

Waveform-Adjusted Visual Equalizer

1st Andy Zheng

Department of EECS

Massachusetts Institute of Technology
akzheng@mit.edu

2nd Christopher Liem

Department of EECS

Massachusetts Institute of Technology
cliem@mit.edu

3rd Julia Kozak

Department of Mathematics

Massachusetts Institute of Technology
juliakzk@mit.edu

Abstract—We present a design for a real-time, camera-controlled audio equalizer and visualizer implemented entirely on an FPGA. Using live video to track the user’s movement, the system maps motion to a flexible set of EQ control points, whose number and positions can vary dynamically. These control points are then interpolated with a Catmull–Rom spline to produce a smooth gain profile. Streaming stereo audio is transformed to the frequency domain via FFT, scaled by the frequency-dependent gains from the interpolated EQ curve and converted back with an IFFT before being sent to the speakers. The modified spectrum and current EQ settings are visualized on an external monitor, demonstrating how reconfigurable hardware can support low-latency multimedia interaction by processing video, audio, and graphics in parallel on a single device.

Index Terms—Digital systems, Field programmable gate arrays, Spline interpolation, Audio processing, Fourier transform

I. SYSTEM OVERVIEW AND IPs NEEDED

Figure 1 shows a high-level diagram of our top-level system. The main submodules of this system include:

1. **Vision.** Includes tracking user movement, and determining control point selection and deselection.
2. **Equalizer Curve Computation.** Includes curve interpolation from control points, into a frequency-to-gain function.
3. **Audio Processing.** Includes I²S transmitting and receiving of audio data, linear gain conversion, and sample processing within the frequency domain.
4. **Visualization.** Includes visualization of camera input streamed from SDRAM memory via DDR3, overlaid with the current equalizer curve and its control points. Additionally, display may be switched to show the live audio’s frequency vs. gain plot, for both unprocessed and processed audio.

We require the following IPs in audio and video processing:

1. **Fast Fourier Transform.** We utilize the LogiCORE FFT IP to convert 1024 24-bit batched audio samples into the frequency domain (1024 frequency buckets) [3].
2. **CORDIC.** We use two LogiCORE CORDIC IP modules to (1) in hyperbolic mode, compute linear amplitude gain from the frequency-to-gain (dB) function (16-bit), and (2) in square-root mode, compute magnitude of frequency-domain buckets, processed as complex values (48-bit) [4].
3. **BRAM.** We use Xilinx True Dual Port RAM (Single Clock) to instantiate BRAM, as well as FIFO wrappers

when clock domain crossing between audio and video signals.

4. **FIFO.** We use Xilinx XPM_FIFO_AXIS to instantiate AXI-Stream FIFOs, which handle clock-domain crossing and provide reliable, protocol-consistent communication with other AXI-Stream-based IP cores.

II. VISION

The vision module tracks the user’s finger and uses its position to update the control points of the equalization curve. These updated control points are passed to the curve interpolation module, which computes the new curve coordinates for real-time visualization.

A. User Movement Tracking

We used the source code from prior labs, storing high-definition raw camera footage into DRAM. To track a user’s movement, we will use let the user hold the pink foam that came with the FPGA kit. With the appropriate filter, we can estimate the center of where the user is holding the pink foam using the Center of Mass Module. A crosshair is overlaid on the live video feed so the user can see and adjust the pink foam’s position in real time. Since the measured center of mass from the filter is noisy, we averaged the past four (x_{com}, y_{com}) values to smooth the motion and stabilize the tracking.

B. Control Point Selection and Movement

We implement control-point selection with a debouncer-style module that uses event counters to detect when the center-of-mass coordinate stays near a control point for a sustained period. The point is then “selected,” highlighted with a box, and moved vertically as long as the center of mass remains inside the box. If the user moves away and causes the center of mass to leave the control point’s range depicted by the selection box, the point is deselected and the box disappears. Currently, we only allow users to move the control points vertically, so the user can deselect a control point by leaving the control point’s range horizontally. The new set of control points will be updated and fed into the curve interpolation modules once per frame.

III. EQUALIZER CURVE

The equalizer curve itself represents applied gain (in dB) as a function of audio frequency, with control points allowing for ease of editing to create custom audio filters such as

