Voice Controlled Car System

6.111 Final Report
Ekin Karasan & Driss Hafdi
December 11, 2016

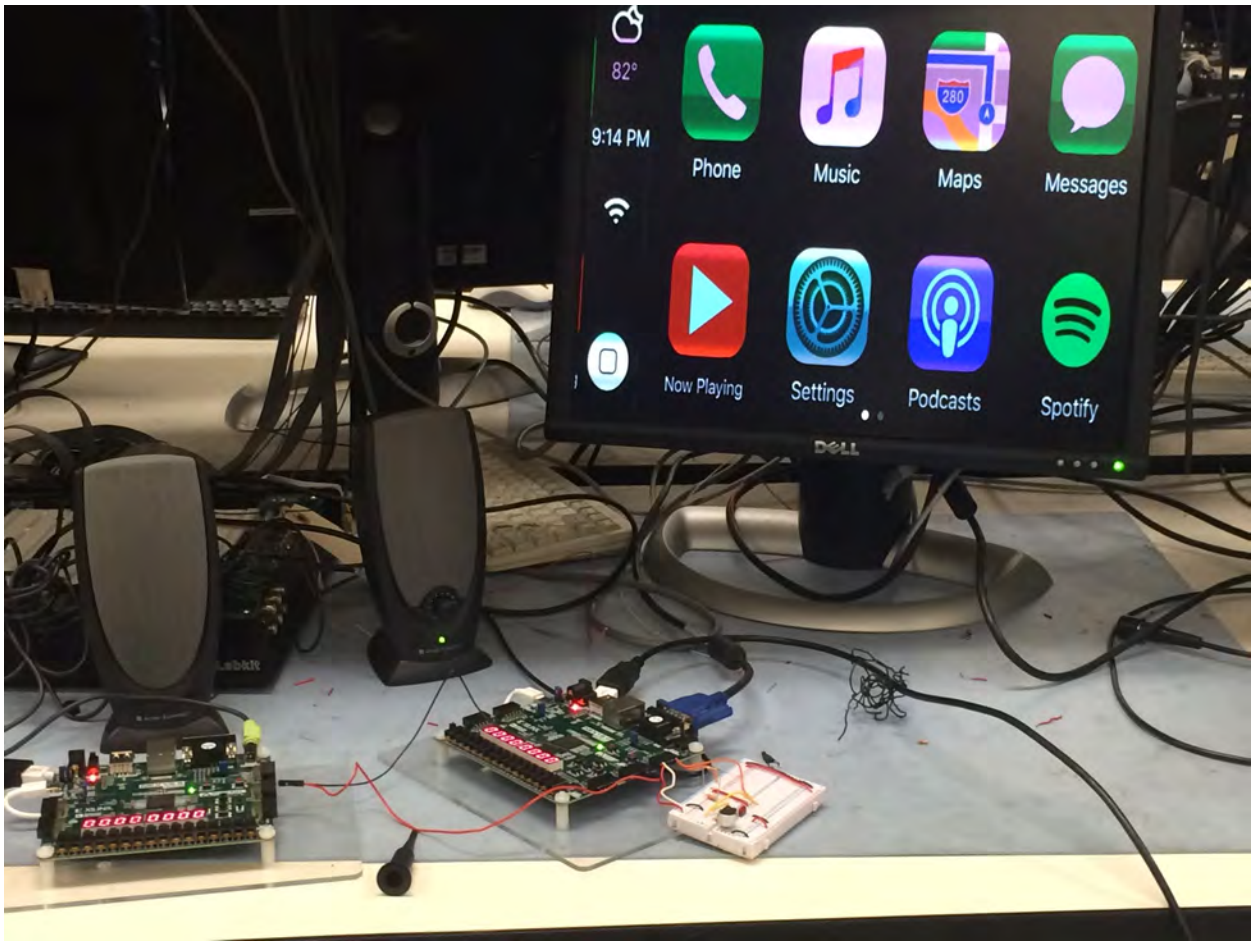


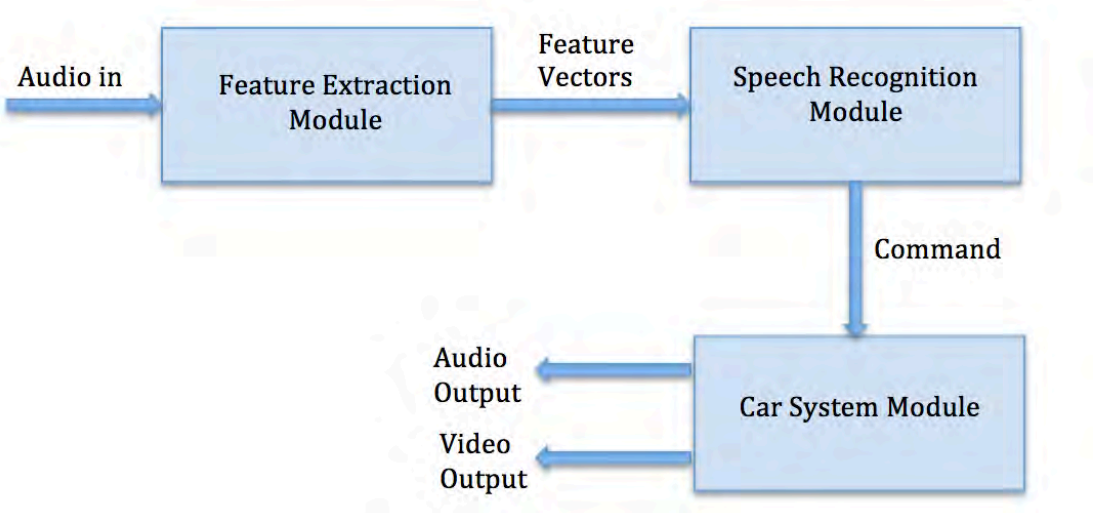
Table of Contents

Overview

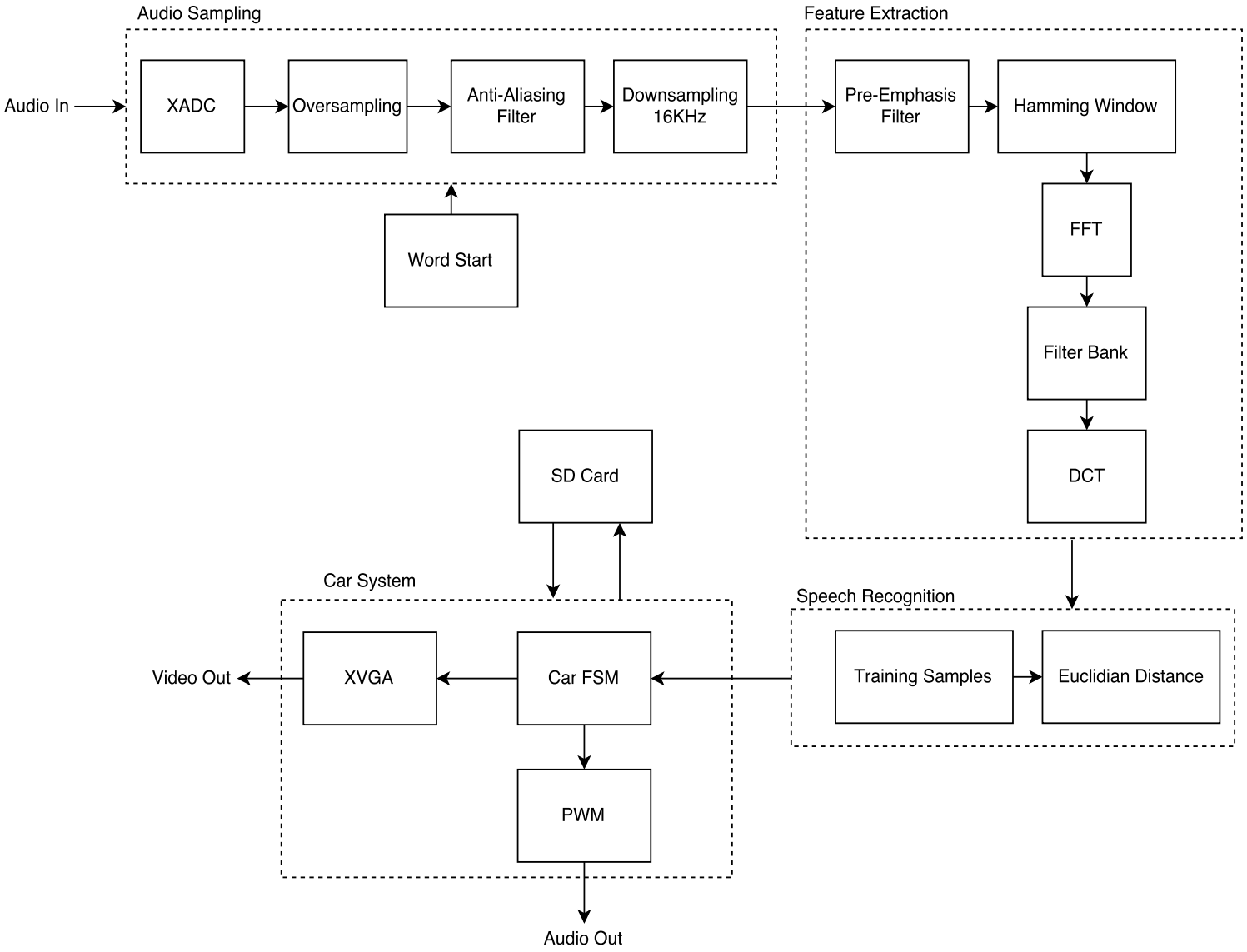
Voice controlled car systems have been very important in providing the ability to drivers to adjust the controls of the car without any distractions. These systems are continuously improving providing drivers more control over the internal car system.

Our main goal in this project was to implement a voice detection algorithm on hardware that would be used to control a virtual car system. The input to the system is through a microphone detecting audio signals. The input audio sample is processed in several steps and a final vector consisting of 1164 values is computed. This vector is called the MFCC feature vector. This feature vector is compared to the pre-computed feature vectors of the training samples and the command with the highest similarity is outputted. The outputted command is used to determine the state of the car system FSM. The car system FSM communicates with an SD card displaying an image for each of the different states.

The overall system consists of three main modules: feature extraction module, speech recognition module and car system module. The brief high-level overview of the flow of information in the system is described in the schematic below:



