ECE 385

Fall 2025

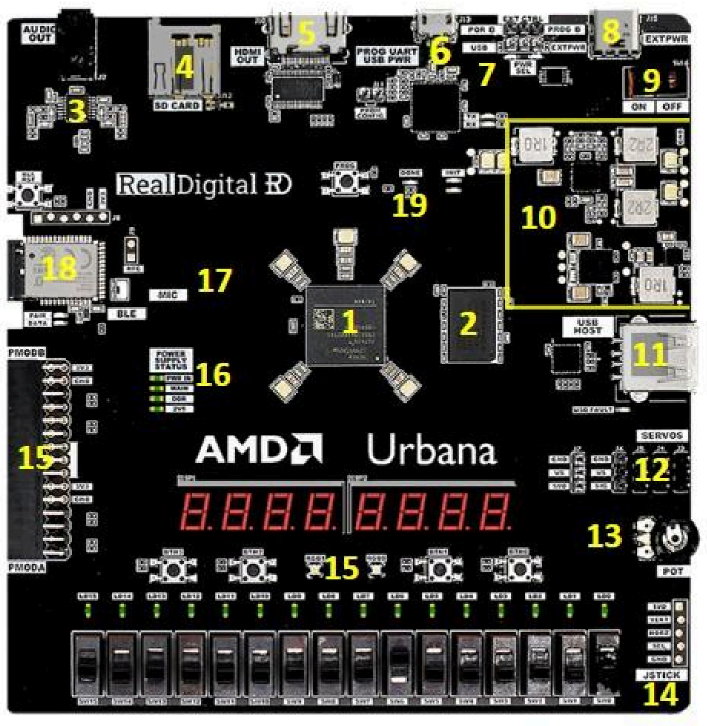
Final Project

FPGA Real-Time Audio Synthesizer and Visualizer

Ayush Jain and Alen Daniel

Elijah Ye (EY)

I. Introduction



The objective of this final project was to design and build a real-time audio processing and visualization system on the Urbana FPGA development board. This specific project differed mainly from previous labs, as it focused on creating a fully functional device from the ground up, rather than a single domain of computing architecture. We had to design all aspects of the device, and we had the opportunity to explore new peripherals on the FPGA board that were previously unexplored, such as the audio pipeline, which included the input microphone on the board, as well as a 3.5 mm audio output jack that can be connected to audio peripherals.

This system processes audio through a user-customizable digital signal processing chain, including distortion, bitcrushing, ring modulation, and pitch shifting, before outputting a pulse-width-modulated signal. Simultaneously, a MicroBlaze soft processor analyses audio amplitude to drive a physics-based HDMI visualizer. This project demonstrates the use of SystemVerilog for hardware design, C for software control, and concepts in analog and digital signal processing.

II. Written Description of System

The system is architected on a System-on-Chip design utilizing the Xilinx Artix-7 FPGA (XC7S50). The design masters the parallel nature of the FPGA to handle high-speed audio intake and video generation simultaneously, a task that wouldn't be as seamless or low-latency on a standard microcontroller setup. The primary clocks that maintain system synchronization in this section are described here. The architecture of this project's design is split into three main systems: Audio Input and Synthesis, Digital Signal Processing (DSP), and Visualization Control.

