Image Buddy: Voice-Controlled Slideshow

By:
Cameron Buttazzoni
Shuyi Wu
Shuran Xu

Overview

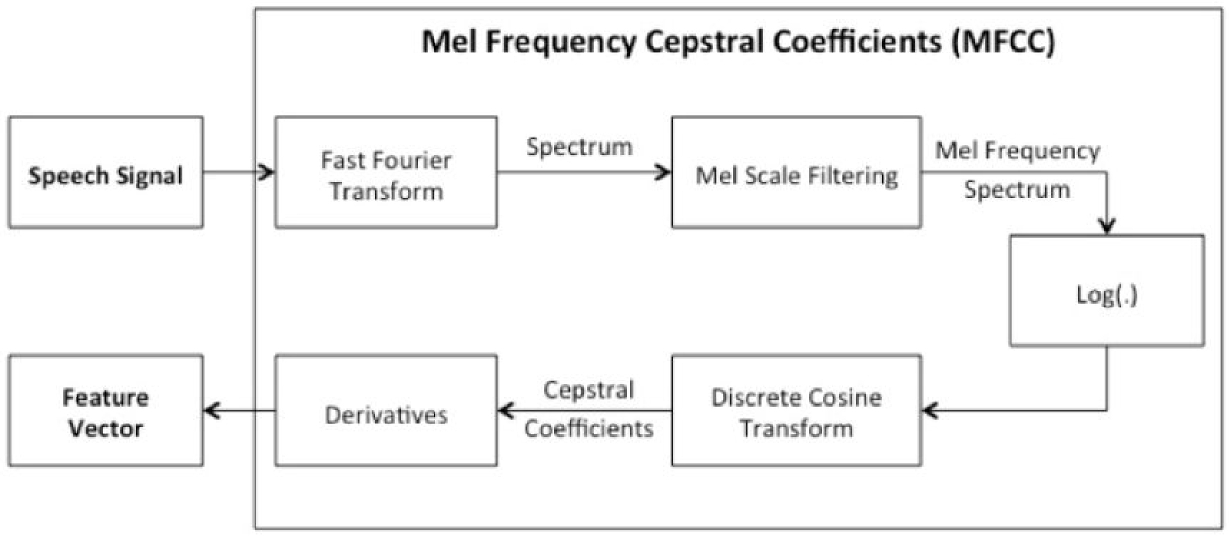
Image buddy is a voice-controlled FPGA project that displays images. The user presses a button on the FPGA and then says one of three words: “alligator”, “baboon”, or “cow”. The FPGA’s built-in microphone records approximately one second of audio data, and then processes it in hardware to extract features using the Mel Frequency Cepstral Coefficients (MFCC) algorithm. The MicroBlaze processor then sends the extracted features to a server over Ethernet. The server uses a pre-trained neural network to classify the speech sample into one of the three animals. An image of the predicted animal is then sent back to the FPGA over Ethernet, where the image is displayed on a PMOD LED screen.

Motivation

The main motivation for this project was learning more about how voice recognition systems are implemented, since they are frequently used in Internet of Things (IoT). We did not have much signal processing background, so we wanted to gain knowledge in this area. Also, we wanted a project that would allow us to have a small machine learning component, which a voice recognition system requires. Current voice recognition products like Google Home and Amazon Alexa are interesting, but we wanted to do something slightly different, so we came up with our image slideshow idea.

MFCC Algorithm

The project uses the MFCC algorithm to extract features from a sample of speech audio. An overview of the algorithm can be found here.



The above diagram depicts the five steps involved in the MFCC algorithm.

1. FFT: Convert short chunks of audio data (~10-20 ms) to frequency spectrum.
2. Mel-Scale Filtering (Bandpass Filters): Use parallel bandpass filters to compute the sum over a small range of magnitudes for different frequency ranges.
3. Logarithm: Compute the logarithms of the outputs of the previous stage. This emulates the human ear, since humans are more perceptive to small changes in frequency at lower frequencies.
4. Cepstral Coefficients: Optional step to remove speaker-dependent characteristics. Generates cepstral coefficients for each chunk of the audio sample by computing the discrete cosine transform.
5. Derivatives: Optional step to compute the first and second derivatives of the cepstral coefficients.