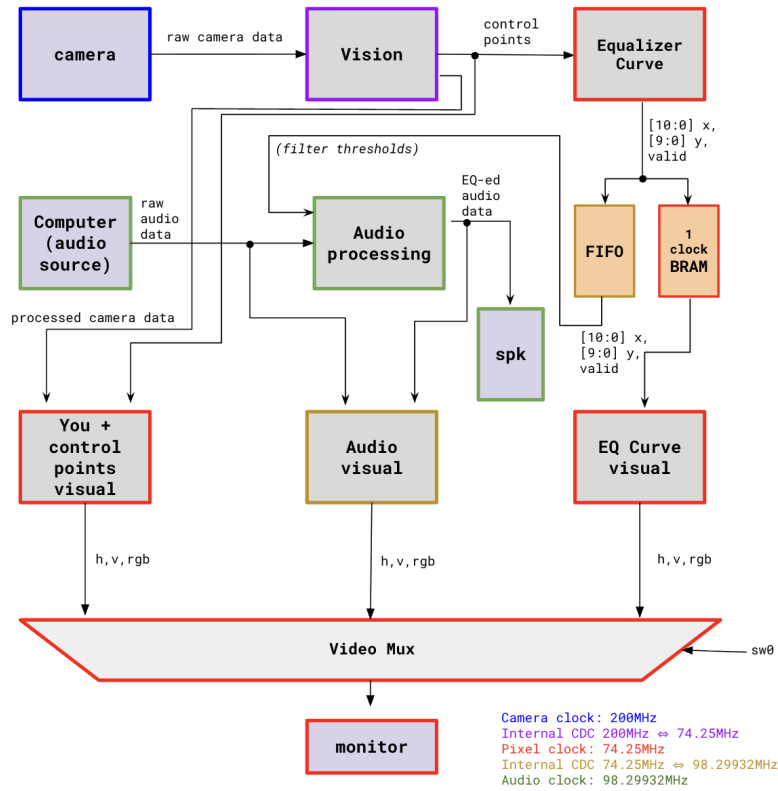


Fig. 1. Top-Level Diagram.

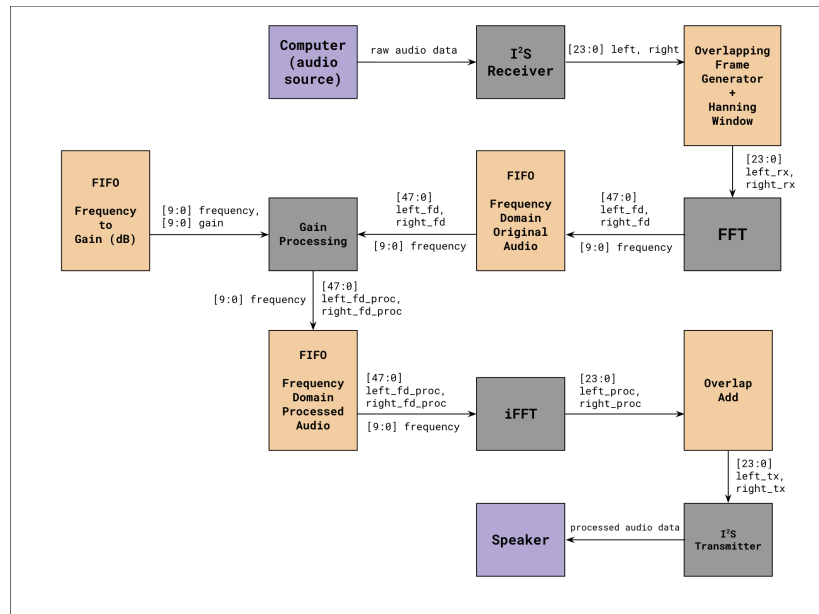


Fig. 2. Audio Processing Pipeline.

high-passing or bass-boosting. We interpolate a smooth curve passing through user-determined control points onscreen, from which we may convert gain in dB to its corresponding linear amplitude gain usable in the audio frequency domain during FFT processing.

A. Curve Interpolation and Sampling

For our system with n control points $p_1, \dots, p_n$ for $n \geq 4$, we choose to draw a curve which is piecewise cubic determined by each sliding-window of four consecutive control points. In total, we obtain $n - 3$ curve pieces $P_i(t)$ each taken on $t \in [0, 1]$. Standard cubic interpolation through four points is possible, though does not guarantee continuity between pieces or ease of editing, so we apply 4×4 spline basis matrix B before interpolating, so curve pieces satisfy:

$$P_i(t) = \sum_{j=0}^3 C_j(t) p_{i+j} \text{ where } [C_3(t), C_2(t), C_1(t), C_0(t)]^T = B \cdot [t^3, t^2, t, 1]^T.$$

We allow for either Bezier or Catmull-Rom interpolation, given by matrices

$$B_{\text{Bezier}} = \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}, B_{\text{CR}} = \frac{1}{2} \begin{bmatrix} 0 & 2 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 2 & -5 & 4 & -1 \\ -1 & 3 & -3 & 1 \end{bmatrix}$$

where Catmull-Rom is preferred as it provides C^1 continuity between curve pieces [1]. In hardware, this is achieved through staged multiplies with fixed-point representation to account for evaluating at fractional t . To evaluate a cubic in $t = c/2^k$, we evaluate monomials

$$c^3, c^2 \ll k, c \ll 2k, 1 \ll 3k$$

so that the final weighted and summed result may be right-shifted by $3k$.

This curve may only be sampled discretely (i.e. at values $t = c/2^k$), in which case we approximate the entire 1024-bit frequency-to-gain function by connecting sample points via polyline. A total of $T = 160$ sample points are stored in BRAM in 21-bit format $\{x, y\}$ for 720p screen resolution, taken along a curve with $P = 5$ pieces so that points are sampled at the multiples of $P/T = \frac{1}{32}$.

B. Polyline Rasterization

We use a common line segment rasterization algorithm named Bresenham's Line Algorithm in order to produce a y -value for each x -value up to 1023 from only the sample points [2]. It is a favorable choice for its use of few resources, notably no DSPs and few additional registers (derived from line segment endpoints `point0`, `point1`) which can be set up in one cycle. It is, however, variable-latency (based on the length of the segment drawn), and requires interfacing with BRAM to obtain new endpoints for each segment along the polyline. We model polyline rasterization with the FSM depicted in Figure 3.

