

Configurable Audio Reactive Output LEDS (CAROL)

Kevin Reyes Palma

Department of Electrical Engineering and Computer Science
Cambridge, MA, USA
kreyesp@mit.edu

Max Wong

Department of Electrical Engineering and Computer Science
Cambridge, MA, USA
mwwong@mit.edu

Abstract—This paper presents Configurable Audio Reactive Output LEDS (CAROL), an FPGA-based real-time audio visualization system that transforms a 48 kHz audio stream into LED facial expressions and HDMI visualizations. The system implements a custom 512-point radix-2 FFT achieving a complete transform in 29.1 μ s. Audio input and output are handled through I2S protocol. The extracted frequency magnitudes are accumulated into low, mid, and high bands that are used to control a face selector state machine. Simultaneously, the system renders real-time waveforms and frequency bars with user-adjustable thresholds on an HDMI display. Comprehensive verification through Cocotb simulation and hardware testing validates both algorithmic correctness and real-time performance on the FPGA.

I. INTRODUCTION

Configurable Audio Reactive Output LEDS (CAROL) is an FPGA-based audio visualization system designed to react to music in real time. The system receives audio through an I2S interface, performs digital signal processing using a custom 512-point FFT, and generates facial expressions and other visualizations on an LED display board. Simultaneously, the real time audio waveform and extracted threshold information are displayed on an HDMI monitor.

To build this system, we developed several custom modules, including an I2S receiver/transmitter, a frame buffer, an FFT module, and a wave visualizer. This report summarizes our system architecture, final progress, evaluation, and implementation insights.

II. SYSTEM OVERVIEW

A. Dataflow Description

Incoming stereo audio enters through the Digilent Pmod I2S2 (Revision A), converting the analog audio signal into 24 bit samples. The I2S receiver deserializes the samples and writes them into the BRAM frame buffer. Once 512 samples have been stored, the FFT Core loads them into the Ping-Pong BRAM and performs a 512-point fixed-point FFT. The resulting frequency-domain magnitudes are accumulated into low, medium, and high bins. These bins are compared against user-defined thresholds to control LED facial expressions, as well as rendered as bar graphs on an HDMI display. In parallel, the waveform renderer displays the audio signal on HDMI. Finally, the samples are passed through the I2S transmitter and sent to the DAC for audio playback.

B. Physical Hardware

The system is implemented on the AMD Urbana FPGA. Audio input and output is handled by the Digilent Pmod I2S2 (Revision A), which integrates the Cirrus CS5343 ADC and Cirrus CS4344 DAC into a single device with 3.5mm connections. The audio visualizations are displayed on an array of 4 8x32 WS2812B LED Panels.

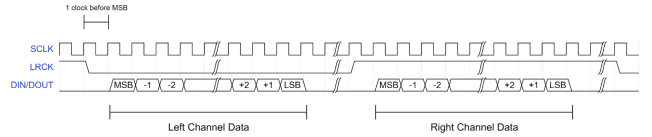


Fig. 1. I2S Protocol Example [1]

III. AUDIO I/O PIPELINE

A. ADC/DAC Hardware

CAROL uses Digilent Pmod I2S2 (Revision A) to handle conversion from analog to 24 bit audio input and output at 48kHz sampling rate. Incoming audio from the 3.5mm source is digitized by the ADC and delivered to the FPGA via the I2S protocol. After passing through the FPGA, the audio is passed through the DAC for playback to a 3.5mm connected speaker.

B. Clock Generation

The ADC and DAC require a 12.288 MHz master clock (MCLK) to operate at 48kHz as specified in their documentation. To support this, the onboard 100 MHz clock is used to produce a 98.304 MHz system clock via the Clock Wizard IP. A simple counter uses this clock to generate the 12.288 MHz MCLK for the ADC and DAC. A second IP is used to generate the 74.25 MHz and 371.25MHz clocks needed for HDMI and TMDS serialization.

C. I2S Protocol Description

The I2S interface uses four signals: a master clock (MCLK), a serial clock (SCK), a word-select signal (WS), and a serial data line (SD). MCLK sets the sampling rate for the ADC and DAC, while SCK determines when audio bits are transmitted/received. WS indicates which channel is currently

being transmitted, with half of one WS period corresponding to one channel.

