Voice-controlled Video Console

Ben Horkley and Jonathan Surick

Abstract

We created a voice-controlled camera system, composed of a video camera and three microphones, that responds to a user's voice. While we were unsuccessful in our goal of a system which responded to different words as distinct voice commands, we were able to create a system that could either automatically track a speaker by measuring differential volume levels between microphones on either side of the camera, or could respond to manual commands through the lab kit hardware to either move the camera or apply color filters to displayed video.

Contents

Overview

Our project was a voice-controlled camera system, which automatically tracked a speaker standing in front of it, and which could respond to user input by adjusting both the camera position and various video filters being applied to the on-screen display.

The camera assembly contained a video camera with a composite video connection to the lab kit, as well as two microphone assemblies, one on either side of the camera. The entire apparatus was mounted on a servo, which was then plugged into a lab power supply for ± 5V and controlled via digital signals from the lab kit. The signal levels from the side microphones were used to determine whether the camera was aimed at the speaker, and control logic adjusted the servo position to keep the camera centered on the sound source. Our original design also included a voice control system to give commands to the camera processing and tracking modules, discussed below; unfortunately, we were not able to finish this module completely and add it to the rest of the system.

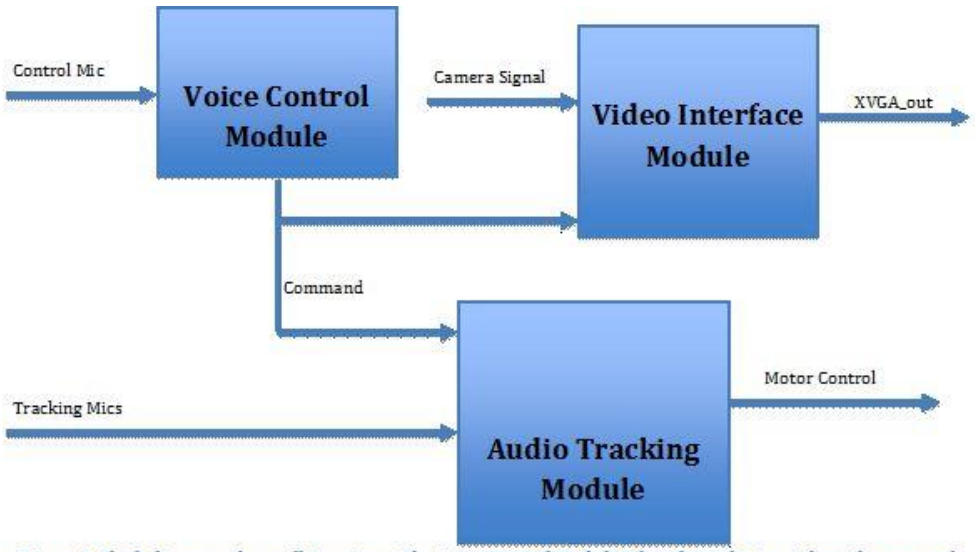

Figure 1: Block diagram of overall structure. The Voice Control Module takes the audio input from the user and determines which of the trained commands has been said. The Audio Tracking Module controls the motion of the camera to either point towards an audio source or to follow commands. The Video Interface Module takes the input from the video camera and applies filters before displaying the output on the screen.

Audio Tracking Module (Jonathan)

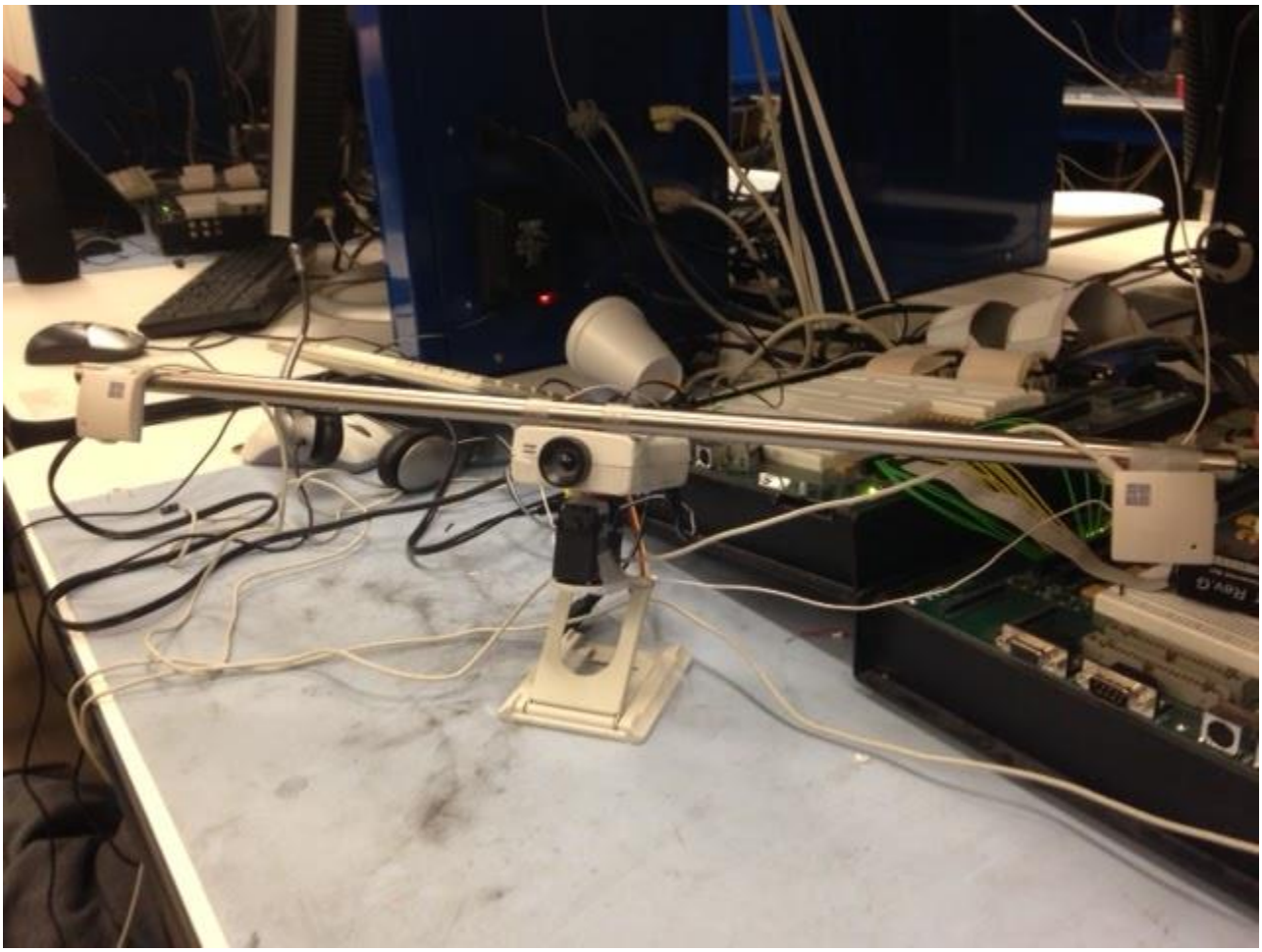

Figure 2: This is the Camera setup used for the audio tracking. The NTSC camera is taped on top of a HS-311 servo motor which is then taped on top of the camera's stand to raise the camera to a better viewing angle. On top of the camera a small steel tube is centered with two different microphones. The microphones send their data to the FPGA boards which allow the motor to turn towards the source of sound.

The audio tracking module used a setup of camera placed on top of a motor with two microphones on either side in order to have the camera track the motion of someone who is speaking. By turning towards where sound is louder, the camera will center on the person speaking since that is when both sources will be at equal amplitude. The final camera set up can be seen in figure 2 above. The farther apart the microphones the better the centering can work. Everything was attached to the structure with scotch tape in order to make things easy to change while still being sturdy.

The block diagram of the final set up of the audio tracking is shown in figure 3 below. Before being processed by any of my modules the audio from each of the mics is processed by the AC97 codec provided for Lab 5. The audio tracking hardware consists of 4 modules, listed in the order of completion: Servo Interface, Override Control Block, Audio Contorl Block, Amplitude2. In addition another two modules were written that took the place of the AC97 codec and Amplitude before the design had to be changed to accommodate issues with those components.

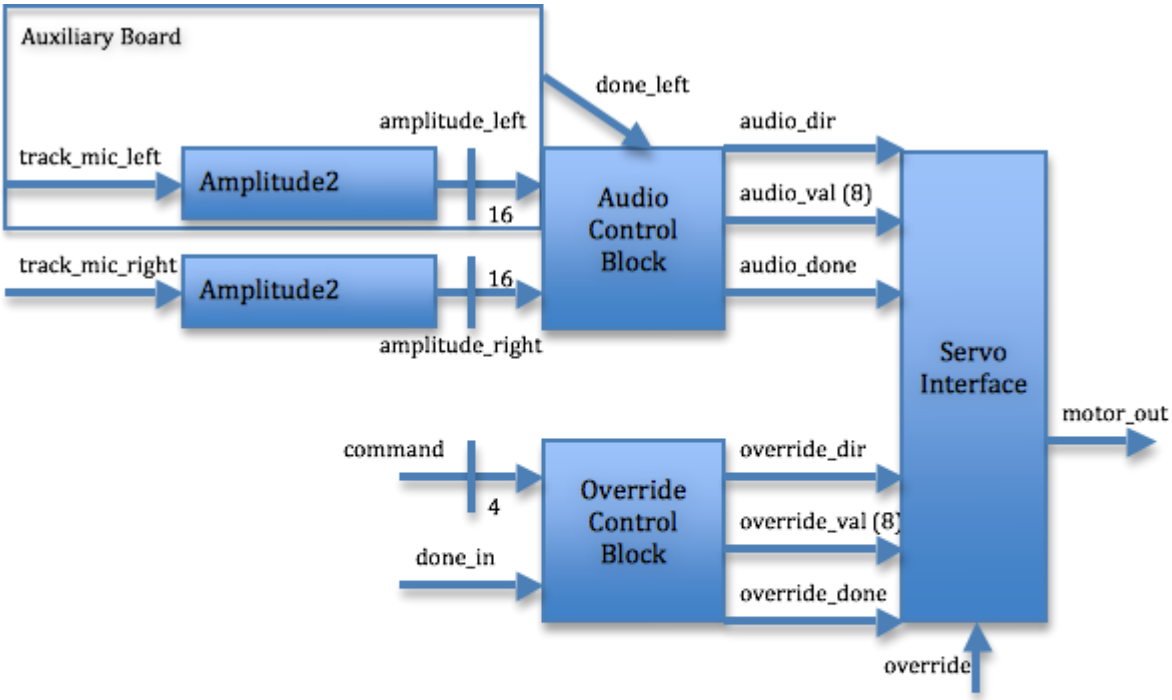

Figure 3:Block Diagram of Audio Tracking Module- This module takes the audio from the two microphones and after getting the data from the AC97 codec, integrates them to determine the overall amplitude in from each mic. The amplitude from the left mic then has to be sent from the auxiliary board to the main board where the audio control block then takes this data and the right mics data and sends signals to the servo to turn the camera towards the louder mic to center the audio source in the camera. The override block takes the voice command generated by the Voice Control Block and passes the instructions on to the Servo Interface. The Interface takes care of turning the signals into PWM so the servo will turn. It also chooses between the override and audio signals based on whether the override signal is on.

Servo Interface

In order to turn the camera to face in different directions we were provided with a Hitec HS-311 servo motor to which the camera was mounted. The servo had a power which was attached to 5 volts, a ground attached to the ground of the labkit, and a signal line upon which a Pulse Width Modulated (PWM) signal was used to control the position of the motor. The specifications required that the signal have a period of 20ms and that the signal should start with a pulse in the range of 1-2ms which determines the position of the motor. The midpoint of the motor was defined at exactly 1.5 ms. Larger pulse sizes meant turning the motor to the right, while smaller pulse sizes meant turning the motor to the left.

I implemented the creation of the pulse signal using a counter and two threshold values.The first threshold was pulse_count, which was the number of clock cycles that the pulse would be at a 1. This value therefore determined what the position of the motor would be. The other threshold was the constant MS20COUNT, which was set the the precise number of clock cycles required for the period of the entire signal to be 20ms. when the count went above this value is restarted the count, starting the next period of the signal and changing the value of pulse_count to its new value.

Three other parameters were also found by testing in order to make the control of the servo motor a lot easier. The first two values, MAXPULSE and MINPULSE, are the values of the largest and the smallest pulse the motor can take before it starts making noises and being unhappy. If something tries to set the pulse width smaller than MINPULSE or larger than MAXPULSE, the pulse width was set to those values to prevent breaking the motor. The third additional parameter is CENTER which is the number of clock cycles required to make a pulse width of 1.5 ms. This therefore gives an easy value to which to center the motor, and in fact on every reset the motor resets to this CENTER position.

In order to choose the value of pulse_count which determines the new position of the motor, the module is given 3 inputs from each of the two control modules. The dir flag (audio_dir and override_dir) determines which direction the motor should turn in by being 1 when the motor should turn to the right and 0 when the motor should turn to the left. The value (audio_val and override_val) determined the magnitude of the turn from the motor's current position and would add or subtract that value depending on the dir flag. To account for the differences between the two value inputs, the value from the override input is scaled up by 16 allowing them to have similar effects. The last signal is the done signal (audio_done and override_done) which tells the module when the other two values are ready for taking. In order to select between the two sets of signals, an override flag is also input from a switch, which when 1 would take the override input data and when 0 would take the audio input data.