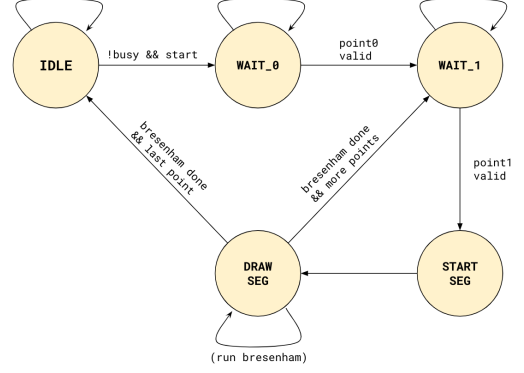


Fig. 3. Polyline Rasterization FSM.

From the DRAW SEG state, the module outputs points (x, y) along the polyline (indicated by a `valid_out` signal). These points are written into a dual-port BRAM addressed by x and containing value y , which may be read by visualization modules to directly display the function curve as well as audio processing modules in order to compute linear gain. Despite variable latency, one can expect even for a longer curve that the BRAM is fully updated within ~ 5000 pixel clock cycles, in which case we trigger `start` on `vsync` (once per frame).

C. Linear Gain Computation

Screen pixel values y must be read from BRAM and converted to their respective gain values g in dB, which is done via linear scaling:

$$g = -\frac{1}{32}y + 11.25$$

which produces a gain between ± 11.25 dB (close to the typical ± 12 dB on equalizer curves). For gain g given in dB, linear amplitude gain G is obtained by the following formula:

$$G = 10^{g/20} = e^{g \frac{\ln 10}{20}} \approx e^{g \cdot \frac{7}{64}}.$$

The LogiCORE CORDIC IP is used to indirectly compute G in hyperbolic mode, where on input t it outputs $\{\sinh t, \cosh t\}$ as two 16-bit floating-point approximations. In particular, it is useful that

$$\sinh t + \cosh t = e^t.$$

IV. AUDIO PROCESSING

The audio processing pipeline is visualized in Figure 2, consisting of the following components:

A. Audio Transmission

In order for the FPGA to receive and transmit analog audio signals, the Digilent Pmod I2S2 is used. To convert analog audio signals into digital signals for the FPGA, the Pmod I2S2 features the CS5343, which is a multi-bit audio A/D converter that receives signals through a 3.5mm stereo audio jack. Moreover, to convert the FPGA's digital signals into

analog audio signals for the speakers to play, the Pmod I2S2 also features a Cirrus CS4344, which is a stereo D/A converter that also receives signals through a 3.5mm stereo audio jack [5].

Information is communicated between the FPGA and the Pmod I2S2 using the I2S protocol. The I2S protocol features three clocks: the Master Clock (MCLK) which keeps everything synchronized, the Left-Right Clock (LRCK), which indicates whether data should be associated with the left or right audio channel, and the Serial Clock (SCLK), which determines when the serialized bits should be sent. In particular, left channel bits should be sent on the serialized data line when the LRCK is low and the right channel bits should be sent when the LRCK is high. Moreover, data should be clocked in (most significant bit first) on the falling edge of SCLK, and each channel's audio sample should be 24-bits wide. When the Pmod I2S2 is set to master mode, then the CS5343 is able to generate the LRCK and SCLK when fed a MCLK. However, the CS4344 is unable to do the same, so we have to generate all of the clocks for it manually.

To communicate with the Pmod I2S2, both I2S receiver and I2S transmitter modules for the FPGA are used. According to the Pmod I2S2 reference manual, feeding in a MCLK of 12.288 MHz results in a sampling rate (LRCK) of 48 kHz. To produce this MCLK, we use a clock wizard to generate a 98.29932 MHz clock and flip the MCLK signal every eight 98.29932 MHz clock cycles. This MCLK is fed into the CS5343, and our receiver module is synced with the LRCK and SCLK that the CS5343 generates. On the other hand, our I2S transmitter generates the LRCK such that the MCLK/LRCK ratio is 256x and the SCLK/LRCK ratio is 64x. This indicates that each LRCK signal is held for 32 SCLK cycles. Since each audio sample is 24-bits wide, the data line is padded with zeros for the last 8 SCLK cycles until the next LRCK edge. These clocks are fed into the CS4344, allowing full synchronization of audio data transfer between the FPGA and audio inputs and outputs.

To ensure that a sample from our audio source isn't lost, we allow our I2S receiver to continuously receive data, storing them in a 1024-depth 48-width AXI Stream FIFO for further processing: 24 bits for the left channel and 24 bits for the right channel. On the other hand, we forbid our I2S transmitter from sending invalid processed audio samples by running a state-machine with two states: *IDLE* and *RUNNING*. The transmitter receives its data from another 1024-depth 48-width AXI Stream FIFO which employs the handshake protocol to transfer data. When the transmitter is in the *IDLE* state, it is always ready to receive data and transitions into the *RUNNING* state once a handshake is reached. While in the *RUNNING* state, the transmitter is unable to receive data and sends the previous data sample under the I2S protocol. Figure 4 visualizes the clocks and the respective data lines in a pipeline without audio processing.