Goals

The primary and stretch goals of our project are listed below. We have also indicated which goals have been completed.

Primary goals:

- ✓ Implement a functional IoT voice recognition system.
- ✓ Achieve over 50% recognition accuracy and < 3 second latency.

Stretch goals:

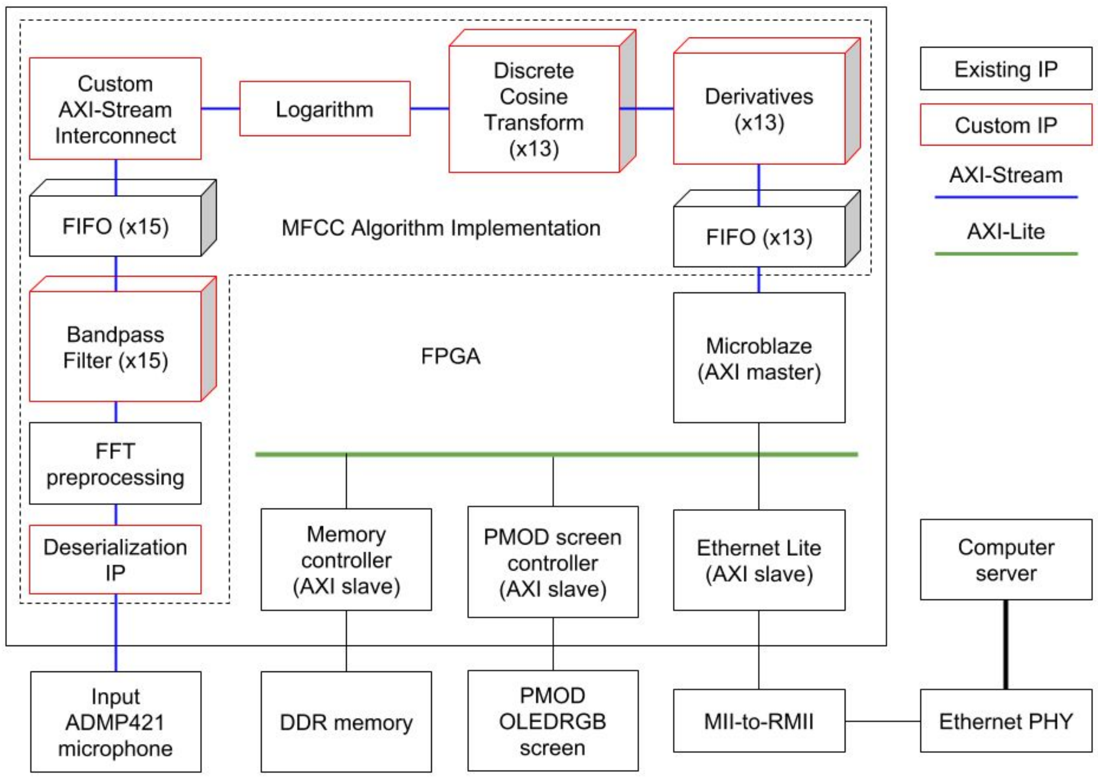
✓ Send images from the server back to the FPGA in an appropriate format, and use the PMOD screen to display the images.

✗ Expand the number of commands recognized by the system (e.g., add additional animal categories, add commands to control the slideshow such as "next" rather than just showing a single image).

✓ Add optional steps from the MFCC algorithm.

Block Diagram

The final block diagram of the project is shown below. The block diagram contains three main paths. The audio path processes the microphone input, implements the MFCC algorithm, and interfaces with the MicroBlaze processor. The server path establishes a connection to the computer server over Ethernet, and is used to send processed data to the server and receive images. The image path contains the PMOD screen, and displays the image from the server on the screen.



The following tables provide a brief description of the blocks in each path.