A. Audio Input and Synthesis Structure

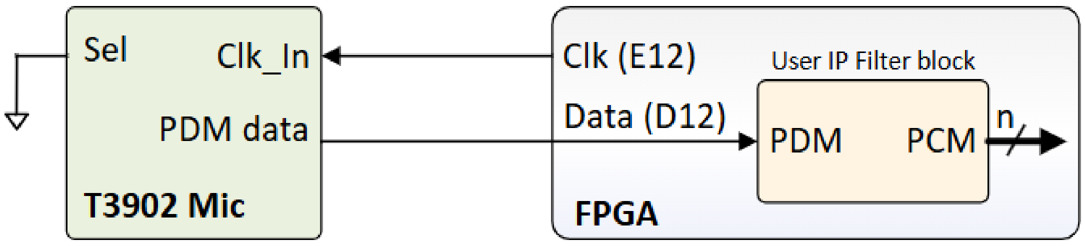


Figure 15. Urbana board MEMs microphone

The audio front-end handles the ingestion of external signals and the generation of internal tunes. The input stage begins with the PDM microphone located on the FPGA board. The input stage interfaces with the onboard TDK T3902 MEMS Microphone. Unlike analog microphones that output a continuous voltage as a function of v(t), this device integrates an amplifier and a sigma-delta modulator to output a Pulse Density Modulation (PDM) bitstream. In this format, the amplitude of the sound pressure is encoded in the temporal density of logical ‘1’ pulses. The T3902 supports a clock range of between 400 kHz and 4.8 MHz. Our design utilizes the microphone at 4.8 MHz to maximize the signal while reducing noise and expanding the audio bandwidth.

Next, we move on to how the decimation filter works. To integrate this 1-bit stream without a 16-bit digital signal processing chain, we implemented a Finite Impulse Response (FIR) Decimation Filter. This filter performs mainly two primary functions. These a low-pass filtering which averages a window of input signal samples and removes the high-frequency quantization noise in the modulator. The second function that the filter performs is to reduce the sample rate from 4.8 MHz to the target audio rate of 48 kHz. In order to make this possible, we utilize a decimation factor of M = 100 for the reduction process. This decimation implements a moving average filter, as demonstrated by the equation depicted below:

$$y[n] \; = \; \sum_{k=0}^{M-1} x[n * M \; + \; k]$$

X is the input bit, which can be positive or negative one, and y[n] is the resulting 16-bit pulse code modulated sample.

The next step of the operation is to implement a direct-current offset correction method. In actual PDM microphones, there is a significant amount of DC drift. To solve this, we implemented a High-Pass filter using a recursive function to center the signal at zero, and thus maximizing the dynamic range of the signed 16-bit integer. The formula of this operation is shown below:

$$y_{clean}[n] \; = \; y_{raw}[n] \; - \; Average[n]$$

B. Direct Digital Synthesis

One small but important aspect of the audio synthesizer project is the internal synthesizer that generates tones using Direct Digital Synthesis technology, using counter-based frequency division. The toggle limit ‘L’ is derived from the audio clock frequency f_{clk} of 4.8 MHz and the target note frequency f_{note}. The equation modeling the resulting toggle limit is shown below:

$$L \; = \; \frac{f_{clk}}{2^* f_{note}}$$

For example, generating Middle C, which is approximately 261.63 Hz, will require a toggle limit of around 9,173 clock cycles.

Another useful feature of the tune synthesizer is the ability to play multiple chords at the same time. This was achieved using Additive Synthesis, which mathematically adds together the outputs of the four independent counters, mapping to four different tones.

$$S_{total}[n] = v_1[n] + v_2[n] + v_3[n] + v_4[n]$$

One critical flaw that our design encountered was integer overflow, especially when combining the synthesized chords with the microphone, which often occurred when both inputs were summed together in their raw form. If the sum exceeded the 16-bit signed limit, the standard binary addition would wrap the value to a negative number. To solve this problem, we implemented a saturating arithmetic function to clamp down the output, as modeled by the equation below.

$$y[n] = clamp(x_{mic}[n] + x_{synth}[n], -32768, 32768)$$

C. Digital Signal Processing Chain

The trademark feature of this device is the ability to morph the inputted audio signal with mathematically implemented hardware effects.

The first implemented effect was hard-clipped distortion. This module uses non-linear amplification to alter the quality of the signal. This is modeled by a piecewise function where the signal is multiplied by a gain factor G and restricted to a threshold of L.

$$y[n] = L, if\ G * x[n] > L$$

$$y[n] = -L,\ if\ G * x[n] < -L$$

$$y[n] = G * x[n],\ if\ -L \le G * x[n] \le L$$

The second audio modification effect was the ring modulator with heterodyning. The ring modulator multiplied the audio signal x(t) by a carrier signal c(t), which was a 30 Hz square wave in our case.

$$y(t) = x(t) * x(t)$$

According to the Heterodyne Principle, multiplying two frequencies f_{in} and f_c generates sidebands at $f_{in} \pm f_c$. Since the 30 Hz signal is not harmonic with the input pitch, the resulting sum and difference frequencies are inharmonic, which is what produces the robotic texture and eliminates the natural human tone.