Block Diagram



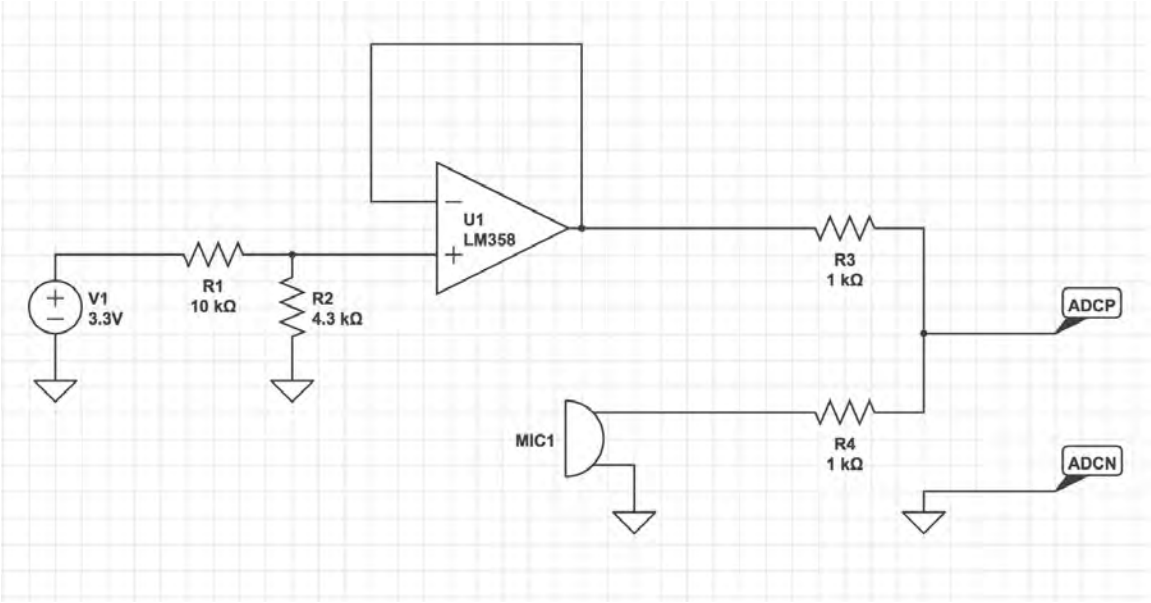
Implementation of Major Modules

Input to the Feature Extraction Module

Microphone Input Circuit and XADC (Driss)

Our audio signal that we obtained from the microphone was in the form of an analog waveform. In order to convert our input analog waveform from the microphone input circuit to a digital input we used the integrated ADC on the Xilinx Artix-7 chip and Vivado's built in XADC IP Core (as recommended to us by Mitchell). The XADC module was clocked with a 104MHz clock obtained from a 100MHz clock using the Clocking Wizard IP Core. This clock allowed us to achieve a sampling rate of 1 MSPs.

The analog input to the XADC module had to be a differential signal ranging form 0V to 1V. In order to achieve this, we used a biasing circuit provided to us by Mitchell, which allowed the signal to be centered at 0.5V. Below is the schematic of our circuit:



The combination of the biasing circuit and the XADC module produces a new output of sample data also asserting an EOC signal. This sampled data is used as input for the rest of the system.

Oversampling the Output of the XADC (Driss)

As stated above the incoming audio signal is sampled at a rate of 1MSPS. However, since we would be considering only a small range of frequencies in the future we would need a higher resolution for each sample in order to obtain better accuracy. We thought that the best way to do this would be to oversample the current signal. Oversampling is the sampling of a signal at a rate that is higher than its Nyquist rate. The extra samples obtained by sampling at a higher rate are averaged, obtaining a higher resolution. For each extra bit of resolution, the signal must be oversampled at a rate of 4 times its Nyquist frequency.

In our system, the signal was oversampled at a rate of 16, obtaining an extra $\log_4 16 = 2$ bits of resolution. The final output of the oversampling module had a resolution of 14 bits at a sampling rate of 62.5KSPs.

Detecting when a Speaker Starts Speaking (Ekin)

Our initial thought on deciding when a speaker started speaking was for them to press one of the buttons and then start speaking. This would provide us with a rough estimate on when a command was spoken and would be enough for us while we computed the feature vectors. However, when we were testing our feature extraction algorithm on MATLAB we realized that one of the main issues was the inconsistency in the time between when the button was pressed and when the command was actually spoken. Therefore, we decided to create a new module following the oversampling module that would detect the exact point when a speaker started speaking.

Each of the sampled values outputted by the oversampling module were centered at a value of 0x2000. Due to this fact, the sampled value itself was not a very good representation of the magnitude of the speech. Since we wanted to measure the magnitude of the speech, the difference of the current sampled value from 0x2000 was calculated and used as the magnitude value for each sample. Two BRAM's with depths of 64 elements were created. Each time a new sample is generated by the oversampling module, its value is added to the first BRAM replacing the last element of this BRAM. The last element of the first BRAM is then added to the second BRAM, replacing the last element of this BRAM. Every time a new sample is added to a BRAM, the current sum of the elements in the BRAM is calculated by adding the newly added sample to the sum and subtracting the removed sample. This way, at each moment in time the two sums of the last 64 magnitudes and the sum of the 64 magnitudes before this are calculated.

For a speaker to be detected, the difference between the two sums of magnitudes must exceed a certain threshold. This means that newly incoming samples have a much higher magnitude compared to previous samples and therefore the speaker must have started talking. This threshold value was determined using a trial and

error process with many different threshold values. Once the difference between the two sums exceeds this threshold value, a signal named word_start is asserted for 1s and deasserted at the end of the 1s interval. This allows for a constant length for each of the detected words. Computations by the following modules only take place during this 1s window.

Down sampling and Anti-Aliasing filter (Driss)

Since the highest frequency component present in speech stands at 8KHz, we decided to down sample our signal to 16KHz. However, before being able to do that, we needed to apply an anti-aliasing filter to our signal that would allow us to only keep the frequency components below 8KHz.
Therefore, we designed a 31 tap low pass FIR filter using MATLAB. The coefficients obtained were scaled to 12 bit signed integers and written to a coe file. They were then stored in a ROM and used in the filter module.
The input to the antialiasing filter is the oversampled audio signal. Once filtered, it is fed into the down sampling module and sampled at a rate of 16KSPs.