Audio Path

IP Name	Brief Description
Deserialization	Custom IP to convert the stream of binary microphone data to 8-bit chunks to pass to the FFT
FFT preprocessing	Xilinx FFT, multipliers and Cordic IP to convert to frequency spectrum and convert the real/imaginary FFT outputs to an AXI Stream FFT magnitude output.
Bandpass Filters	Parallel set of 15 custom IP blocks that each sum over a different range of FFT magnitudes.
FIFOs	Used 15 Xilinx AXI4-Stream Data FIFOs in parallel to store the results of bandpass filters.
Custom AXI-Stream Interconnect	Custom IP used to convert the parallel FIFO data into a single AXI-Stream of data.
Logarithm	Custom IP that uses a Block RAM as a lookup table to take the logarithm of the input.
Discrete Cosine Transform	13 parallel custom IP blocks and Xilinx multiplier IP to compute the cepstral coefficients from the logarithm values.
Derivatives	Parallel set of 13 custom IP blocks that computes the first and second derivatives of the cepstral coefficients.
FIFOs	Set of 13 Xilinx AXI4-Stream Data FIFOs to store the cepstral coefficients and derivatives.
Microblaze	Reads from the FIFOs until all features for one voice sample have been read in, and then sends the extracted features to the server over Ethernet.

Server Path

IP Name	Brief Description
Microblaze	After receiving all features, it sends them to the server over Ethernet, and then sends a GET command to the server and receives new image data. It overwrites the stored image array and updates the PMOD screen.
Ethernet Lite	Supports an interface for accessing Ethernet PHY MII and communicates to a processor via AXI4 or AXI4-Lite interface as described by Ethernet Lite Documentation.
MII To RMII	Converts MII data on a Xilinx Ethernet MAC core to RMII data as described by RMII description.
Ethernet PHY	Provides required logic to access the Ethernet MAC Core via either MII or RMII as described by Ethernet PHY documentation.
Computer Server	Gets the MFCC features from the FPGA and passes them to a pre-trained neural network for classification. It then sends an image of the predicted animal back to the FPGA after receiving a GET command.

Image Path

IP Name	Brief Description
Microblaze	Reads an image stored in memory and updates the screen to display this image.
AXI-Lite Bus	Xilinx IP that connects the MicroBlaze with AXI Slave peripherals used in the system : DDR memory controller, PMOD screen controller and Ethernet Lite.
Memory Controller	Xilinx IP that allows the MicroBlaze to interface with the DDR off-chip DDR memory.
DDR Memory	Off-chip DDR memory to store program executable and data.
PMOD Screen Controller	Digilent IP to interface with the PMOD OLEDRGB Screen collected from Digilent's PMOD Library.

PMOD OLEDRGB Screen	PMOD peripheral to display the image received from the server.

Outcome

The final project implements the entire MFCC algorithm in hardware, including all of the optional steps, which was one of our stretch goals. We also completed the stretch goal of displaying the images on the PMOD screen connected to the FPGA, rather than on the server.

The primary goal of implementing a functional system was achieved. The system works consistently, except the server would sometimes not receive all of the data sent over the network from the FPGA. The time from clicking the record audio to having a new image displayed on the screen was about 2.5 seconds, which was less than our goal of 3 seconds. The system could also achieve 75% test set accuracy when trained on 50 audio samples per class with a test set of 10 samples per class. Unfortunately, the system did not perform as well when tested on the fly in a new environment. The word “cow” was particularly hard to classify, whereas the other two were reasonably consistent.

The voice classification did not work very well in the live demo. “Alligator” and “baboon” were classified correctly most of the time, but it did not manage to correctly classify “cow”. The demo was done in a different room with background noise that was very different from the one in which all of the training data was recorded, which caused the system to perform significantly worse, as it lacks normalization or noise filtering. Also, adding the optional steps of the MFCC algorithm caused the system to be less accurate than before they were added. Because the neural network was trained on only Cameron’s voice data, when speaker-dependent information was removed from the data, the neural network had less information and was not as accurate as before at successfully classifying Cameron’s speech. We also only used a basic neural network architecture for classification, rather than trying to fine-tune the architecture to perform better (e.g., by using a recurrent neural network). Since this was a hardware project, we focused our efforts on adding additional hardware complexity, rather than trying to improve the classification accuracy. Overall, the hardware implementation, which was the main point of this project, was a success.

