ECE 5775: Final Project Report

Voice Recognition using FPGA-based Signal Processing and Neural Networks

Taylor Pritchard, Shiva Rajagopal, Ian Vermeulen (tjp79, svr24, iyv2)

revision: 2015-12-15-04-18

1. Introduction

The objective of this project was to design a system that could reliably identify a limited set of vocal commands among samples of human speech, using signal processing and machine learning techniques. The system is implemented on an FPGA, using a high-level synthesis (HLS) work flow. The set of commands recognized includes `GO FORWARD`, `TURN LEFT`, `TURN RIGHT` and `REVERSE`. This limited set of vocabulary was chosen for for the purpose of controlling a car. Taking into consideration some key characteristics of human speech, our system made use of a series of signal processing techniques to generate a characteristic "fingerprint" of each word based on when certain syllables were uttered in order. Normalizing the input sound enabled the generation of similar fingerprints from the same word. We implemented a neural network which we could train offline, thereby minimizing the need for data on the FPGA. Finally, we were able to use the Xilinx SDK to activate the necessary peripherals and connections to make this a fully embedded system. Our system was able to achieve an accuracy rate of 90% on a small test set of 20 sounds, which was primarily due to the neural network's ability to classify similar, but different samples of audio. Our high level synthesis gave us a resource usage of about half of the Zedboard's resources, leaving us plenty of room to add features to the system if we desired. The system was able to recognize speech in less than a second after recording, making it very close to a real time system. Even without optimizations, our project was able to meet all of our specifications, and exceeded even our own expectations for its accuracy and speed.

2. Techniques

Our project is a combination of hardware and software, and makes use of many algorithms. From a high level, the signal processing portion uses a series of windowed samples of the sound to create a fingerprint that is then fed into the neural network classifier. The neural network is trained offline to not only save space on the FPGA, but also to ensure that no external computing source is needed once the FPGA is programmed in order to classify the code. On the hardware side, we make use of the FPGA's line-in interface and the Xilinx IPs necessary to transfer data into our synthesized hardware for classification. We explore each of these portions in detail here.

2.1 Fingerprinting Sound

This project creates a fingerprint for a given sound, according to the diagram in Fig. 1. This is a complex

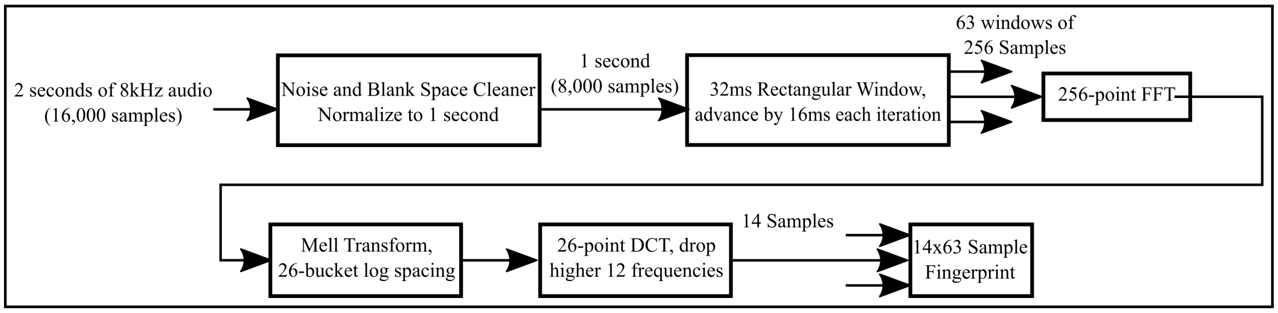

2 seconds of 8kHz audio (16,000 samples)
Noise and Blank Space Cleaner Normalize to 1 second
1 second (8,000 samples)
32ms Rectangular Window, advance by 16ms each iteration
63 windows of 256 Samples
256-point FFT
Mell Transform, 26-bucket log spacing
26-point DCT, drop higher 12 frequencies
14 Samples
14x63 Sample Fingerprint

