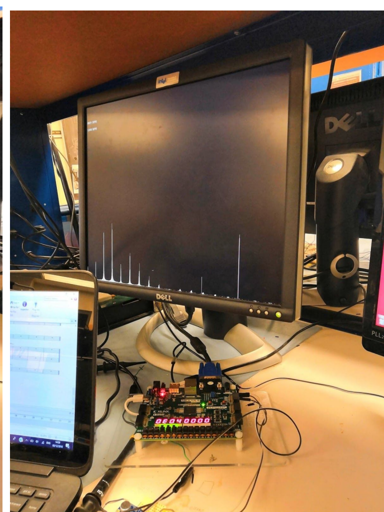
It was around this time that I was also running into compiling issues, in that each of my implementations would take over half an hour to run, despite me using very little resources, which made debugging painful. This ended up being a crossed clocks issue, where I was trying to use a 104MHz synchronised vsync cycle in the histogram module, where I should have been using the normal vsync, with a 65MHz clock, as required by the VGA. Once that was sorted, I was able to fix my issue with hpeaks, by creating two ‘peak’ registers, one of which (hpeaks) did as before and counted up the number of peaks, but this time only on one specific value of vcount, the value for which determines the sensitivity. The other, ‘peaks’, is much more steady, and at the end of each frame **(vcount = 766)**, it would get set to whatever value is in hpeaks, which represents the number of peaks at the set volume limit.

At this point we were still relying on the microphone, which was less than ideal due to the amount of noise and the lack of clean peaks. Elina and I talked about other ways of processing this, including loading sounds through a sim card, but we really wanted it to be in real time, through something with a lot of flexibility in terms of potential sounds to play with. After some research, we settled on a TRS aux cable, as they were fairly cheap and common, and we cut, split and soldered it, and then simply ignored one of the connections and only used the ground and ‘right ear’ wires. Again, we weren’t entirely certain how this was going to work, so I used the oscilloscope to check the output signal before plugging it in, to ensure the signal was within voltage limits and wasn’t going to fry our Nexys, but it ended up being the saviour of this project. With some filtering, we were able to produce very clear and smooth FFT data, especially when playing professionally recorded MIDI sounds, which allowed us to directly use the full range of instruments in the Sibelius library.

Flute



Violin



Trombone

So now that we had a working peak detector, the next step was to translate this to instruments. After testing and recording the number of harmonic peaks, I tuned the threshold down to 20 and realised that even then, it was not going to be possible to tell the difference

between instrument families in any detail. We could get some broad ideas, but there simply weren't enough repeatable differences between the instrument families to tell the difference only using the number of harmonics.
Whilst we did further research into how to better distinguish between instruments, I started work on the frequency detection, as I knew this was going to be key to whatever further parameter we decided to use.

Fundamental Detection

This one fell out fairly smoothly. I simply added a 'locking' register to store the hcount location of the first pixel found in the histogram on each frame. The one sticking point was the existence of some unseen pixels at hcount = 0, so I had to add a check so it only triggered if hcount was greater than 1, which felt a little morally dubious, but which fixed the problem completely and so, for lack of more time to discover what these pixels were, that was the solution I kept. The other issue I discovered was that my frequency calculation from the address, which was $f = vaddr \times \frac{f_{nyquist}}{4096/2}$ was off by 40Hz, as tested against pure tones at given frequencies on Youtube. This led to a subtraction of 40Hz in my calculation which, once again, felt a little morally dubious, but fixed the issue and did not cause any further issues, and so, is the solution we kept.

Therefore our final calculation was $f = vaddr \times \frac{sampleRate}{2} \times \frac{1}{4096/2} - 40$

Once I had the fundamental peak's frequency, I passed this to my currently unused peak_detector module, which I had now assigned the role of processing the information that the histogram module outputted. I created a lookup table to output a visual hex representation of the pitch in the form: note (where 6=G), sharp (8) or flat (b), octave (for example, F#4 was output as 12'hF84). Eventually Elina took these and turned them into displayed ASCII characters, but for the time being I was testing them on the 8 character hex display. Here I found my biggest problem, which had been a recurring issue with a lot of what I'd been doing up until this point, which was that the peaks would reset too early and so the lowest note in my lookup table was constantly displayed.