To further improve this project, a good next step would be to add normalization and noise rejection techniques so that Image Buddy can perform well regardless of its environment. If higher classification accuracy were required, more training data could be collected from different speakers in different environments. If we had more time, the next step would have been to add a custom IP to normalize the output of the FFT module. If starting over, we may have chosen to prioritize this more, since having a better classification accuracy would have made for a more interesting demo.

Project Schedule

This section discusses the proposed milestones at the beginning of the course and the actual project milestones that we reached. A discussion of the differences between them is included as well.

The following table describes the details of the original proposal milestones.

Original Proposal Milestones	
Milestone 0	● Be able to connect to a computer server via Ethernet.
Milestone 1	● Research voice recognition algorithms and FPGA implementations (all members). ● Find noise reduction IP core (Xilinx FFT) for audio preprocessing (Cameron). ● Understand how the screen works and be able to print simple shapes, like lines (Aaron). ● Be able to incorporate UART IP with Microblaze processor (Shuyi).
Milestone 2	● Be able to display an image (that has already been placed into memory) on the screen (Aaron). ● Begin implementing voice recognition module (Cameron). ● Begin setting up a simulation testbench for the voice recognition module (Shuyi).
Milestone 3	● Develop voice recognition module and verify the correctness of the implementation in simulation (all members).
Milestone 4	● Be able to take input from the microphone and feed it into the noise reduction module (all members).
Milestone 5	● Basic functionality of the voice recognition subsystem. Be able to take input from the microphone and feed it through the audio pipeline, and get the output in the Microblaze (all members). ● At this point, all three subsystems (voice recognition, screen, network) should be working in isolation.
Milestone 6	● Integration of all three subsystems (all members).
Milestone 7	● Improve voice recognition accuracy (all members). ● Fix any lingering stability issues (all members).

The following table describes the details of the actual accomplished milestones.

Accomplished Milestones	
Milestone 0	● Be able to connect to a computer server via Ethernet (completed warm-up demo).
Milestone 1	● Research voice recognition algorithms and FPGA implementations. ● Found similar FFT projects that show how to customize the Vivado FFT IP and convert its output data from real/imaginary to a magnitude frequency domain output.
Milestone 2	● Made microphone module to get data from microphone and deserialize it (convert from 1-bit to n-bit numbers). ● Integrated microphone module with MicroBlaze and computer server so pressing a button on the FPGA will record ~1 second of audio and send the data to the server, where a Python script does both FFT and MFCC.
Milestone 3	● Converted microphone module to an AXI-Stream IP so that it interfaces better with the FFT IP. ● Added multipliers and adders to convert FFT IP output to an AXI-Stream magnitude output, and verified it works as expected. ● Trained neural network on image data and achieved test set accuracy of 90% on 30 audio samples, but having consistency problems. ● Changed microphone sampling rate to use frequencies similar to those transferred in a telephone call (up to 6 KHz). ● Attempted to implement bandpass filters needed for MFCC by using FIR Compiler IP with coefficients generated by Matlab.
Milestone 4	● Created custom bandpass filter, where the passband is parameterizable. ● Verified its operation in hardware using the ILA.
Milestone 6	● Added PMOD LED screen to project and added code to MicroBlaze application to display a stored image on the screen. ● Changed server to send an image of predicted animal to the FPGA after classification, which the MicroBlaze stores it in memory and then displays it on the PMOD screen.

Milestone 7	● Created custom interconnect IP to convert parallel AXI-Stream data paths to one single AXI-Stream data path. ● Implemented custom logarithm IP that uses RAM lookup table. ● Produced discrete cosine transform IP to calculate cepstral coefficients. ● Implemented derivatives IP in hardware to calculate first and second derivatives of cepstral coefficients.

The largest difference between the planned and actual milestones is that we were originally going to start working on the PMOD screen from the start, but we moved that to a stretch goal at the end so that we could focus on building a functional voice-recognition system first before adding additional hardware complexity.