Figure 1: Flow Diagram of Sound Fingerprinting

diagram, but we explain each step of the flow here. The first step is cleaning the sound and removing blank space. From the 2 seconds of sound we record, the words that we say can easily fit into one second, but we do not know when the speech started, nor do we know how much space is between each word. Therefore, we first send the 2 second sample into a preprocessor which normalizes the data for fingerprinting. It does this by first removing all values that are under a threshold, and then normalizing the beginning of the speech to the beginning of the trimmed clip. Once this is done, we normalize it to 1 second by inserting blank space at the end. The next step in the signal processing is the windowing function, which processes the signal in overlapping chunks of 32ms each, advancing by 16ms each time. This leads to 63 different windows, which are all processed the same way, based on the information found in [1].

The first step in the signal processing is running a length-256 Fast Fourier Transform on the chunk. An FFT is a fast way of performing a Discrete Fourier Transform, which is a method of finding the power spectrum of a signal over discretized chunks. The DFT operates using equation 1.

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-\frac{jkn2\pi}{N}} \tag{1}$$

For our project, since we have 256 samples, we use a 256-sample DFT. Although usually the top half of DFT values should not be used, as they contain negative frequency content, we keep these values since we do not particularly care about what this content means. The neural net will simply train on any data we give it, so as long as we use the same data each time, we will be okay. Additionally, this content will be shrunken when we apply the Discrete Cosine Transform later. The Fast Fourier Transform accomplishes the same result as the DFT, but it uses properties of the data to parallelize some of the calculations, and give us a faster result. We use an implementation of the FFT from [2], rather than using the Xilinx IP FFT, as we did not have enough time to figure out how to integrate the Xilinx FFT with our code. Our FFT still runs quickly, and gives us 256 values for the frequency content of the 32ms chunk. Although FFT algorithms are usually recursive, we make use of one with nested for-loops that accomplishes the same task. make use of one with nested for-loops that accomplishes the same task. This will give us a length-256 power spectrum of the chunk, which is more than enough resolution for our purposes.

Once we have this, we must then separate the frequency content based on human speech. In this case, since human speech frequencies are separated logarithmically, we use a Mell transform to find 26 logarithmically spaced buckets that we can combine our 256 frequency values into. For the Mell transform, we take the lowest and highest frequencies that we need, which in this case, is 300Hz and 8000Hz, respectively. Using the equations given by [1], we can find the forward Mell transform by Eqn. 2 and the inverse by Eqn. 3.

$$M(f) = 1125ln(1 + \frac{f}{700}) \tag{2}$$

$$M^{-1}(m) = 700(exp(\frac{m}{1125}) - 1) \tag{3}$$

We use the highest and lowest frequencies that we want in terms of Mels with (2). Once we have this, we can find 26 equally spaced values in Mels, and then convert from Mels back to regular frequency, using (3). We combine the defined buckets and take the logarithm of the combined amplitude to reduce the irrelevant frequency content.

After we have these 26 buckets, we want to somewhat consolidate the overlapping data through these transforms. In order to do this, we use a Discrete Cosine Transform on all 26 buckets. The DCT can characterize a set of data as a sum of cosine functions at different frequencies. The DCT can decorrelate any overlaps that we have while summing the Mell filterbanks together, which means that we will be able to have a higher quality fingerprint. The DCT we use has the form shown in Eqn. 4

$$X[k] = \sum_{n=0}^{N-1} x[n]cos\left[\frac{\pi}{N}(n + \frac{1}{2})k\right] \tag{4}$$

Once this is done, the highest frequency buckets actually only contain noise that is not relevant to our fingerprint. Thus, we drop the top 12 results from the DCT. This finally gives us our fingerprint. While the later windows in the sample are usually primarily zeros, we can't know how long the voice sample will last, so this gives us a safe margin to ensure we get all of the speech. In addition, people generally say words at similar speeds, so these zeros in the later windows can actually be good for differentiating valid words form invalid words.

