Feline Programmable Gate Array

Final Report

Andi Qu
Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
andiqu@mit.edu

Richard Beattie
Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
rbeattie@mit.edu

***Abstract*—We present *Feline Programmable Gate Array*: a robotic cat that can recognize and move toward its owner's voice. *Feline Programmable Gate Array* is implemented on a RealDigital Urbana development board and leverages an FPGA's digital signal processing capability to perform real-time audio inference. The system features several different applications of digital systems — audio capture and filtering through I2S microphones, feature extraction and voice recognition using Mel-frequency cepstrum coefficients, sound localization using a time-difference-of-arrival algorithm, Bluetooth communication, and motor controls. This paper reports the project's outcomes, implementation details, and the insights we gained from designing a large-scale digital system.**

***Index Terms*—digital systems, field programmable gate array, voice biometrics, sound localization, autonomous systems**

I. Introduction

It is well-known that cats are four-stage finite state machines wrapped in fur: they can recognize their owner's voice, move toward that voice, get distracted by red laser dots, and meow. As such, this project aimed to build a robotic cat that can identify and move toward its owner's voice.

We used a Xilinx Spartan-7 FPGA on a RealDigital Urbana board to achieve this goal. FPGAs were ideal for this project because they support a wide range of sensors and outputs (for example, microphones and Bluetooth) and can process sensor inputs in real time. This real-time signal processing capability enabled us to implement capabilities like sound localization that would otherwise be difficult on the traditional von Neumann architecture.

By the end of the project, we successfully implemented the following capabilities, all running on a single robot/FPGA:

- I2S-driven audio capture.
- Finite impulse response (FIR) audio anti-aliasing.
- Bluetooth low energy (BLE) wireless communications.
- Mel-frequency cepstrum coefficient (MFCC) extraction.
- Real-time voice recognition.
- Real-time audio localization.
- Real-time image recognition.
- Autonomous servomotor controls.
- Autonomous DC motor controls.
- Audio playback.

The source code for this project is freely available at https://github.com/dolphingarlic/feline-programmable-gate-array

II. System Architecture

Figs. 1, 2 and 3 shows how the project was structured. First, audio data would be captured, filtered, and downsampled to 4 kHz. The FFT of the audio data would then be computed, which would be fed into the sound localizer and biometrics modules. These two modules would output control signals to determine whether and how the robot moves.

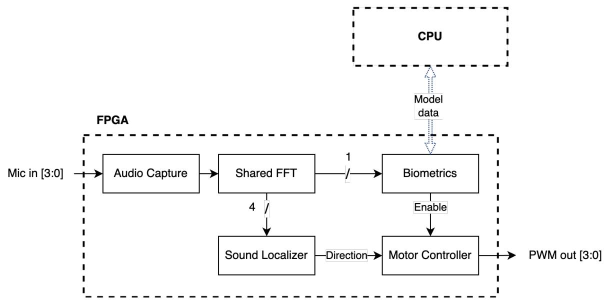

Fig. 1. The top-level system diagram. Data flows along the arrows, and the blue dashed arrow represents Bluetooth communication.

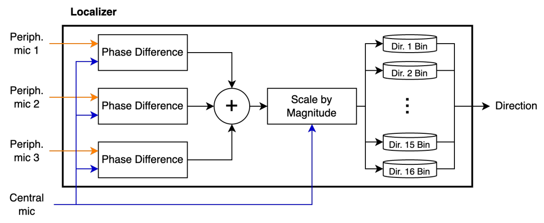

Fig. 2. The structure of the localizer module.

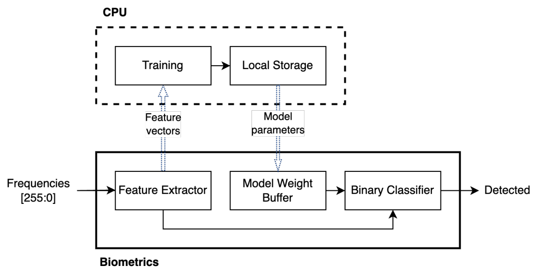

Fig. 3. The structure of the biometrics module.

III. Physical Construction

The robot consists of:

- A aluminum robot chassis previously designed and fabricated for 2.S007
- Four SPH0645LM4H-B I2S MEMS microphones on a solderless breadboard
- Two TT Motors and a L298N Dual H-Bridge Motor Driver to control them
- One TowerPro SG92R micro servo
- A 4 AA Battery Pack to power the motors and the servo
- A Xilinx Spartan-7 FPGA

IV. Audio Capture

The audio was captured through four SPH0645LM4H-B I2S MEMS microphones on Adafruit breakout boards. These microphones were arranged with a central microphone surrounded by three peripheral microphones.

To provide maximum resolution for sound localization while minimizing cycle-intensive calculations, the system assumes the central microphone is placed at $(0, 0)$ and the peripheral microphones at $(0, 1)$, $(-1, -1)$, and $(1, -1)$. Physically these coordinates correspond to $(0, 0)$, $(0, 3\,\mathrm{cm})$, $(-3\,\mathrm{cm}, -3\,\mathrm{cm})$, and $(3\,\mathrm{cm}, -3\,\mathrm{cm})$.

This $3\,\mathrm{cm}$ distance was chosen to be less than half the wavelength of the frequency interest to avoid spatial aliasing. In this case, $\lambda/2 = 5.72\,\mathrm{cm}$ because $f = 3\,\mathrm{kHz} \implies \lambda = 11.44\,\mathrm{cm}$.