Another big difference was that when working on this project, we encountered a few issues with the network and IP from Vivado's IP Catalog that slowed down progress for that week's milestone. However, since we worked closely on all parts of the project, there were no integration problems at all at the end of the project, other than not having enough FPGA resources, which simply required the redesign of the derivatives IP block to be more efficient with its resource use.

Overall, it took longer to get each of the subsystems working than originally proposed. However, since we were continually integrating components as we worked on them, the lack of integration problems allowed us to make up for the extra time spent earlier in the project.

Detailed Description of Blocks

This section describes the details of each IP used in the final design.

Deserialization

This custom IP takes a stream of digital microphone data from the FPGA's ADMP421 onboard microphone and converts it to an AXI-Stream signal with n-bit data (the number of bits is customizable), to be sent to the FFT module. The system is clocked at 100 MHz, but the onboard microphone can only be clocked at 1 MHz to 3.3 MHz, so the IP uses a counter to toggle the microphone's clock input every 32 cycles, giving a frequency of 3.125 MHz (Nexys 4 DDR Documentation).

The project uses this IP with 8-bit outputs. In this case, the IP has an accumulator register that accumulates 256 microphone data points. This gives an 8-bit value that is sent through AXI-Stream to the FFT IP by setting the tvalid signal high. An 8-bit value was chosen because this allows the FFT to capture frequencies up to 6.1 KHz, keeping in mind that human speech is within the range 250-6000 Hz (Speech Information).

This IP had another counter to count the number of values stored in a chunk of audio. The tlast signal is set high when the microphone sends 256 data samples to the FFT, to signal that it can start processing the input data. Each chunk represents 21 ms of the audio sample. A final counter checks the number of chunks sent to the FFT. Our IP sends 64 chunks of data, which represents 1.34 seconds of audio. This was chosen because it gives enough time for a speaker to finish saying the longest word “alligator”, taking into account that they will likely have a small delay between pressing the button and starting to speak.

This was initially tested in simulation with a testbench to check that it was accumulating values correctly, setting tvalid and tlast at the correct times, and producing the correct number of chunks and samples per chunk. Next, an ILA was used to verify that the output values were reasonable. For quiet inputs, the output value should remain near 128 (the midpoint of 0 and 255), whereas for loud inputs it should have values closer to 0 and 256.

FFT Preprocessing

Functionally speaking, the custom FFT preprocessing IP generates a frequency domain representation of the input AXI-Stream data using the Vivado FFT IP. It then calculates the magnitude of the generated frequency spectrum from its real and imaginary parts using Vivado’s Multiplier and Adder IPs, as well as the CORDIC IP, which is configured to calculate the square root. The magnitude output is also in AXI-Stream format.

Structurally speaking, the FFT preprocessing IP consists of the Vivado Fast Fourier Transform IP, 2 AXI4-Stream Register Slice IPs, custom delay IPs, 2 Multiplier IPs, one Adder IP, and one CORDIC IP.

The detailed description regarding the usage of IPs is described in the following table.

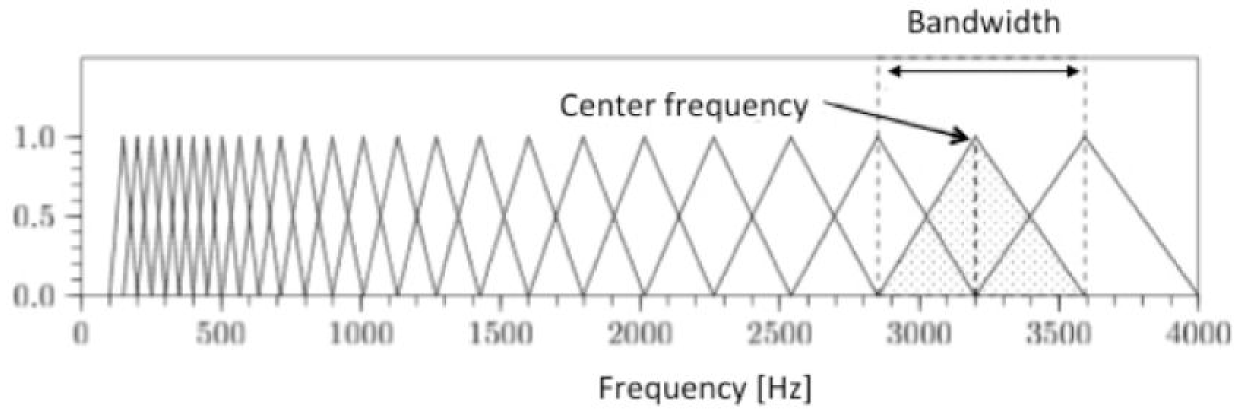
IP Name	IP Description
FFT	Configured in Pipelined, Streaming I/O mode. It takes 256 data samples, which are represented as unsigned integers, and performs the computation in one stage using Block RAM and delivers 48-bit AXI4-Stream data output.
AXI4-Stream Register Slice	Separates tdata from the rest of the AXI-Stream signals.
Custom delay	Delays the other FFT output signals while tdata is undergoing processing. Specifically, the IP takes tvalid, tuser, and tlast and then outputs them without any modification five clock cycles after.