Once we perform this series of processing on all 63 chunks of data, we have a 14x63 fingerprint of the sound sample, bringing our initial 16,000 points of data to a size-882 fingerprint, which is a large amount of compression.

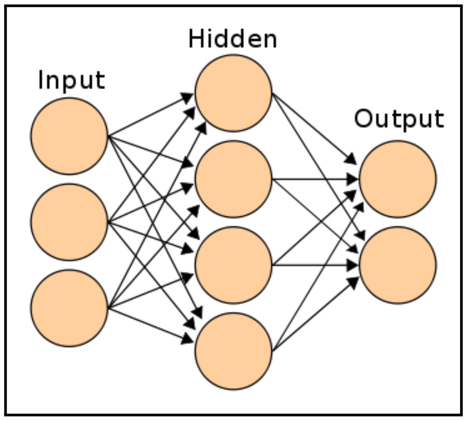


Figure 2: Three Layer Artificial Neural Network

2.2 Neural Network

We will be using a three-layer artificial neural network to recognize the speech. This network consists of an input layer, a hidden layer, and an output layer. The output from the fingerprint will be fed into the input layer, and each output neuron will correspond to a different classification. The hidden layer is used to capture different features in the fingerprints. Each layer consists of multiple "neurons" and every neuron has connections to every other neuron in adjacent layers. Each of these connections has a weight value associated with it.

The network is trained by feeding many example inputs through the network. This feed-forward process consists of first inputting all the fingerprint values into separate input neurons. Then the value of each hidden neuron is calculated by summing the products of all the input neurons and the edge weight between the input and hidden neurons. Next this value is input into the sigmoid function shown below to give the neuron a value between 1 and 0. Then the same process is performed to determine the output neuron values.

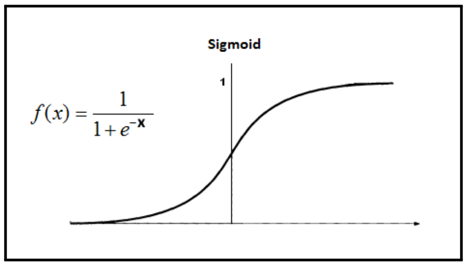


Figure 3: Sigmoid Function for Determining Neuron Values

The output values are then compared with the desired values and adjustments to the edge weights are propagated back through the network. This training process is called back propagation. After a sufficient amount of training, the weights can be saved, allowing the network to be trained before being used to classify speech samples. The classification process consists of feeding the current sample into the network and examining which output neuron ends up with the highest value.

3. Implementation

The final version of the system was implemented on a ZedBoard proof-of-concept development kit. The ZedBoard features a Xilinx Zynq-7000 All Programmable SoC, which offers software-programmability and hardware-programability in a single chip. The functionality of the system on the Zynq is split between the Processing System (PS) and the Programmable Logic (PL). For our voice recognition system the PS operates as the controller for the other components of the system, which are implemented on the PL.

Implementation of our system followed a multi-step encapsulation-based workflow, illustrated by Fig. 4. First the code for the signal processing and machine learning techniques used were implemented in C++. This code was then synthesized into logic, generated using High-Level Synthesis techniques. Next, the synthesized logic was incorporated into the rest of the system using the Vivado system integrator tool. The system integrator tool was used to compose all of the logic to be implemented on the programmable logic fabric. Additionally, the tool was used to generate a board support package, containing the necessary drivers used by the control software to communicate with the rest of the system. Finally, the control software is written in the SDK, and the SDK is used to program both the PS and the PL for a fully functional system.