D. I2S Receiving

Our module takes in WS, SCK, and SD. We model the protocol as a FSM, where we have states RECEIVE_L, RECEIVE_R, SKIP_FIRST_L, and SKIP_FIRST_R. The ADC sends in valid data on rising edge of SCK, and we sample on the falling edge. When a channel is done, WS changes, and we skip one SCK cycle before starting collection again. Once we collect one channel, we send out a single cycle valid signal to our system. We then place the value in corresponding left and right BRAMs of width 24 bits and depth 1024.

E. I2S Transmitting

Our module takes in valid stereo data from our BRAMs, and uses a FSM to transmit. We have states: IDLE, START, TRANSMIT_L, TRANSMIT_R. The IDLE state places valid data in buffers that we use for transmitting. The START state skips 1 SCK cycle between WS change, and the other two states transmit data. We transmit 24 bits of a channel and then hold low until it is time to change WS. With a counter, we generate our own SCK and WS signals, 3.072 MHz and 48 KHz respectively, that the DAC will sample based on.

F. Verification

The audio I/O pipeline was verified through a combination of simulation and hardware tests. Cocotb was used to test and debug our I2S modules, which revealed incorrect samples due to the invalid bits. After passing in simulation, the ADC/DAC and logic analyzer was connected to PMODA and PMODB to test audio play-through on hardware. These tests validated the correctness of our receiver and transmitter.

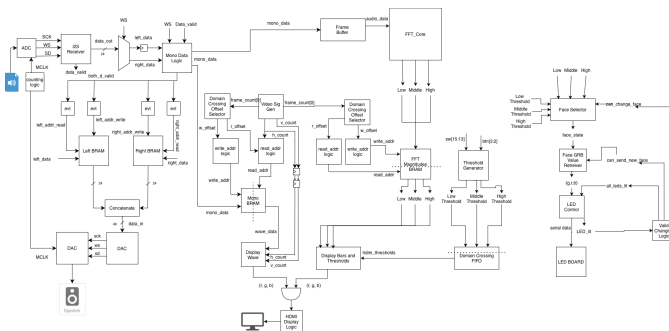


Fig. 2. CAROL Block Diagram

IV. FFT OVERVIEW AND IMPLEMENTATION

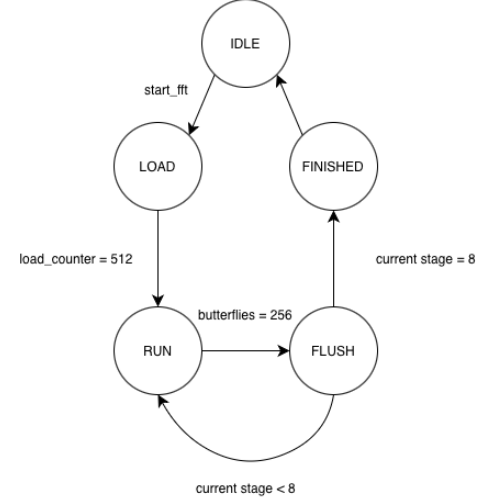


Fig. 3. FFT State Machine

A. Design

The central challenge of CAROL is performing real-time frequency analysis on a continuous 48 kHz audio stream. A 512-point FFT was selected to balance resolution and complexity, providing 256 usable frequency bins over a range of 0 Hz to 24 kHz [2]. At a sampling rate of 48 kHz, new samples arrive every $20.8 \mu\text{s}$ meaning computation must occur within 10.6ms to prevent overwriting any of the 512 samples.

A custom FFT implementation was deliberately chosen to maintain control over the data flow throughout the system. The FFT was parametrized for number of points, data size, twiddle size, and fixed point representation. This provided flexibility in how data is loaded, how butterflies are performed, and how the frequency bins become available to the rest of the system, avoiding overhead in formatting input/output and interfacing with IP.

B. Fixed-Point Format

Fixed point arithmetic is used throughout the FFT pipeline to ensure the device remains efficient and simple. For CAROL, the audio input does not need the dynamic range a floating-point representation provides. Rather, the input samples are well-suited for conversion to fixed-point representation. Additionally, fixed-point arithmetic maps directly to simple integer operations, with addition and multiplication requiring minimal adjustments to align the implied binary point.