Multiplier	Calculates the square. In the design, two multiplier IPs are used; one computes the square of the real part of the generated frequency spectrum, while the other computes the square of the imaginary part. The computation is pipelined to use 5 clock cycles to avoid timing violations. This is why the delay block is required.
Adder	Adds the output from the two multiplier IPs.
CORDIC	Computes the square root of the output of the Adder IP to yield the magnitude.

The FFT module was first tested using simulation to ensure that it was giving a tvalid signal after receiving the correct number of input values. Also, we confirmed that the correct output was generated for a constant input (the 0th index was a constant, all other indices were 0). Next, the FFT was tested with the ILA. A low-pitch tone was played from a phone into the onboard microphone and the FFT output was confirmed to match the frequency of the input tone. This was repeated for a high-pitch tone.

Bandpass Filters

Bandpass filters are used to compute the Mel-frequency spectrum from the FFT output. A set of 15 parallel bandpass filters that each target a different range of frequencies is used. Each filter takes the magnitude AXI-Stream output from the FFT, accumulates the magnitudes in its range, and outputs the final accumulated value through an AXI-Stream interface. The IP is parameterizable; the start and end index for accumulating can be customized. Once the end index is reached, the IP sets the tvalid signal high. This is done for each chunk in the audio sample. Filters for the higher frequencies cover a larger range of sound to better mimic human auditory perception (see figure below). Note that square bandpass filters were used instead of triangular ones since some research shows it can achieve better results (MFCC Description).



The bandpass filters were tested in simulation by passing a stream of numbers to the module. It was verified that it started accumulating at the begin point, and would set tvalid output to high upon reaching the end point. We also confirmed that the accumulator value was reset when the input set the tlast value high.

FIFOs

Each band-pass filter connects to a Vivado AXI4-Stream Data FIFO IP. The FIFO was set to store 64 values, since there are 64 chunks in an audio sample. This prevents the possibility of overfilling the FIFO if downstream blocks cannot process the data fast enough. The documentation for this IP can be found here: AXI4-Stream DATA FIFO Documentation.

AXI-Stream Interconnect

Since the logarithm is using a large RAM lookup table, there can only be one instance of it, or else there would not be enough board resources. This custom IP was created to read from the FIFOs containing the bandpass filter data in a round-robin fashion and send them through the logarithm IP. The later Discrete Cosine Transform step required this to be done in a round-robin fashion.

This IP has a counter that selects the next input to read from. Once the downstream AXI-Stream is ready to receive data and sets the tready signal, it propagates this signal to one of the FIFOs. It waits for the upstream FIFO to set its tvalid signal, at which point it sends the data downstream, and increments its FIFO selection counter.

This IP was tested using an ILA connected to the inputs and output. We confirmed that it selected the inputs in round-robin fashion, and once an input was valid it would set tvalid high then move on to the next input.