The Feature Extraction (MFCC) Module

Pre-Emphasis Filter (Driss)

Before the calculation of the MFCC coefficients a pre-emphasis filter is required in order to emphasize the higher frequencies, which are suppressed during the sound production mechanism of humans, relative to lower frequencies. The output of he pre-emphasis filter is calculated by the following equations for each of the samples in the time domain:

$$y[n] = x[n] - 0.96875 * x[n-1]$$

Usually a reasonable value to multiply the previous sample with ranges from 0.9 to 1.0. Since a division must be done with bitshifting and multiplication in Verilog, a value that can be easily obtained by these operations was chosen (0.96875). The output of the pre-emphasis filter is stored in a circular buffer BRAM. The values of samples in the BRAM correspond to the last 32ms of audio.

Application of a Hamming Window (Driss)

In the MFCC algorithm, the FFT is applied to a series of overlapping frames. The 32ms length of audio in the buffer corresponds to each of the frames. Since we compute the FFT on windows of data, we need to reduce discontinuities at the boundaries of each one of our frames. Therefore, we apply a Hamming window to each one of our frames as they're sent to the FFT module. This way, we minimize the effects of spectral leakage.
We used MATLAB to obtain 512 Hamming window coefficients and scaled them to 16 bit unsigned integers. We then generated a coe file containing the coefficients and stored them in a ROM. Each audio sample would then be multiplied by its corresponding Hamming window coefficient and sent to the FFT module.

The Fast Fourier Transform (Driss)

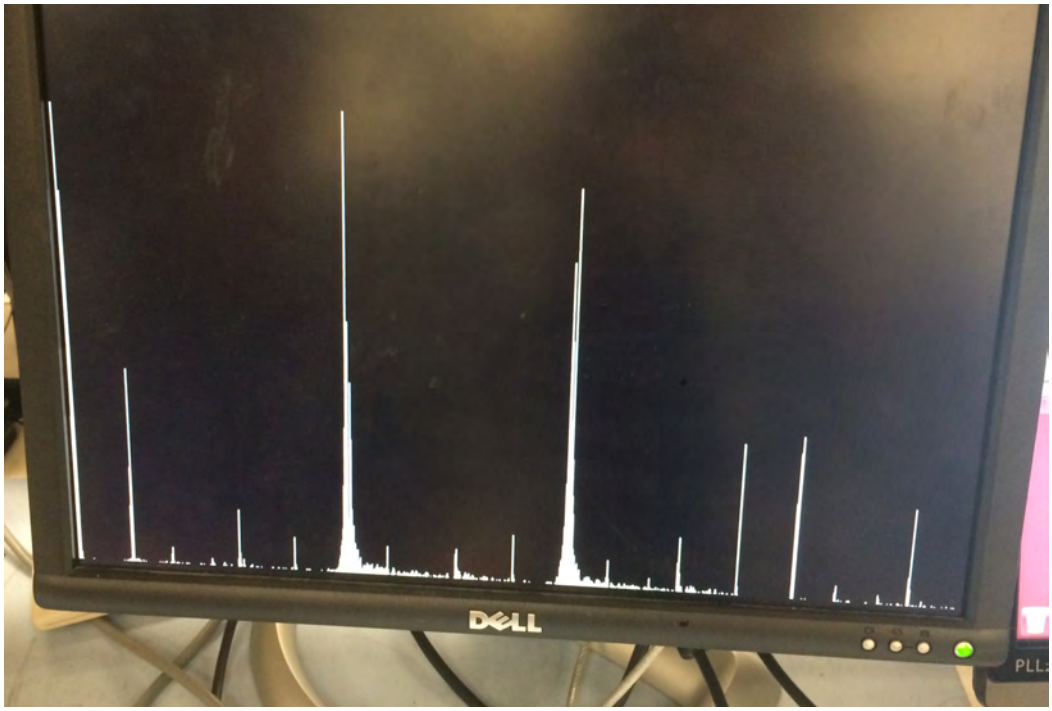
The first step in the application of the Fast Fourier transform is the streaming of the data into the FFT module. The FFT module uses a handshake signaling process. The index of the beginning of the current frame in the BRAM is kept in a variable and incremented every time a new sample is added. The FFT is computed with the samples in the BRAM every 10ms. Every 10ms the variable t_ready is asserted and the FFT module asserts t_valid once it is ready to receive new samples. This completes the handshaking process and the samples in the BRAM are streamed into the FFT starting from the head address. The output of the FFT module is a 512-point FFT with a 16 bit unsigned number for each value.

The FFT core outputs both the real and the imaginary parts of the FFT. However, since the following MFCC computations need the square of the magnitude of the FFT

value, this value is calculated. This is done by squaring the real component and the imaginary component of the FFT and adding them together.

The frequencies are not outputted in any particular order by the FFT core. Therefore the FFT module outputs the current frequency index in a bus named t_user. This is very important in allowing the output of the FFT to be stored in the correct order. One other important feature is that when the last output of the FFT is missing, a signal named last_missing is asserted. This means that the last value of the FFT is being outputted and the values that are outputted afterwards that do not belong to the FFT can be discarded.