Fixed-point numbers are represented in the Qm.n format, where m denotes the integer bits and n denotes the fractional bits as shown in Fig.4. The default parameters for the FFT pipeline define a Q16.8 representation. This format provides a range of $\pm 32,768$ with precision of 2^{-8} . The default parameters for the twiddle factors define a Q2.3 representation. Since twiddle factors are bounded in magnitude by the unit circle, only 2 integer bits are needed to represent a magnitude of ± 1 . This allows for reduced memory storage at the expense

of some precision loss. In both cases, however, these representations are parameterized to allow for experimentation with different precisions.

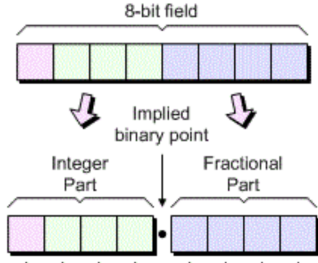


Fig. 4. Fixed-Point Representation [3]

C. Butterfly Module

The butterfly operation is the fundamental computational unit of the radix-2 FFT algorithm. Each butterfly takes two complex inputs (X_0 and X_1), one complex twiddle factor (W), and produces two complex outputs according to:

$$\begin{aligned} Y_0 &= X_0 + WX_1 \\ Y_1 &= X_0 - WX_1 \end{aligned}$$

To support this operation in hardware, the inputs and outputs are separated into their real and imaginary parts. The complex multiplication step expands into four real multiplications:

$$\begin{aligned} \text{Re}(WX_1) &= \text{Re}(W)\text{Re}(X_1) - \text{Im}(W)\text{Im}(X_1) \\ \text{Im}(WX_1) &= \text{Re}(W)\text{Im}(X_1) + \text{Re}(X_1)\text{Im}(W) \end{aligned}$$

To support fixed-point computation, the module widens intermediate bit widths to avoid overflow errors. The products are then shifted by the number of twiddle fraction bits to align the decimal point, with extra bits added to prevent overflow errors. Saturation logic clips any value exceeding the maximum representable value.

To meet timing requirements, DSP blocks are explicitly instantiated for the butterfly multiplications. Additionally, the module is pipelined into multiplication, addition, and saturation stages. This pipelined architecture, combined with the DSP usage, eliminates timing violations and maintains correctness.

D. FFT Core

The FFT core implements a 512-point Cooley-Tukey algorithm using an iterative architecture. The FFT is broken down into stages, with $\log_2(512) = 9$ stages needed. During each stage, butterfly operations are performed on pairs of samples 2^{s-1} distance in memory, as shown in Fig.5

The FFT core is controlled by an FSM with states for loading, running a stage, flushing a stage, and completing the FFT. During the load stage, samples are written into the BRAM in bit reversed order to align with standard Decimation in Time format. In the computation phase, the controller iterates through all butterfly indices, generating the appropriate memory addresses and twiddle factor indices. The butterfly operations are computed using these addresses until all 256

butterfly operations for that stage have been initiated. Due to the latency of the butterfly module, a flushing state is necessary to write the remaining butterfly outputs to memory before proceeding to the next stage. In the final stage, the FFT begins summing the desired frequency bins. Due to the Nyquist theorem, the frequency range of the FFT is 0 to 24 kHz, spread among the first 256 bins. Thus, bin i represents $i * 93.75$ Hz, which yields bins 1-5, 6-40, and 41-100 for the low, middle, and high frequencies, respectively. When the FFT is completed, it enters its final state to indicate completion and valid outputs.

A ping-pong BRAM architecture is used to prevent overwriting information. Two 512-entry BRAMs each storing 48 bit words (24 bits real, 24 bits imaginary) are used each cycle, with one being read while the other is written to, alternating each stage.

Twiddle factors are precomputed in Python according to $\cos(-2\pi k/N) + j \sin(-2\pi k/N)$ and read from memory.