A. I2S Communication

A variant of the I2S protocol was used to receive audio data from the microphones. In the standard I2S protocol, shown in Fig. 4, data is written on the *falling* edge of the serial clock (SCK). However, the SPH0645LM4H-B writes the data on the *rising* edge instead, as shown in Fig. 5.

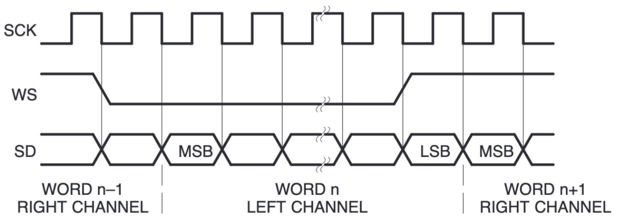

Fig. 4. Standard I2S timing diagram, adapted from [2]. DATA changes on the *falling* edge of the clock.

An I2S controller module was implemented to generate the serial clock (SCK) and word select (WS) signals. Normally, WS represents the sample rate of the microphone. However, I2S assumes a *stereo*-microphone, while the SPH0645LM4H-B is a *mono*-microphone, so our effective sample rate is only half the WS frequency. The microphones had a maximum clock frequency of $4.096\,\mathrm{MHz}$ and an oversampling rate of 64, so the maximum WS frequency was $64\,\mathrm{kHz}$, which corresponds to a sampling rate of $32\,\mathrm{kHz}$ [1].

The SCK and WS signals were then connected to the four microphones, along with an I2S receiver module (operating

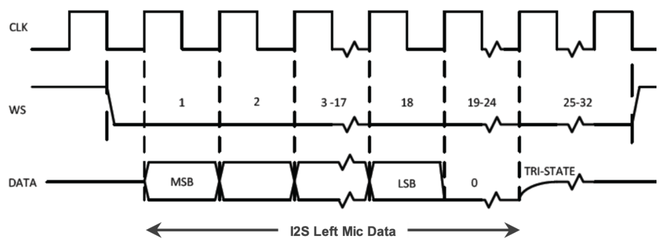

Fig. 5. SPH0645LM4H-B timing diagram, adapted from [1]. DATA changes on the *rising* edge of the clock.

in secondary mode). The I2S receiver was implemented by following the I2S spec [2], adjusting for the non-standard protocol.

B. Filtering and Decimation

Human speech typically occurs from between $20\,\mathrm{Hz}$ to $3\,\mathrm{kHz}$. As such, it was necessary to pass the audio data through a low-pass finite impulse response (FIR) filter. We used the Xilinx FIR Compiler IP to perform the filtering and decimation.

First, a low-pass filter was designed using Matlab's `filterDesigner` tool. F_s (the effective sampling frequency) was set to $32\,\mathrm{kHz}$, F_{pass} to $3\,\mathrm{kHz}$ and F_{stop} to $6\,\mathrm{kHz}$, which generated a sequence of coefficients between -1 and 1. Because the Xilinx FIR Compiler IP required *integer* coefficients, all coefficients were then divided by the coefficient with minimum absolute value and rounded to the nearest integer.

These values were then used in the Xilinx FIR Compiler IP, which was set to decimation mode with a decimation rate of 8, reducing the sampling frequency to $4\,\mathrm{kHz}$.

C. Increasing Gain

Analysis of the microphone outputs using an oscilloscope revealed that the dynamic range was much greater than expected, and the six most significant bits were always 1 during testing. To increase sensitivity to human speech (for testing with headphones, localization, and biometrics), the filtered and decimated audio data was left-shifted by six positions.

D. Signal Windowing

Before passing the filtered and decimated audio data to the fast Fourier transform (FFT), it was necessary to pass it through a Hanning window. FFT assumes that an integer number of periods of a signal is captured (that is, the first and last captured values are equal), but this is rarely the case with audio data. Thus, to avoid spectral leakage (frequencies present in the output that were not in the input), the input data should be windowed as shown in Fig. 6 [3].

A Hanning Window was chosen, as both ends of the window touch zero, removing any discontinuities from the original signal when applied.

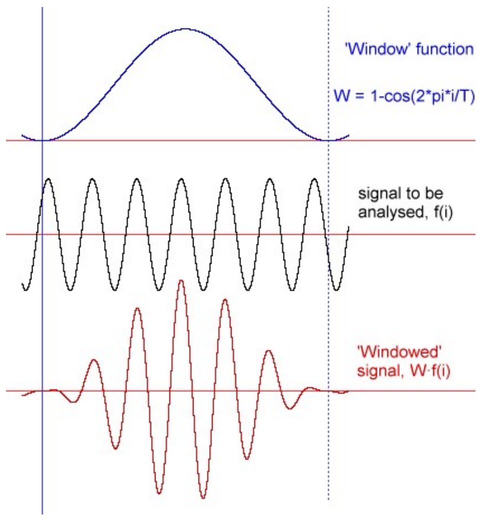

Fig. 6. The effects of the Hanning window on a signal. Adapted from [3]

E. Shared FFT

To conserve resources, the sound localization and biometrics modules shared an FFT instance. A Xilinx FFT LogiCORE™ IP block was used, configured with four channels (one for each microphone) to operate on 512 samples and output coefficients in natural order. Because audio data is sampled so slowly relative to the system clock, the Radix-2 Burst I/O architecture was used. After processing an audio frame, the FFT instance would send the 512×4 calculated real and imaginary components frequency downstream.

V. Voice Biometrics

This module served three main purposes: extracting useful features from the raw audio, training a voice recognition model on those features, and using that model to predict whether the incoming voice matches the owner's voice.