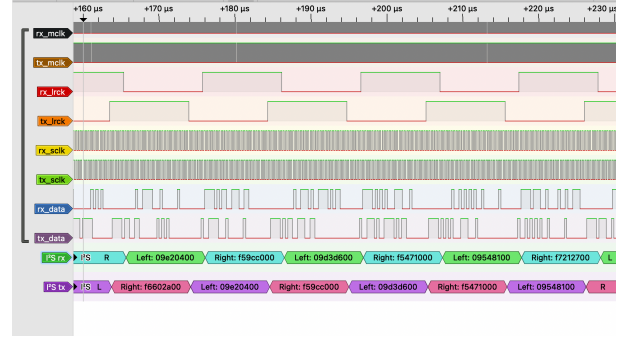


Fig. 4. Receiver/Transmitter clocks and their respective data lines. The FPGA transmits the audio samples during the next transmitter LRCK corresponding to the sample's channel and the last two hex digits of the data values are padded zeros.

B. Fast Fourier Transform (FFT)

To apply the gains computed in the previous section, the audio samples must first be mapped to the frequency domain. To accomplish this, a Fast Fourier Transform (FFT) IP generated by Vivado is used with the following configuration:

- Transform length: 1024
- Number of channels: 2
- Target clock frequency: 100 MHz
- Architecture choice: Radix-4, Burst I/O
- Data width: 24 bits
- Scaling option: scaled
- Output order: natural

The FFT IP is used because it efficiently computes the Fourier Transform using the Cooley-Tukey algorithm, which recursively divides the Discrete Fourier Transform (DFT) into smaller DFTs until they are trivial to compute. In the forward direction, the amplitude of the k th frequency bin $X(k)$ computed by the DFT is

$$X(k) = \sum_{n=0}^{N-1} x(n)e^{-jn k 2\pi/N} \text{ for } k = 0, \dots, N-1$$

where N is the transform size and $x(n)$ is the n th time domain sample. Similarly, the amplitude of the n th audio sample $x(n)$ computed by the inverse DFT is

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k)e^{jn k 2\pi/N} \text{ for } n = 0, \dots, N-1$$

Thus, N time domain samples map to N frequency bins, and vice versa.

Using the Radix-4 architecture, a size M DFT is decomposed into four interleaved DFTs of size $M/4$. This decomposition introduces radix-4 butterflies that combine four input samples at a time with appropriate twiddle factors. For $N = 1024 = 4^5$, this results in five stages for our algorithm. According to the LogiCORE documentation, the output of each butterfly can increase by a factor of $1 + 3\sqrt{2} \approx 5.24$, which is a growth of three bits. This motivates the use of a scaling schedule, which encodes how many bits to the right the

outputs of each stage should be shifted to prevent overflow. As recommended by the documentation, the conservative scaling schedule [10 10 10 10 11] is used to avoid any overflow; each pair of bits represents the number of bits to shift after each stage in binary.

Two instances of the FFT IP are instantiated: one for the forward FFT, which maps time domain samples to the frequency domain, and one for the inverse FFT, which maps frequency domain bins to time domain samples. The configuration bit which corresponds to the direction that an FFT instance computes is set at runtime: 1'b1 for the forward direction and 1'b0 for the inverse direction.

Stereo audio samples are streamed into the FFT IP through the AXI Stream protocol, where the data lines for both the slave and master axes are 96-bits wide: 48 bits for the left channel samples and 48 bits for the right channel samples. For each channel, 24 bits correspond to the real component of a sample and likewise for the imaginary component. Since the time domain samples are entirely real, the imaginary bits sent to the forward FFT are set to 0 while the real bits encode the values of the original audio samples. On the other hand, the frequency domain samples have both real and imaginary components which are sent to the inverse FFT for processing. Since our generated IP has two channels, both the left and right samples can be processed in parallel using the same IP instance.

C. Equalization

The equalization stage applies a gain to each of the 1024 complex frequency bins produced by the FFT. Although the control curve internally stores 1024 gain values, only the first 512 bins (0-24 kHz) correspond to audible frequencies. To ensure the full 1024-point control curve is still utilized, we map every even-indexed gain value from the control curve onto the 512 audible bins. This effectively samples the curve at twice the resolution for the audible range while preserving its overall shape. The remaining 512 bins (24-48 kHz), which are outside the range of human hearing, are still processed by assigning their gains using the property:

$$|X(k)| = |X(1023 - k)|, \quad 512 \leq k \leq 1023$$

which is required for correct reconstruction of real-value time-domain audio after the inverse FFT. This operation is orchestrated by a small FSM with two states, IDLE and RUNNING. While in the IDLE state, the gain processor monitors the FIFO that buffers the FFT outputs of the original audio stream. When the FIFO signals that a full batch of data is available, the FSM transitions to RUNNING and begins applying gain factors sequentially across the frequency bins.

After each bin has been scaled, the resulting complex frequency-domain samples are written into a dedicated Frequency-Domain Processed Audio FIFO. This FIFO feeds the inverse FFT (iFFT) module, which reconstructs the time-domain processed audio to be transmitted to the speaker system. Because the FFT module requires significantly more clock cycles to generate a batch of frequency bins than the

gain processor requires to apply the gains, the Frequency-to-Gain FIFO can be assumed to be fully populated before each FFT result batch becomes available.

FIFO structures are used throughout this stage for reliable buffering and clock domain crossing. In particular, the Xilinx FIFO IP integrates cleanly with the AXI-Stream interface used by the FFT module, ensuring correct synchronization and avoiding the need for manually writing BRAM signals.