Override Control Block

The override control block is a very simple block that translates commands given by the Voice Control Module into direction and value signals to be sent to the servo interface. This was implemented as a case statement that takes the command signals used for controlling the camera and outputs their proper values, while ignoring any commands that would be used for either the video interface or not used command signals. The encoding for these commands is discussed in the Integration section below. The commands were only processed by the module once the module received a done_in signal for the command signal by being stored in a command_register. Once a command was implemented the command_register was cleared to prevent a command from continuously running. There were four commands that were implemented: LEFT, RIGHT, GO and STOP.

The LEFT and RIGHT commands worked by sending out set values for the 3 signals, the value of the dir flag was chosen depending on whether the signal was to step to the left or to the right while the magnitude of the turn was a set parameter of STEPSIZE which determined how large of a step to take per clock cycle. The done signal was set to 1. This sent a signal for one clock cycle to the motor and had it take a single step in the desired direction.

In order to have continuous motion with the GO and STOP commands, a simple two state, state machine was implemented. Whenever a GO command is received, the module sets a persist bit to 1 as well as setting the value to a parameter GOSIZE which allows a different speed between continuous motion and taking steps. Then as long as persist is true, done will remain 1, thereby continuously stepping the motor GOSIZE in the given direction until the STOP command is given at which point persist is set to 0 and done will be set to 0 on all clock cycles where no command is given.

Go does not choose which direction to turn the motor, that is chosen by whichever one of LEFT or RIGHT has been said before the GO command was received. Therefore if after the GO command is received, a LEFT or RIGHT command is received, the direction in which the motor is panning can be changed to the given direction. However, this leads to a small problem, once the new command is said, the value being sent is changed to STEPSIZE instead of GOSIZE. This was not noticed in testing because STEPSIZE and GOSIZE were almost exactly the same size and therefore the change in speed was not noticeable. Though this is easy to change by having the value set to GOSIZE as long as persist is true.

One additional change I would make to this module in the future would be that I would abstract away the override commands from the main code by having them as parameters. Since there a very few commands that is not really an issue at this moment, but if there were more commands that were more complicated, it would be a lot better to have the 4 bit command values just stored in parameters than could be changed and then implementing the case statement blind to those values themselves.

Audio Control Block

The audio control block was designed to take the inputs of the amplitude of two different mics and use them to send a signal to the servo interface telling it which direction and how much to turn the camera so that the signal is balance between the two microphones. In order to deal with difference between microphones two parameters are declared to make changes between different microphone setups. The first is THRESHOLD, which determines how large of a difference between the two levels of the microphone the incoming signal has to be before the camera will start turning. The second one is STEPUNIT which determines how large of a value should be sent to the servo_interface given a specific difference between the two input amplitudes.

The implementation of the Block was really simple in that it compared the right and left amplitudes and if the difference was larger than THRESHOLD it would send out the direction towards the louder mic with a value of:

(largerAmp - smallerAmp - THRESHOLD)*(STEPUNIT/32)

This allowed the motor to turn towards the source of louder noise, but not turn so fast as to create its own source of additional noise.

Audio Input and Amplitude

Our design requires 2 mics to be attached to the camera assembly for audio tracking in addition to a third used for the voice control of the entire system. But since the labkits AC97 codec and chip can only handle a single microphone source per labkit, we initially decided to use to PmodMIC chips that contained their own preamp and 12bit ADC that could be attached to the setup and used for rotating the assembly. However, after overpowering one of the units and breaking it we were forced to change plans. I will discuss the initial progress we got with the PmodMIC chips and what lead to their failure before discussing the final design because a good portion of my time was spent working on this version before having to restart.

PModMIC Interface

The Digilent PmodMIC uses an TI ADCS7476AIM 12 bit A/D converter in order to take the audio input to the mic and output a serial digital signal. The mic included its own interface code written in VHDL for use in both creating a single 12 bit bus from the input signal as well properly clocking and enabling the mic. However, this codec was written specifically for Digilent's boards and therefore did not match the clocking of our board. Additionally, the codec is just the simpler state machine shown in Figure 4, so it seemed easy to just implement my own version in verilog using their code andknowledge of the state machine.

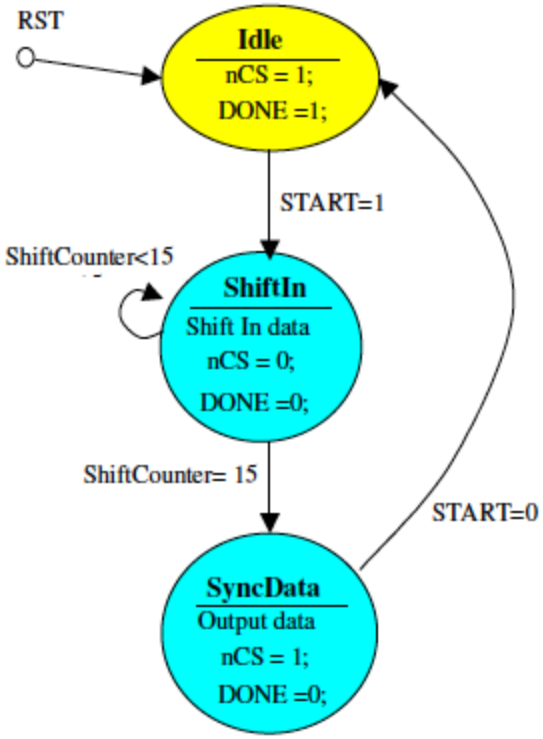

Figure 4: The state Machine for the PmodMIC interface as given by Digilent. After recieving a start signal data is serially shifted into the registers in the module before the data is output as a single parallel grouping of the 12 data bits from the A/D converter. (PmodMIC Reference Component, Digilent)

The PmodMIC itself specified that it needed a 12.5Mhz clock signal. However, it was impossible to produce that speed of clock using a DCM and the 27Mhz signal. Therefore we looked at the specs of the TI A/D convertor which the clock was running and found that it could work at ranges from 10 kHz to 20 MHz. Therefore we implemented the clock divider so it would simply do it by dividing by powers of 2 in order to get a slower clock. We settled on a clock speed of 52 kHz because it provided the best data acquisition when used along with the integrator. But we also had used a 13.5 MHz signal initially to be as close to the original required signal as possible.

The rest of the module ran off this slower clock in order to run the state machine, following the state machine shown above in Figure 4. In the IDLE state, the module waits for a start signal while the last collected data is stored in the data bits. once a start signal is recieved it goes to the SHIFT_IN state where the bits of data from the microphone are pushed in serially till 16 bits of data are collected (the 12 data bits plus 4 leading zeros). Once 16 bits have been collected, the data goes to the SYNCDATA state where the 12 data bits are sent to the output and can be received by the integrator module.

Integrator Module

The encoding of the audio data from the PmodMic was in a weird unsigned form. When no audio data was picked up by the microphone the interface would output a constant baseline DC signal that would register at a magnitude of about 480. Then as an audio signal was input into the mic the wave of the signal would be added to this baseline with an added offset for how loud it was. This caused it so the signal was never below the baseline value of 480.

In order to get a sense for how loud the audio coming into each mic was, I originally planned on having a circular buffer that would store the last N signals from the PMOD mic and would add up their magnitudes in order to get a good estimate as to the current amplitude of the signal. However, because the center of the signal increased as the volume increased, this method was unworkable at getting a consistent signal. Additionally the Pmod had very small changes in amplitude even for really loud sounds, rarely ever using all 12 bits of data.

Therefore we switched to finding the local maximum of the signal and holding at that value till another local maximum was found. This had the benefit of requiring less data be stored (only the previous bit rather than the previous N) and worked well for a signal where the maximum of the signal was raised based on how loud the signal was, even when it was a very small amount. However, in order to deal with the different clock speeds of the PmodMIC and the integrator, we had to make sure that a single signal was only processed once instead of 512 times. Therefore, once a signal has been processed a *done_stop* bit is raised to tell the circuit not to run the integrate till the next time done is raised after *done* has been at 0 for at least a clock cycle. Through this method we were able to implement something that got realistic audio levels for the two mics.

Testing the System and Breaking the Pmod

As each module was added to the system, I tested them to make sure they worked together. I first tested the *servo_motor* interface with the override control block, but the audio control block would be meaningless without audio data to run it, so I first tested one PmodMIC interface with one integrators to make sure I would get good values. Then I attached in another PmodMIC and tested to see that the output integrated to appropriately weighted values for each of the two microphones. At this point everything for the audio tracking had been implemented, so it was time to test the system all together.

Rather than attaching the mics to the camera to test it, I just placed the microphones some distance apart and spoke into them while the motor was running over towards the side. As soon as the motor was attached the microphones started to act funny, having their levels change by large margins, which caused the motor to move a lot and changed the audio level even more creating a weird oscillating effect for the motor.

It turned out that whenever the motor was running, it would change the readout from the PmodMICs because the motor drew so much of the power from the labkit there wasn't enough to power the microphones which caused some wonky results. In order to prevent this I tried attaching the additional 5 volt power supply to the servo motor rather than the lab kits 5 volts. This causes the motor to turn and not work properly on it's own, because the ground of the pulse signal and the ground of the servo were not the same ground. However I did not realize this at the time and thought that if I couldn't run the motor off the external power supply maybe I could run the PmodMICs off of it.

This was a really bad idea, because once again, the ground of the Pmod did not match the ground of the digital signals it was receiving, it completing overpowered the device and burnt out one of the mics before I was able to shut it off. This left us with a huge problem less than a week before our checkoff since we no longer had two mics for the audio tracking. After a bit of discussion and realizing there was no way we were going to be able to get a replacement Pmod, decided that we could use the AC97s on multiple boards in order to get the required mic inputs by using two old desk microphones that Gim found for us. The changing of the system to use these required changing from one board to three boards, one board for each mic input of the two audio tracking mics and one board to handle the audio from the voice control, because each labkit only one microphone preamp hooked up to the AC97.

Amplitude2

Now, the audio from each of the mics is processed by the AC97 chip on each of the two boards using the codec provided for Lab 5. The right mic is processed on the main board, where the rest of the tracking and video logic was encoded. On the other hand, the left mic is processed on an auxilary board, that just processes the audio signal. The audio from both mics is sampled at 48kHz by the AC97 and is sent to the Amplitude2 module which needed to replace the Integrator module from before in order to get the amplitude data from the new audio output of the AC97. The two biggest changes from the PmodMIC to the AC97 was that instead of 12 bits of data from the Pmod, we were not dealing with only 8 bits or data giving us a smaller range. More importantly though, was that the data output from the AC97 was signed and centered at 0 which allowed the possibility of actually integrating of the audio signal to get the amplitude.

However, the biggest reason we couldn't use the integrator module was because now that we needed to send data from one board to another, we couldn't send it over at such a high bitrate when dealing with two different clock domains. Therefore we decided to implement the system so that we could send over the amplitude data from one board to the other at only 60Hz which is slow enough to deal with the different clock domains of the two lab kits.

In order to get a 60Hz amplitude signal, I took the easiest route possible and decided that I would integrate the square of the magnitude over (48kHz/60Hz =) 800 clock cycles and at the end of

those 800 cycles I would take the sum and output the higher order 16 bits as the amplitude. This created a signal that was 60 Hz and properly represented the amplitude of the data.

In order to deal with differences between the microphones and the different lab kits, two parameters were implemented to scare the system. The first parameter is THRESHOLD, which determines if the integrated value after 800 cycles is not larger than this THRESHOLD, it's amplitude is set to 0 instead. The other parameter is MULTIPLY which scales of the signal just in case there are differences in scaling between the two microphones and lab kits. In our case, the values for the Threshold were set really high because the right microphone had a lot of noise even when no mic was connected to the system. This forced the Threshold to be as high as 6500 for both mic to prevent no signal from being seen as a difference between the two mic and turn the camera. This additionally forced the threshold for the audio control module to be larger than 6500 in order to not mistake that cutoff as being the two mics being at different levels. This overall made the camera less responsive since it takes a really loud different between the mics to get it centered.

Inter-FPGA Communication

After the use of amplitude2, the 16 bit amplitude data from the left mic is still on the auxiliary board and needs to be sent over to the main board. Since the information is so slow and relatively small, I decided to use more wires rather than more Verilog code by sending a parallel signal using 17 wires from one board to the other. The 17 signals were the 16 bits from the *amplitude_left* data as well as a done signal so that the data from *amplitude_left* would only be taken when there was new data available. The 16 bits were reversed going out of the left board so because the two boards were back to back, this enabled the main board to read them in without any additional hassle.

This changed also required a small change to the *audio_control_block* on the main board. rather than comparing the two amplitude values every clock cycle, which would cause the motor to turn too much for a single input, it will only compare the values and send out a signal whenever the done signal from the auxiliary board is active meaning only once there is new data from the left. These two changes were all that was needed to deal with the additional board, making the entire implementation a lot easier.

Testing notes

In order to test the assembly as a whole there were two main things to be tested. First in order to test the override control, the command signal was implemented using the switch[3:0] and the done signal was implemented by using button0 on the main board. The override switch was attached to switch[7]. Then it was as simple as seeing the output of what position the motor was turning towards on

the 16hex display to make sure that the motor is moving towards where we want it to move for each of the different signals. This can also be seen by witnessing if the motion of the motor is as expected.

The testing of the second component, the audio tracking was a lot more complicated. The most useful thing for seeing the audio coming in to the microphones was the logic analyzer. It allowed us to see the magnitude of the audio signals during all the different stages of processing from output of the AC97 through the signal after the amplitude module or stages in between. In order to gauge the threshold and multiply values within the amplitude modules, audio data was given into the mics from different positions while the override switch was turned on in order to get data in the logic analyzer without worrying about the camera turning. An output from such a run can be seen in Figure 5.