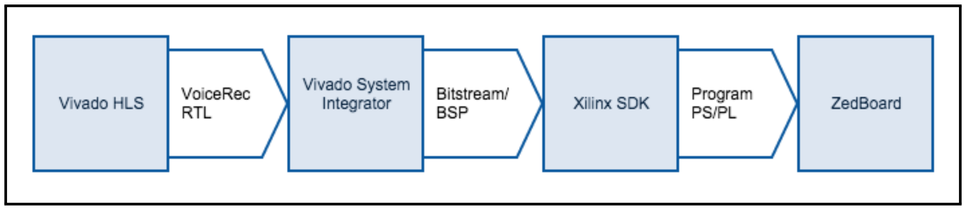


Figure 4: Implementation Process Flow Chart

3.1 C++ Code and High Level Synthesis

The software portion of our code for this project was written entirely in C and C++. Before synthesizing the C++ into hardware, AXI-4 lite interface directives were added to the top level function of the Voice Recognition system. This provides an interface for control and data exchange between the HLS block and the rest of the system. We had to take care to ensure that our code would play nicely with the High-Level Synthesis, however, which we accomplished through some clever tricks. The signal processing piece of the software was initially implemented in MATLAB to ensure proper functionality before continuing translating to C. When porting this MATLAB code to C, we had to take care to not follow our instinct to optimize the code using variable-bound loops. While this would make for a more optimized software runtime, it would not play kindly with hardware, which is traditionally done with fixed latency loops. While this meant a potentially worse maximum runtime, we would be able to tell what our latency was simply through high-level synthesis, and not be required to go through cosimulation.

Additionally, MATLAB has many in-built mathematical functions that might not be as efficient when written in C. We had the advantage in this project of knowing exactly which trigonometric values we would need for both the FFT and DCT, and therefore were able to use constant arrays to replace these values, rather than forcing the hardware to calculate these each time. Additionally, we were able to implement a fast logarithm function based on [3], thereby making the need for the math.h library obsolete. This would make our high-level synthesis easier, as well as lead more predictable program behavior. If we were required to calculate arbitrary trigonometric values, we could have made use of the CORDIC algorithm implemented earlier this semester, but we decided against this since it was not within the scope of our project.

The final implementation trick we used for the high-level synthesis was modifying the neural network code to only contain the classification methods. While it might be nice to be able to train the system on the FPGA itself, this code would not only be difficult to synthesize, but would also add a lot of complexity to our system. Thus, it made the most sense to train the system offline and only synthesize the hardware with the neural network weights and the classification methods. This also gave us the power to keep very high accuracy while training, and test how much this was compromised by adjusting datatypes.

3.2 PL and The Vivado System Integration Tool

The Vivado System Integration tool was used to incorporate the HLS-generated logic into the rest of the system, as illustrated in Fig. 5. The other components of the system were IP provided by Xilinx, including the audio controller, which was responsible for handling the input/output audio stream, the GPIO controller, responsible for handling user control, and the AXI interconnect. PS-PL communication is carried out through the use of an memory-mapped AXI bus. The processing system has an AXI master port, while all of the other components in the system have AXI slave ports. This includes the HLS-generated voice recognition logic, as mentioned earlier. Utilizing a standard memory-mapped protocol across modules gives the PS one consistent view of the rest of the system and allows the components of the system to easily be connected to each other to facilitate the exchange of control signals and data. The tool was also used for other configurations of the PL and I/O, such as the incorporation of constraint declarations, setting the clock speed, and establishing the memory map for the peripheral control registers. After integration was complete, the tool was used to generate a bitstream file, to be used in programming the PL, as well as a board support package, or BSP, providing drivers to be incorporated into the software for the processing system.