For each command, the FFT is computed on 32ms windows every 10ms. This results in 97 FFT computations for a 1 second command. Each FFT computation outputs 512 values since it is a 512-point FFT. However, we only need to store the first 257 values since the rest of the values are basically a mirror image of the values indexed from 1 to 256. Therefore for each command the FFT outputs a total of $97 * 257 = 24929$ values. We decided that in order to make debugging of the FFT easier and reduce problems with timing it would be best to wait until all 24929 values have been outputted by the FFT before moving on to the next module. Once the FFT module has produced all of these values it asserts a done signal, which signals the next module to begin its computations. All of these values are stored in a single BRAM. The first 257 values in the BRAM correspond to the FFT of the first frame, the second 257 to the FFT of the second frame and similarly for the rest of the frames. A histogram of four consecutive FFT frames are shown below:



Application of the Mel-Frequency Filterbank (Ekin)

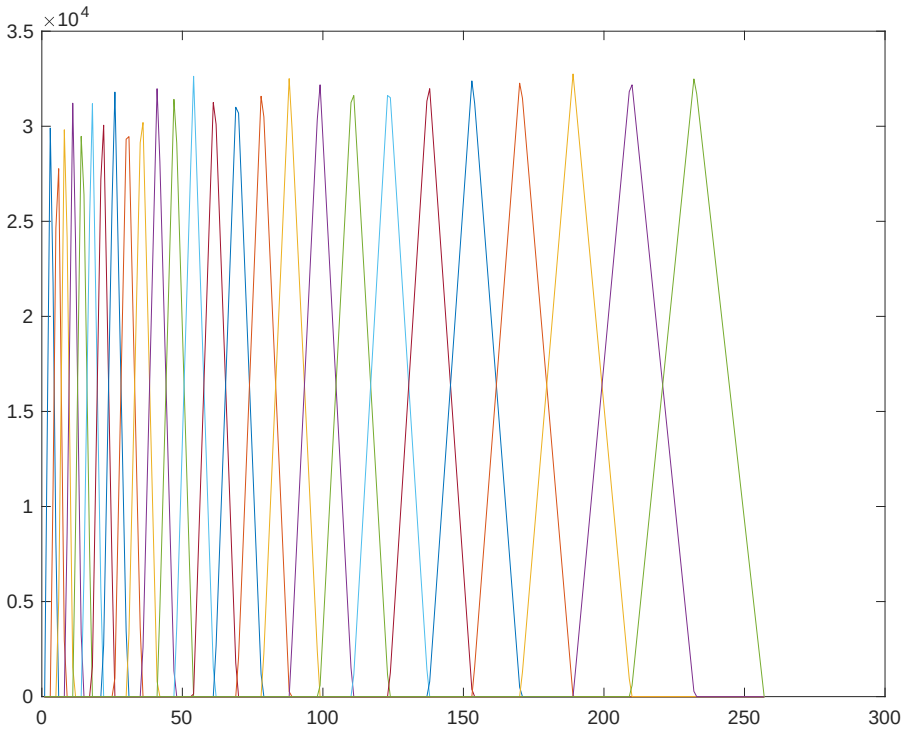
The next step of the MFCC computation is the application of the triangular filterbank to the FFT. One important feature that is important in understanding how the filterbank works is the mel-scale conversion. Generally, the actual measured frequency of a pitch is different than the one perceived by humans. Humans are better at distinguishing between changes in lower frequencies compared to higher frequencies. The mel-scale is a method to convert the actual measured frequency to the one perceived by humans. Here is the equation for this conversion:

$$M(f) = 1125 * \ln\left(1 + \frac{f}{700}\right)$$

The computation of the triangular filterbank involves the conversion of the highest and lowest frequencies in the signal to the mel-scale. Depending on the number of triangular filters a number of equally spaced values are chosen between these two points. These equally spaced values are converted back to actual frequency by using the inverse of the equation stated above. Since we are taking the inverse logarithm of values in this process, the equally spaced values in the mel-scale become closer together in lower frequencies and farther apart in higher frequencies. For each set of three consecutive values a triangular filter is created that starts form the first point, reaches a maximum at the second point and goes back to zero in the third point creating a triangle.

Once these triangular filters are created, the energies after each filter is applied to the FFT of the signal are calculated. The energies are computed by multiplying the filters with the FFT of the signal and adding up the coefficients. Therefore for each FFT frame we get one value for the application of each filterbank. After some research we decided that 26 was a reasonable value for the number of triangular filters.

The computation of the triangular filters involved many complicated mathematical operations such as taking logarithms. Therefore, we decided that it would be best to perform the computation of the triangular filters on MATLAB using the available MFCC function. In the MATLAB MFCC function the coefficients of the triangular filters were in the range from 0 to 1 and contained many decimal numbers. These values had to be converted to 16 bit integers by scaling them by 2^{15} and rounding them to the closest integer. Here is a plot of all 26 triangular filters produced after this process:



As seen above, each of the filters contain 257 coefficients, with most coefficients having a value of 0. The coefficients for each of the triangular filters are converted into a .coe file that can be used in Vivado in order to store these coefficients in a BRAM. Once the set of 24929 values have been outputted by the FFT module, each of the filters are applied to the FFT computations for each of the frames. For each frame a total of 26 32-bit unsigned filterbank energies are calculated. The 26 filterbank energies for each frame are stored once again in a new BRAM. This BRAM contains a total of $26 * 97 = 2522$ values. Similar to the FFT module, the computations in the next module do not begin before all 2522 values have been calculated.

The Decision to Not Use a Logarithm (Ekin)

The next step in the MFCC process is to take the natural logarithm of each of the values calculated in the previous module. In our initial design plan, we had decided to use IP core Floating-Point Operator offered by Xilinx in order to calculate the logarithm. This module required a 64-bit floating number. Conveniently, the same IP core contained another function that could convert a 32-bit integer to a 64 bit floating number. However, the conversion to a floating number would result in a maximum of 11 bits for the exponent with the rest of the bits as fraction. This meant that we would be losing 32-11=21 bits of accuracy when we did this conversion. Since taking the logarithm once again reduces the magnitude of the value we would generally be ending up with a single hex digit in the end of these computations.

Therefore most of the output values of the logarithm were very similar and we were losing a lot of accuracy.

We decided to test the MFCC function on MATLAB without incorporating the logarithm. Surprisingly, we realized that the accuracy in detecting commands remained very similar. Hence, we decided that it would be best if we removed this module from the MFCC computation.