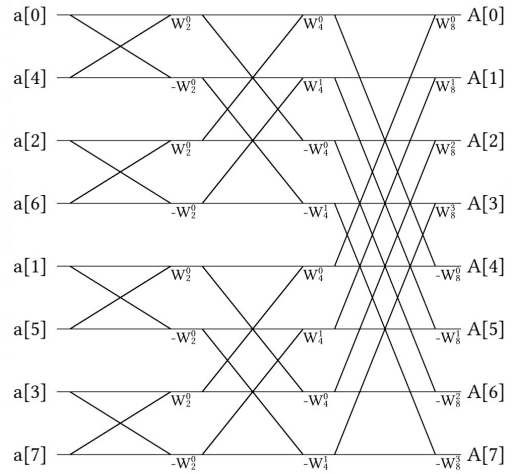


Fig. 5. Radix-2 8 Point FFT [4]

E. Verification

The butterfly module was tested extensively using randomized complex inputs and edge cases designed to expose indexing, sign, and overflow errors in the fixed-point arithmetic. These tests included values near the limits of the representable range, purely real and purely imaginary inputs, and random combinations of twiddle factors. This identified several issues early in development, including incorrect bit slicing, swapped real-imaginary components, and overflow. Iterative debugging based on these results ensured that the butterfly module produces correct outputs across all inputs.

Verification of the FFT core began by validating the data loading mechanism. A debug parameter was added to the FFT core that allowed it to load samples into BRAM and immediately return to IDLE, enabling inspection of BRAM contents to confirm correct sample storage and bit-reversed addressing. Once loading was verified, a comprehensive cocotb test bench was developed to test the full FFT module. The test bench

generates 512 fixed-point input samples using Fxp, initiates the FFT computation, waits for the done signal, and compares the output against a NumPy reference. Test samples included sine waves, impulse signals, and random distributions. Each test type revealed different aspects of FFT behavior. Sine waves produced predictable single-peak outputs that validated frequency bin alignment. Impulse signals exposed magnitude scaling issues. Random distributions stressed the full pipeline and revealed accumulation overflow conditions. Finally, the FFT was validated against real audio samples from WAV files. For each test case, correctness was verified by visual comparison of frequency bin magnitude plots, numerical comparison of individual bin values, and correctness of the accumulated low, mid, and high frequency range outputs. Importantly, the low, middle, and high magnitude values are computed using the approximation $|re|+|im|$ to avoid additional combinational delay. This introduces a max overestimation of $\approx 41\%$ but provides sufficient accuracy for audio visualization while meeting timing requirements.

The FFT was integrated into the top-level systematically, with test points added at each stage of the pipeline. Intermediate signals were monitored, including I2S sample reception, BRAM write operations, BRAM read operations, butterfly module outputs, and final accumulator outputs. This incremental approach allowed early detection of issues such as pipeline timing mismatches and signed-math errors.

Final verification on the FPGA involved displaying the low, mid, or high frequency magnitudes on the seven-segment display and comparing against expected values using the same input data.

V. VIDEO AND DISPLAY SYSTEM

A. Waveform Renderer

The HDMI system follows the same TMDS pipeline used in earlier labs, but extended with a real-time audio waveform renderer. Mono audio samples are written into BRAM at sampling frequency on the system clock and read on the 74.25 MHz clock. To support reliable cross-clock operation, the BRAM is divided into distinct read and write sections, with an offset pointer determining the active section. The offset pointers update based on a new frame signal that is generated on the 74.25 MHz clock. We do a domain crossing by using a double buffer to avoid metastability. The display module then extracts the upper 8 bits of the audio data to determine positioning, calculating the y-coordinate relative to the waveform center position. Pixels are colored red when their y-offset is equal to v_count , as seen in Fig.6

B. Frequency Bar Visualization

In addition to the waveform, the system displays vertical bars representing the low, mid, and high frequency magnitudes extracted by the FFT. Each bar's height is proportional to the accumulated energy in its frequency range, with user-controlled thresholds rendered as horizontal green lines overlaid on the bars. This visualization enables real-time adjustment of the LED response thresholds. The frequency magni-