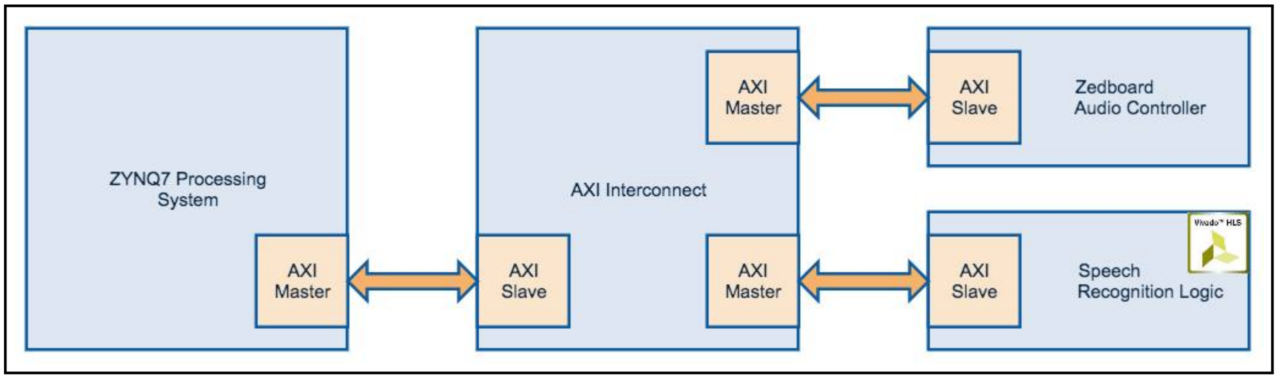
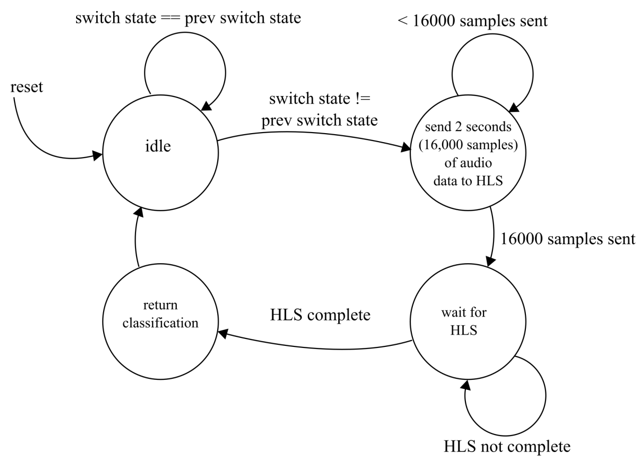


Figure 5: Block Diagram For Processor-based Audio System Integration

3.3 PS and The Xilinx SDK


switch state == prev switch state
reset
idle
switch state != prev switch state
< 16000 samples sent
send 2 seconds (16,000 samples) of audio data to HLS
16000 samples sent
wait for HLS
HLS not complete
HLS complete
return classification

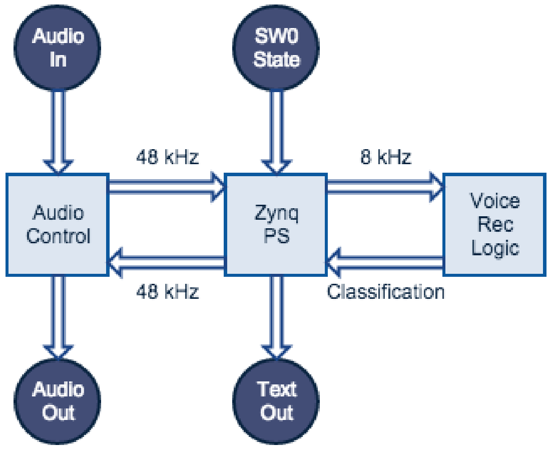


Figure 6: Control and Datapath for Voice Recognition System

The final stage of the development process was performed using the Xilinx SDK. The bitstream and BSP produced by the system integration tool were imported into the SDK, and the SDK is used for writing software for the PS, and for the programming of both the PL and the PS. All code for the PS was written in C, and we can model the flow of data by the right-side figure in Fig. 6.

The PS of the Zynq is a dual core ARM Cortex-A9, and serves as the controller for the rest of the system. The drivers contained in the BSP provided a level of abstraction for easily interacting with the PL components via the control software to be implemented on the PS. The software state machine is detailed in Fig. 6. BSP functions were leveraged for retrieving audio data from the audio controller, determining the switch state, sending/receiving data from the HLS voice recognition block, and controlling the start and end of the classification process. The PS receives audio samples from the audio controller at a sample rate of 48 kHz. In its idle state, the PS continually polls the state of switch SW0 until it finds that it has changed. For 2 seconds after it detects a change in the switch-state, it down-samples the audio stream to 8 kHz and sends it to the voice recognition logic. After 16,000 samples are sent to the voice rec module, the PS waits for a classification. Throughout the processes information about the state of operation, including the command classification, is communicated over UART to a console on the host PC.