The Discrete Cosine Transform Module (Ekin)

A discrete cosine transform is similar to a discrete fourier transform except the signal is represented only as a sum of cosine oscillations. This allows the small high frequency components of the signal to be discarded. Since this transform also involves complex mathematical operations such as the calculation of cosines we decided to compute the DCT coefficients on MATLAB and then perform the application of the DCT on the current signal on Verilog. Similar to the filterbank coefficient calculation, we scaled all of the coefficients to 16-bit signed integers.

The application of the DCT was very simple as it involved a multiplication of the corresponding DCT coefficient with each of the output values of the filterbank module. This resulted in a 32-bit signed integer for each coefficient with 26 values for each of the 97 frames. However, not all 26 values computed for each of the frames were part of the final cepstral coefficients of the MFCC. Only the coefficients from 1-13 represented the Mel Frequency Ceptral Coefficients and therefore were the only ones stored while the other coefficients were discarded. In summary, after the application of the DCT we obtained $12 * 97 = 1164$ 32-bit signed Mel Frequency Ceptral Coefficients, which were stored in a new BRAM.

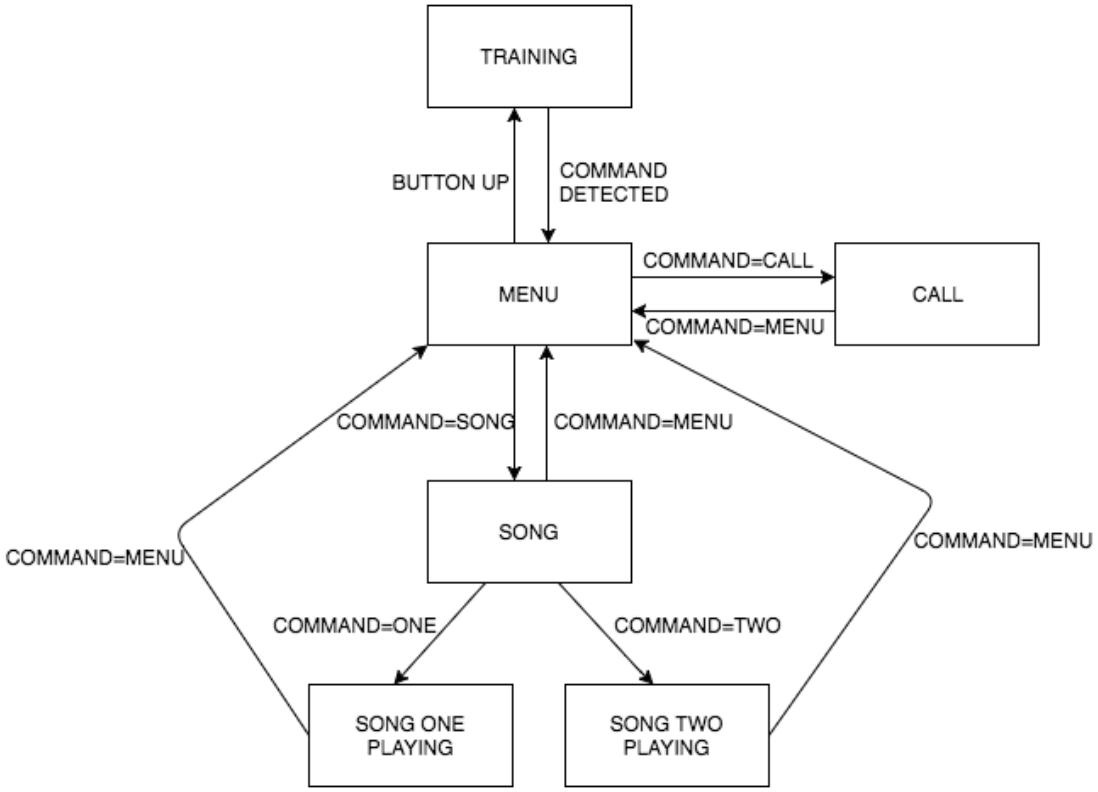
The Speech Recognition Module (Ekin)

The main purpose of the speech recognition module is to compare the Mel Frequency Ceptral Coefficients of the current command with that of the training samples, each corresponding to one command, and output the most similar command. The coefficients for each of the training commands are stored in five separate BRAM's. When a new command is streamed through the feature extraction module and done is executed by the DCT module the speech recognition module begins its comparison. The comparison takes place using a basic Euclidian distance computation.

At this point, the coefficients of the current command and that of the training samples are all stored in separate BRAM's. The speech recognition module reads from all of the BRAM's simultaneously and takes the square of the difference between each of the values for the current command and the training samples. The total sum is accumulated as the difference between each new coefficient is computed. When the module has reached the end of the coefficients, it compares the accumulated sums for each of the training samples and outputs the command with the lowest value.

The Main Car System (Ekin)

Car System FSM Diagram



The state transitions of the main car system FSM is displayed above. There are a total of six states and five distinct commands. A different VGA display is associated with each of the separate states. The two states “Song One Playing” and “Song Two Playing” have an audio output along with the VGA display. The details of the VGA display and audio output will be explained in detail in the next section.

The Training State

Initially our plan was to use training coefficients that were calculated in MATLAB and then stored in BRAM’s. These coefficients would always remain constant. However, we realized that the FFT used by MATLAB might not be the same as the one performed on Vivado. We also realized that this would not allow for many people to use the system, as the training samples would only be for one person. Therefore, we decided that the best approach would be to give the users the opportunity to set the training samples by repeating the commands once every time they started using the system.