D. Audio Batch Processing

The FFT processes the audio in batches of 1024 samples, but naively feeding in disjoint time-domain audio batches results in spectral leakage. Intuitively, the spectral leakage causes issues at the boundaries of the audio batches, introducing audible static artifacts whenever a batch is done processing. To resolve this, audio samples are batched into overlapping frames which are sent to the audio processing pipeline with smoothed transitions. To achieve this, three stages are necessary: creating the overlapping frames, applying a filter to smooth out the transitions between frames, and adding up the resulting equalized audio samples from the overlapping time domain regions.

1) *Overlapping Frame Generator*: The overlapping-frame generator converts a continuous audio stream into the overlapping batches specified before. In particular, samples are batched with 50% overlap with frames of size $N = 1024$, meaning that the last $N/2 = 512$ samples of the current frame will overlap with the first $N/2$ samples of the next frame. To achieve this, a finite-state machine with three states is used: IDLE, COPY, and EMIT. Incoming samples are written into a circular BRAM buffer of size N , which ensures that the most recent N samples are always being tracked. Moreover, two counters are used to keep track of (1) whether a full frame of N samples has been accumulated and (2) the number of new samples have arrived since the last frame was emitted. In the IDLE state, this circular buffer is populated and no output is produced by the module. To transition out of the IDLE state, at least N samples must exist and at least $N/2$ samples must have passed since the previous frame was emitted. At this moment, the current write pointer is saved for frame reconstruction, the counter tracking the number of new samples is reset, and the FSM enters the COPY state.

In the COPY state, the most recent N samples from the circular buffer are copied over to another frozen-frame BRAM of size N that stores the next frame to send. Using the stored write pointer, the module sends read requests to the circular buffer, copying the value at the specified address over to the frozen-frame buffer. This continues to happen until the frozen-frame buffer is fully populated with N samples. Once the last sample is stored, the FSM transitions to the EMIT state and the frozen-frame buffer is no longer mutated. It is important to note, however, that the circular buffer continues to be populated with incoming audio samples while in the COPY state. In terms of timing, the 48 kHz audio sample rate is substantially slower than the 98 MHz clock that this module

runs on, so data won't be overwritten before it is copied over to the frozen-frame buffer.

In the EMIT state, the module streams the frozen frame to its external ports using the AXI-Stream protocol. Read requests are sent to the frozen-frame BRAM in order and are only sent when a handshake is reached. When the final sample is sent, the TLAST signal is high for a single cycle, and the FSM returns to the IDLE state, repeating the entire process. This pipeline guarantees that frames with 50% overlap are accurately produced, which is visualized in Figure 5.

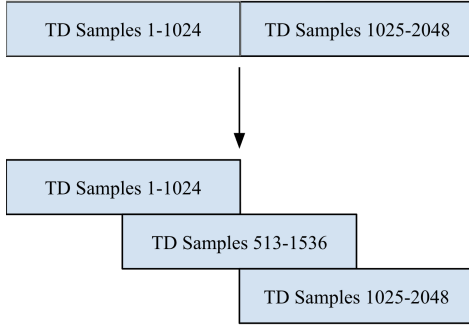


Fig. 5. The overlapping frame generator takes the time domain samples and batches them into overlapping frames, where 50% of the samples between consecutive frames are batched into both frames. Instead of batching the samples into the two disjoint frames, an intermediate frame containing half the samples from each frame is introduced.

2) *Hanning Window*: The Hanning window is a filter that minimizes spectral leakage by smoothly reducing the signal amplitudes towards zero at the beginning and end of each frame, which is visualized in Figure 6. In conjunction with the 50% overlapping frames and overlap-add, perfect reconstruction can be achieved without amplitude distortion or spectral artifacts.

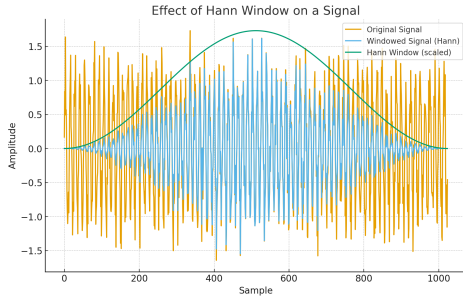


Fig. 6. The Hanning window "pinches" down the ends of a frame, introducing smooth transitions between our overlapping frames.

The Hanning window coefficient $w(n)$ for the n th time domain audio sample is given by

$$w(n) = 0.5 \left(1 - \cos \left(2\pi \frac{n}{N} \right) \right)$$

where $N = 1023$ and $n \in \{0, \dots, N\}$. To implement the Hanning window, the coefficients $w(n)$ are precomputed,

rounded and scaled up before being stored in a single-port read-only BRAM. In particular,

$$w(n) = \text{round} \left(0.5 \left(1 - \cos \left(2\pi \frac{n}{N} \right) \right) \cdot 2^{24} \right)$$

where $N = 1023$ and $n \in \{0, \dots, N\}$. The scaling by 2^{24} is necessary to produce noticeable results when applying the filter to the audio frames. These values are stored in a BRAM of size N with bitwidths 28, where four 0 bits are prepended to the binary representation of the coefficients to prevent issues with signed-math.

The module takes in audio samples from the overlapping frame generator under the AXI-Stream protocol, multiplies both the left and right channels by the corresponding Hanning coefficient, undoes the scaling from before by applying an arithmetic right shift of 24 bits, and streams the filtered output to the FFT for further processing. An internal counter is used to keep track of which sample of the frame is currently being processed.