Due to time and resource constraints, model training did not happen on the FPGA. Instead, the FPGA would send the extracted feature vectors over BLE radio to an external laptop for training, and the laptop would then send the trained model parameters back to the FPGA.

A. Feature Extraction

MFCCs were the main features used for biometrics in this project. Modern speech recognition systems commonly use MFCCs as features [4] because they concisely mimic how humans perceive pitch and timbre [5].

MFCCs are based on the Mel scale, which relates *perceived* pitch to measured frequency. Humans are more sensitive to low-frequency than high-frequency pitch changes, so Mel frequencies grow logarithmically with Hertz frequencies. The conversion from Hertz to Mels is:

$$M(f) = 1125 \ln(1 + f/700)$$

Likewise, the inverse conversion is:

$$M^{-1}(m) = 700(\exp(m/1125) - 1)$$

In this project, MFCCs were computed via the following five-step process [5]:

1) First, the real and imaginary components of the FFT output are squared and summed. This step yields the power spectrum of the signal.
2) The power spectrum then passes through a Mel filterbank — a set of 32 triangular filters with peaks evenly spaced in the Mel scale. Fig. 7 shows a Mel filterbank with only 10 filters.
   The 32 filters run in parallel for maximum throughput. A description of our filter design and implementation is presented in the Appendix.
3) The output of each filter is accumulated, which yields 32 filterbank energies. Each filterbank energy represents the loudness of its respective Mel frequency band.
4) The logarithms of the 32 filterbank energies are then computed in parallel. This step is also motivated by human hearing, as humans do not hear loudness on a linear scale [5].
   The natural logarithm is normally used here, but we decided to use the base-2 logarithm instead because it is simpler to compute and differs from the natural logarithm by a constant factor. A detailed description of our implementation is presented in the Appendix.
5) Finally, the discrete cosine transform (DCT) of the log energies is computed using the Xilinx XFFT IP module. This step decorrelates the feature vectors' components, which is necessary because the triangular filters in the Mel filterbank overlap.
   Only the lower half of the DCT coefficients are kept after this step to simplify the feature vectors.

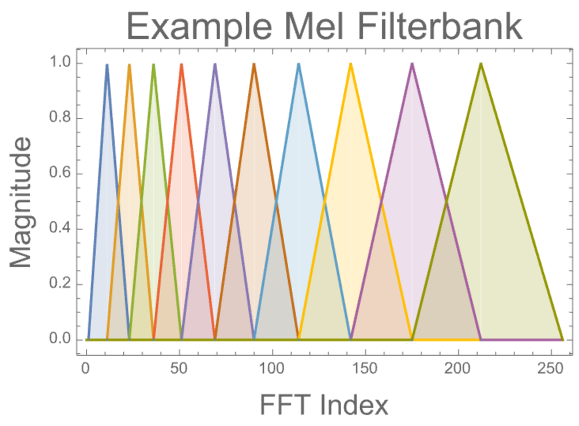

Fig. 7. An example Mel filterbank with only 10 triangular filters. Note that the filters are narrow at low frequencies and wider at higher frequencies.

These five steps were implemented as standalone modules that communicate with each other over AXI4-Stream.

B. Bluetooth Communication

Each RealDigital Urbana board came equipped with a Nordic Semiconductor nRF52832 BLE device, which was used for wireless communication between the FPGA and external laptop (an M1 MacBook Pro). The BLE device communicated with the FPGA using a two-wire (RX/TX) UART protocol, running at a baud rate of 115 200 bps [6].

During testing, it was discovered that the nRF52832 BLE device required a newline character (hex code `0x0A`) to be sent over UART to flush the output. This condition posed a few challenges to our design:

- The BLE device can only hold up to 256 bytes of data in its output buffer, so the FPGA needed to send a newline character periodically to prevent a crash.
  Because the feature extractor used the AXI4-Stream protocol, a natural solution to this problem was to send a newline character after sending a feature vector over BLE. Fig. 8 shows an FSM describing this behavior.
  Note that this extra newline character would not be necessary if the last byte of the feature vector was `0x0A`.
- The first byte sent after a newline character could not be `0x0A`, or else the BLE device would ignore that byte.
  To address this problem, we hard-wired the lowest bit of each MFCC feature to be 1 and then sent the lower 8 bits before the upper 8 bits. Although this approach meant losing one bit of information per feature, it guaranteed that every byte of information would be intact.

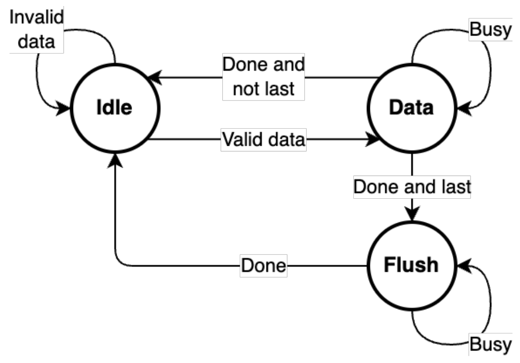

Fig. 8. The FSM controlling how the FPGA sent data over BLE. Note the need to flush the output after each burst of data.

It is worth noting that the receiving end of the BLE device mostly did not have these challenges. The only roadblock we encountered on that end was a data transmission rate limit, which we addressed by having 0.01 s idle periods between sending bursts of data from the laptop.