I tried various fixes to this, but nothing was helping, so in the end I gave up on automatically resetting the peak values. Instead, I forced it to update 'peaks' only when 'hpeaks' was greater than its current value, so it was always displaying the maximum number of peaks detected up til that point, and I also put in a manual reset on the BTNC button, which would put 'peaks' back to zero. This way, for now, we would only be able to test instruments played one at a time, manually triggered with a manual reset in between.

Top Harmonic

So, finally, we had a working detector for number of peaks and frequency detection, but still no accurate instrument detection. Coming back to this problem, I looked at the shapes of the FFT I was seeing for different instruments and the differences between them were clear, I just had to work out how to tell the computer these differences. The main thing I was looking for was how quickly the overtones fell away, but I wasn't entirely sure how to check this and, the more I looked online, the more I discovered just how little hard fact there was out there.

Everything was vague and conceptual, whereas I needed hard distinctions. I decided to see if I could get a clearer distinction based off of the frequency of the uppermost overtone (that exceeded the volume limit I had set). The ratio of this to the fundamental would, I hoped, give me an idea of the way in which the overtones fell off, and thus give me another filter. My last resort was going to be a detailed look at the intensity (**vheight**) of each harmonic, but I was still trying to avoid that if possible, as I knew it was going to be tough to achieve in the scope of the project.
I used a simple 'if' test to find the address and therefore frequency of the top harmonic in the series (stored in **top_har**). By this stage, I had become far more adept at the structure of the fft and how to manipulate the data streams, and so the implementation of this went fairly smoothly.
Finally I started testing again, with the new parameter in place. It took a long time to get the parameters correct, and even then there were still issues with distinctions. Keyboards were coming up as strings, and the line between woodwind and brass was far too blurry. The location of the split between them seemed to change every time I came back to the station and so, even if I could get it to detect it correctly once, it was unlikely to still be working after a coffee break, for example, leading me to be constantly changing the parameters in the hopes of nailing it down. I was also forced to add a manual test button, so that it would only change instrument when that button was held down, to allow us to actually see the results of the test, as otherwise it would keep flipping back and forth, even when no note was being played. With this manual button we were able to see some correct classification of instruments, at least enough to distinguish wind/brass from keyboard, but it was still far from ideal.

Getting it working

So, whilst I tried to work out what to do about this, I added some polishing factors. I increased the pitch LUT to cover 2 octaves, set the pitch to '0' when there was nothing being played **(fundamental vheight < 50**) rather than just defaulting to the lowest value in the LUT as before (it had been constantly displaying E4 as a default value). I got rid of the manual test and manual reset buttons and replaced them with a volume check on the fundamental, so that it would start testing when it was above a certain value and reset hpeaks, peaks and note when it dropped below it, thus allowing for smooth, automatic updating. I tried to add a volume component, so that the vcount level at which peaks were being counted would vary based off the height of the fundamental on that frame. However, this ended up messing with our counted peak values too much and made the system far less reliable, so I went back to a set sensitivity.

Finally, with a module now functioning as it should, I came back to the crux of the issue. How do I discriminate between instruments based on the information available? For the keyboard, I added another check **(hpeaks <= (peaks >> 1))**, which takes into account how quickly the note dies away. This allows for distinction between instruments that hit a note (e.g. keyboard, guitar) and instruments that can sustain notes (e.g. wind, string with a bow). Again, this is a distinguishing factor that could be developed further, but served its purpose in our case.