Logarithm

The logarithm uses a Block RAM lookup table to output a fixed-point representation of the log of the input. It uses the input value to index the RAM and gives a valid output one clock cycle after the input becomes valid. Logarithm is always positive for an input greater than or equal to 1, so it outputs a 16-bit unsigned value, where there are 5 bits for the integer part and 11 bits for the fractional part. This module works for input values up to 131,071 (17 bits). This number was empirically chosen; based on many audio samples taken, there was never an input to the log greater than about 70,000, so it is unlikely that any numbers would exceed this range. As a safeguard, should the input exceed this range, this module sets the output to 0x8800, which is one larger than the largest possible output value. This is to prevent overflow from introducing a significantly wrong output value.

The logarithm custom IP was tested using a simple testbench that entered in a few values from 0 to 131,071 and checked that the correct output was produced. The fixed-point values for these were checked manually as well to ensure that the generated memory file was correct.

Discrete Cosine Transform

The cepstral coefficients are calculated through the following formula:

$$c_{\tau,j}^{(4)} = \sum_{j=1}^{N_d} c_{\tau,j}^{(3)} \, cos \left[\frac{k\,(2j-1)\pi}{2N_d} \right] \qquad k = 0, 1, ..., N_{mc} < N_d$$

where k = 0, 1, …, 12 for a total of 13 parallel computations, and j = 0, 1, …, 14 are the indices of the 15 log filters. Each cepstral coefficient is the sum of all filters' log result multiplied by a constant based on k. This gives a total of 832 cepstral coefficients. This step is explained in more detail at MFCC Description.

IP Name	Description
Get_values	Sends the output of the logarithm and the cosine value to a set of 13 parallel multipliers. The cosine values are loaded from a memory file where they were precomputed. Each of the parallel multipliers is sent a different value corresponding to the different k values. Each cosine value has 12 bits, stored in two's complement signed fixed-point format. There is 1 bit to account for the sign, 1 bit for the integer part (since cosine is within [-1, 1]), and 10 bits for the fractional part. The j value is incremented after each input is received.
Multiplier	The Multiplier is the Vivado Multiplier IP, which multiplies the unsigned log input with the signed cosine value. It is set to pipeline the output by 3 cycles. The multiplier output is sent to an accumulator block.
Accumulator	Sums over the 15 filters to produce the cepstral coefficient. Once it has accumulated 15 values it sends the value to the derivative stage and resets its output to 0. The output is a 32-bit two's complement signed number with 1 sign bit, 10 bits integer part, and 21 bits fractional part.

A simple testbench was created that sends in an input of all 1s, and then we verified that the correct output values were produced (matching the memory files) in simulation. Some values from cosine memory files were calculated manually to verify that the stored results were computed correctly.

Derivatives

The MFCC algorithms uses cepstral coefficients along with the first and second derivatives of the cepstral coefficients as its feature vectors. The derivative for a chunk is calculated by getting the difference between the next chunk and previous chunk's cepstral coefficient. The second derivative is calculated the same way. Since the audio is separated into 64 chunks, there are 64 cepstral coefficients, 62 first derivatives and 60 second derivatives for each of the 13 different k values. This gives a final number of 2418 features that will be passed to the machine learning classifier.

Due to resource limitations, the IP needs to calculate the derivatives on the fly, rather than storing all the values first. It has 3 registers to store the 3 most recent cepstral coefficients and 3 registers to store the 3 most recent first derivatives. It uses a finite state machine to determine which value to output. For each audio sample chunk, it first outputs the cepstral coefficient, then the first derivative, and then the second derivative, before moving on to the next audio chunk. The first two chunks have no first derivatives and the first four chunks have no second derivatives.

The derivatives custom IP was tested with a testbench. We verified that the correct first and second derivatives were calculated, and that a correct number of values were outputted.

FIFOs

The outputs of the derivatives are stored in separate instances of the Vivado AXI4-Stream Data FIFO, where they can be read by the MicroBlaze. There are 13 parallel FIFOs, one for each of the k value data paths. For each k value, there are 186 values produced (64 cepstral coefficients, 62 first derivatives, and 60 second derivatives), so the FIFOs were set to a capacity of 256 to ensure all the values can be stored.