We created a separate Training State in the FSM that is accessible from the Menu state if the up button is pressed. The switches (SW[6:2]) determine the command that is being trained and the word that should be said is displayed on the screen. The system does not leave training mode unless a command is detected. If the system is in training mode, the output of the feature extraction module (the mel-frequency ceptral coefficients) is not used as input for the speech recognition module. Instead it is stored into the corresponding training sample BRAM depending on which of the commands are being trained.

Once all of the commands have been trained, the user can start using the car system. This method to create training samples also allows for the user to update a training sample if it is not working properly.

The SD Card Interface (Driss)

Graphics

Once we decided on the different images we wanted to display, we used GIMP, an image manipulation program, to modify the aspect ratio of the pictures to be 1024x768, which is required by our XVGA module. Additionally, since the Nexys4's VGA port only accepts 12-bit color depth data, we had to reduce the color depth of the images to contain 4-bit per color. Lastly, the pictures were converted to the Bitmap format.
Then, we used MATLAB to extract the color tables of each image, which we then wrote to coe files and stored in different ROMs. Then, we extracted the value of each pixel in the image and stored the 4-bit unsigned integer pixel value in the SD card. This was done using a HexEditor and recording the start address at which the image is stored. Those values represent an index in the color table. This process was repeated for each image that we wanted to display.
We wrote a module that is able to read the indices from an SD card. Whenever the state of the Car FSM changes, we generate a pulse that triggers a read from the SD. The image which we want to display is fed into the module, which selects the appropriate address to start reading from. Then, we store the image data into a 4-bit wide 1024x768 deep BRAM. The said BRAM is being written to with a 12.5MHz clock, as it is the speed at which the SD SPI module operates at.
However, the XVGA module continuously reads from the BRAM at a speed of 65MHz. The indices read back from the BRAM are used to select the value of the Red, Blue and Green component per pixel. The color tables are selected through a mux depending on the image to be displayed.
An issue we encountered with this module is the speed at which the images transition. The issue lies in the speed at which the image is loaded from the SD card. As the XVGA module continuously displays the content of the BRAM, the transition between images is observable, as the BRAM is slowly getting filled with the data from the new image.
Since the BRAM usage of the FPGA was reaching saturation, we would not have been able to solve this issue by creating a second 1024x768 BRAM and only alternate between the two once it is filled completely. Hence, the only possible solutions would have involved using the on board RAM. However, due to time constraints, we were not able to solve this issue.

Audio

Once we selected the two audio files to be played, we had to modify the data such that it can be stored in the SD card. We started by reading the mp3 file using MATLAB. The audio data obtained was in the range from -1 to 1, and had to be rescaled to an 8-bit unsigned integer. Once this is done, we store the file into the SD card using a HexEditor and record the start address of the audio file.

Then, we wrote a module that communicates with the SD card and starts by reading the first two blocks containing audio data. Those 1024 bytes worth of data were stored into two separate 8-bit wide 512 deep BRAM. Once both are full, we stop reading from the SD card and start reading out the data from one of the BRAMs at a rate of 44.1KHz. The data is output to the PWM module.
Then, whenever we're done playing out samples from one of the BRAMs, we start outputting samples from the other one and execute a block read in the SD card. The data from the card is then stored in the BRAM that we just finished reading from. The rate at which a full SD card block is read is much higher than that of playing out all samples from a BRAM. Therefore, we just keep executing block reads as we finish reading from a BRAM to fill it up with new audio data.
However, when we tried to implement the design, we realized that adding the audio module would cause the speech recognition module to stop working due to timing issues. Therefore, we decided to implement the audio module on another FPGA. The Car FSM outputs two control lines, song_playing, which tells the other FPGA that a song needs to be played, and song. If song = 0, then we play the first song, else, we would play the second one.

Analysis of the System (Ekin)

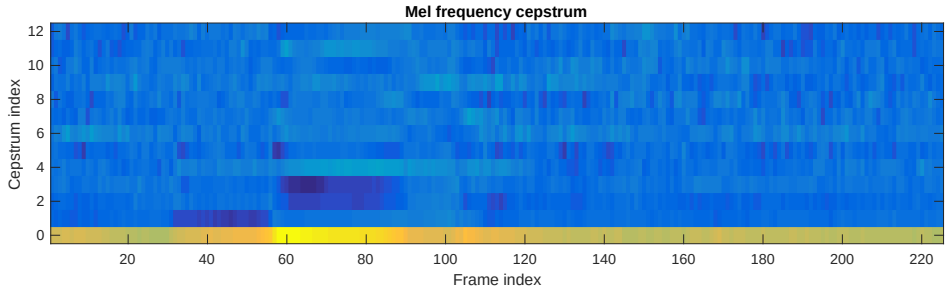
Testing for the Accuracy of the System

We tested the accuracy of the system firstly by using our own training samples and then also using each other's training samples. Each of the commands was repeated 20 times for each of the 4 cases. Here are the results summarized in a table displaying the percentage of accuracy:

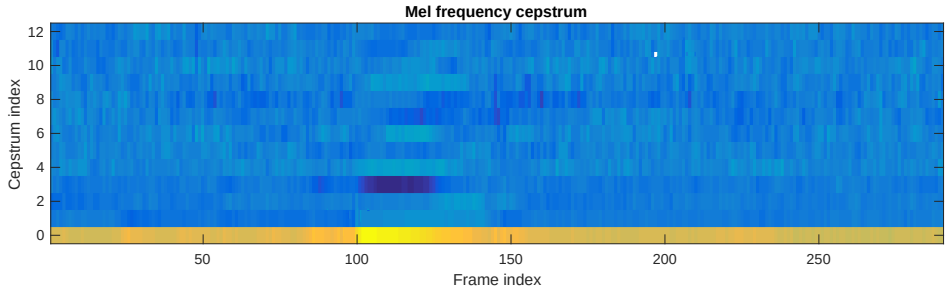
	Menu(%)	Song(%)	One(%)	Two(%)	Call(%)
Driss with Driss' Training Sample	95	85	85	85	100
Driss with Ekin's Training Sample	0	0	95	0	95
Ekin with Ekin's Training Sample	90	100	100	65	100
Ekin with Driss' Training Sample	25	15	50	85	70

As seen from the results above, we have a fairly good accuracy in recognizing commands when a person's own training samples are used. However, when another person's training samples are used, the accuracy is much lower and can be as low as 0%. This shows that our system does not work for multiple speakers with the same set of training commands. This is mainly because we have used a simple speech recognition module instead of a more resilient module such as dynamic time warp or a Gaussian probabilistic model that would work with different speakers.