Finally, I sat down and crunched some numbers, in the hopes that patterns would present themselves. The instruments I had been testing with were:

Woodwind: Flute, Oboe, Clarinet, Alto Sax
Brass: Trumpet, Trombone
Keyboard: Piano, Keyboard
Strings: Violin, Cello

And here were my results (tested on different notes to ensure generalisable to different pitches)

Family	Instrument	No. Peaks	Fundamental /Hz	Top harmonic /Hz	Ratio fund:top
Woodwind	Flute	16	~660 (E4)	8425	12.76
	Oboe	16	~587 (D5)	6370	10.85
	Clarinet	15	~523 (C5)	6190	11.83
	Alto Sax	***41***	***440 (A4)***	***7825***	***17.78***
Brass	Trumpet	38	440 (A4)	7810	17.75
	Trombone	28	220 (A3)	5005	22.75
Keys	Piano	14	440 (A4)	3085	7.01
	Keyboard	13	440 (A4)	3535	8.03
Strings	Violin	16	~784 (G5)	10855	13.84
	Cello	23	440 (A4)	6085	13.8

As you can see, the patterns are fairly obvious and easily distinguishable, with one exception. My issue had been that I was forcing the saxophone into the woodwind section. Saxophones have always been a grey area, in that they are, technically speaking, woodwind instruments, since they produce sound from reeds, but they actually have a timbre very similar to brass. As a saxophonist, I can't count the number of times my school band stuck us in the orchestra when they needed another horn, and in jazz bands, the section labelled 'horns' actually comprises of brass instruments and saxophones. It made complete sense for there to be a scientific explanation for this common substituting of saxes for brass, namely that they actually have an overtone series similar to a brass instrument and therefore can blend in a similar way with an orchestra.
So, in order to solve this, I simply gave up on 'correctly' classifying the saxophone. It's timbre classed it as brass, and so I let my detector, which is designed to discriminate based on timbre, classify it as brass, and, once I'd programmed in my new values, the detector worked surprisingly well. We were able to successfully distinguish between woodwind, strings, keys and brass on the instruments we'd been testing with, on different notes, and even using other instrument sounds that we imported for testing, such as the cornet.

Our final test was to plug an electric guitar into the system and have Shreeyam play live. It was not able to detect the instrument type successfully, due to the excessive noise added by the amplifier, but it was still able to identify the pitch of the note being played, including the pitch of the root in a triad chord, which I was impressed by.

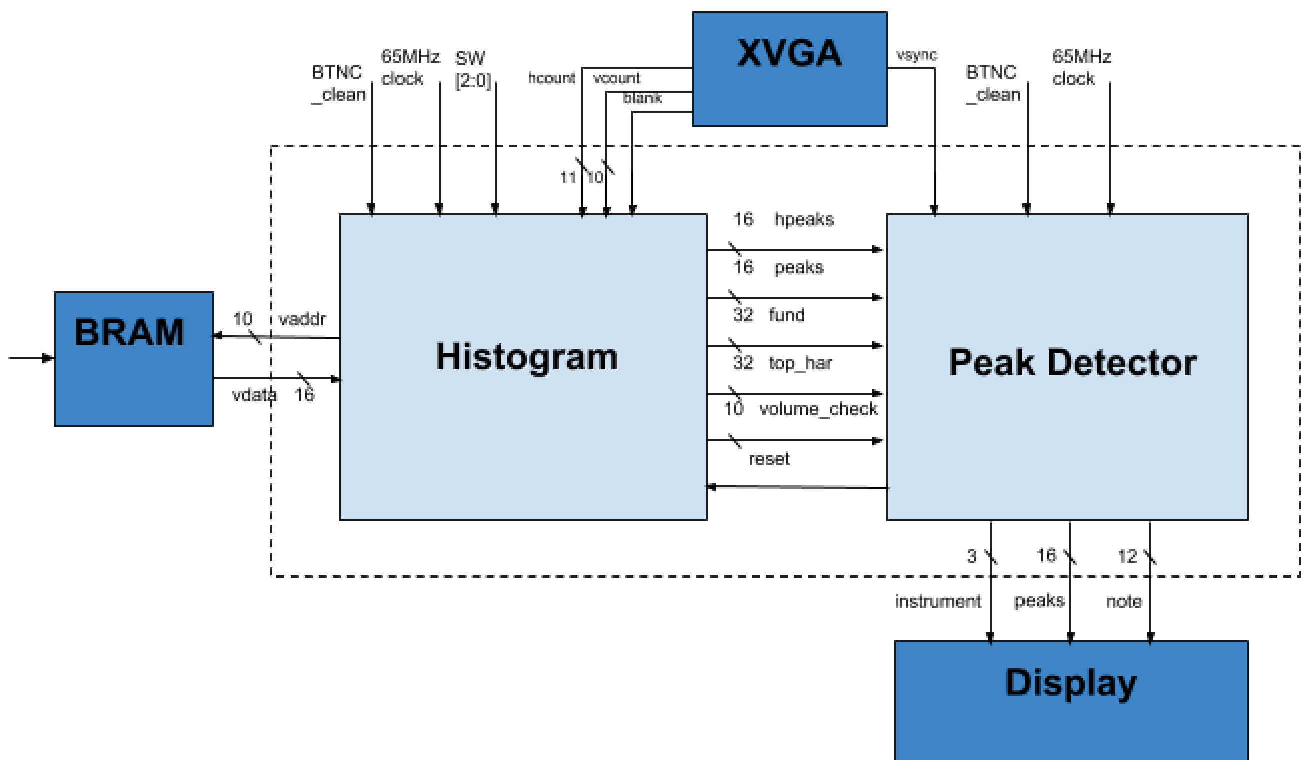
The most satisfying thing about this project was learning something new from something we had built. To be able to see the reason why saxophones are treated as pseudo-brass, and for the distinction to be so clear, was fascinating.