After overcoming these challenges, we were able to use Bleak [7] (an open-source Python BLE client) to send and receive hundreds of bytes of data wirelessly between the FPGA and laptop. Thanks to the low audio sampling rate, we also did not need to discard any data to accommodate the (relatively slow) BLE baud rate and MacBook Bluetooth polling rate.

C. Inference

Predicting whether the incoming voice matches the owner's voice is a binary classification problem. Although there are many approaches to binary classification, the fact that training data consists almost entirely of positive data greatly limits the set of feasible approaches. Ultimately, we decided to use the One-Class SVM [8] — an outlier detection algorithm that has seen use in the literature for sound classification [9]. This algorithm is designed to train on mostly positive training data and is also relatively straightforward to implement in hardware for real-time classification.

One-Class SVM works by learning a decision boundary (often the minimal bounding hypersphere) around a single class of data, allowing them to identify whether new data points deviate from the "normal" class. This decision boundary is defined by a small set of "support vectors" — a subset of the input data points lying closest to the boundary.

To classify a data point $\mathbf{x}$, the algorithm computes the decision function:

$$f(\mathbf{x}) = \sum_{i \in S} \alpha_i K(\mathbf{x}_i, \mathbf{x}) + b$$

where S is the set of support vectors, α_i are weights between 0 and 1, b is a bias offset, and K is a "kernel function" (an arbitrary function that computes a scalar from two vectors). If $f(\mathbf{x}) > 0$, then $\mathbf{x}$ is classified as an inlier (that is, the incoming sound matches the owner's voice); otherwise, it is classified as an outlier.

For simplicity, we chose K to be the linear kernel:

$$K(\mathbf{x}_i, \mathbf{x}) = \mathbf{x}_i \cdot \mathbf{x}$$

Although higher-dimensional kernels like the popular RBF kernel $K_{\text{RBF}}(\mathbf{x}_i, \mathbf{x}) = \exp(-||\mathbf{x}_i - \mathbf{x}||^2/2)$ would potentially have classified data more accurately, we found that the added complexity was not worth the marginal gains in accuracy.

Another advantage of the linear kernel arose from linearity:

$$\alpha_i K(\mathbf{x}_i, \mathbf{x}) = K(\alpha_i \mathbf{x}_i, \mathbf{x})$$

which meant that instead of storing both α_i and $\mathbf{x}_i$, the FPGA only needed to store the scaled coefficients of $\alpha_i \mathbf{x}_i$. This also allowed the FPGA to save one multiplication operation per support vector, which led to lower-latency classification.

Model training was implemented using scikit-learn's `sklearn.svm.OneClassSVM` [8]. The model training pipeline worked as follows:

1) First, a script would connect with the FPGA over BLE, ingest a stream of extracted feature vectors, and filter out data below an empirical loudness threshold.
   Each feature vector would consist of 16-bit fixed-point numbers, which would then be rescaled and interpreted as signed integers.
2) Next, another script would train an SVM model on the ingested data.
3) Finally, that same script would reconnect with the FPGA over BLE and stream the model parameters (scaled support vectors and bias offset) to it.
   Each support vector would consist of 16-bit signed integers, and the bias offset would be a 32-bit signed integer. Fig. 9 shows how such a stream of data would be structured.

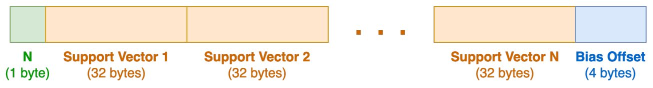

Fig. 9. The communication protocol over BLE to get model parameters from the external laptop to the FPGA.

After receiving the model parameters over BLE, the FPGA would store them in BRAM, with each dimension of the support vectors having its own dual-port BRAM. We decided to use 16 BRAMs in parallel here because it allowed us to compute $K(\alpha_i \mathbf{x}_i, \mathbf{x})$ with maximum throughput.

Computing the sum of the dot products was achieved using a simple counter-controlled loop and accumulator. After computing this sum, the inference module would compare it with b and output a binary "detected" signal based on the result. For extra accuracy, audio inputs below a loudness threshold were ignored, just like in the model training pipeline.

VI. Sound Localization

This module determines the direction of an audio source relative to the microphone array. As discussed previously, four microphones were used — three peripheral microphones arranged in a triangle around a central microphone. The key idea behind our approach toward this module is the fact that there is a small but detectable time difference of arrival (TDOA) between when each microphone detects a sound. We used this TDOA to calculate the direction of the source.

A. Filtering frequencies

Of the 512 calculated FFT coefficients, only the lower 256 are usable, by the Nyquist-Shannon sampling theorem. We further restricted the coefficients to account for the human speech frequency range. Based on experiments, we decided to take the 10th to the 226th coefficients — roughly corresponding to the range 100 Hz to 2.65 kHz.

B. Rectangular to Polar Conversion

FFT outputs coefficients in the rectangular form (x, y), but the polar form (r, ϕ) is needed to calculate the TDOA. As such, a Xilinx LogiCORE™ CORDIC IP block was used to convert between the two forms. Four CORDIC modules were instantiated to convert all four channels in parallel.

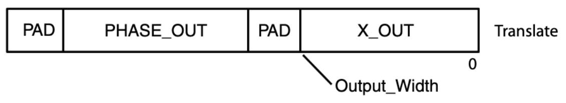

Fig. 10. TDATA Structure for Output for CORDIC IP in Translate Mode, adapted from [12]

The CORDIC module would output a 32-bit value, as shown in Fig. 10. The most significant 16 bits would be the phase ϕ, expressed in radians as a 3.13 fixed point two's complement number, and the least significant 16 bits would be the magnitude, expressed as a 2.14 fixed point two's complement number. The magnitude is scaled by a constant factor; however, as the magnitude is not used, this scaling factor did not matter in this project [12].