3) *Overlap Add*: Following frequency-domain processing, the batched frames must be recombined into a continuous audio stream. Overlap-add reconstructs a continuous equalized audio output by adding the results of overlapping time-domain regions. The process is visualized in 7.

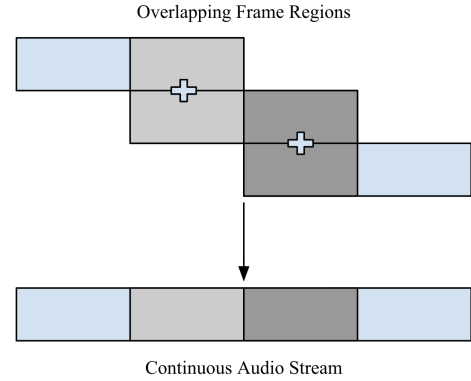


Fig. 7. The overlap-add module adds the equalized outputs of overlapping regions in consecutive frames to produce a continuous audio stream. The audio samples in the light-gray region are added together and the audio samples in the dark-gray region are also added together independently.

Samples are streamed from the inverse FFT to the overlap-add module through the AXI-Stream protocol. The inverse FFT produces both imaginary and real outputs, but only the real outputs are utilized in overlap-add. To perform the overlap-add operation, an internal counter is used to keep track of which sample in the frame is currently being received. This is necessary to decide whether the current sample should be stored. In particular, a BRAM of size $N/2$ to store the second half of each frame is maintained. When the sample index is greater than or equal to $N/2$, the current sample is stored into the BRAM at address $\text{idx} - N/2$. Conversely, when the index is less than $N/2$, the module retrieves from the BRAM the sample stored during the second half of the preceding frame. This value represents the previously computed, overlap-aligned contribution that must be added to the current sample

to reconstruct the equalized original signal. In this case, the value read from the BRAM is added to the current sample and is streamed to the FIFO bridging the overlap-add module with the I2S transmitter. In particular, the reconstructed signal $\{r'[n], l'[n]\}$ is given by

$$r'[n] = r_{\text{prev}}[n] + r_{\text{curr}}[n], \quad l'[n] = l_{\text{prev}}[n] + l_{\text{curr}}[n]$$

where $r_{\text{prev}}[n]$ and $l_{\text{prev}}[n]$ are the right and left channel values read from the BRAM and $r_{\text{curr}}[n]$ and $l_{\text{curr}}[n]$ are the values most recently received by the module. Since audio is played at 48 kHz and the overlap-add module runs on a 98 MHz clock, this process produces a continuous audio stream for the I2S transmitter to receive.

V. VISUALIZATION

To visualize the control curve, we store the 1024 (x, y) points from the curve interpolator in a BRAM, using x as the address and y as the stored value. To render the curve in the correct screen position, we pipeline the video timing signals (h_count_hdmi , v_count_hdmi). The BRAM is then addressed by $h_count_hdmi - 128$, which offsets the curve to be centered on the screen. Since the horizontal resolution (1280 pixels) exceeds the curve's 1024 points, the curve is only visible for $h_count_hdmi - 128$ values between 128 and 1151. We will use a similar approach to display the original and processed stereo audio in the frequency domain. In this case, each of the 1024 (x, y) points will represent the magnitude of the complex values produced for each frequency bin by the FFT and iFFT modules, rather than the values generated by the curve interpolator.

A. Equalizer Curve Visualization

To ensure that the interpolated curve is rendered with visually uniform thickness, we implement a single 4×4 vertical neighborhood check during rasterization. At each pixel column, the visualization module receives the most recent five sampled y coordinates of the curve, stored in a five-entry buffer $\{y_0, y_1, y_2, y_3, y_4\}$. For the current video position ($h_{\text{hdmi}}, v_{\text{hdmi}}$), we compute the signed vertical differences:

$$d_i = v_{\text{hdmi}} - y_i, \quad i = 0, 1, 2, 3, 4$$

and determine whether any d_i lies within a smaller tolerance window. If $|d_i| \leq 4$ for any i , the pixel is classified as lying "near" the curve is colored white; otherwise, the pixel is drawn as background from the camera feed.

B. Audio Visualization in Frequency Domain

To visualize the audio spectrum, we obtain the complex-valued frequency bins from the FFT modules for both the original and equalized stereo audio streams. Each bin contains real and imaginary components corresponding to the Fourier coefficients. We compute the magnitude of each complex coefficient using the LogiCORE CORDIC IP in square-root mode, which implements

$$|X(k)| = \sqrt{\text{Re}(X(k))^2 + \text{Im}(X(k))^2}$$

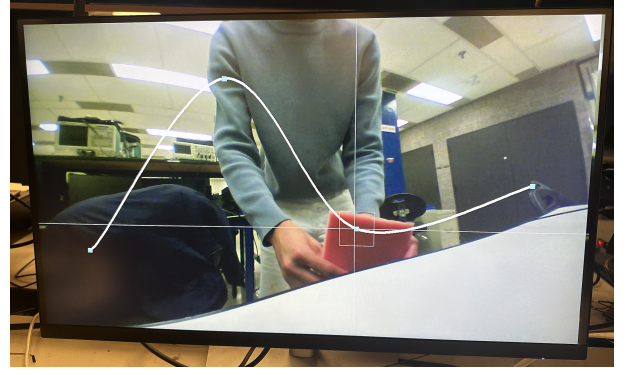


Fig. 8. Curve Visualization with Control Point Selection

and provides an efficient hardware approximation without consuming DSP resources. The resulting magnitudes represent sound pressure in the frequency domains. For visualization, we truncate the magnitude output to its top 10 bits, yielding a value scaled to the display's vertical resolution. These values are then plotted column-wise as histograms on the screen to render the real-time spectrum for both the original and processed audio.