Future

I now fully believe that more time to develop this model could lead to fairly accurate instrument detection, even between instruments within families. Things I would have liked to implement with more time include using the pitch to further classify as bass/tenor/alto/soprano within a family, and then an attempt at distinctions between instruments themselves, and more testing with live instruments to try to get it work outside of the scope of MIDI sounds. Furthermore, I now believe that one of our other ideas, which we did not manage to implement, chord progression detection, is entirely possible and is something that I would also have wanted to implement.

Block Diagram

This ended up a little different to our original design. We combined a couple of modules into one, and got rid of a couple that ended up being unnecessary, but the basic form of FFT to detector, to information processor, to display was maintained. Below is the block diagram of what my side of the project ended up looking like.



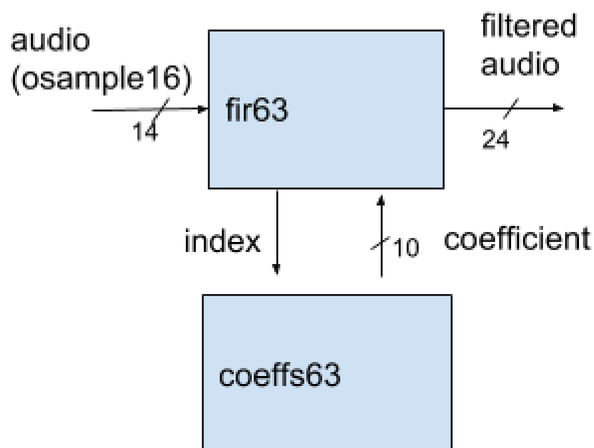
Elina: LPF, Beat Detection, and Display

Low-Pass Filter

Overview

We wanted to implement a 63-tap finite impulse response low-pass filter to complement our beat detection system, since the beats of a song are usually more prominent in the bass. The plan was to run the beat detection system on low-pass-filtered audio in order to limit noise and interference from high-frequency parts of the song, such as voices and instruments that are not drums or bass. Although we did not end up integrating the low-pass filter and beat detection system, for reasons mentioned in the “Difficulties” subsection of the “Tempo” section, we have a working low-pass filter.

Block Diagram



The design of this module was heavily influenced by the fir31 module from Lab 5. Effectively, the fir63 module does the processing of audio and relies on the coeffs63 module for the necessary coefficients to multiply the samples by.