C. Direction Calculator

Once converted to polar form, an overall direction was determined. Cross-correlation is often used for this step and has been accomplished in 6.2050 before [10]. However, since we only need the direction and not the distance to the sound source, we implemented a different algorithm, as outlined in [13]. The idea is to scale each peripheral microphone's location by its delay to get an average location.

This algorithm was accomplished in three steps:

1) First, the phase difference $\Delta\phi$ between each peripheral microphone and the central microphone is calculated and constrained to be between π and $-\pi$. We know that the time difference between two signals (of the same frequency f) is related to the phase difference by:

$$\Delta t = \frac{\Delta\phi}{2\pi f}$$

However, since the frequency is the same for all three microphones this scaling factor can be ignored.

2) Each microphone's location is then scaled by its phase difference. As discussed previously, $(0, 1)$, $(-1, -1)$ and $(1, -1)$ were chosen as microphone locations to avoid cycle-intensive multiplication in this step.
3) Finally, the x and y components from each scaled location are summed to give an overall direction vector.

The phase difference calculation and two additions mean the summed location would be 19 bits (in 6.13 fixed point form), so the three least significant bits are discarded. Due to the limited arithmetic operations, this module was implemented combinationally.

D. Direction Binner

For each frequency, the overall direction vector is put into one of 16 direction bins, each representing $\frac{\pi}{8}$ radians, as shown in Fig. 11.

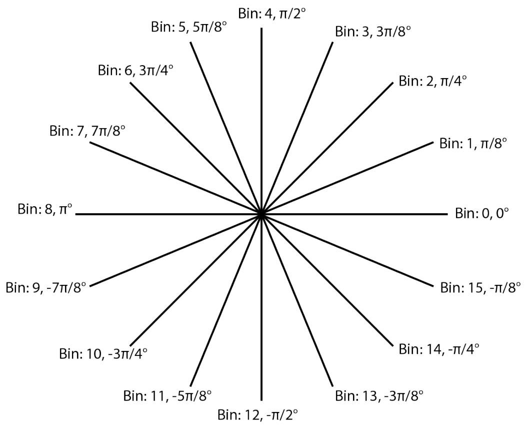

Fig. 11. 16 angle bins and their corresponding angles.

First, another CORDIC module translates the direction vector into polar form. The resulting phase is compared to predetermined bin values, and the magnitude of the vector is added to the matching bin.

Once all 216 frequencies have been binned, the maximum magnitude and its corresponding bin are determined. These values are provided over AXI4-Stream out of the localizer module. Finally, the CORDIC IP applies back pressure on the translate module.

VII. Motor Controls

A. Servo Motor Control

To test sound localization, a TowerPro SG92R micro servo was used. This servo expects a PWM signal with a fixed period of $20\,\mathrm{ms}$ and a pulse width from $1\,\mathrm{ms}$ to $2\,\mathrm{ms}$ [14] that varies its position from $0°$ to $180°$, as shown in Fig. 12.

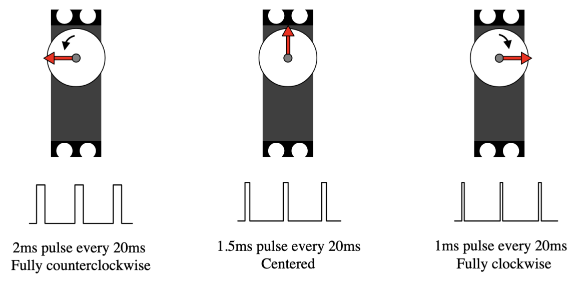

Fig. 12. Servomotor PWM signal, adapted from [14]. Not that the minimum and maximum duty cycles are somewhere between 0% and 100%.

The servo was powered by an external $5\,\mathrm{V}$ battery supply attached to the `VS` and `GND` pins on the Urbana board. Each bin from the localization module was mapped to a specific duty cycle to make the servo point in the direction of the sound source.

B. DC Motor Control

Continuous servos were initially used to move the robot. However, these motors generated enough noise to interfere with the localization module. Instead, two Adafruit DC Gearbox TT Motors were used. The higher voltage required to power these meant a L298N Dual H-Bridge Motor Driver was also required.

Each motor had three ports: `IN1`, `IN2`, and `EN`. `IN1` and `IN2` controlled the direction of the motor and were always set to `low` and `high` respectively, as the motors only moved clockwise. `EN` controlled whether the motor is enabled.

A PWM signal was generated for the `EN` pin to vary the motor speed. A frequency of $20\,\mathrm{kHz}$ was chosen to not interfere with localization or voice biometrics.

C. Robot Motion Control

This module determined the speed of the left and right motor based on the bin and magnitude from the localization module, and whether the biometrics module detected the correct person talking.

First, the detected bin was stored in a register. If the newly-detected bin from the localizer module had a magnitude greater than 5000 (an experimentally determined value), then the register would be updated. This is done to reduce false positives from pauses during speech.

If the biometric module did not recognise the person, both motors would be turned off. If it did recognise the person, then there were three cases:

1) If the source is in bins 6 to 11 (on the left), then only the right motor would turn on.
2) If the source is in bins 0 to 2 or 12 to 15 (on the right), then only the left motor would turn on.
3) If the source is in bins 3 to 5 (in front), then both motors would turn on.