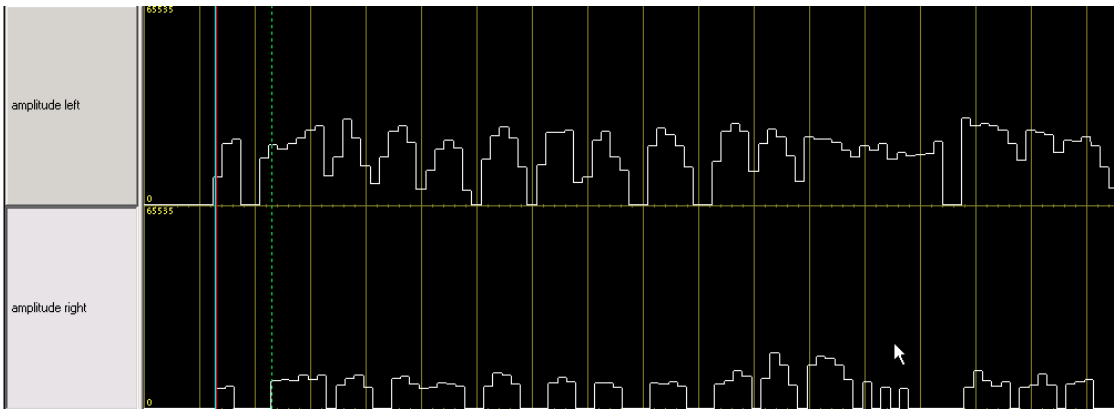

Figure 5: Output from the Logic analyzer when speaking directly into the left tracking mic. As can be seen the amplitude data from the left mic is a lot larger than the amplitude data on the right; the motor will therefore turn towards the left to try and balance out the levels.

Once the audio levels matched what would be expected with higher values when the speaker was closer to one or the other, it was time to try it out with the camera moving. It was found that whether or not the camera moved towards a voice depended on the tone of the speaker greatly. For example, If I talked to the camera like it was a baby, or made weird noises or sand, the camera would successfully turn to face towards me. However, if I shouted at the camera or whistled it would occasionally turn away from me. Clearly this shows that the system has already gained sentience and needs to be treated kindly. More seriously this shows that the audio signal coming in should probably have more signal processing in order to distinguish between the different sorts of ways one can sound while making noises as a microphone and deal with stray echoes.

Additionally the camera turned a bit too slowly, it takes multiple times of talking at the camera to get it to face you rather than a single moment of speaking. This would be helped by both a larger step

size and smaller threshold, but because of the noise in the right microphone this was not possible, though more analysis could have made the final result better than it turned out.

Overall the Audio Tracking Module was a complete success, doing everything it was designed to do even though the audio tracking had to be redesigned at the last step. In fact the transition to using the AC97 made the audio processing so much easier that if I was restarting the project now I would rather stick with 3 boards than try using a different mic to restrict on the number of lab kits. though maybe that would change if we had fully implemented the voice recognition system.

Voice Control Module (Ben/Jonathan)

The voice control module was responsible for taking in audio data from a third microphone, not mounted on the audio tracking assembly, and processing it to recognize spoken commands. While we were unable to complete and test all of the modules in the system, and thus were unable to actually recognize commands, we did complete or make substantial progress toward completion on all. The infrastructure of the other modules in the system was configured such that a completed voice command interface would be able to run off an entirely separate lab kit if required, with 5 data lines running between this module and the rest of the system.

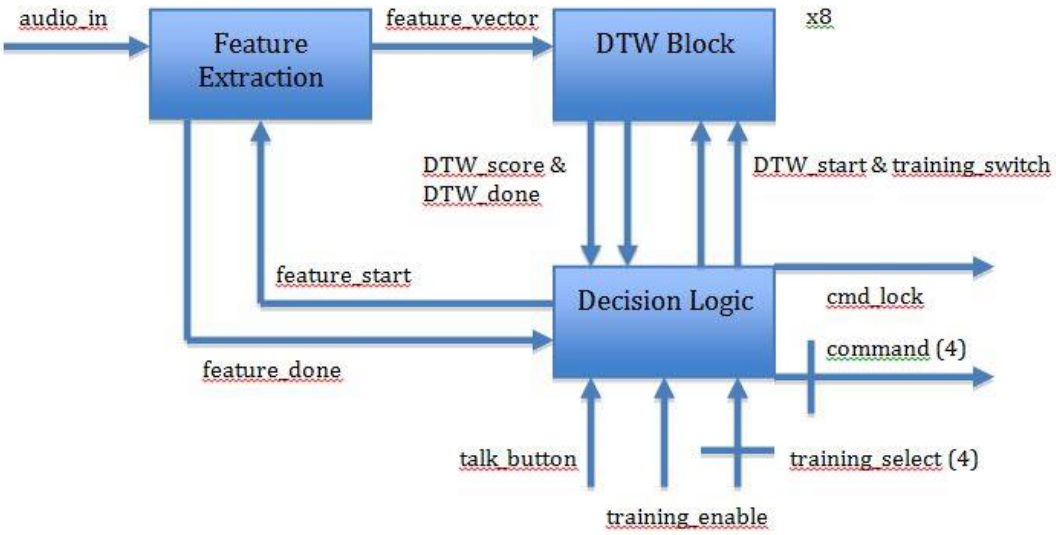

Figure 6: Block diagram of Voice Control Module. Incoming audio is buffered and fed in overlapping chunks of about 20 ms to the feature extraction module, which feeds data about the signal to the DTW blocks for training or comparison scoring. The decision logic keeps track of when calculations finish, sets the modes on the DTW blocks as needed (from external signals), and processes comparison scores to determine what command if any should be output.

The top-level module (*voice_main.v*) was intended to contain three main submodules; the feature extractor, the comparison scoring module, and the control logic module. Each of these is described in more detail below. On the recording of a voice sample, the overall flow of the system would

be to divide the data into overlapping chunks of about 30 ms of data (with the start of each chunk separated by about 10 ms, such that adjacent chunks overlap), and operate on each chunk to generate a feature vector based on the frequency content of the chunk. Based on whether the module was set to recording or training mode, this feature vector would either be stored as a template in one of the comparison blocks, or simultaneously compared to all existing templates; in the latter case, the control logic would then read in each of these comparison scores, and output the command with the best comparison score to the other modules in the system (or no command if none of the comparisons gave a value within a threshold).

For feature extraction, we chose to use Mel spectral coefficients, after consulting with our project mentor and looking at examples of successful previous 6.111 projects that dealt with speech recognition. The Mel scale is an alternative frequency scale, logarithmic with hertz, that is suitable for spectral analysis of speech. Mel coefficients represent the total power of spectral data within a number of bins, covering the full frequency range with bins at higher frequencies covering a larger span of frequencies. To calculate these coefficients, a series of overlapping triangular filters are constructed, with endpoints spaced equally on the Mel scale (such that higher-frequency filters are wider on the hertz scale), and multiplied with the Fourier transform of the data being analyzed. The filters overlap such that each section of the spectrum is included by two bins, along with both higher and lower frequencies. The picture below gives an idea of what these filters look like, for a smaller number of filters than our design. The set of coefficients generated from all of the filters forms a vector in “feature space”, and the series of vectors generated from calculating the coefficients on many chunks of speech data can be compared to a template to determine whether the words being said during the two separate audio samples was the same.

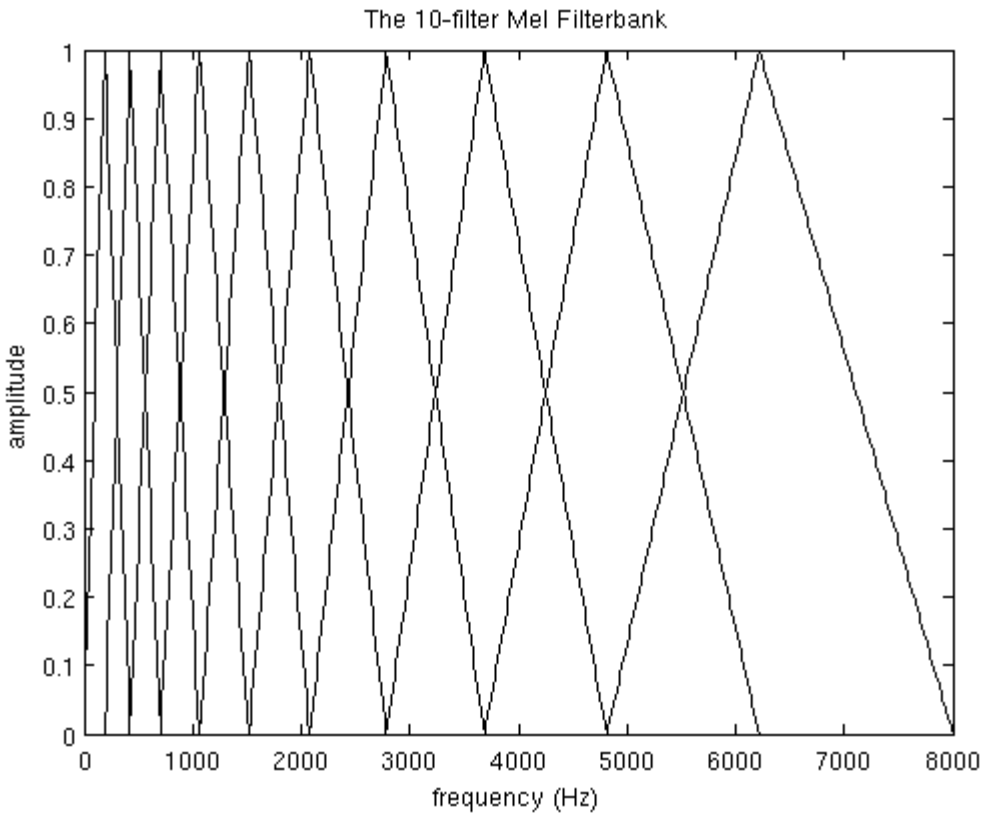

Figure 7: Triangular filters for a 10-filter system with max frequency of 8 kHz. From <http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/>.

Our system used 20 filters, with endpoints spaced by 135 Mels; this gave a top frequency of 2835 Mels, almost exactly 8000 Hz, the maximum frequency content available in our system.

For audio input, we used a standard headset microphone, plugged into the lab kit's microphone jack. The AC 97 audio interface code, as used in a previous lab, was modified to take in data at 16 kHz, and store 16000 samples (exactly 1 second of data) in a BRAM. The sampling frequency was chosen to cover a full representative range of human speech, while being easily obtainable from the standard sampling frequency of 48 kHz by only recording every third sample.

Feature extraction (Ben)

A block diagram of the feature extraction module is included below. The major components are the Mel coefficient generator and the FFT; additional calculation would be done in this module as well, to multiply the values computed by the FFT by the filter coefficients and sum the results to give a measure of the total spectral content of each bin. The order of the calculations was enforced by an internal state machine, with one major state per computation step, as well as secondary states within each of those for keeping track of progress.

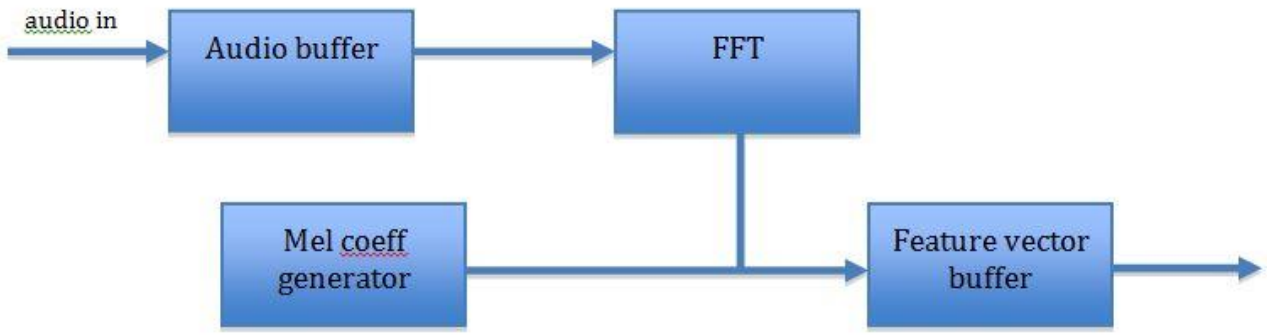

Figure 9: Block diagram of feature extraction module. Incoming audio is stored in chunks of about 30 ms and run through a 512-point FFT. The squared magnitude of the FFT is run through a series of triangular Mel filters to create a feature vector that can be read out.

The FFT module was provided by Michael Price, with minor modifications for compatibility between different versions of ISE. This code is unfortunately not licensable for redistribution, but ISE's built-in core generation tools are also capable of providing similar functionality. We used a 512-point FFT, with 8-bit audio data, which output 8-bit signed real and imaginary coefficients. These coefficients would then be squared and added to produce a magnitude.

From the 512-point FFT, only the first 256 coefficients were saved to module state; these correspond to the positive-frequency signal component, of the most relevance to the speech data.

The Mel frequency coefficients were calculated in a separate module, called *mel_filter*. The inputs to this module were *filter_num* (ranging from 0 to 18) and *coeff_num* (ranging from 0 to 255); on each clock cycle, the module would output the Mel filter coefficient corresponding to the FFT point at position *coeff_num*, as it would be in the *filter_num*'th triangular filter. A Python script was used to generate these coefficients, and is included in the appendix. Each coefficient is a 10-bit value, between 0 and 1023.