4. Evaluation

For our baseline implementation, we tested our code in C running on the Zedboard, and compared it to our alternative design, which would execute using the FPGA hardware as well. Both of these would use the same signal processing and neural network code, but our experiment would compare how much we could speed up our implementation by using a dedicated piece of hardware. We also analyze our hardware's resource usage on the FPGA to evaluate its feasiblity as a hardware module.

4.1 Analysis

4.1.1 Qualitative Analysis

Qualitatively, our system was mostly successful. The microphone interface to the turned out to be our most unpredictable component, as it sent in 48kHz 24-bit audio, and our system was based on 8kHz 8-bit audio. Therefore, we had no guarantee that the sound values coming in would be similar to those we received from our training through the MATLAB interface, and it seemed that our recognition system would not work reliably with the FPGA microphone interface. Had we trained the system based on samples from the FPGA microphone itself, we might have achieved a better result.

While the FPGA microphone was not extremely reliable, we were able to verify that the hardware's functionality was equivalent to that of the software implementation. We used a test set of 5 recordings for each command from header files, some of which the system recognized incorrectly. The hardware system behaved in the exact same manner as the software in all 20 test cases, verifying that our synthesis to hardware was successful, and that our system integration was successful. The system was extremely easy to use. Upon power-up, all the user had to do was flip a switch to make the FPGA start recording, and wait for the response. We returned the value through the UART connection for our own debugging purposes, rather than using a signal on the FPGA itself. This improved human readability, but also showed us that should this project be part of a larger system on the FPGA, we could interface with it very easily through our ARM core code.

4.1.2 Quantitative Analysis

The process of generating training and test data was a relatively easy process, but we chose 20 samples of Taylor's speech, 5 of each phrase, for testing. Our results were very good, achieving 100% recognition on everything but `GO FORWARD`. We expect this is because of uneven training, as we did not have a lot of time to train the system further since we were trying to make it work on the FPGA. However, this means that we have achieved over 90% accuracy with our training set. One important decision we made during development was to use the word `REVERSE` instead of `GO BACK`, for two reasons. First of all, the first word "GO" could possibly lead the system to confuse `GO FORWARD` and `GO BACK`, since the first time frames in the fingerprint would contain similar data. Secondly, the word "BACK" has two sounds, the "B" and the "K" that do not generate very rich and unique frequency content. For these two reasons, our neural network was not recognizing `GO BACK` well at all. Thus, we decided to switch to `REVERSE`, which gave us much better results.

4.2 Latency

Our latency estimate from the high-level synthesis gave us some idea of how long the system would take to recognize our commands in hardware. According to the Vivado HLS synthesis report, the latency estimates for the system were as shown in Table 1. Our synthesis predicts that we will be able to meet our target clock period of 10ns, so we expected to, and were able to run this system at 100MHz easily. At this clock speed, Vivado HLS pre-

Module	Latency (cyc)	Time (ms)
ARM-sw		68.5
FPGA-all	41614325	416.14
Preprocessor	232085	2.3
Fingerprint	674308	6.7
Classifier	249327	2.5

Table 1: HLS Module Latency in cycles

dicted an average case runtime of around 400ms. Although we did not have time to actually measure our hardware runtime speed using Xilinx SDK, this runtime was found to be somewhat accurate. From the end of the recording until the system gave a response almost felt like a real-time response.