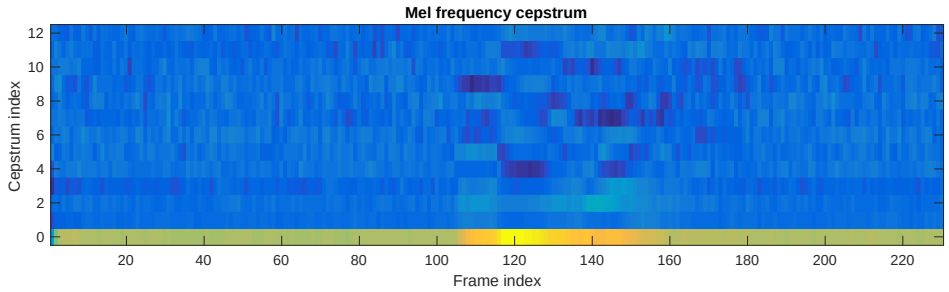
We tested to see how similar the MFCC plots for different speakers were for the MATLAB computations. The plots show that the MFCC calculations for different speakers are very different. Here are the MFCC plots for two of the commands spoken by the two of us:



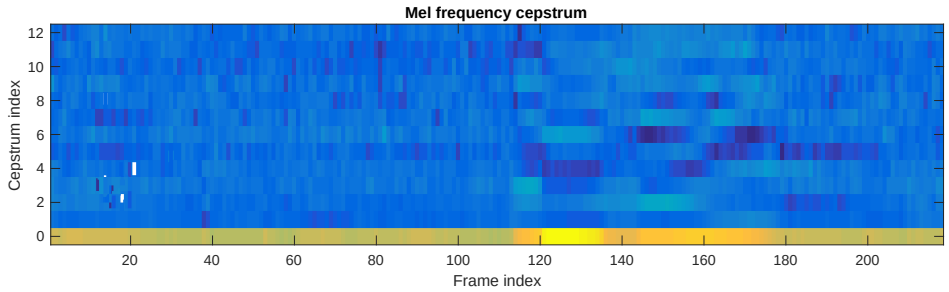
Mel frequency ceptrum of "Song" by Ekin



Mel frequency ceptrum of "Song" by Driss



Mel frequency ceptrum of "Menu" by Driss



Mel frequency ceptrum of "Menu" by Ekin

Since the above plots for the same command are very different than each other it is not surprising that our system does not work for multiple speakers either.

Lessons Learned and Advice for Future Projects

As we were implementing this system, we came across many different problems. One of the main issues was that we were not able to use floating point numbers. Since the MFCC process contained many complex mathematical operations such as cosines and logarithms, not being able to use floating point numbers really reduced our accuracy. In most cases, we needed to use bitshifting operations and only were able to keep a portion of our values. This problem also caused us to not be able to use the logarithm operation while we were implementing the MFCC.

One other important issue we had was in the implementation of the FFT. Although the FFT operation is a one line command in MATLAB, we realized that it was much harder to implement on the FPGA. More specifically, we had a lot of problems with timing and how to write the values in the correct order of frequency into the BRAM. In this process we extensively used the Vivado internal logic analyzer. The logic analyzer was an extremely useful resource in order to look at the values of signals at several points in time and figure out the errors in or system.

We also had timing issues throughout the project that were not apparent until we decided to add audio to the project. When we tried to read the audio from the SD card, we realized that our voice recognition system did not work properly anymore. The system was not able to detect certain commands. Therefore, we decided to use an extra FPGA in order to solve this issue. This FPGA would read audio from the SD card and communicate with the main FPGA which would signal it when to start playing audio. Although we could not figure out the reasons behind our timing error, we did not encounter any other problems in our system.

One of the main lessons we learned after doing this project was that it was better to focus on a smaller aspect of the project and put a lot of effort into that portion than to work on too many aspects of the project. In our case, we concentrated in computing the MFCC of the audio signals and did a good job in doing this. Therefore, we were able to get away with using a very simple Euclidian distance algorithm to detect a command, and not a more robust probabilistic model. One other very important lesson we learned was the importance of the logic analyzer. We were able to debug much easier and in a more efficient way thanks to the logic analyzer.

Overall, we enjoyed our experience with this project a lot and improved many of our skills. Our main goal starting this project was to implement a voice recognition system on hardware that is able to detect at least 5 commands and we were successful in doing that.

Acknowledgements

Micheal Price for taking the time to meet with us in the beginning of our project and advising us throughout the project.

Gim Hom for his guidance on deciding on a project and his help all throughout the project.

Mitchell Gu for all of his help and support throughout the project.

References

Lutter, M. (2014). Mel-Frequency Cepstral Coefficients. Retrieved December 16, 2016, from http://recognize-speech.com/feature-extraction/mfcc

Michael Price PhD Thesis - Energy-scalable Speech Recognition Circuits(2016)

Vivado IP Catalog