The next step in the calculation would be to combine the Mel filters with the FFT data, and then sum the results in each bin to generate a feature vector of all the results. This is about the point in implementation where we ran up against deadlines, and were not able to complete the design. The results from this calculation were to be pushed to a BRAM buffer, to be read out by the vector comparison blocks described in the next section.

Vector comparison (Jonathan)

Our initial design called for using the Dynamic Time Warping (DTW) algorithm to compare samples of audio data to templates. The DTW algorithm uses dynamic programming to calculate the best mapping between two series of vectors, such that different ways of saying the same word would get similar scores in the voice recognition. However, due to time constraints, we simplified our comparison blocks to use Euclidean distance between feature vectors as the comparison metric. This type of system would not be ideal for an actual recognition application, as it would require exact time matching between different recordings of the same word, but would serve for a proof-of-concept prototype.

In DTW the comparison between feature vector and the stored value occurs for each possible word in the vocabulary of the system. Each DTW engine therefore needs to be trained with the word that is supposed to be compared to for the audio information. This means that each engine also needs a training mode, which is why each module has a training enable switch.

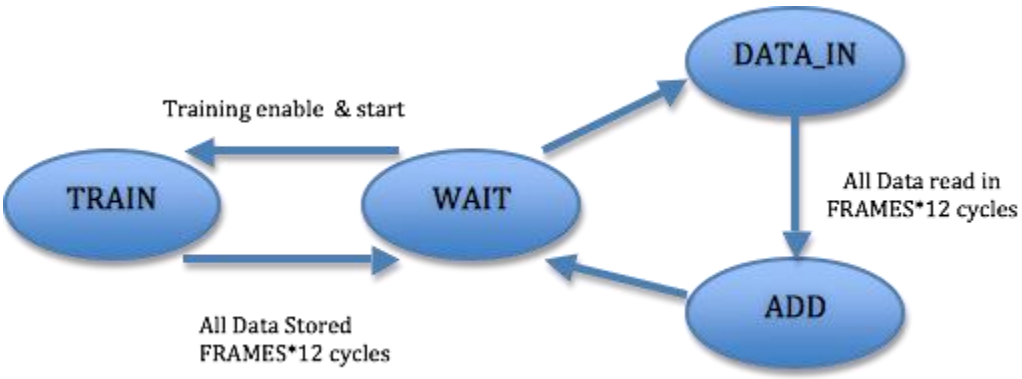

Figure 10: The state machine for each DTW engine. The four states take care of waiting for new data in WAIT, storing the data as training for given word in TRAIN, storing in data for the comparison while calculating the Euclidian distance of each point in DATA_IN and finally in the adding these values up and returning the score in ADD.

In order to implement the reduced Euclidean distance scoring for the voice recognition, the module DTW_score was implemented at a State Machine with four different states as viewed in Figure X. During WAIT, the DTW engine waits for a start signal. If the Training enable bit is one when start set to high, the engine enters the TRAIN mode. If train_enable is not enabled than the state machine enters the DATA_IN state. In the TRAIN mode the data is serially inputted into the test array of registers to store the data where is can be compared to an actual input signal. The ordering is following the length 12 feature vectors times the number of Frames taken (which is a parameter to allow changing) each containing 8 bits. Once all the bits have been stored in the test register, the state returns to WAIT to wait for an actual feature vector to compare.

Once in the DATA_IN state, the data is brought in serially from the feature extraction and as each 8 bit value is brought in, the magnitude of the difference squared between this value and the stored test data is calculated and stored in a temp register array. Once all the data has be brought in and the calculation stored in tmp, the module enters the ADD state. In the ADD state the module simply goes through all the entries in tmp and adds them together to get the final euclidean distance between the two vectors. This calculated value is then sent back to the control logic along with a done signal. Nine of these DTW engines would have been implemented, eight for each of the eight signals and a ninth for background noise so we could see if the signals did not match any of them well.

Control Logic (Jonathan)

The control logic takes care of organizing and ordering the use of the Voice control system, as well as interfacing with the other two major components through the use of the command signal and deciding which of the DTW scores is the best thereby choosing the command signal. The command signals were chosen to be 4 bits in order to hold all of the 8 commands we wanted to implement for the camera motion and the video filters and leave additional space for any other commands we wanted to implement, especially a command to let both components know no real command was recorded. These commands were separated into two groups based on the higher order bits so that all commands with only bit[2] high would change the video filter and any command with only bit[3] high would only control the motion of the camera. The commands chosen for each of the Modules is shown in Table 1.

Table 1: Voice Command Table: This is a reference between the internal code for each command, the Voice command word that creates that command and what each command should do. Better descriptions of each command is given in each components own section.

Command[3:0]	Command Word	Command Description
0100	None	No Video Filter
0101	Red	Red Filter
0110	Blue	Blue Filter
0111	Black	Greyscale Filter
1000	Left	Step the Camera to the Left
1001	Right	Step the Camera to the Right
1010	Go	Continuous Camera motion is last given direction
1011	Stop	Stop the camera at the current position

In order to implement the control logic and control the flow of the voice recongnition, a state machine with four states was implemented as seen below in Figure Y. During the initial state of WAIT, the logic simply waits for the talk button to be pressed and once the talk button has been pressed starts the feature extration described above, At this point the system goes to the state RECORD. In the state RECORD, it waits for the feature extraction to send its done signal, Once it recieves the done signal it sends the data off the the needed DTW engines. If the training enable switch is on, it chooses the appropriate DTW engine and both enables the training enable on that DTW engine and only starts that single engine. If its a normal signal it starts all of the DTW engines. At this point it enters the DTW state. This state is just a state to wait for the DTW engines to complete. Once all of the activated engines complete, if training enable is on, it returns to the WAIT state. However, if the training switch is off, the scores need to be compared to determine which command to send. At this point therefore the logic enters the COMPARE state which is where this module does most of its work.

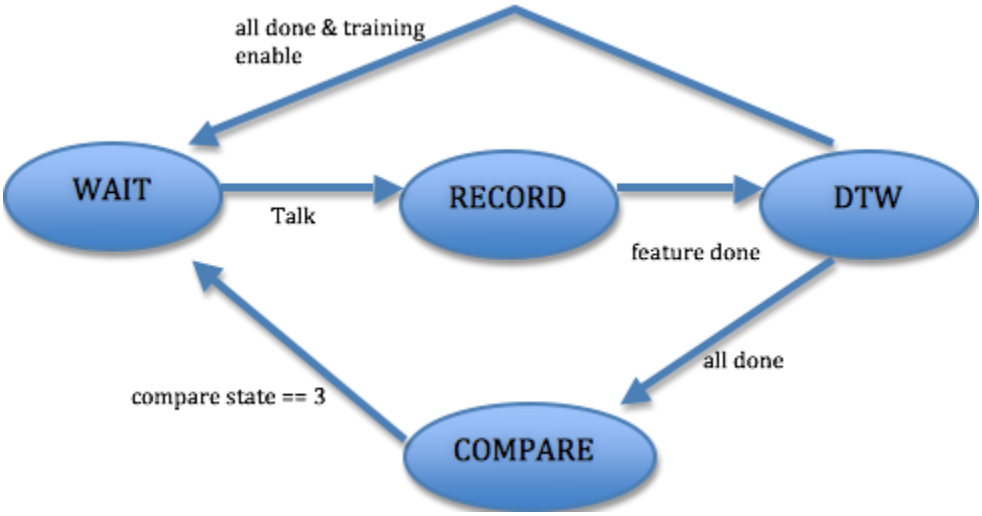

Figure 11: The state machine for the control logic of the voice recognition component. It goes through the states of Waiting for something to start to be recorded in WAIT, then waiting for that recoding and feature extraction to finish in RECORD. After that it starts the DTWs the number they start depending on training_enable in the DTW state. If the system is in training it then returns to the WAIT state. If it is actually running, it will go the COMPARE state where it will choose which DTW score is the lowest thereby choosing that message before returning to the initial WAIT step.

The COMPARE state itself has a number of substates which separates the comparing of all the scores into 4 steps. In the first 3 steps the scores are compared in tournament style with the lower score winning. In the first quarter finals round DTW1 is compared to DTW2, DTW3 is compared to DTW4 and so on. the lowers scores value and number are also stored for the following round. In the second and third rounds the winners of the previous rounds face off in a similar manner so that by the end of the third cycle only one DTW score remains and it is the lowest score.

However before this winner sends its command out, it is compared to the score from DTW9 in the fourth cycle to confirm the signal is more like one of the words that background noise. If the signal is a better signal than random noise than the command corresponding with that DTW (ordered exactly as the commands are in Table 1) and the system returns to the WAIT state.

This module and the DTW module were unable to be tested because we were never able to get a signal through the feature extraction to really see a good signal worth comparing. However in the future with more time this would be nice to see so that both of these modules could be improved into a better form.

Video Interface Module (Ben)

The video interface module dealt with everything between the camera, which was plugged into the lab kit's composite video port, and the VGA display. An overall block diagram of the system is below.

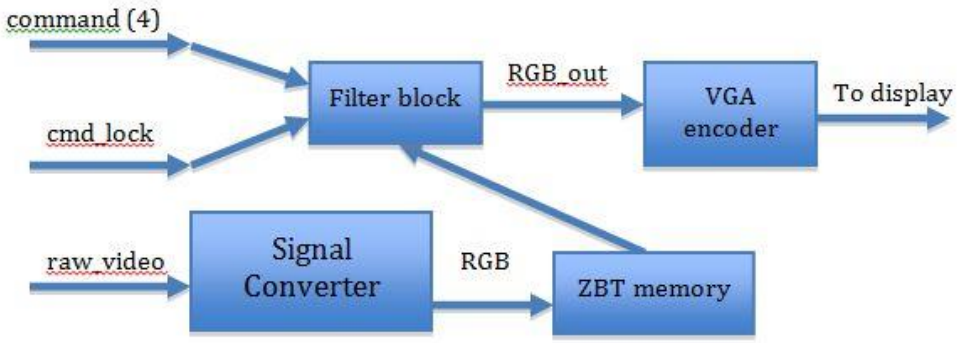

Figure 12: Block diagram of Video Interface Module. This module converts the raw NTSC video from the camera to RGB, and stores frames in the ZBT memory. These data are which are sent to a filter block, which selects from several color filters. A pulse on cmd_lock locks in a command, which updates the filter being displayed.

We started with the lab-provided template code for converting NTSC video data to black and white data for an RGB monitor, storing the intermediary data in the lab kit's onboard ZBT memory to allow the pixel data to be pushed to the screen asynchronously from it being sent from the camera to account for different clocks and timings between the NTSC and VGA standards. The lab-provided code stores four samples of 8-bit intensity data in each 36-bit entry of the ZBT memory. This intensity data can be directly read out from the decoded NTSC data, which uses YCrCb encoding, the Y corresponding to intensity (and thus the brightness of a grayscale pixel).

Our first modification was adding in another lab-provided module to convert the full 30-bit YCrCb data into 24 bits of RGB data, 8 bits for each of the three color channels. However, to store the full 24 bits of data would require a separate entry in the ZBT memory for each pixel of video, for which the size of the ZBT memory is insufficient. To account for this, we truncated the RGB data to 18 bits (6 for each channel), and then modified the NTSC interface code such that only two pixels were stored per memory entry, which used twice as much memory as the original black and white video but could still be accommodated by the kit. The memory addresses used to read out data were also modified to account for this change in storage location. During this conversion, we encountered a persistent bug which caused vertical bars to appear in the displayed image; it turned out to be a mismatch between the clocking signals used to access the ZBT memory and to read out raw camera data.

The filter module took in a 2-bit value to select which (if any) filter would be applied, 24-bit buses for RGB input and output data, and a single bit value *in_frame* generated by the top-level module which signified whether the pixel currently being operated on was in the portion of the screen occupied by the camera video display. The *in_frame* bit was necessary to prevent unintended display artifacts caused by the ZBT interface process, such as partially duplicated displays or seemingly random color values in the area outside the video display. From the 1024x768 VGA screen, we used only pixels in columns 35 to 743, and rows 79 to 563, and otherwise displayed black, which made a frame around the display (cutting off a couple pixels on each edge). All of the filters used operated on a per-pixel basis; this was done for simplicity of implementation, as the filters were not intended as a major goal of the project. However, the system was easily expansible to select for more filters by adding additional control lines to this module.

The interface for sending commands to the video interface module was designed to accommodate commands being sent from a completely external module (possibly on another board), from which as few connecting wires would be desirable. The only command inputs were a 2-bit bus and a *command_lock* line to indicate that the command currently on the bus was valid data. The integrated main module described below generated each of these from the full 4-bit command bus and global command lock line.

Integration (Ben)

The video and tracking master modules were combined into a single integrated module, for programming onto the same lab kit. The only functionality added to this module was to distinguish which commands were intended for the tracking module and which were for the video module. Four of the kit switches were used as an input bus to encode the command, with one of the buttons providing a *command_lock* line. Each module recognized 4 distinct commands, encoded using the two lower-order bits of the command. The two higher-order bits encoded the target device, a value of “10” corresponding to the tracking module and “01” to the video module; other combinations would be an invalid command, and not generate a command locking signal on either. This was consistent with the design for the voice recognition module, which used 4 lines to encode a command and a single line which would pulse high on a recognized command to indicate completion to other modules. The integrated file, *track_and_video.v*, is included in the appendix. The device encoding was used to generate a single control bit for each module to use, raised high when that module should be paying attention to the command.

Conclusions

We were able to create a functioning video console, which automatically tracked a user's voice to keep a camera pointed toward them, and which could take in commands to both override the automatic camera tracking to adjust position and apply filters to the video display. Although we were ultimately not successful in adding a voice recognition system, we were able to implement two of the three main modules, and make substantial progress toward the third.