We compared this to running our software on a Zedboard running Xillinux, without any hardware acceleration, which achieved roughly 68.5 ms runtime. This was consistently found to be faster than the hardware, but this is not a bad thing. The advantage of having dedicated hardware is that we always know what latency we will be facing, which may not be the case with hardware. Additionally, having a hardware implementation lets us integrate our system with other components and offload this potentially costly operation of voice recognition to a place other than the main processor. While we could have possibly brought this hardware latency down through use of pipelining and using accelerated versions of code such as Xilinx's IP FFT module, we did not have time to try synthesizing this code, as it would have taken a long time to run through attempting to pipeline this module. To pipeline the code, we would need to significantly optimize the code further, especially the fingerprinting module. Looking at Table 1, we can see that each piece of the system has a significant latency. The fingerprint generator is called 63 times in our system, each time taking about 675,000 cycles, which leads to a large delay and makes pipelining difficult. The preprocessor and sound classifier could probably benefit from loop unrolling and array partitioning as well, as both contain a multitude of loops and operate on large sets of data.

4.3 Resource Usage

Looking at our synthesized hardware, we can gather a fair amount of insight about our system from the resource usage. We see from Table 2 that overall, our system took a fair amount of all types of resources, which we expected for such a complicated module. However, in terms of our target Zedboard, this resource usage is not very much. Our usage for BRAMs, DSPs, FFs, and LUTs was 45%, 31%, 12%, and 38%, respectively, of what the Zedboard can handle. This means that in terms of integrating this fingerprint generator with a larger system, such as one that could control some outer system, we have plenty of room to add more logic. Looking at the Table 2 further, we can see a clear characteristic of resource usage for each main part of the synthesized code. The audio preprocessor is the most simple module, as it follows the same behavior for the entire input sample, and therefore can reuse a lot of logic with limited storage needed. With the fingerprint generator, we see a large increase in the resource usage, primarily in the way of DSPs needed, but also for flip-flops and LUTs. This makes sense, as we keep many temporary arrays as we go along, and need to do some fairly complex math along the way. This will take many cycles, of course, and require a wide variety of LUTs to accomodate all of our logic. We see somewhat similar resource usage for our neural network classifier, but the most shocking statistic is the BRAM usage. This is a major increase from the other two, and we do expect this. The main reason for this is that our neural network, while only having three layers, has a large amount of neurons and weights that need to be considered. The storage of all of these variables is the primary reason for such excessive BRAM usage, but our percentage of BRAM usage tells us that we could easily fit in an even larger neural network should our system require it.

Module	BRAM	DSP	FF	LUT
All	**128**	**69**	**13477**	**20312**
Preprocessor	2	0	1161	1949
Fingerprint	7	35	5321	8193
Classifier	69	34	6851	9882

Table 2: HLS Module Resource Usage

5. Project Management

A summary of our tasks and milestones is given in Table 3. For this project, we followed an agile development methodology, which was best for this type of project, as we had to change our plans numerous times. We divided the work in the most parallel way possible, ensuring that each of us could develop our part without need for the other member's contribution. To do this, we first established the necessary interfaces. Ian and Shiva agreed on the interface that the neural network would require from the signal processing, and ensured that these interfaces were met and that the data types were in agreement. Taylor and Shiva agreed upon a

Task	**Members**	**Dates of Task**	**Milestone Completion Date**
Task 1: Project Proposal and Research	All	11/3-11/16	
Task 2: Implement Neural Network in C++	Ian	11/15-12/4	
Task 3: Implement Signal Processing in MATLAB, then C++	Shiva	11/15-12/4	
Task 4: Generate Training and Test Data for Neural Net	Taylor	11/30-12/4	
Milestone 1: Software Implementation			**12/4**
Task 5: Ensure synthesizability of Signal Processing	Shiva	11/24-12/2	
Task 6: Ensure synthesizability of Neural Network	Ian	11/24-12/2	
Milestone 2: Synthesize C++ code with HLS			**12/2**
Task 7: Complete example lab for audio interface on Zedboard	Taylor	11/16-11/23	
Task 8: Generate FPGA Bitstream with Xilinx IP Connections	Taylor	11/30-12/4	
Task 9: Implement ARM controller code and integrate system	Taylor	12/4-12/12	
Milestone 3: Functional Hardware Implementation			**12/12**
Task 10: Evaluate Design on Zedboard Software and Hardware	All	12/12-12/14	
Task 11: Finalize code and report	All	12/12-12/14	
Milestone 4: Project Completion			**12/14**

Table 3: Timeline of Project

hardware interface that could send 2 seconds of 8kHz audio to Shiva's signal processing code, and return a single classification value from Ian's code. Once we had these, however, the modularity of our project made parallel development easy.