Design

I got the coefficients using Matlab’s fir1(n, Wn) command, which takes in the order of the filter (n) and the normalized frequency (Wn). For our purposes, I wanted a cutoff frequency of 100Hz, which would isolate the bassline and drums. For the sampling rate of 62.5kHz, I needed n=62, corresponding to order 63, and Wn = 100Hz/62.5kHz *2π = .01. I put these coefficients into the coeffs63 module.

Testing

The first step of testing the filter was to see its impulse response, similarly to in Lab 5. The desired output was the coefficients of the filter, in order. This would confirm basic functionality, but not necessarily correct operation on more complex signals.

After confirming basic functionality, I tested the LPF by piping in its output to the FFT module, because you can more clearly see the frequency response in the frequency domain. I saw an extreme attenuation of high frequencies, with a sharp peak at 0Hz. There was an increase in the magnitude throughout the whole spectrum during heavy beats. Because of this, I assumed that the filter was functional.

In the final design, you can switch what is being passed into the FFT using the 3rd switch from the right.

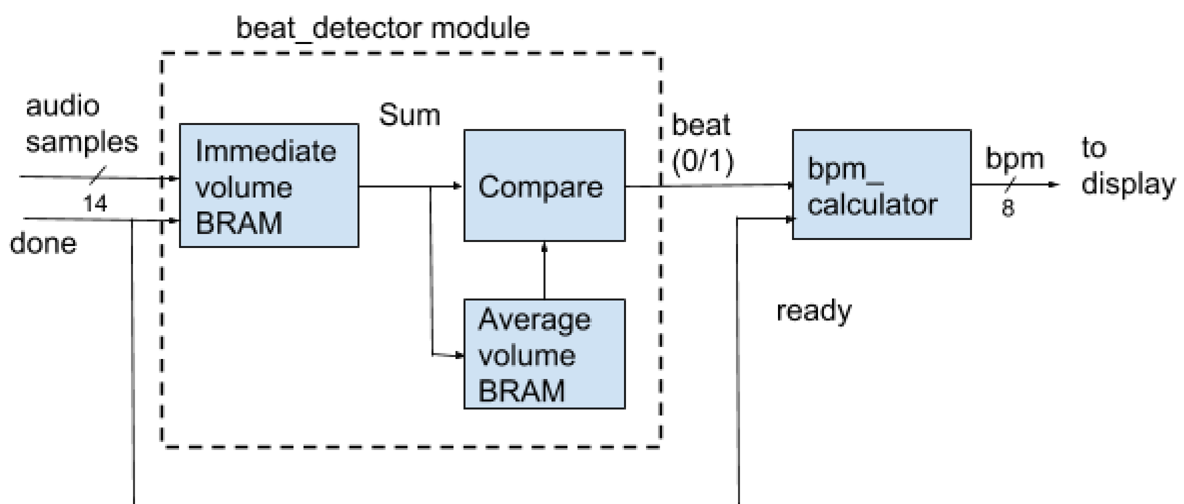
Tempo

Overview

One of the most important and basic aspects of a song is its speed, so I decided to detect the tempo of a song in the most commonly used units, beats per minute (BPM). The tempo usually manifests itself in songs via heavier drumbeats and bass-line on the beats, which humans instinctively hear and respond to. However, it is slightly harder for computers to tell the tempo of a song, due to the noisy nature of music and sounds that emphasize the spaces between beats, which are prevalent because they make music interesting.

On a high level, our approach was to detect when the volume of the song increased momentarily, signalling a beat, and then keep track of the number of audio samples between beats in order to calculate the beats per minute.

Block Diagram



The tempo detection portion of the project consists of two main modules: the beat_detector module and the bpm_calculator module. Above is the basic block diagram of how the two modules work and interact.

Beat Detection Algorithm

The beat_detector module uses an algorithm from [1] that uses samples from the time domain, before they are put through the Fourier transform module. The high-level idea is to compare the volume from a short period of time to the volume from a slightly longer period of time to see if it is significantly higher, which would classify the current sample as belonging to a beat.