If starting the project over again from scratch, we would definitely want to think again about the allocation of time to the different modules, and what additional components to have alongside the voice recognition system. We ended up spending more time toward the beginning on finishing both the video and tracking modules first, when this might have been better spent on prototyping the voice recognition module (using Matlab or Python) for easier implementation later on.

We also found the AC 97 codec, and the mics that used it, better suited for the audio tracking module. Although it did require an extra lab kit to use, if the resources are available, it made the task of extracting audio data much easier, and the mics were of better quality for the particular application.

Appendix

Note that some lab-provided code has been redacted from the below.

Appendix A – video_filter Verilog

```
/*
Basic video filter module. Applies various color filters to RGB data, and selects between outputs.
Filters (from value of "option" bus):
0 - none
1 - red tint
2 - blue tint
3 - grayscale

in_frame input makes display black outside of video area

*/

module video_filter(clk, rgb_in, option, rgb_out, in_frame);

  input clk;
  input [23:0] rgb_in;
  input [1:0] option;

  input in_frame;

  output reg [23:0] rgb_out;



  // Individual color components for RGB in/outs
  wire [7:0] r_in, g_in, b_in;
  assign r_in = rgb_in[23:16];
  assign g_in = rgb_in[15:8];
  assign b_in = rgb_in[7:0];

/*
  wire [7:0] r_out, g_out, b_out;
  assign r_out = rgb_out[23:16];
  assign g_out = rgb_out[15:8];
  assign b_out = rgb_out[7:0];
*/

  // Recalculate intensity value from RGB coefficients
  wire [7:0] intensity;
  assign intensity = (r_in >> 2) + (r_in >> 5)
                   + (g_in >> 1) + (g_in >> 4)
                   + (b_in >> 4) + (b_in >> 5);

  // Mux between outputs
  always @(posedge clk) begin
    if (~in_frame) begin
      rgb_out <= 24'b0;
    end
    else begin
      case (option)
        2'd1: rgb_out <= {r_in, g_in >> 2, b_in >> 2};
        2'd2: rgb_out <= {r_in >> 2, g_in >> 2, b_in};
        2'd3: rgb_out <= {intensity, intensity, intensity};
        default: rgb_out <= rgb_in;
      endcase
    end

  end // always
```

```
endmodule
```

Appendix B – audio_amplitude2 Verilog

```
module audio_amplitude2(
  input clock,//system clock
  input reset,//system reset
  input ready,//ready signal from ac97
  input [7:0] audio_in,//audio data from ac97
  output [15:0] amplitude,//integrated output data
        output [17:0] temp,//temp value used for debugging
        output done//done output
  );

        //PARAMETERS
        parameter MULTIPLY = 1; //multiply value for right mic
        parameter THRESHHOLD = 6500;//Threshhold for right mic

        reg [9:0] count;//count for data coming in
        reg [15:0] amplitude_reg;//register for amplitude data
        reg [17:0] tmp_reg;//tmp register holding the sum
        reg done_reg;//done register

        wire [7:0] data;

        assign data = ((audio_in[7])? (~audio_in + 1): audio_in); //data comes in signed so take the magnitude

        always @(posedge clock)begin
                if(reset) begin//if reset clear the values
                        amplitude_reg <= 0;
                        count <= 0;
                        tmp_reg <= 0;
                        done_reg <= 0;
                end
                else begin
                        if(ready) begin//when ready take in data entry square it and divide by 2^6
                                tmp_reg <= tmp_reg + ((data*data)>>6);
                                count <= count+1;
                                done_reg <= 0;
                        end
                        else if(count == 10'd800) begin //once 800 sample have been taken multiply by MULTIPL value and take upper
16 bits
                                amplitude_reg <= MULTIPLY*((tmp_reg[17:2] > THRESHHOLD)? tmp_reg[17:2]: 0);
                                tmp_reg <= 0;
                                count <= 0;
                                done_reg <= 1;
                        end
                end
        end

        //assignment of registers to outputs
        assign amplitude = amplitude_reg;
        assign temp = tmp_reg;
        assign done = done_reg;

endmodule
```

Appendix C – audio_control Verilog

```
module audio_control(
  input clock,//system clock
  input reset,//system reset
  input [15:0] intL,//integrated data from left mic
  input [15:0] intR,//integrated data from right mic
        input intL_done,//ready signal from left mic (since sent from other board)
  output dir,//output direction 0 for left, 1 for right
```

```
  output [7:0] val,//output value (how many step to turn)
  output done//output done signal
  );

        //registers for outputs
        reg dir_reg, done_reg;
        reg [7:0] val_reg;

        //PARAMETERS
        parameter THRESHHOLD = 16'd8000; //threshhold difference needs to be for motor to turn
        parameter STEPUNIT = 8'd16; //how large a step should be taken for each unit of amplitude difference


always @ (posedge clock) begin
        if (reset) begin// on reset clear the registers
                dir_reg <= 0;
                val_reg <= 0;
                done_reg <= 0;
        end
        else if(intL_done) begin//when we've gotten data from the left mic
                if(intR > THRESHHOLD + intL) begin //if the right mic is louder turn right
                        dir_reg <= 1;
                        val_reg <= STEPUNIT*((intR-intL-THRESHHOLD)/32);
                        done_reg <= 1;
                        end
                else if(intL > THRESHHOLD + intR) begin// if the left mic is louder turn left
                        dir_reg <= 0;
                        val_reg <= STEPUNIT*((intL-intR-THRESHHOLD)/32);
                        done_reg <= 1;
                        end
                else done_reg <= 0;
        end

end

//Assign output registers
assign dir = dir_reg;
assign val = val_reg;
assign done = done_reg;

endmodule
```

Appendix D – track_and_video Verilog

```
  //////////////////////////////////////////////////////////////////////////////
  //
  // Reset Generation
  //
  // A shift register primitive is used to generate an active-high reset
  // signal that remains high for 16 clock cycles after configuration finishes
  // and the FPGA's internal clocks begin toggling.
  //
  //////////////////////////////////////////////////////////////////////////////
  wire power_on_reset;
  SRL16 reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(power_on_reset),
         .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
  defparam reset_sr.INIT = 16'hFFFF;

  // ENTER button is user reset
  wire reset,user_reset;
  debounce db1(power_on_reset, clock_27mhz, ~button_enter, user_reset);
  assign reset = user_reset | power_on_reset;

  //////////////////////////////////////////////////////////////////////////////
  //
```

```
// Control logic
//
//////////////////////////////////////////////////////////////////////////////

// for non-voice command selection
//
wire set_tracking, set_filter;
assign set_tracking = switch[3] & ~switch[2];
assign set_filter = ~switch[3] & switch[2];

wire [1:0] param_sel;
assign param_sel = switch[1:0];

// Button 0 locks in commands
debounce button0_dbounce(.clk(clock_27mhz), .reset(reset),
                .noisy(~button0), .clean(b0_db));

wire cmd_lock; // gives positive pulse on debounced button press
reg b0_db_prev;
always @(posedge clock_27mhz)
  b0_db_prev <= b0_db;

assign cmd_lock = b0_db & ~b0_db_prev;


//////////////////////////////////////////////////////////////////////////////
//
// Audio tracking modules
//
//////////////////////////////////////////////////////////////////////////////

        wire readymic; //is 1 when data from lab5audio is ready
        wire [7:0] audio_right; //audio in from right mic
        wire [15:0] amplitude_right; //integrated value from right mic
        wire [17:0] temp;//temp value for debugging

        reg [7:0] audio_right_reg; //register to reduce noise for debugging of audio_right
        reg [15:0] amplitude_left; //integrated value from left mic sent over from other board

        //lab5audio takes in the audio from the mic and outputs the digital data for use
        lab5audio a(.clock_27mhz(clock_27mhz), .reset(reset),
                                                .volume(0), .audio_in_data(audio_right),
                                                .audio_out_data(0), .ready(readymic),
            .audio_reset_b(audio_reset_b), .ac97_sdata_out(ac97_sdata_out)
                                , .ac97_sdata_in(ac97_sdata_in),.ac97_synch(ac97_synch),
                                .ac97_bit_clock(ac97_bit_clock));

        //audio_amplitude2 takes the audio from the mic and integrates every 800 samples to get 60Hz of amplitude data
        audio_amplitude2 amp2(.clock(clock_27mhz),
                        .reset(reset || !button_down),
                                                                .ready(readymic), .audio_in(audio_right),
                                                                .amplitude(amplitude_right), .temp(temp));


        always@(posedge clock_27mhz) begin
                        if(readymic)begin
                                        audio_right_reg <= audio_right; // storing audio in audio_right_reg for debugging
                                        end
                        if(user2[16]) begin
                                        amplitude_left <= user2[15:0]; // amplitude data from second board
                        end
        end



        //Logic Analyzer data
```

```
  assign analyzer3_data = {amplitude_left};
  assign analyzer3_clock = clock_27mhz;

        assign analyzer1_data ={amplitude_right};
        assign analyzer1_clock = readymic;


        //CONTROL LOGICs
        wire audio_dir, override_dir,done_in;//
        wire override;
        debounce dir_bounce(.clk(clock_27mhz), .reset(reset), .noisy(~button0), .clean(done_in));
        debounce over_bounce(.clk(clock_27mhz), .reset(reset), .noisy(switch[7]), .clean(override));

        wire [15:0] count, new_count;
        wire ready, override_done,audio_done;
        wire [7:0] override_val,audio_val;
        wire [3:0] command;


        //audio control module takes amplitude data and returns a direction and value for the motor to turn
        audio_control aud_control(.clock(clock_27mhz), .reset(reset |!button_down),
                                                                                .intL(amplitude_left),
.intR(amplitude_right),
                                                                                .intL_done(user2[16]),
.dir(audio_dir), .val(audio_val),
                                                                                .done(audio_done));

        //over_ride control takes a command and uses it to output a direction and value for the motor to turn
        override_control over_control(.clock(clock_27mhz),
                        .reset(reset || !button_down),
                                                                        .command(switch[3:0]), .done_in(done_in),
                                                                        .dir(override_dir), .com_debug(command),
                                                                        .val(override_val), .done(override_done));

        //servo_interface takes the value and direction information and uses it to create PWM and control the servo
        servo_interface servo(.clock(clock_27mhz),
                .reset(reset || !button_down),
                                                        .audio_dir(audio_dir), .audio_val(audio_val),
                                                        .audio_done(audio_done), .override_done(override_done),
                                                        .override(override),
                                                        .motor_out(user4[0]), .override_dir(override_dir),
                                                        .override_val(override_val), .count(count), .new_count(new_count));

        //DISPLAY INFORMATION

        assign led[7] = !done_in;
        assign led[6] = !override_done;
        assign led[0] = !override;

        assign led[5:1] = 5'b11111;

        //16 hex dispay
        display_16hex disp(.reset(reset), .clock_27mhz(clock_27mhz),
.data_in({amplitude_right,audio_val,3'b0,audio_dir,3'b0,audio_done,override_val,command,4'b0,count}),
                .disp_blank(disp_blank), .disp_clock(disp_clock), .disp_rs(disp_rs), .disp_ce_b(disp_ce_b),
                .disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));



  ////////////////////////////////////////////////////////////////////////
  // Demonstration of ZBT RAM as video memory

  // use FPGA's digital clock manager to produce a
  // 65MHz clock (actually 64.8MHz)
  wire clock_65mhz_unbuf,clock_65mhz;
  DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
```

```
  // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
  // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
  // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
  // synthesis attribute CLKIN_PERIOD of vclk1 is 37
  BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

//   wire clk = clock_65mhz;  // gph 2011-Nov-10


            wire locked;

            //assign clock_feedback_out = 0; // gph 2011-Nov-10

  ramclock rc(.ref_clock(clock_65mhz), .fpga_clock(clk),
                                                        .ram0_clock(ram0_clk),
                                                        //.ram1_clock(ram1_clk),   //uncomment if ram1 is used
                                                        .clock_feedback_in(clock_feedback_in),
                                                        .clock_feedback_out(clock_feedback_out), .locked(locked));



  // generate basic XVGA video signals
  wire [10:0] hcount;
  wire [9:0]  vcount;
  wire hsync,vsync,blank;
  xvga xvga1(clk,hcount,vcount,hsync,vsync,blank);

  // wire up to ZBT ram

  wire [35:0] vram_write_data;
  wire [35:0] vram_read_data;
  wire [18:0] vram_addr;
  wire        vram_we;

  wire ram0_clk_not_used;
  zbt_6111 zbt1(clk, 1'b1, vram_we, vram_addr,
                           vram_write_data, vram_read_data,
                           ram0_clk_not_used,   //to get good timing, don't connect ram_clk to zbt_6111
                           ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

  // generate pixel value from reading ZBT memory
  wire [17:0]              vr_pixel;
  wire [18:0]              vram_addr1;

  vram_display vd1(reset,clk,hcount,vcount,vr_pixel,
                           vram_addr1,vram_read_data);

  // ADV7185 NTSC decoder interface code
  // adv7185 initialization module
  adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
                              .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
                              .tv_in_i2c_clock(tv_in_i2c_clock),
                              .tv_in_i2c_data(tv_in_i2c_data));

  wire [29:0] ycrcb;     // video data (luminance, chrominance)
  wire [2:0] fvh;        // sync for field, vertical, horizontal
  wire       dv;         // data valid

  ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
                              .tv_in_ycrcb(tv_in_ycrcb[19:10]),
                              .ycrcb(ycrcb), .f(fvh[2]),
                              .v(fvh[1]), .h(fvh[0]), .data_valid(dv));


  // convert ycrcb to RGB
  wire [7:0] R, G, B;
```

```
  wire [9:0] Y, Cr, Cb;
  assign Y = ycrcb[29:20];
  assign Cr = ycrcb[19:10];
  assign Cb = ycrcb[9:0];
  YCrCb2RGB rgb_conv ( .R(R), .G(G), .B(B),
                       .clk(tv_in_line_clock1), .rst(0),
                       .Y(Y), .Cr(Cr), .Cb(Cb) );

  // for storage, take highest-order RGB bits
  wire [17:0] RGB_trunc;
  assign RGB_trunc = {R[7:2], G[7:2], B[7:2]};

  // code to write NTSC data to video memory

  wire [18:0] ntsc_addr;
  wire [35:0] ntsc_data;
  wire        ntsc_we;
  ntsc_to_zbt n2z (clk, tv_in_line_clock1, fvh, dv, RGB_trunc,
                                ntsc_addr, ntsc_data, ntsc_we, 1'b1);

  // code to write pattern to ZBT memory
  reg [31:0]                zbt_count;
  always @(posedge clk) zbt_count <= reset ? 0 : zbt_count + 1;

  wire [18:0]               vram_addr2 = zbt_count[0+18:0];
  wire [35:0]               vpat = ( switch[5] ? {4{zbt_count[3+3:3],4'b0}}
                                               : {4{zbt_count[3+4:4],4'b0}} );

  // mux selecting read/write to memory based on which write-enable is chosen

  wire        sw_ntsc = 1;//~switch[7];
//   wire     my_we = sw_ntsc ? (hcount[1:0]==2'd2) : blank;
  wire        my_we = sw_ntsc ? hcount[0] : blank;
  wire [18:0]               write_addr = sw_ntsc ? ntsc_addr : vram_addr2;
  wire [35:0]               write_data = sw_ntsc ? ntsc_data : vpat;

//   wire     write_enable = sw_ntsc ? (my_we & ntsc_we) : my_we;
//   assign  vram_addr = write_enable ? write_addr : vram_addr1;
//   assign  vram_we = write_enable;

  assign     vram_addr = my_we ? write_addr : vram_addr1;
  assign     vram_we = my_we;
  assign     vram_write_data = write_data;

  // select output pixel data

  reg [17:0]                pixel;
  reg         b,hs,vs;


  always @(posedge clk)
    begin
//       pixel <= switch[0] ? {hcount[8:6],15'b0} : vr_pixel;
      pixel <= vr_pixel;
      b <= blank;
      hs <= hsync;
      vs <= vsync;
    end

  // Video filter module for each pixel
  wire[23:0] pixel_filtered;

  // Only generate pixel values for in-frame locations; otherwise, black.
  wire in_frame;
  assign in_frame = (hcount > 34) & (hcount < 744)
             & (vcount > 78) & (vcount < 564);
```

```
  // Control logic to change filter only on voice command (or simulated)
  wire [1:0] video_option;
  reg [1:0] video_option_prev;

  assign video_option = (cmd_lock & set_filter) ? param_sel : video_option_prev;

  always @(posedge clk)
    video_option_prev <= video_option;



  video_filter filter(.clk(clk),
    .rgb_in({pixel[17:12], 2'b0, pixel[11:6], 2'b0, pixel[5:0], 2'b0}),
    .rgb_out(pixel_filtered),
    .option(video_option),
    .in_frame(in_frame));


  // VGA Output.  In order to meet the setup and hold times of the
  // AD7125, we send it ~clk.

  assign vga_out_red = pixel_filtered[23:16];
  assign vga_out_green = pixel_filtered[15:8];
  assign vga_out_blue = pixel_filtered[7:0];
  //assign vga_out_sync_b = 1'b1;    // not used
  assign vga_out_pixel_clock = ~clk;
  assign vga_out_blank_b = ~b;
  assign vga_out_hsync = hs;
  assign vga_out_vsync = vs;

  // debugging

//   assign led = ~{vram_addr[18:13],reset,switch[0]};


endmodule
```