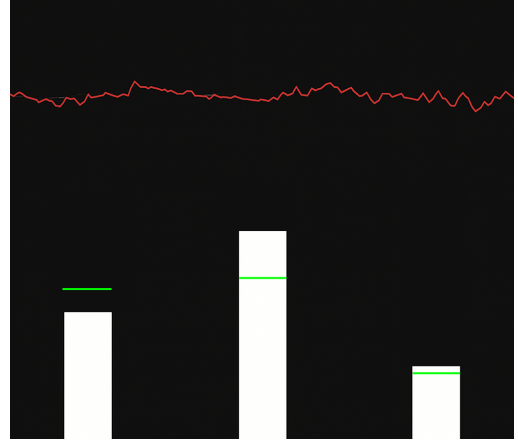


Fig. 6. Sound wave and frequency bars visualized on HDMI monitor

tude data requires clock domain crossing from the 98.304 MHz computation domain to the 74.25 MHz pixel clock domain. A dual-port BRAM is used in a ping-pong configuration, with one section written at 98.304 MHz while the other is read at 74.25 MHz, alternating on frame boundaries to ensure data stability. Additionally, to prevent rapid flickering, the low, middle, and high magnitudes pass through simple moving average filters before display.

C. Threshold Generation

Three independent thresholds are implemented for the low, mid, and high frequency ranges. Threshold selection is controlled via switches 15, 14, and 13, while buttons 3 and 2 adjust the selected threshold upward or downward, respectively. To allow smooth adjustments during button holds, threshold updates are limited to occur at most once every 1/32 of a second. Threshold generation operates on the 98.304 MHz clock domain, allowing direct comparison with FFT outputs for face selection logic without additional clock domain crossing. For display purposes, the three threshold values are transmitted through a FIFO that crosses from the 98.304 MHz domain to the 74.25 MHz pixel clock domain, enabling proper rendering of the threshold indicator lines on the HDMI display.

VI. LED CONTROL SYSTEM

A. LED Hardware and Protocol

The LED display consists of a 32×32 matrix of WS2812B addressable RGB LEDs arranged as 1024 serially-connected pixels. The WS2812B protocol uses a single-wire interface where each bit is encoded as a pulse-width modulated signal. A logical zero is transmitted by holding the data line high for 0.5 μs followed by 0.75 μs low, while a logical one requires 0.875 μs high and 0.376 μs low. Each LED consumes 24 bits in GRB (green-red-blue) order, and a reset signal consisting of a 50 μs low pulse latches the data into the LEDs and prepares the chain for the next frame.

The LED controller implements a finite state machine with four states: RECEIVE_SAMPLE, TRANSMIT_ZERO, TRANSMIT_ONE, and WAIT. Operating at 98.304 MHz, the

controller uses cycle counters to generate the precise timing required by the WS2812B specification. Once all 1024 LEDs have been updated, the controller enters the WAIT state to transmit the reset signal, after which the process repeats for the next frame.

B. Sprite System

Eight distinct 32×32 face images are stored in BRAM, with each pixel represented as an 8-bit palette index rather than a full 24-bit RGB value. A separate palette BRAM maps these indices to full RGB colors. This approach reduces the storage requirement from 24 KB for direct RGB encoding to 8 KB for indexed images plus 0.76 KB for the palette.

C. Face Selection

The face selector implements an eight-state FSM that maps frequency content to facial expressions. Each state represents a combination of eye position (open or closed) and mouth shape (closed, mid, open, or wide). The state transitions are determined by the concatenation of the comparisons between low, middle, and high bins against their user-defined thresholds.

Low frequencies control eye state, causing the eyes to close when bass exceeds the threshold. Mid frequencies influence intermediate mouth positions, while high frequencies trigger wide mouth expressions. State updates occur only when the update signal is asserted, which is synchronized to the completion of LED frame updates to ensure smooth visual transitions.

VII. EVALUATION

A. Performance Analysis

The custom FFT core completes a 512-point transform in 2862 clock cycles. This cycle count is derived directly from the FSM implementation and can be verified through simulation. At the system clock frequency of 98.304 MHz, the FFT completes a computation in 29.1 μ s. The system also processes audio frames of 512 samples at 48 kHz, corresponding to a frame period of 10.67 ms. The FFT computation consumes approximately 0.3% of this available time, leaving substantial margin for other operations.

For comparison, a 512-point FFT using Numpy on a modern CPU was measured to complete in approximately 7.5 μ s. While the FPGA FFT implementation is roughly four times slower in practice, it operates on a clock frequency 10-30x slower than modern CPUs. Additionally, with further optimizations and utilizing greater resources for parallel butterfly computation, the custom FFT's runtime could be improved significantly.

Overall, our system met timing with a worst negative slack of 0.776.