VIII. Additional Cat-like Behaviors

In addition to reacting to audio, the robot can meow and chase lasers like a real cat. These behaviors were stretch goals implemented after all other functionality was achieved.

The meowing was achieved using pulse-density modulation to play a pre-recorded meowing sound effect, sampled at $12\,\mathrm{kHz}$ and stored in BRAM.

Laser chasing was achieved by using the camera module and red chroma-keying to detect whether the robot could see a red laser dot, and if so, whether the dot was to the robot's left or right side. Because we were not displaying the camera frame, we were able to use a "racing the beam" approach to perform this calculation. This approach meant that a BRAM frame buffer (which would have been prohibitively memory-intensive) was not necessary.

As these additional behaviors were stretch goals, they are still somewhat limited in scope. Specifically:

- There is only a single, static meowing sound effect that the robot can produce.
- If the owner's voice is sufficiently high-pitched, then the meows may interfere with voice biometrics.
- Laser chasing does not work with non-red lasers or on brightly-colored terrains.
- The laser dot must be no further than $20\,\mathrm{cm}$ away from the camera.
- We did not have sufficient time to integrate the laser dot with the physical assembly and motor controls

Furthermore, we did not have enough time to integrate these behaviors into the main system, but we have verified that they work independently. We also do not expect the integration to be complicated.

IX. Evaluation of the System

Overall, the system's performance met our expectations set at the start of the project. We were able to meet our commitment (real-time audio localization, MFCC feature extraction, Bluetooth communications, and software-based inference), accomplish our goals (hardware-based inference and motor control), and implement two out of our three stretch goals (image recognition and audio playback). Nonetheless, there are still a few opportunities for further optimization.

A. Resource Usage

Table I shows the overall resource usage of the system (not including the contribution from laser dot chasing or meowing). A significant portion of the DSP blocks used was due to feature extraction, as each of the 32 triangular filters used a DSP block to perform multiplication. Given the large amounts of parallel, real-time digital signal processing performed by the system, this usage is significantly lower than what we initially expected.

Resource	Raw Usage	Percentage Usage
Slice LUTs	9861	30.25%
As logic	8848	27.14%
As memory	1013	10.55%
BRAM	13	17.33%
DSP blocks	81	67.50%

TABLE I
RESOURCE USAGE OF THE PROJECT, AS CALCULATED BY VIVADO.

This relatively low resource usage suggests that we could use the fully-pipelined FFT IP block (and other more resource-intensive IP configurations) to lower latency.

We also initially expected the IP used for sound localization and biometrics to be more memory-intensive, as we had envisioned (in our block diagram report) using BRAM to store the results from CORDIC in the sound localizer [16]. However, we ended up using a four-channel FFT and parallel processing instead of the BRAM because we had enough DSP blocks left over for it.

Finally, we were surprised that a single Urbana board had enough ports to accommodate a camera, four microphones, two DC motors, *and* an external speaker. This means that it would be possible to integrate our stretch goals without needing extra ports.

B. Latency and Throughput

We used $98.304\,\mathrm{MHz}$ clock throughout the design as it cleanly divides the audio sampling frequencies.

We were fortunate to be working with audio data which at most was being sampled at $64\,\mathrm{kHz}$ — much slower than the the $98.304\,\mathrm{MHz}$ system clock. As such, most modules have a latency much less than the period between successive audio samples. For example, filtering and windowing the audio incurs a latency of $71.2\,\mathrm{ns}$, which is negligible compared to the $0.25\,\mathrm{ms}$ between audio samples.

The localization module used AXI-Streaming to let it apply back pressure on the FFT. This back pressure ensured that no data was ever lost. For each FFT, the total latency came primarily from the two CORDIC calculations downstream. Each CORDIC had a latency of 20 clock cycles (that is, $203.5\,\mathrm{ns}$). With 216 samples, this gives the localization module a maximum throughput of 1 per $440\,\mu\mathrm{s}$ — again, less than the delay between audio samples. As such we could localize audio in real time.

Likewise, the biometrics module worked in real time because it was fully pipelined.

C. Accuracy

Throughout the project, we tested individual modules with simulations and groups of modules (for example, UART and I2S) with end-to-end simulations to ensure accurate transmitting and receiving of data.

We initially tested the localization and biometrics modules using Manta and LED indicators. Using these, we were able to detect between the trained voice and another person's voice. A limitation of both modules is the presence of false positives, especially when someone spoke at a similar pitch to the trained voice. These false positives were expected, based on only having positive data and how MFCCs work; despite this, the system was very accurate and had almost no false negatives.

The localization module struggled with pauses while people were talking due to the speed of the module. Once we implemented volume thresholding, though, it drastically improved its accuracy.

When fully integrated, the robot successfully identified its owner's voice, turned towards them, and navigated to them. When presented with another person's voice, it moved significantly less (often not moving at all). This behavior can be seen in the video submitted alongside this report.

X. Retrospective

This was our first time designing and implementing a large-scale digital system from scratch, so naturally, the project involved much experimentation to get things working. Many of our ideas worked well and made development easier, while some others created more problems than they solved.

Through this process, we learned the importance of careful planning, thorough testing, and a willingness to adapt our approach as needed. These lessons will undoubtedly guide our decisions as we continue to design new digital systems in the future.

A. Collaboration and Git

A key driver of success in this project was our highly effective teamwork and distribution of labor. In the early planning stages of the project, we designed the system architecture to be modular and parallelizable. This architecture (with a dependency graph shown in Fig. 13) maximized throughput because it ensured that nobody would be blocked by the other teammate's incomplete work.