As the project progressed, Taylor ran into some issues with the integration, including buggy software, some unclear instructions, and compatibility issues. Eventually he was able to figure out the interface and generate a functional result for the FPGA implementation through consulting documentation and online forums.

Ian encountered a few problems while constructing the neural network. The basic construction was straightforward, but there were several constants that had to be tuned experimentally to get accurate and quick training. This took a fair amount of trial and error. The main issues came while attempting to synthesize the network. Since the training code used some non-synthesizable constructs, another separate classifier had to be created for synthesis. The original size of the neural network also had to be scaled down in order to fit in hardware. Another issue with the neural network was the volume of training data required to obtain accurate classifications. We settled for a relatively small training set with only one speaker due to time constraints, but with enough time and data the classification set could be expanded in size and include multiple speakers. This would likely increase the accuracy and general usability of the system.

Shiva ran into a variety of issues through the development process, mostly relating to the conversion of the signal processing code from MATLAB to C++ code. These primarily related to removing data-dependent loops, as well as memory allocation issues and ensuring that the execution of the two languages was equivalent. Another major challenge during the project was attempting to implement HLS optimizations into the project. The signal processing piece of this project in theory lent itself very well to pipelining, as we run the same set of operations on different pieces of an array as the majority of our signal processing. However, attempts to synthesize a pipelined version of this code took an excessively long time to synthesize with Vivado HLS, and the compile time would've been unacceptable to finish this project in time.

While development was still in progress with the FPGA, we experimented with adjusting bitwidths and checking accuracy. Although time was short, we found that we could replace our double-precision floating point operations with 10:8 fixed-point representation and still achieve the same accuracy. At first glance, this appeared to roughly halve our hardware latency. We did not have time to fully validate this, however, and settled for using single-precision floating point representation in our final code.

6. Conclusion

Although we did not have enough time to fully optimize and expand our set of training data, we still developed a fully functional system to recognize voice commands. We successfully synthesized the signal processing and neural network classification into hardware and integrated these components with user input. Classification had a few issues due to the differences in audio systems used for training and for real-time classification, but this could be easily resolved in the future by using consistent audio hardware for both training and classification. Running our system on a Zedboard achieved low latency and produced a response very quickly. By achieving 90% accuracy with a relatively small training set and designing hardware that runs in effectively real-time to a human, we can declare this a successful experiment For future work, a developer could explore hardware optimizations such as array partitioning, loop unrolling, and pipelining that could be used to further reduce the system latency. Additionally, future work could involve use of more Xilinx IP, such as the FFT function that is available to the Zedboard. It also might be worth exploring whether we can modify the hardware such that the ARM core would not even be necessary, and we could route the audio directly to the FPGA hardware block. However, the fact that we were able to train with such accuracy and achieve such good timing even without these optimizations is a positive for our project. Our project has demonstrated that using signal processing and neural networks on an FPGA is very feasible and reasonable, and implies that embedded applications can strongly benefit from a hardware/software codesign such as this.

References

[1] Lyons, J. Mel Frequency Cepstral Coefficient (MFCC) tutorial. 2012.

[2] Exstrom Laboratories. Digital Signal Processing. 2014.

[3] Mineiro, P. Fast Approximate Logarithm, Exponential, Power, and Inverse Root 2011.

[4] Mikulic, E. Discrete Cosine Transform. 2015.

[5] Anguelov, B. Basic Neural Network Tutorial : C++ Implementation and Source Code. 2008.

[6] Xilinx Inc. Vivado Design Suite User Guide: High Level Synthesis. 2015.

[7] Xilinx Inc. Vivado-Based Workshops: High-Level Synthesis Flow on Zynq using Vivado HLS. 2014.