Resource Utilization

In total, CAROL utilizes 2411 LUTs, 1604 FFs, and 8 DSP blocks, well within the FPGAs resources

We utilized BRAM across multiple modules for domain crossing, buffering, and computation, totaling roughly 30 KB, as shown in Fig.7

Name	Width (bits)	Depth	Total Size (KB)
Frame Buffer	24	1024	3.07
FFT Ping-Pong	48	1024	6.14
Twiddle Rom	5	256	0.16
Threshold Crossing	96	64	0.77
Display Wave	24	1600	4.80
Display Bar	96	20	0.24
Image Memory	8	8192	8.19
Palette	24	256	0.77

Fig. 7. CAROL Memory Usage

B. Goals

We were able to meet our commitment goals, including receiving and transmitting audio through I2S, displaying an LED pattern, and showing live amplitude bars on HDMI.

We completed our goals, including implementing a working and tested FFT module, develop a state machine for feature control, and display threshold lines for each frequency band on HDMI.

We made progress towards our stretch goals, including implementing user-configurable thresholds and weighted sum.

VIII. IMPLEMENTATION INSIGHTS

Testing the butterfly module thoroughly before integrating it into the FFT core was one of the most valuable decisions in the project. This verification exposed several errors that would have been far more difficult to diagnose within the full FFT pipeline. By confirming the butterfly module's correctness first, debugging the FFT core became straightforward. When issues arose during FFT integration, we could eliminate the butterfly computation as a source of error and focus exclusively on the control logic, address generation, and timing.

The value of parametrization became apparent throughout implementation, particularly when design requirements shifted. All bit widths, fractional bit counts, and fixed-point representations were parametrized from the beginning. This decision became useful when we needed to reduce the twiddle factor size. Initial testing showed that the twiddle factors had excessive precision at the expense of larger delay. By changing a single parameter from Q2.10 to Q2.3, we reduced the multiplication width while maintaining sufficient precision. We also parametrized the butterfly module's latency in the FFT core's control logic. This parametrization allowed the FSM to automatically adjust its cycle counts by simply incrementing a parameter rather than manually recalculating every state transition.

Integration into the full system presented issues that simulation did not expose. While cocotb testbenches validated the FFT's algorithmic correctness against NumPy reference implementations, timing and interface issues were only revealed in hardware. The seven-segment display became the primary debug tool for these hardware-specific problems. We systematically routed intermediate signals at each stage to the

display, including BRAM read addresses and data values, butterfly module inputs and outputs, address generator counters, and the low, mid, and high frequency accumulator results. This process required rebuilding the bitstream and testing after each change, but it proved faster than attempting to reason where the problem should be.

CONTRIBUTIONS AND ACKNOWLEDGMENTS

Kevin worked on interacting with the hardware: sound wave, thresholds, and FFT output visualization on HDMI display, the I2S protocol, and controlling signals sent to LEDs. Max researched and developed the FFT, including the related butterfly module, twiddle factor generator, frame buffer, and magnitude smoother. He also created the image sprites and face selector. Both Kevin and Max wrote their relevant portions of the papers and presentation.

We'd like to thank Professor Joe Steinmeyer for providing guidance, Teaching Assistant Hasan Zeki Yildiz for hardware debugging support, and the rest of the 6.205 course staff for their availability and support.

CODE REPOSITORY

<https://github.mit.edu/6205F25/fa25-6205-team18>

REFERENCES

- [1] Pmod I2S2 Reference Manual. <https://digilent.com/reference/pmod/pmodi2s2/reference-manual#i2s-overview>
- [2] C. Diatkine, "FFT Size," Ircam.fr, 2011. <https://support.ircam.fr/docs/AudioSculpt/3.0/co/FFT%20Size.html>
- [3] Dr. Steve Arar, "Fixed-Point Representation: The Q Format and Addition Examples," Allaboutcircuits.com, Nov. 30, 2017. <https://www.allaboutcircuits.com/technical-articles/fixed-point-representation-the-q-format-and-addition-examples/>
- [4] Meiyappan, Somasundaram. Implementation and performance evaluation of parallel FFT algorithms.
- [5] R. Keim, "Introduction to the I2S Interface - Technical Articles," www.allaboutcircuits.com, Mar. 04, 2020. <https://www.allaboutcircuits.com/technical-articles/introduction-to-the-i2s-interface/>