Appendix E - servo_interface Verilog

```
module servo_interface(
   input clock,//system clock
   input reset,//system reset
   input audio_dir,//input direction from audio control
   input [7:0] audio_val,//input value from audio control
   input audio_done,// done input from audio contorl
   input override_dir,//input direction from override_control
   input [7:0] override_val,//input value from override_control
   input override_done,//done signal from over_ride control
   input override,//override signal (when 1 use data from override, otherwise data from audio)
   output motor_out,//output signal to the servo motor
              output [16:0] new_count, //new pulse width count for debugging
              output [16:0] count //old pulse width count for debugging
   );

              //PARAMETERS
              parameter MS20COUNT = 20'd540000; //number of counts needed for the 20ms
              parameter MAXPULSE = 17'd67500; //Largest pulse width count
              parameter MINPULSE = 17'd13500;//smallesr pulse width count
              parameter CENTER = 17'd40500; //pulse width needed to center the servo

              //output registers
              reg motor_reg;
              reg [19:0] cycle_count;
              reg [16:0] pulse_count;
```

```
	reg [16:0] new_pulse_count;


	always @(posedge clock) begin
		if(reset) begin//on reset center the camera
			motor_reg <= 0;
			pulse_count <= CENTER;
			cycle_count <= 0;
			new_pulse_count <= CENTER;
		end
		else begin //make the pulse by having the signal be a 1 till the pulse count
			if(cycle_count == 0)begin
				motor_reg <= 1;
				cycle_count <= cycle_count + 1;
				end
			else if(cycle_count == pulse_count) begin
				motor_reg <= 0;
				cycle_count <= cycle_count + 1;
				end
			else if(cycle_count >= MS20COUNT) begin //and then 0 till the full 20ms passes
				cycle_count <= 0;
				motor_reg <= 1;
				if(new_pulse_count > MAXPULSE) pulse_count <= MAXPULSE; //keep pulse size within the set
bounds
				else if(new_pulse_count < MINPULSE) pulse_count <= MINPULSE;
				else pulse_count <= new_pulse_count;
				end
			else cycle_count <= cycle_count + 1;

			if(override) begin //if override take new pulse count from override signals
				if (override_done) begin
				new_pulse_count <= (override_dir)? pulse_count + (override_val<<4): pulse_count -
(override_val<<4);
				end
				else new_pulse_count <= new_pulse_count;
				end
			else if (audio_done)begin //otherwise take new pulse count from audio signals
				new_pulse_count <= (audio_dir)? pulse_count + (audio_val): pulse_count - (audio_val);
				end
			else new_pulse_count <= new_pulse_count;


			end

end

//output assignments to registers
assign motor_out = motor_reg;
assign new_count = new_pulse_count;
assign count = pulse_count;

endmodule
```

Appendix F – override_control Verilog

```
module override_control(
  input clock,//system clock
  input reset,//system reset
  input [3:0] command, //voice (or switches) command
  input done_in,// command is ready for use signal
  output dir,//output direction (0 is left, 1 is right)
  output [7:0] val, //output value of how much to turn
  output done,//output saying val and dir are ready
		output[3:0] com_debug//output of the stored command for debugging purposes;
  );
```

```
	//PARAMETERS
	 parameter STEPSIZE = 8'b00000010; //step size when direction is said
	 parameter GOSIZE = 8'b00000001; //step per clock cycle when in GO mode

	 //module registers including output registers
	 reg [3:0] command_reg;
	 reg dir_reg, done_reg;
	 reg [7:0] val_reg;
	 reg persist;

	 always @(posedge clock)begin
		if (reset)begin//on reset clear all the registers
			dir_reg <= 0;
			done_reg <= 0;
			command_reg <= 0;
			val_reg <= 0;
			persist <= 0;
		end
		else begin//choose outputs based on the given command
			case(command_reg)
			4'b1000: begin //Step Right
				dir_reg <= 0;
				val_reg <= STEPSIZE;
				done_reg <= 1;
				command_reg <= 0;
				end
			4'b1001: begin //Step Left
				dir_reg <= 1;
				val_reg <= STEPSIZE;
				done_reg <= 1;
				command_reg <= 0;
				end
			4'b1010: begin //Go
				val_reg <= GOSIZE;
				persist <= 1;
				done_reg <= 1;
				command_reg <= 0;
				end
			4'b1011: begin //STOP
				val_reg <= 8'b00000000;
				persist <= 0;
				done_reg <= 0;
				command_reg <= 0;
				end
			default: begin//for GO keep sending the signal otherwise stop after one clock cycle
				if(persist) begin
					done_reg <= 1;
					val_reg <= GOSIZE; //added after checkoff, makes it so changing direction while in GO
does not change speed
					end
				else begin
					done_reg <= 0;
					val_reg <= 8'b000000;
				end
			end
			endcase
		end

		if (done_in && !reset)command_reg <= command; //only store command when given the done_in signal
	end

//output registers assignments
assign dir = dir_reg;
assign val = val_reg;
assign done = done_reg;
assign com_debug = command_reg;
```

endmodule

Appendix G – feature_extract Verilog

```
/*
Computation engine for feature extraction.
- Runs FFT on chunks of audio data stored in BRAM; stores squared magnitude
-

*/
module feature_extract(clock, reset, start, done, bram_addr, bram_data, chunk_num, debug_out, status, state);

  input clock, start, reset;
  output reg done;
  output reg [13:0] bram_addr;

  input [7:0] bram_data;
  input [5:0] chunk_num;

  output reg [15:0] debug_out;
  output reg status;

  // 3e80 is highest address of BRAM
  parameter MAX_ADDR = 14'h3E80;

  // Main state machine state; keeps track of computation step.
  parameter IDLE_STATE = 4'h0;
  parameter READING_AUDIO = 4'h1;
  parameter COMPUTING_FFT = 4'h2;
  parameter READING_FFT = 4'h3;
  parameter FILTER_MULT = 4'h4;
  output reg [3:0] state = IDLE_STATE;

  parameter SAMPLES_PER_CHUNK = 200;

  // Internal state
  reg [8:0] sample_num;
  reg [13:0] addr_base;
  reg init;
  reg [4:0] filter_num;


  // FFT module I/O
  reg fft_start, fft_reset;
  reg [8:0] fft_addr_in, fft_addr_out;
  reg fft_we, fft_re;
  wire fft_done;
  reg [7:0] fft_data_in;
  wire [7:0] fft_real_out, fft_imag_out;

  fft #(.M(9), .B(8)) fft_inst(.clk(clock), .reset(fft_reset),
        .start(fft_start), .done(fft_done),
        .addr_in(fft_addr_in), .addr_out(fft_addr_out),
        .write_enable_in(fft_we), .read_enable_out(fft_re),
        .data_real_in(fft_data_in), .data_imag_in(8'b0),
        .data_real_out(fft_real_out), .data_imag_out(fft_imag_out));


  reg [15:0] fft_readout_in,
           fft_real_sq, fft_imag_sq, fft_bram_addr;
  wire [15:0] fft_readout_out;
  reg fft_readout_enable;
```

```
mybram #(.LOGSIZE(9), .WIDTH(16)) fft_out_ram(.clk(clk),
              .din(fft_readout_in), .dout(fft_readout_out),
              .addr(fft_bram_addr),
              .we(fft_readout_enable));

// Mel coefficient generator (from mel_filter.v)

reg [7:0] mel_coeff_num;
reg [4:0] mel_filter_num;
wire [7:0] mel_coeff;

mel_filter mel(.clock(clock), .coeff_num(mel_coeff_num),
            .filter_num(mel_filter_num), .coeff(mel_coeff));

always @(posedge clock) begin
  if (reset) begin
    state <= IDLE_STATE;
  end // reset
  else begin
  // Main FSM
  case(state)
    ////////////////////////////////////////////////////////////////////////
    IDLE_STATE: begin
    // Waiting for signal to begin computation.
      if (start) begin
        init <= 1;
        state <= READING_AUDIO;
        status <= 1;
      end // if
      else
        status <= 0;
      fft_we <= 0;
      fft_re <= 0;
      done <= 0;
      fft_reset <= 0;
      debug_out <= 0;


    end // IDLE_STATE

    ////////////////////////////////////////////////////////////////////////
    READING_AUDIO: begin
    // Read audio data from BRAM into FFT module, one per clock cycle.

      // On first cycle, set BRAM address to start of relevant section.
      if (init) begin
        addr_base <= chunk_num * SAMPLES_PER_CHUNK;
        init <= 0;
        sample_num <= 0;
      end // if
      // TODO make sure timings work for getting first sample
      else begin
        fft_we <= 1;
        bram_addr <= addr_base + sample_num;
        sample_num <= sample_num + 1;
        fft_addr_in <= sample_num;
        debug_out <= {bram_data, 8'b0};

        // Go to next sample, or end

      end // else

      // If on last sample, start FFT running (after
      if (sample_num == 9'd511) begin
        state <= COMPUTING_FFT;
```

```
    fft_start <= 1;
  end // if


end // READING_AUDIO


////////////////////////////////////////////////////////////////////////
COMPUTING_FFT: begin
// Wait for FFT module to finish


  if (fft_done) begin
    state <= READING_FFT;
    fft_re <= 1;
    init <= 1;
  end // if

  fft_we <= 0;
  fft_start <= 0;
  fft_reset <= 0;
  sample_num <= 0;

end // COMPUTING_FFT

////////////////////////////////////////////////////////////////////////
READING_FFT: begin
// Read out data from FFT, store results in a BRAM
// do multiplication on numbers being read out
  // TODO

  // 1: read out real & imag (needs sample_num set)
  // 2: multiply both
  // 3: store sum in BRAM

  sample_num <= sample_num + 1;

  // account for 2-cycle delay between readout sample and bram store
  fft_bram_addr <= sample_num - 1;

  // square numbers being read out

  fft_real_sq <= fft_real_out * fft_real_out;
  fft_imag_sq <= fft_imag_out * fft_imag_out;

  // store sum of squares
  fft_readout_in <= fft_real_sq + fft_imag_sq;
  debug_out <= fft_readout_in;

  // stop when writing last data point
  if (~init & (sample_num == 0)) begin
    state <= FILTER_MULT;
  end // if


end // READING_FFT

////////////////////////////////////////////////////////////////////////
FILTER_MULT: begin
// Read through FFT data from BRAM and Mel filter coeffs
// Generate vector of feature values
  // TODO



end // FILTER_MULT
```

```
    endcase
    end // else
  end // always

endmodule
```