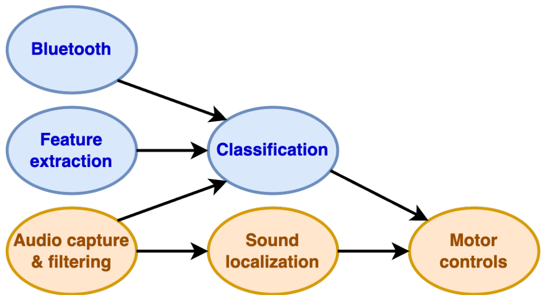

Fig. 13. The dependency graph of the system. Blue indicates modules implemented by Andi, and orange indicates modules implemented by Richard.

Work was assigned to leverage each team member's expertise. Richard had taken 2.007 (Design and Manufacturing) and 6.3100 (Dynamical System Controls), so he was well-suited for implementing sound localization and motor controls. Likewise, Andi had taken 6.3900 (Machine Learning) and 6.1220 (Design and Analysis of Algorithms), so he was well-suited for implementing biometrics.

As with any project involving shared code, Git was an invaluable tool for asynchronous teamwork and version control. Using separate branches, we were able to work on separate parts of the project simultaneously without interfering with each other's work. The Git history was also very useful for tracking down breaking changes that had gone unnoticed.

B. General Debugging Techniques

As expected, not everything worked on the first try. Writing thorough test benches, simulating code using iVerilog, and inspecting waveforms helped us catch the most egregious bugs and fix them before attempting to build the code. Although these test benches undoubtedly saved us from countless hours of debugging, there were still a few bugs that slipped through the cracks and required *in vivo* debugging.

To that end, we relied heavily on Manta [11] — an FPGA debugging tool that allowed us to capture signals on the FPGA using Python. In particular, the IO Core was invaluable as it allowed us to view in realtime the bins the localization was returning. We attempted to use Manta's Logic Analyzer Core at points to get more in-depth waveforms from our modules but were unsuccessful in this.

On a few occasions (most notably when debugging the servomotor controls and I2S microphones), we also used an oscilloscope to inspect the signals the FPGA was generating. This allowed us to track down why the servomotor wasn't initially working (using the board's 5V port to power the motor caused the FPGA to crash), and why the microphones initially were very quiet (the six most significant bits were always 1).

C. Integrating and Testing IP

One of the most time-consuming parts of this project was integrating and testing IP. Because of all the digital signal processing done by the FPGA, the project relied heavily on Xilinx's IP blocks; in total, it used seven different types of IP.

Although these IP blocks saved us a lot of development time by allowing us not to have to create our own FFT/CORDIC/etc. from scratch, they also introduced a host of problems that significantly slowed down our development velocity. In particular:

- The IP blocks (especially XFFT and the FIR compiler) were slow to compile on Vivado. This slowness meant that builds would take several minutes to fail, even from minor syntax errors like forgetting a comma.
- We were unable to simulate the IP locally using iVerilog, which was very frustrating in the project's early stages.
- Even when using Vivado to simulate IP, the code would often crash with a cryptic error message that would take a few hours to fix.
- Simulations of the IP blocks were slow and took upwards of an hour to complete in the case of the FIR compiler.
- The documentation was often poorly written and contained contradictory information.

It quickly became clear to us that we had severely underestimated the complexity of using the supposedly "plug-and-play" IP blocks. Consequently, if we had to redo this project, we would probably try implementing it without relying as heavily on IP, or at least build the project in Vivado's project mode so that it is easier to test the IP.

Nevertheless, we found the experience of working with IP to be a valuable learning experience because IP *is* used in industry, and the IP blocks we used afforded us much greater complexity than we likely could have achieved without them.

D. Integrating and Testing Peripherals

We encountered several challenges when integrating peripherals like the DC motors. The Urbana board includes four servomotor connectors, but there is no public documentation for the Urbana board. We eventually found documentation for the Boolean board (another RealDigital board), which specified two ways to power the motors.

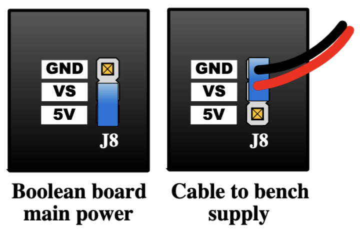

Fig. 14. Two ways of powering the servos on the Real Digital Boolean Board, adapted from [15]

Based on Fig. 14 we initially tried powering the servo from the board's 5V supply. Unfortunately, this setup did not work, even with the micro servos that we were using. While the servo would move once, it would then crash the FPGA, causing the flashed program to be lost. Eventually, we realised this flaw and powered the servo with an external 6V battery pack instead.

XI. Conclusion

Ultimately, the project was a success, and we are proud of what we have accomplished. Our *Feline Programmable Gate Array* successfully identified and moved towards its owner, even if it did in classic cat fashion — getting distracted along the way. We are grateful for the tremendous learning experience this project has bestowed, both in digital system design and in debugging a large hardware-based project.

Acknowledgments

The authors thank Joe Steinmeyer and Joseph Feld for providing valuable feedback during the design process, and the other 6.2050 Fall 2023 TAs and LAs for technical assistance with implementing the design. The authors also thank Alayah Hines for providing her voice as audio testing data.

Author Contributions

Both authors contributed equally to writing this report. Andi implemented voice biometrics and the additional cat-like behaviors, while Richard implemented audio capture, sound localization, and motor controls.