For visualization, the original stereo audio is shown in red and the equalized stereo audio in blue, with overlapping regions rendered in purple. Because only the lower half of the FFT spectrum (0-24kHz) and the control curve is only used to provide unique gains for the lower half of the audio spectrum, we display only the 512 magnitudes. A sample control curve and its corresponding processed stereo audio along with the original stereo audio are shown in Figures 9 and 10.

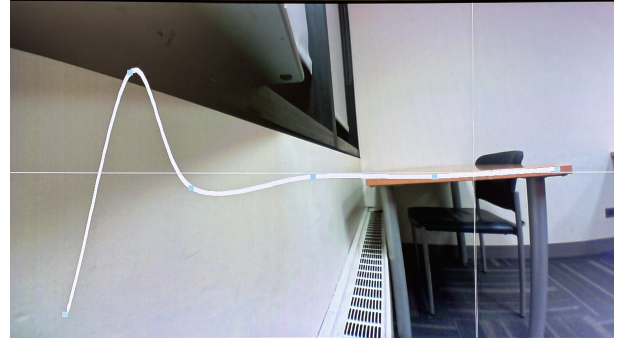


Fig. 9. Sample Control Curve

From the visualization, we observe that the lowest frequencies for both the left and right audio are attenuated by the control curve: the original audio in red is much more prominent in this region, while the equalized audio in blue is nearly absent, producing a purple bar that is much lower than the red component. In the next frequency range, the sharp peak in the control curve leads to noticeable blue spikes, indicating that these frequencies have been amplified in the processed audio. Beyond this region, the control curve flattens, and the corresponding portions of the spectrum show the original and

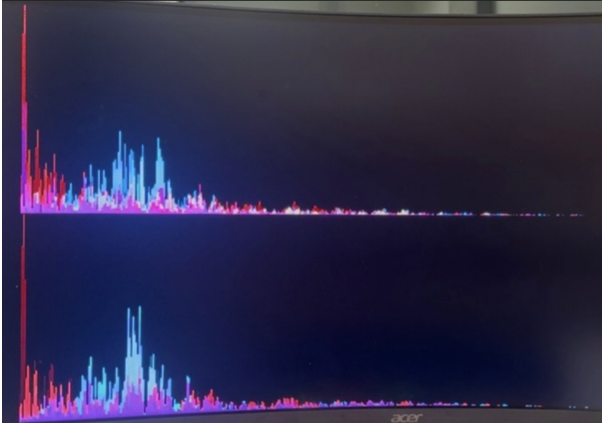


Fig. 10. Audio Snapshot Corresponding to the Sample Control Curve

processed audio at comparable levels, reflecting that these frequencies have been left unchanged.

VI. DESIGN EVALUATION

A. Design Choice

Before implementing anything on hardware, we implemented a proof-of-concept audio equalizer using Python. We tested out three different design choices: FFT based filtering without overlap add, FFT based filtering with overlap add, and Infinite Impulse Response (IIR) filtering. For all three methods, we used an interpolated Bezier curve to choose the gains, which mimics our final hardware design setup.

To implement FFT-based filtering without overlap added, we batched samples in frames of 1024 and processed them in the frequency domain, similar to our final design. However, the audio outputs had substantial static that overpowered the audio when listened to closely. However, those listening from afar were unable to pick up the static, possibly due to the restricted range of frequencies that humans can hear.

On the other hand, we also implemented IIR filtering by creating low-shelf, high-shelf, and band-pass filters for the different frequency bins by computing the coefficients according to the Robert-Bristow Johnson cookbook for audio equalizers. This eliminated the static completely, but it defeated the purpose of having the vision-controlled gain curve since we essentially computed a different gain control curve in creating the biquad filters. We opted out of this approach.

To compensate for the static, we also tested the FFT-based filtering with overlap add, which smooths out the transitions between frames to eliminate static. The audio output achieved was comparable to the IIR-based filtering, so we ended up choosing this approach as the final design.

B. Resource Utilization

With our final product, we still have substantial BRAM and DSP resources available. This is largely because the high-definition video stream is buffered in external DRAM

rather than on-chip BRAM. Additionally, our use of hardware-efficient, shift-based modules such as CORDIC avoids consuming DSP slices during computation. Our estimated resource usage in these two categories is summarized below:

Resources	Available	Current Usage
DSP	120	79
BRAM	75	46.5

Based on the post-place utilization report, the vision and equalizer-curve modules currently consume approximately 5.5 BRAMs, while the audio-processing pipeline uses an additional 25 BRAMs. We also store both the original and processed audio in the frequency domain for visualization. With left and right channels represented as 48-bit complex values, the required storage per frame is $96 \cdot \frac{1024}{36000} \cdot 2 \approx 5.46$ BRAMs. So we allocate approximately 6 BRAMs for this stage. Finally, implementing overlapping add along with Hanning Window between audio batches requires an additional 10 BRAMS for audio buffering and coefficient storing. For DSP resources, the curve-computation module uses 19 DSP slices, the FFT requires 36 DSP slices, and the gain-processing, magnitude-computation, and audio-optimization components together require an additional 12 DSP slices.