I calculate the immediate volume by summing the volume of the last 1024 audio samples, downsampled by 4, so that there are 256 samples saved to be summed over. This corresponds to about .016 seconds of audio at a 62.5 kHz sample rate. This value is

compared to the average volume over the last second, which was achieved by saving and summing the last 64 values of the immediate volume, corresponding to 1.05 seconds of audio. These values were approximately suggested by the article from which I took the algorithm. I tested having various amounts of immediate volume samples saved, such as 43, which was the exact number recommended by the website, but found that saving 64 samples gave good results in terms of accurately finding the BPM and was easier to implement since 64 is a power of 2.

Symbolically, this looks like:

$$V_{immediate} = \sum_{i=0}^{255} a[4 * i]$$, where a[i] are the most recent sound samples

$$V_{average} = \frac{1}{64} \sum_{i=0}^{63} e[i]$$, where e[i] are the most recent immediate volumes

To compare the immediate and average volumes, I picked a ratio of 1.375 to determine if a beat is present or not. If $V_{immediate}$

Specifically, this value is 11/8, which is practical because I can compare the immediate volume right shifted by 9, in order to normalize the average volume (divide the sum of the values stored in the average volume BRAM values by 64) and then to multiply by 8 for the denominator of the ratio, to the sum of the values in the average volume BRAM multiplied by 11. Doing these arithmetic gymnastics is useful because only integers are used, and there is no need for a divider module.

The value of 1.375 was close to the value suggested by the article, but was higher than their value because I found that the system was too sensitive to off-beat noises. This specific value was found mostly through testing and checking to see if the output was reasonable and accurate.

BPM Calculator

The output of the beat detection module is wired into the bpm_calculator module, which processes the serial stream of beat/no beat decisions and outputs an integer between 28 and 256 representing the beats per minute corresponding to the time between beats.

The output is calculated by dividing 60 sec/min * 62.5k samples/second = 3,750,000 samples/min by the number of audio samples that pass between beats, which will give a number with the units beats per minute, as intended. I used the LogiCORE IP Divider Generator that comes with Vivado to perform this calculation. The number of samples between beats is counted by saving the value when the beat input is 1 and the number of samples that has passed between beats is greater than 62.5kHz * 60 sec/min / 256bpm = 14,648 samples/beat, which is the number of samples between beats for 256 beats per minute, the maximum that I am checking for. This is because 256 beats per minute is very fast, and almost no song will be faster than that, so waiting until at least this many samples have passed is a way to ignore the off-beats that might trip the beat detector, and account for the fact that the beat signal might be 1 for more than one clock cycle, since beats usually last for more than 0.000016 of a second (1 sample length). From the output of the division, I

took the lowest 8 bits of the quotient, which was sufficient and did not truncate the result, since the lower limit on the number of samples between beats limits the BPM to 256.

The 8-bit output of this module is passed on to the display_info module, which is discussed in greater detail later in the report.

Testing and Results

The simplest beat is that of a metronome, which is a pure straight rhythm and has very short, clear beats. This made it ideal for testing out our system in a controlled environment. At first, I did not get results, because I was using the “eoc” signal from the XADC rather than “done_osample16” as the ready signal for the beat detector and bpm calculator. “eoc” is 1 at the end of every conversion, which happens much quicker than the 62.5kHz audio sampling rate, which is the frequency of “done_osample16”. Therefore, the beat detection actually looked at a much smaller portion of the song for both the immediate and average volume. More importantly, the count of samples per beat was much too high, leading to an extremely low BPM, which registered as zero.

Once I changed to the correct ready signal, I got the correct BPM value consistently for the metronome audio. After that, I switched to the 100 BPM drum beat ([2]) for testing, because it has some off-beat noise from the cymbals and less clean beats, so it is better for adjusting the sensitivity of the system.

When I put the output of the low-pass filter into the beat detector, I found that there was no sensitivity at which the beat detector reliably found the correct beats per minute. Either the system detected a beat almost all of the time, or no beats whatsoever. There was also the issue of which indices of the filtered audio to use, since passing the audio through the LPF increases the bus width by 10 but also attenuated the signal significantly. I tried taking the top 14 bits and some other selections from the middle, but the sensitivity was such a finicky issue that eventually, I decided to use the beat detection system on the unfiltered audio, since it worked better without the filter.