References

[1] Knowles. "I2S Output Digital Microphone." SPH0645LM4H-1. https://www.knowles.com/docs/default-source/default-document-library/sph0645lm4h-1-datasheet.pdf (accessed Nov. 19, 2023).

[2] NXP Semiconductors. "I2S bus specification." UM11732. https://www.nxp.com/docs/en/user-manual/UM11732.pdf (accessed Nov. 19, 2023).

[3] ATX7006. "Signal windowing for incoherent signals." http://www.atx7006.com/articles/dynamic_analysis/windowing (accessed Dec. 12, 2023).

[4] M. Siafarikas, T. Ganchev, N. Fakotakis, and G. Kokkinakis. "Overlapping wavelet packet features for speaker verification." in *Proc. Interspeech*, 2005, pp. 3121-3124.

[5] J. Lyons. "Mel Frequency Cepstral Coefficient (MFCC) tutorial." Practical Cryptography. http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/ (accessed Nov. 16, 2023).

[6] Real Digital. "Urbana Board." https://www.realdigital.org/hardware/urbana (accessed Nov. 20, 2023).

[7] Bleak. "Bluetooth Low Energy platform Agnostic Klient." https://bleak.readthedocs.io/en/latest/index.html (accessed Nov. 20, 2023).

[8] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python." JMLR 12, pp. 2825-2830, 2011.

[9] A. Rabaoui, M. Davy, S. Rossignol, Z. Lachiri and N. Ellouze, "Improved One-class SVM Classifier for Sounds Classification." *2007 IEEE Conference on Advanced Video and Signal Based Surveillance*, London, UK, 2007, pp. 117-122, doi: 10.1109/AVSS.2007.4425296.

[10] J. Feld, J. Poliniak. "Sound Localisation - Final Report" https://fpga.mit.edu/videos/2021/team13/report.pdf

[11] F. Moseley. "Manta: An In-Situ Debugging Tool for Programmable Hardware." Diss. Massachusetts Institute of Technology, 2023.

[12] XILINX. "CORDIC v6.0 LogiCORE IP Product Guide." https://docs.xilinx.com/v/u/en-US/pg105-cordic (accessed Dec. 13, 2023).

[13] J. Rajewski. "Detecting the Direction of Sound With a Compact Microphone Array", https://cs229.stanford.edu/proj2014/Rajewski,Detecting%20The%20Direction%20Of%20Sound%20With%20A%20Compact%20Microphone%20Array.pdf (accessed Nov. 22, 2023)

[14] Real Digital. "Servo Motors. An Overview of Servomotors and Servo Control". https://www.realdigital.org/doc/d0c3c99063edf31ea853d517847ac488 (accessed Dec. 13, 2023).

[15] Real Digital. "Boolean Board. Complete Reference Manual". https://www.realdigital.org/doc/02013cd17602c8af749f00561f88ae21 (accessed Dec. 13, 2023).

[16] Qu. A, Beattie. R. "Feline Programmable Gate Array, Block Diagram Report". https://docs.google.com/document/d/1t80gmxlvWl5ObLAPkVF3Igm7_9RtXCgvxwZP-1kXq6g/edit?usp=sharing (accessed Dec. 13, 2023).

Appendix

A. Mel Filterbank Implementation

For maximum , we developed a Python script that uses parameters like the number of filters to generate code for the triangular filters in the Mel filterbank. Because each filter's bandwidth is so narrow, a lookup table was used instead of direct multiplication to compute filter amplitudes.

Thanks to this lookup-table approach, each triangular filter used a single DSP block. Consequently, the entire Mel filterbank only used 32 DSP blocks (26.67% of what is available). We initially anticipated that the Mel filter would use much more resources, so we were able to be more aggressive with the other computation-intensive modules.

B. Logarithm Approximation Algorithm

We present an algorithm that leverages the slow growth of logarithms to approximate $\log_2(x)$ for arbitrary 32-bit numbers. Suppose we want to compute $\log_2(2^k + m)$ where $k \in \mathbb{Z}$ and $0 \leq m < 2^k$. Consider the first-order Taylor approximation of $\log_2(x)$ centered at $x = 2^k$:

$$\begin{aligned}\log_2(x) &= \log_2(2^k) + \frac{x - 2^k}{2^k} + \cdots \\ &\approx k + \frac{m}{2^k}\end{aligned}$$

This result suggests that $\log_2(x)$ can be approximated as a piecewise linear function with breakpoints at powers of 2. Thus, we get the following algorithm:

1) Compute k by repeatedly shifting x to the left until the uppermost bit is 1, while counting the number of shifts.
2) Shift x once more to the left, and take the resulting bits as the fractional part of $m/2^k$.

This algorithm is fast — it only uses bitwise operations and addition, so it can run in a single clock cycle on our FPGAs. It is also reasonably accurate — for any k and m, the approximation differs from the true value by at most 0.086.

C. Computing DCT via FFT

Suppose we want to compute the DCT of some sequence $a_1, a_2, \ldots, a_n$. Although Xilinx does not include a DCT module as IP in Vivado, the following algorithm adapts an FFT IP module to compute the DCT:

1) Concatenate the sequence with itself in reverse to get a sequence $a_1, a_2, \ldots, a_n, a_n, a_{n-1}, \ldots, a_1$ of length $2n$.
2) Interdigitate this sequence with zeroes to get a sequence $a_1, 0, a_2, \ldots a_n, 0, a_n, \ldots, a_2, 0, a_1, 0$ of length $4n$.
3) Take the FFT of this sequence and keep only the real parts of the first n coefficients. These n real numbers form the DCT of the original sequence.