MicroBlaze

The MicroBlaze is the Xilinx soft processor core IP (MicroBlaze Documentation). The AXI-Stream interfaces were enabled to connect to the 13 FIFOs. We found that the 0th Interface was producing erroneous values, so it was not used. The MicroBlaze runs a program that reads from the FIFOs in round-robin fashion. It blocks on a read until it receives data from it, and then stores the value in an array of integers. Once it receives 186 values from each of the 13 FIFOs, it sends the array over Ethernet to a server using the TCP protocol. It then sends a GET command to the server and waits to receive an image from the server. After receiving an image, it updates the stored image array and calls a function to refresh the screen. To make it clear that an update has happened, it first clears the screen by setting it all black.

Memory Controller

The memory controller is used to interface with the on-board DDR2 memory. The C program executable that the MicroBlaze processor runs and its data are stored in this memory. The MicroBlaze connects to it through an AXI-Lite bus. The MicroBlaze was given 32 KB of instruction memory and 32 KB of data memory. The DDR2 memory has a maximum capacity of 128 MB. The datasheet for the MIG can be found here: MIG Documentation.

MII-to-RMII

According to the documentation, the Vivado MII-to-RMII IP converts the 16-pin Media Independent Interface (MII) on a Xilinx 10/100 Ethernet MAC core to a 6-pin Reduced Media Independent Interface (RMII) interface, allowing the MAC to connect to RMII compliant PHYs. In addition, automatic detection of Receive side throughput is provided by the RMII module and the network throughput can be customized (RMII description).

Ethernet PHY

The PHY interface provides required logics to access the Ethernet MAC Core via either MII or RMII (Ethernet PHY documentation).

PMOD Screen Controller

The PMOD Screen Controller is a Digilent IP obtained from PMOD Controller Library. This library also contains example code for the MicroBlaze processor to display an image on the PMOD screen.

Description of Design Tree

https://github.com/cameronutoronto/image_buddy

The design tree contains two main subdirectories: “server”, which contains the server Python script and a saved TensorFlow model, and “vivado”, which contains source files and Tcl scripts for building the Vivado project. A third subdirectory, “doc”, contains this document and a powerpoint presentation.

Under the “vivado” subdirectory, “constrs” contains the constraints file, “ip_repo” contains all of our custom IPs and Digilent’s PMOD library, “sdk” contains the application project, and “sim” contains testbenches for some modules. The “src” subdirectory contains the Verilog source files (in “hdl”), memory files (in “mem”), and a Tcl script for generating the top-level block design (in “bd”). These files are used by the “build.tcl” script to generate the Vivado project from the Tcl Console.

After generating the project and all its outputs, the application project in “vivado/sdk” must be imported into the SDK. Under "File -> Import...", select "General -> Projects from Folder or Archive". Select the "vivado/sdk" directory and import "screen_proj" and "screen_proj_bsp".

Tips and Tricks

- When using IP blocks that you didn't make yourself, never assume that it works how you think it should, even if it's from the Vivado IP Catalog. Simulate everything. For example, we tried to use the AXI-Stream Interconnect IP from the IP Catalog to combine many streams into a single stream, and did not bother to simulate it. We didn't notice until hours later that the stream data was corrupted, and it took more time still to trace the issue back to the interconnect.
- If using Vivado doesn't work correctly, try another method that should achieve the same result. Sometimes we experienced crashes/problems doing things one way, but when we tried to do it in a different way, it would work correctly, even though intuitively there should be no difference between the two ways of doing things. A strange example of this is that we found our design did not work correctly when using the 0th AXI-Stream interface on the MicroBlaze, but all of the other interfaces worked correctly.
- Verify that archiving a project works correctly. We found that we should have at least 1.5GB of free disk space when archiving a project, otherwise there is a chance that the project will fail to archive correctly. Vivado will still say that everything archived correctly, but there will be missing files in the archive. It seems like many intermediate files are created during archiving and if there isn't enough space for all of them, then Vivado will not throw any warnings/errors. Always make sure you have lots of extra disk space.
- Start with something that is not difficult to finish, but has room to add improvements to. This way, if you experience problems that make progress much slower than anticipated, you will still have something working at the end. Our strategy for this was to get things working at the start mostly in software, and then moving single components one at a time to hardware, ensuring that things still work correctly at each step.