Difficulties

One of the difficulties in extracting the tempo is that many times, there are heavy drum-beats on the off-beats, because it makes songs more interesting to listen to than if there were simply straight beats, which is the term for one drum beat per beat of the measure and sounds boring. However, this varies by genre; for instance, rock and roll tends to have more straight beats, while Latin music such as reggaeton has strong syncopation, or emphasis on off-beats. This is why I decided to test the module on a basic straight rock and roll beat from Youtube.

Future Work

As stated in the article I used, one possible improvement for the beat detection algorithm is to dynamically calculate the cutoff ratio for what is considered a beat based on the variance

in average energy. This would make the algorithm more robust to changes in beat volume between songs, or even within the same song.

In addition, with more time, I would like to incorporate the LPF with the beat detection fully. As such, it is possible to input the filtered data, but the module will not give accurate results.

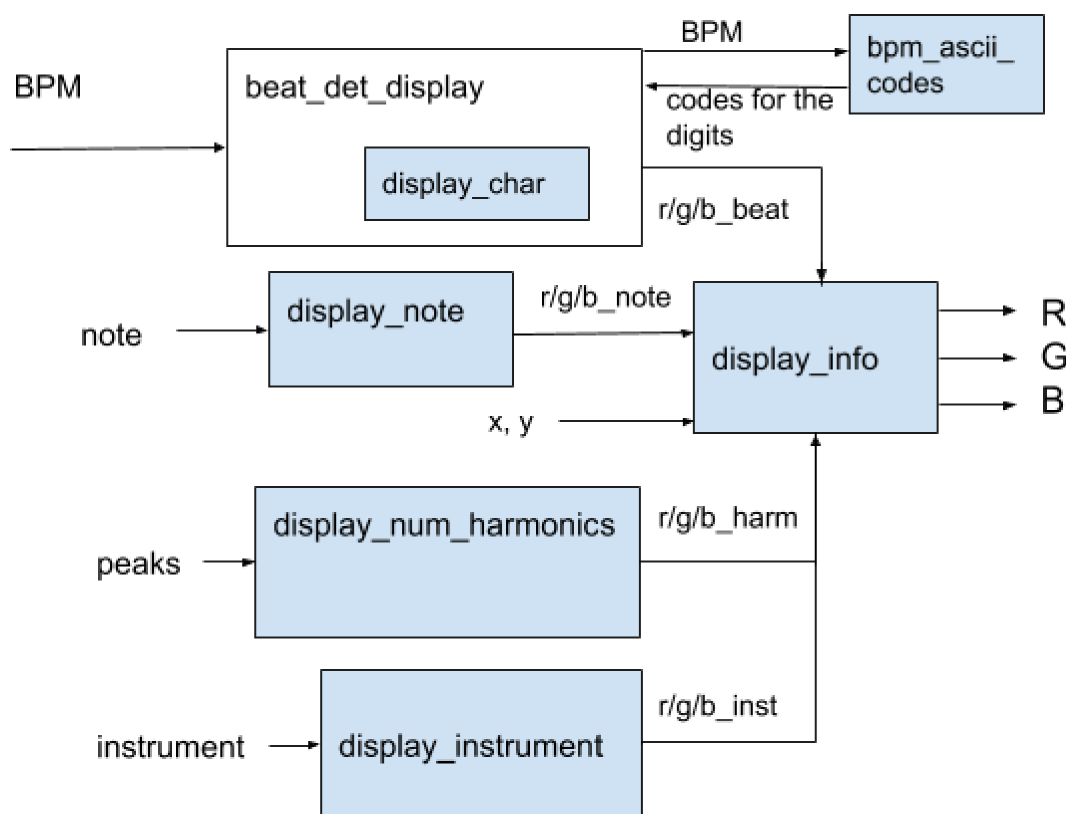
One other possible improvement is to use the data in the frequency domain to calculate this. It can be done by summing up the magnitude of each bin of data, and this can be used to emulate a LPF by only summing up to some frequency. Alternatively, we could have used a comprehensive set of combtooth filters to decide which frequency contributes most to the BPM of the piece, by seeing the response of the signal to a wide range of combtooth filters. This can be done much quicker in the frequency domain, but still requires quite a bit of power and time.