Appendix H – mel_filter Verilog

Note: the large lookup table blocks have been snipped; they can be generated via the script in Appendix I.

```
/*
Generate Mel filter coefficients, one per clock cycle, for 257-point FFT result (from 512-point transformation discarding negative frequencies).
Parameterized by filter number.
One cycle latency between receiving coefficient & filter number, and outputting Mel filter coefficient.
*/
module mel_filter(clock, coeff_num, filter_num, coeff);

  input clock;
  input [7:0] coeff_num;
  input [4:0] filter_num;

  output [7:0] coeff;

  reg [7:0] coeff_num_p;  // buffer coeff number for next cycle
  reg [7:0] descending, ascending;
  reg [7:0] low, mid, high;

//   boundaries = [0,3,6,10,14,18,24,30,36,44,52,62,72,85,98,114,131,150,173,197,225,256]


  // Implemented as 256-entry tables, with 2 entries for each index.
  // Assumes fixed placements of bin edges, as determined by Mel scale.
  // Based on low, mid, and high, assigns entry 0, 1023, or table entry.

  // Select between possible values of coefficient based on bin selected
  assign coeff = (coeff_num_p > mid) ? ((coeff_num_p < high) ? descending : 8'b0)
                              : ((coeff_num_p > low) ? ascending : 8'b0);

  always @(posedge clock) begin

    coeff_num_p <= coeff_num;

    // Set boundaries for bins
    case(filter_num)
      5'd0: begin
        low <= 0;
        mid <= 3;
        high <= 6;
        end
      5'd1: begin
        low <= 3;
        mid <= 6;
        high <= 10;
        end
      5'd2: begin
        low <= 6;
        mid <= 10;
        high <= 14;
        end
      5'd3: begin
        low <= 10;
        mid <= 14;
        high <= 18;
        end
      5'd4: begin
```

```
  low <= 14;
  mid <= 18;
  high <= 24;
  end
5'd5: begin
  low <= 18;
  mid <= 24;
  high <= 30;
  end
5'd6: begin
  low <= 24;
  mid <= 30;
  high <= 36;
  end
5'd7: begin
  low <= 30;
  mid <= 36;
  high <= 44;
  end
5'd8: begin
  low <= 36;
  mid <= 44;
  high <= 52;
  end
5'd9: begin
  low <= 44;
  mid <= 52;
  high <= 62;
  end
5'd10: begin
  low <= 52;
  mid <= 62;
  high <= 72;
  end
5'd11: begin
  low <= 62;
  mid <= 72;
  high <= 85;
  end
5'd12: begin
  low <= 72;
  mid <= 85;
  high <= 98;
  end
5'd13: begin
  low <= 85;
  mid <= 98;
  high <= 114;
  end
5'd14: begin
  low <= 98;
  mid <= 114;
  high <= 131;
  end
5'd15: begin
  low <= 114;
  mid <= 131;
  high <= 150;
  end
5'd16: begin
  low <= 131;
  mid <= 150;
  high <= 173;
  end
5'd17: begin
  low <= 150;
```

```
        mid <= 173;
        high <= 197;
        end
      5'd18: begin
        low <= 173;
        mid <= 197;
        high <= 225;
        end
      default: begin
        low <= 0;
        mid <= 0;
        high <= 0;
        end
    endcase


    case(coeff_num)
      // Values generated with python (create_coeffs.py)
      8'd0: ascending <= 1023;
     …
      8'd255: ascending <= 990;
    endcase

    case(coeff_num)
      8'd0: descending <= 1023;
     …
      8'd255: descending <= 33;
    endcase
  end //always

endmodule //mel_filter
```

Appendix I – Mel coefficient python script

```
boundaries = [0,3,6,10,14,18,24,30,36,44,52,62,72,85,98,114,131,150,173,197,225,256]
ascending = [0]*257
descending = [0]*257

for i in xrange(len(boundaries)-1):
  left = boundaries[i]
  right = boundaries[i+1]
  for j in xrange(left, right):

    ascending[j] = int(round(1023.0*(j-left)/(right-left)))
    descending[j] = 1023 - int(round(1023.0*(j-left)/(right-left)))


for b in boundaries:
  ascending[b] = 1023

# Generate verilog code for assignments

'''
for i in xrange(257):
  print "assign ascending[" + str(i) + "] = " + str(ascending[i]) + ";"
for i in xrange(257):
  print "assign descending[" + str(i) + "] = " + str(descending[i]) + ";"
'''

for i in xrange(256):
  print "  8'd" + str(i) + ": ascending <= " + str(ascending[i]) + ";"
for i in xrange(256):
  print "  8'd" + str(i) + ": descending <= " + str(descending[i]) + ";"
```

Appendix J – Voice logic Verilog

```
module logic(
  input clock,//system clock
  input reset,//syetm reset
        input talk,//push to record button
        output reg feature_start,//start trigger to record/feature vector extractor
        input feature_done,//done signal from feature vector
  input training_enable,//training_enable switch
  input [3:0] training_select,//training select choices
  input [25:0] DTW_score1,//scores for the 9 DTWS
        input [25:0] DTW_score2,
        input [25:0] DTW_score3,
        input [25:0] DTW_score4,
        input [25:0] DTW_score5,
        input [25:0] DTW_score6,
        input [25:0] DTW_score7,
        input [25:0] DTW_score8,
        input [25:0] DTW_score9,
  input DTW_done1, //done and start for the 9 DTWs
        output reg DTW_start1,
        input DTW_done2,
        output reg DTW_start2,
        input DTW_done3,
        output reg DTW_start3,
        input DTW_done4,
        output reg DTW_start4,
        input DTW_done5,
        output reg DTW_start5,
        input DTW_done6,
        output reg DTW_start6,
        input DTW_done7,
        output reg DTW_start7,
        input DTW_done8,
        output reg DTW_start8,
        input DTW_done9,
        output reg DTW_start9,
  output reg DTW_train1,//NONE  //Training signals for the 9 DTWs
        output reg DTW_train2,//RED
        output reg DTW_train3,//BLUE
        output reg DTW_train4,//BLACK
        output reg DTW_train5,//LEFT
        output reg DTW_train6,//RIGHT
        output reg DTW_train7,//GO
        output reg DTW_train8,//STOP
        output reg DTW_train9,//NOISE no clear word?
  output reg [3:0] command, //output command
        output reg done //done output
  );

        //STATES
        parameter WAIT = 2'b00;
        parameter RECORD = 2'b01;
        parameter DTW = 2'b10;
        parameter COMPARE = 2'b11;


        reg [1:0] state; //overall state
        reg [1:0] compare_state;//state in COMPARE section

        wire all_done;
        reg [2:0] min12, min34, min56, min78,min_next1, min_next2, min; //compare storage registers
        reg [24:0] min12_value, min34_value, min56_value, min78_value, min_nextvalue1, min_nextvalue2, min_value;//compare value
registers
```

```
//is true only once all 9 DTW engines are complete
assign all_done = (DTW_done1 & DTW_done2 & DTW_done3
        & DTW_done4 & DTW_done5 & DTW_done6
        & DTW_done7 & DTW_done8 & DTW_done9);


always @(posedge clock) begin
        if(reset) begin // On reset clear all the train switches and starts
                state <= WAIT;
          command <= 0;
                feature_start <= 0;
                done <= 0;
          DTW_train1 <= 0;
                DTW_start1 <= 0;
          DTW_train2 <= 0;
                DTW_start2 <= 0;
          DTW_train3 <= 0;
                DTW_start3 <= 0;
          DTW_train4 <= 0;
                DTW_start4 <= 0;
          DTW_train5 <= 0;
                DTW_start5 <= 0;
          DTW_train6 <= 0;
                DTW_start6 <= 0;
          DTW_train7 <= 0;
                DTW_start7 <= 0;
          DTW_train8 <= 0;
                DTW_start8 <= 0;
          DTW_train9 <= 0;
                DTW_start9 <= 0;
        end
        else begin
                case (state)
                WAIT: begin //WAIT till talk button is pressed, then go to RECORD and start feature vector extraction
                        DTW_train1 <= 0;
                        DTW_train2 <= 0;
                        DTW_train3 <= 0;
                        DTW_train4 <= 0;
                        DTW_train5 <= 0;
                        DTW_train6 <= 0;
                        DTW_train7 <= 0;
                        DTW_train8 <= 0;
                        DTW_train9 <= 0;
                        if(talk) begin
                                state <= RECORD;
                                feature_start <= 1;
                                done <= 0;
                        end
                end
                RECORD: begin
                        //wait till Feature vector is generated , then if training activate only the respective DTW engine
                        if(feature_done) begin
                                if(training_enable) begin
                                        case(training_select)
                                                4'b0000: begin  //NOISE
                                                        DTW_train9 <= 1;
                                                        DTW_start9 <= 1;
                                                        state <= DTW;
                                                        end
                                                4'b0100: begin //NONE
                                                        DTW_train1 <= 1;
                                                        DTW_start1 <= 1;
                                                        state <= DTW;
                                                end
                                                4'b0101: begin //RED
                                                        DTW_train2 <= 1;
```

```
                                        DTW_start2 <= 1;
                                        state <= DTW;
                                        end
                                4'b0110: begin //BLUE
                                        DTW_train3 <= 1;
                                        DTW_start3 <= 1;
                                        state <= DTW;
                                        end
                                4'b0111: begin //BLACK
                                        DTW_train4 <= 1;
                                        DTW_start4 <= 1;
                                        state <= DTW;
                                        end
                                4'b1000: begin //LEFT
                                        DTW_train5 <= 1;
                                        DTW_start5 <= 1;
                                        state <= DTW;
                                        end
                                4'b1001: begin //RIGHT
                                        DTW_train6 <= 1;
                                        DTW_start6 <= 1;
                                        state <= DTW;
                                        end
                                4'b1010: begin //GO
                                        DTW_train7 <= 1;
                                        DTW_start7 <= 1;
                                        state <= DTW;
                                        end
                                4'b1011: begin //STOP
                                        DTW_train8 <= 1;
                                        DTW_start8 <= 1;
                                        state <= DTW;
                                        end
                                default: begin //DON'T TRAIN
                                        DTW_train1 <= 0;
                                        DTW_train2 <= 0;
                                        DTW_train3 <= 0;
                                        DTW_train4 <= 0;
                                        DTW_train5 <= 0;
                                        DTW_train6 <= 0;
                                        DTW_train7 <= 0;
                                        DTW_train8 <= 0;
                                        DTW_train9 <= 0;
                                        state <= WAIT;
                                        end
                                endcase
                        end
                else begin //if not training all the DTWs are activated
                        DTW_start1 <= 1;
                        DTW_start2 <= 1;
                        DTW_start3 <= 1;
                        DTW_start4 <= 1;
                        DTW_start5 <= 1;
                        DTW_start6 <= 1;
                        DTW_start7 <= 1;
                        DTW_start8 <= 1;
                        DTW_start9 <= 1;
                        state <= DTW;
                end
        end
end
DTW: begin //Wait till all the DTWs are done
        if(training_enable) begin
                if(all_done) state <= WAIT;
                end
        else if(all_done) begin
```

```
                state <= COMPARE;
                compare_state <= 2'd0;
                end
        end
COMPARE:begin //compare the scores to find the score with the lowest value
        case(compare_state)
        2'd0: begin //compare level 1: quarter finals
                min12 <= (DTW_score1 < DTW_score2)? 0: 1;
                min12_value <= (DTW_score1 < DTW_score2)? DTW_score1: DTW_score2;

                min34 <= (DTW_score3 < DTW_score4)? 2: 3;
                min34_value <= (DTW_score3 < DTW_score4)? DTW_score3: DTW_score4;

                min56 <= (DTW_score5 < DTW_score6)? 4: 5;
                min56_value <= (DTW_score5 < DTW_score6)? DTW_score5: DTW_score6;

                min78 <= (DTW_score7 < DTW_score8)? 6: 7;
                min78_value <= (DTW_score7 < DTW_score8)? DTW_score7: DTW_score8;

                compare_state <= 2'd1;
                end
        2'd1: begin//compare level 2: semi finals
                min_next1 <= (min12_value < min34_value)? min12: min34;
                min_nextvalue1 <= (min12_value < min34_value)? min12_value: min34_value;

                min_next2 <= (min56_value < min78_value)? min56: min78;
                min_nextvalue2 <= (min56_value < min78_value)? min56_value: min78_value;

                compare_state <= 2'd2;
                end
        2'd2:begin//compare level 3: finals
                min <= (min_nextvalue1 < min_nextvalue2)? min_next1: min_next2;
                min_value <= (min_nextvalue1 < min_nextvalue2)? min_nextvalue1: min_nextvalue2;

                compare_state <= 2'd3;
                end
        2'd3: begin//compare level 4: compare to Noise and send out command
                if( min_value < DTW_score9) begin
                        case(min) //MY LIST OF command values, change if you think neccesary
                                4'd0: command <= 4'b0100;//NONE
                                4'd1: command <= 4'b0101;//RED
                                4'd2: command <= 4'b0110;//BLUE
                                4'd3: command <= 4'b0111;//BLACK
                                4'd4: command <= 4'b1000;//LEFT
                                4'd5: command <= 4'b1001;//RIGHT
                                4'd6: command <= 4'b1010;//GO
                                4'd7: command <= 4'b1011;//STOP
                                default: command <= 4'b0000; //NO COMMAND
                        endcase
                end // if
                else command <= 4'b0000;
                done <= 1;
                state <= WAIT;
                compare_state <= 0;
                end
        default: begin
                compare_state <= 0;
                end
        endcase
end
default: begin
        state <= WAIT;
        end
endcase
// Each start is only active for 1 clock cycle
if(DTW_start1) DTW_start1 <= 0;
```

```
                                        if(DTW_start2) DTW_start2 <= 0;
                                        if(DTW_start3) DTW_start3 <= 0;
                                        if(DTW_start4) DTW_start4 <= 0;
                                        if(DTW_start5) DTW_start5 <= 0;
                                        if(DTW_start6) DTW_start6 <= 0;
                                        if(DTW_start7) DTW_start7 <= 0;
                                        if(DTW_start8) DTW_start8 <= 0;
                                        if(DTW_start9) DTW_start9 <= 0;
                                        if(done) begin //done is only active for 1 cycle.
                                                done <= 0;
                                                command <= 4'b0000;

                                end // else
  end // always



endmodule
```