C. Future Directions

In its current form, our design represents a viable implementation of a real-time visual equalizer that meets timing by 0.265 ns, leaving space for additional processing stages or future architecture refinements. The system supports six user-controlled points on the equalizer curve, with more control points concentrated on the lower half of the frequency range, as they contribute to most of the audio. For audio optimization, we incorporated both a Hanning window before the FFT and an overlap-add reconstruction stage after the iFFT. These additions reduce spectral leakage and smooth transitions between consecutive audio frames, resulting in noticeably improved audio quality to the preliminary design.

Despite these improvements, some ringing and static noises remains at certain frequencies. Several further enhancements could mitigate these artifacts. A longer FFT size would provide finer frequency resolution, reducing the spread of energy across neighboring bins. A more dynamic windowing strategy optimized for low-ringing applications could also be helpful with this issue. Another challenge arises from the pink foam used for motion tracking. Background objects with similar colors can interfere with segmentation, causing the center-of-mass estimate to drift or become inaccurate. In addition, the camera exhibits noticeable distortion near the edges of the frame, making it difficult for users to precisely manipulate control points located near the boundaries of the equalization curve. Allocating time to develop a more robust color-filtering pipeline and using more easily trackable luminescent markers would substantially improve reliability and overall user experience.

Our final design met the commitments and goals outlined on our checklist. We achieved real-time equalized audio output and visualized both the original and equalized signals in the frequency domain using an EQ curve interpolated through six control points. We also visualized user movement through the center-of-mass tracker. For our reach goals, we optimized the audio pipeline, and our design is also easily configurable to accept microphone input instead of streamed audio. Although our control-point logic does not currently support horizontal movement, we intentionally disabled that feature to keep deselection simpler.

VII. REFLECTION

Overall, we are happy with our top-level system design and its effective use of resources. We were able to reduce memory usage by cutting out usage of a video frame buffer early on, which gave us flexibility to use AXIS FIFO and BRAM modules in the audio-visual pipeline when clock-domain crossing. Looking back, the development process taught us that:

- Working together is very effective. We began the project by each individually working on different pipelines, which created some challenges with integration and version control. Ultimately, working together in the lab worked best for us, and helped to ensure our top-level design was making progress.
- Integration is a bigger challenge than expected. In our case, our top-level design involved a lot of clock-domain crossing as well as different streaming and memory protocols (both AXIS and direct addressing with BRAM), which added unexpected complexity when writing our top-level module. Overall, we would have preferred to anticipate this, and start on integration early.
- Agreeing on consistent protocols is important. We ended up needing to refactor our gain processing module to use an AXIS protocol rather than BRAM after reconsidering audio pipeline design choices, which caused a bit of a mess. It would have saved us time to finalize the design beforehand.

VIII. AUTHOR CONTRIBUTIONS

The authors would like to thank Kiran Vuksanaj and Joe Steinmeyer for their feedback on design choices and paper writing, and for providing the Pmod I2S2 and OV5640 modules. They would also like to thank Professor Dennis Freeman for his technical guidance on audio signal processing, introducing key techniques such as overlap-add.

Julia designed and wrote equalizer curve modules (spline interpolation and polyline drawing), curve rendering/neighborhood checking, linear gain computation with CORDIC, and gain magnitude calculation with CORDIC. Julia also created report block diagrams.

Andy refactored code from earlier labs by removing unnecessary components to reduce memory usage and shorten critical paths for the purpose of this project. Andy also

designed and implemented modules responsible for control-point selection, deselection, control-point movement, and gain processing for the audio samples produced by the FFT.

Christopher implemented the I2S receiver and transmitter modules and generated the clocks necessary to cooperate with the Pmod I2S2 device. Moreover, Christopher designed the audio processing pipeline to use AXI-Stream and combined the I2S modules with the LogiCORE FFT IP to achieve perfect reconstruction prior to gain processing. Finally, Christopher implemented the overlapping frame generator, Hanning window, and overlap-add modules to reduce spectral leakage in audio output.

All authors collaborated on the audio visualization component.

IX. SOURCE CODE

Our source code is available on GitHub, linked here: <https://github.mit.edu/6205F25/fa25-6205-team07>.

REFERENCES

- [1] Catmull, E.E., & Rom, R. (1974). A CLASS OF LOCAL INTERPOLATING SPLINES. *Computer Aided Geometric Design*, 317-326. <https://api.semanticscholar.org/CorpusID:118383557>.
- [2] Bresenham, J.E. (1965). An algorithm is given for computer control of a digital plotter. *IBM Systems Journal*. <https://ibm-1401.info/Pics3/bresenham1965.pdf>.
- [3] *Fast Fourier Transform v9.1 LogiCORE IP Product Guide*. Vivado Design Suite. (2022). https://www.xilinx.com/support/documents/ip_documentation/xfft/v9_1/pg109-xfft.pdf.
- [4] *CORDIC v6.0 LogiCORE IP Product Guide. AMD Technical Information Portal*. (2021). <https://docs.amd.com/v/u/en-US/pg105-cordic>.
- [5] *Digilent Pmod I2S2 Reference Manual* <https://digilent.com/reference/pmod/pmodi2s2/reference-manual?redirect=1>