Display

Overview

We displayed a logo, the FFT histogram that was already in the demo module, and the instrument, pitch, tempo, and number of harmonics of the music.

Block Diagram



Above is the basic block diagram of how the display modules work together. display_note, display_num_harmonics, and display_instrument all use display_word and

display_character, but it is only drawn in for beat_det_display, which only uses display_character. All of the higher level displaying modules are synthesized in display_info.

display_char
This module is the basis for displaying text on the screen. It takes in the x and y coordinates of the character and its ASCII code, and all the usual display signals such as hcount, vcount, and the 65 MHz clock. Using the font COE file from Piazza to look up the shape of the letter, it outputs whether or not the current pixel is a part of the character.

display_large_char
Just like display_char, but with a "scale" input that determines by how many factors of 2 to scale.

display_word
Takes the length of the word as a parameter, the usual display signals, the position, the scale of the word to display (it uses display_large_char to display the individual characters), and the ASCII codes of the letters in the word (backwards, due to how it loops through the codes), and outputs whether or not the current pixel is a part of the word. This is done by looping through and generating a display_large_char instance for each character in the word.

bpm_ascii_codes
Outputs the ascii codes of the 3 digits of the BPM, assuming that it is less than 299 (which makes sense, since it is capped at 256). For any value over 299, the most significant digit will remain a 2.

beat_det_display, display_note, display_num_harmonics, display_instrument
These modules use display_character and display_word to display the information the backend has found. The information passed through is in the form of a number that maps to a result. For instance, if the instrument input is a 2, that means woodwinds are being detected.

display_info
This module determines what information is shown and how, by employing all the modules specified above. Since the tempo and instrument detection were most likely not to work at the same time on the same piece of music, we decided to only show one panel of information at a time. That is, by using the 4th from the right switch below the LEDs, the system can either display the tempo or the instrument and pitch information.

display_logo
This module displays the logo we came up with, which is "The Music Box™" in large black letters in a white rectangle with a blue shadow.

Testing and Results
Testing these modules was relatively easy and painless, since I could see the effects of every change I made visually without needing special debugging to expose the outputs of

the modules. There were some initial errors with displaying words where the words were backwards, due to the nature of the loop in display_word. However, overall this part of the project was the least stressful, and very satisfying to see work.

Main Module

Nexys4_fft_demo (plus auxilliary modules util, bram_to_fft, bram_fft, fft_mag)
Authors: Phoebe, Elina and Mitchell Gu

We organised all of our separate modules through this 'main' module, in which we would create instances of the modules we had made and coordinate the data flow between them. The basis of this code and it's associated modules were written for a demo, by Mitchell Gu, and it was into this module that we wrote our own and put it all together to output the information we wanted.

SUMMARY

Through a mixture of our own algorithms, and algorithms found online, we were able to take the FFT module in the demo and extract instrument family, pitch and tempo data, which we then displayed through VGA. As such, we met all of our commitment goals, and some of our main goals. This project has a lot of scope for expansion and we are interested in future improvements such as more detailed instrument detection, chord progressions and, finally, marrying it all together to make a music information retrieval black box, so that a complete song could be passed through our hardware and analysed at once.

We had a lot of fun working on this project and would like to thank Driss, Diana and all the other TAs, and, of course, Gim and Joe, for their expertise and help.

References

[1]: “Beat Detection Algorithms” by Frederic Petin (URL: http://archive.gamedev.net/archive/reference/programming/features/beatdetection/)

[2]: “100 BPM - Simple Straight Beat” by LumBeat (URL: https://www.youtube.com/watch?v=zZbM9n9j3_g)