Appendix K – voice_main Verilog

```
  //////////////////////////////////////////////////////////////////////////////
  //
  // Reset Generation
  //
  // A shift register primitive is used to generate an active-high reset
  // signal that remains high for 16 clock cycles after configuration finishes
  // and the FPGA's internal clocks begin toggling.
  //
  //////////////////////////////////////////////////////////////////////////////
  wire power_on_reset;
  SRL16 #(.INIT(16'hFFFF)) reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(power_on_reset),
                          .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));

  // button 3 is user reset
  wire reset,user_reset;
  debounce db1(power_on_reset, clock_27mhz, ~button3, user_reset);
  assign reset = user_reset | power_on_reset;

  wire [7:0] from_ac97_data, to_ac97_data;
  wire ready;

  // Alias main system clock for easier use below
  wire clock;
  assign clock = clock_27mhz;

  // allow user to adjust volume
  wire vup,vdown;
  reg old_vup,old_vdown;
  debounce bup(.reset(reset),.clock(clock_27mhz),.noisy(~button_up),.clean(vup));
  debounce bdown(.reset(reset),.clock(clock_27mhz),.noisy(~button_down),.clean(vdown));
  reg [4:0] volume;
  always @ (posedge clock_27mhz) begin
    if (reset) volume <= 5'd8;
    else begin
              if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
              if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
    end
    old_vup <= vup;
    old_vdown <= vdown;
  end

  // AC97 driver
  lab5audio a(clock_27mhz, reset, volume, from_ac97_data, to_ac97_data, ready,
                    audio_reset_b, ac97_sdata_out, ac97_sdata_in,
```

```
                ac97_synch, ac97_bit_clock);

  // push ENTER button to record, release to playback
  wire playback;
  debounce benter_db(.reset(reset),.clock(clock_27mhz),.noisy(button_enter),.clean(playback));

/*
  // debouncer for enter (start record)
  wire button_enter_db;
  debounce benter_db(.reset(reset),.clock(clock_27mhz),.noisy(button_enter),.clean(button_enter_db));

  // debouncer for button 3 (start play)
  wire button3_db;
  debounce b3_db(.reset(reset),.clock(clock_27mhz),.noisy(button3),.clean(button3_db));
*/

  // switch 0 up for filtering, down for no filtering
  wire filter;
  debounce sw0(.reset(reset),.clock(clock_27mhz),.noisy(switch[0]),.clean(filter));

  // light up LEDs when recording, show volume during playback.
  // led is active low
  // assign led = playback ? ~{filter,2'b00, volume} : ~{filter,7'hFF};
  wire fft_status;
  wire [3:0] fft_state;
  assign led[7] = ~fft_status;
  assign led[6:3] = ~fft_state;
  assign led[2:0] = {3{1'b1}};

  // record module
  wire [7:0] mem_out;
  wire feature_start;
  wire [13:0] fft_mem_addr;
  recorder r(.clock(clock_27mhz), .reset(reset), .ready(ready),
        .playback(playback), .filter(filter),
        .from_ac97_data(from_ac97_data),.to_ac97_data(to_ac97_data),
        .mem_out(mem_out),
        .start_fft(feature_start),
        .fft_addr(fft_mem_addr), .fft_override(fft_status));

  // output useful things to the logic analyzer connectors
  assign analyzer1_clock = ac97_bit_clock;
  assign analyzer1_data[0] = audio_reset_b;
  assign analyzer1_data[1] = ac97_sdata_out;
  assign analyzer1_data[2] = ac97_sdata_in;
  assign analyzer1_data[3] = ac97_synch;
  assign analyzer1_data[4] = feature_start;
  assign analyzer1_data[15:5] = 0;

  wire [15:0] debug_out;
  assign analyzer3_clock = clock_27mhz;
  assign analyzer3_data = debug_out;

////////////////////////////////////////////////////////////////////////////////
//
// Voice processing modules
//
////////////////////////////////////////////////////////////////////////////////

// Feature extractor (feature_extract.v)
// DTW scoring module (from dtw_score.v)
// control logic (from logic.v)



  wire feature_done;
```

```
  reg [6:0] chunk_num = 0;


  feature_extract extractor(.clock(clock), .reset(reset), .start(feature_start),
      .done(feature_done), .bram_addr(fft_mem_addr),
      .bram_data(mem_out), .chunk_num(chunk_num),

      .debug_out(debug_out), .status(fft_status), .state(fft_state));

endmodule

//////////////////////////////////////////////////////////////////////////////////
//
// Record/playback
//
//////////////////////////////////////////////////////////////////////////////////

module recorder(
  input wire clock,                // 27mhz system clock
  input wire reset,               // 1 to reset to initial state
  input wire playback,             // 1 for playback, 0 for record
  input wire rec_button,
  input wire play_button,
  input wire ready,               // 1 when AC97 data is available
  input wire filter,              // 1 when using low-pass filter
  input wire [7:0] from_ac97_data, // 8-bit PCM data from mic
  output wire [7:0] to_ac97_data,    // 8-bit PCM data to headphone
  output wire [7:0] mem_out,
  output reg start_fft,
  input [13:0] fft_addr,
  input fft_override
);


  // Internal wires for configuring memory
  wire [13:0] mem_addr;
  reg [13:0] rec_addr;
  reg mem_we;
  wire [7:0] mem_in;
  reg [13:0] highest_addr;

  //assign mem_addr = fft_override ? fft_addr : rec_addr;
  assign mem_addr = rec_addr;

  // 64K x 8 memory for storing samples
  mybram #(.LOGSIZE(14),.WIDTH(8))
    audio_bram(.addr(mem_addr),.clk(clock),.we(mem_we),.din(mem_in),.dout(mem_out));

  // used to detect edges on ready signal
  reg ready_prev;
  reg playback_prev;

  // Used to only take every 3rd audio sample
  reg [1:0] upsample_counter;

  reg enable_record;

/*
  // create pulse to start recording on release of enter
  reg recording;
  wire start_record;
  reg button_enter_db_prev;

  assign start_record = button_enter_db & ~button_enter_db_prev;
```

```
  // create pulse to signal start of playback
  reg playing;
  wire start_play;
  reg button3_db_prev;

  assign start_record = button_enter_db & ~button_enter_db_prev;
*/

////////////////////////////////////////////////////////////////////////////////
  always @ (posedge clock) begin

//////////////////////////////////////////
    // Reset memory address when switching modes; supercedes other actions
    if (playback != playback_prev) begin
      rec_addr <= 14'h0;
      upsample_counter <= 2'h0;

      // if recording, also reset highest address location
      if (~playback) begin
        highest_addr <= 14'h0;
        enable_record <= 1;
      end
    end

//////////////////////////////////////////
    else if (ready && ~ready_prev) begin  // posedge on ready


      // only update outputs every 3 ready cycles (downsample to 16 kHz)
      if (upsample_counter == 2'b00) begin
        if (playback) begin  // playback mode
//            to_ac97_data <= mem_out;   // read sample from memory
          rec_addr <= (rec_addr == highest_addr) ? 14'h0 : rec_addr + 1;
        end // playback mode

        else begin       // record mode
          // stop recording after 16000 samples (1 second)
          if (rec_addr == 14'h3E80) begin
            enable_record <= 0;
            start_fft <= 1;
          end

          if (enable_record) begin
            // pulse write enable high (returned to low outside loop)
            if (~mem_we) begin
              mem_we <= 1;
            end
//            mem_in <= from_ac97_data;    // taken care of by combo logic
            rec_addr <= rec_addr + 1;    // after recording at FFFF, will stop recording
            highest_addr <= rec_addr;

          end // if enable_record


        end // record mode

      end  // upsample_counter



      // state machine for upsample_counter to cycle every 3 clocks
      case(upsample_counter)
        2'b00: upsample_counter <= 2'b01;
        2'b01: upsample_counter <= 2'b10;
        2'b10: upsample_counter <= 2'b00;
```

```
        endcase

    end   // posedge ready

    // make sure that write enable only pulsed for 1 cycle
    if (mem_we) begin
      mem_we <= 0;
    end

    // pulse feature extractor start for 1 cycle
    if (start_fft)
      start_fft <= 0;

    // always update previous signals
    ready_prev <= ready;
    playback_prev <= playback;
//     button_enter_db_prev <= button_enter_db;
//     button3_db_prev <= button3_db;

  end // always

////////////////////////////////////////////////////////////////////////////////
  // Audio filtering

  wire [7:0] filter_in;
  wire [17:0] filter_out;

  fir31 filter_inst(.clock(clock), .ready(ready), .x(filter_in), .y(filter_out));

  // Combinational logic for implementing filtering
  assign mem_in = filter ? filter_out[17:10] : from_ac97_data;
  assign filter_in = playback ? mem_out : from_ac97_data;
  assign to_ac97_data = playback ? (filter ? filter_out[17:10] : mem_out) : mem_in;

// TODO: multiplex BRAM address and WE lines to let FFT read out



endmodule
```

Appendix L – dtw_score Verilog

```
module dtw_score(clock,reset, in, train_enable, score, start, done);

  // possible modification: take in data sequentially (since not all needs to be processed at once); would need next_data line, or something
similar

  input clock; //system clock
            input reset; //system reset

  // Sequences of feature vectors: 40 frames, 12 features, 8 bits each
  input [7:0] in;
  input train_enable; //when true sequence is stored rather than compared
  output reg [25:0] score; //final DTW score

  // receive high pulse to start, emit high pulse on completion
  input start;
  output reg done;

            //STATES
            parameter WAIT = 2'b00;
            parameter DATA_IN = 2'b01;
```

```
parameter ADD = 2'b10;
parameter TRAIN = 2'b11;

parameter FRAMES = 50;

//Counting Registers
reg [3:0] j;
reg [5:0] i;

reg [1:0] state;
reg [7:0] test[FRAMES - 1:0][11:0]; //data just recorded
reg [25:0] tmp[FRAMES - 1:0][11:0]; //tmp storage of compared square values

//BEGIN STATE MACHINE
always @(posedge clock) begin
	if(reset) begin //on reset clear out registers
		for(i=0;i<FRAMES;i=i+1) begin
			for(j=0; j<12; j=j+1) begin
				test[i][j] <= 8'b0;
				tmp[i][j] <= 26'b0;
				end
			end
		score <= 0;
		state <= WAIT;
		done <= 0;
	end
	else begin
		case (state)
		WAIT: begin //WAIT until start signal then if train is enabled go to TRAIN, otherwise go to DATA_IN
			if(train_enable && start)begin
				state <= TRAIN;
				i<=0;
			end
			else if(start) begin
				state <= DATA_IN;
				score <= 0;
				done <= 0;
				i <= 0;
			end
			else state <= WAIT;
			end
		TRAIN:begin//Store the incoming 40*12 bytes in test[i][j] then go back to WAIT
				if(i < FRAMES) begin
					if(j < 12) begin
						test[i][j] <= in;
						j <= j+1;
					end
					else begin
						j<= 0;
						i <= i +1;
					end
				end
				else begin
					state <= WAIT;
					done <= 1;
				end
			end
		DATA_IN: begin //Square the difference between test[i][j] and the input (EUCLIDIEAN DISTANCE) once done
for all 480 bytes go to ADD
				if(i < FRAMES) begin
					if(j < 12) begin
						tmp[i][j] <= (in>test[i][j])? ((in - test[i][j])*(in - test[i][j])):((test[i][j]
- in)*(test[i][j] - in));

						j <= j+1;
						end
					else begin
```

```
                                                j <= 0;
                                                i <= i + 1;
                                                end
                                        end
                                else begin
                                        state <= ADD;
                                        i<= 0;
                                end
                        end
                ADD: begin //ADD together the 480 square bytes to get a full DTW score, then return the value and go back to
WAIT
                                        if(i < FRAMES) begin
                                                if(j < 12) begin
                                                        score <= score + tmp[i][j];
                                                        j <= j + 1;
                                                        end
                                                else begin
                                                        j <= 0;
                                                        i <= i + 1;
                                                        end
                                        end
                                        else begin
                                                state <= WAIT;
                                                done <= 1;
                                                i<= 0;
                                        end
                                end
                default begin
                        state <= WAIT;
                        end
                endcase
        end
    end
endmodule
```