Another effect that we implemented was a bitcrushing effect using quantization. This effect reduces the depth with a logical mask.

$$y[n] = x[n]\ \&\ 0xF000$$

This increases the voltage step size and introduces Quantization Noise e[n] that is correlated with the signal.

Finally, to create a pitch shift effect, we employed a variable-speed playback utilizing a Circular Buffer in Block RAM. This is accomplished by decoupling the Read Pointer from the Write Pointer. This is where the playback speed comes into play, where the read pointer increments by a step size α provided by the potentiometer, whose functionality will be explained in a later section. For example, if alpha was 2.0, the system reads every second sample, doubling the frequency, resulting in an increased pitch as well. On the other hand, if alpha was at around 0.5, the system reads samples twice, halving the frequency, resulting in a decreased pitch. The range in playback speed goes from 0x all the way to 2.0x double speed.

D. User Control: XADC and Potentiometer

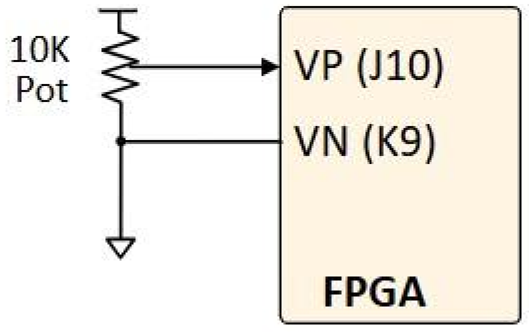


Figure 19. XADC

To implement the analog pitch control, we utilized the Spartan-7 FPGA’s internal XADC block. The XADC consists of two 12-bit, 1 Mega-Samples Per Second analog-to-digital converters. The XADC receives a voltage from the onboard 10k potentiometer (Vp / Vn), into a 12-bit value ranging from 0 to 4095. This value is read by the audio looper module to dynamically calculate the address step size of alpha to control the playback speed for the pitch shift effect.

E. Audio Output: PWM and Analog Reconstruction

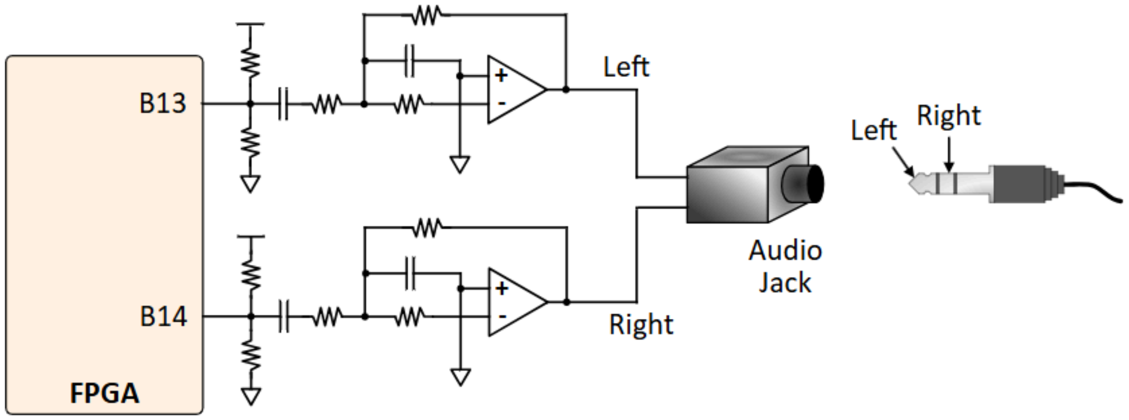


Figure 13. Urbana board audio output circuits

The digital-to-analog conversion is performed to convert the code signal, which was easy to modify, into an analog signal that speakers can output so that it can be heard as a matter wave. The signed 16-bit PCM is converted to an unsigned 8-bit Duty Cycle target. A counter compares these targets with another time to generate general PWM pulse sequences. The Urbana-Board includes a 2-pole Sallen-Key Low-Pass Filter on the output jack. This filter reconstructs the continuous analog waveform from the digital pulses by integrating voltage over time.

F. HDMI Video Design

The visual output system relies on an on-time pixel generation pipeline. The signal path begins with the VGA-to-HDMI IP code provided by Real Digital, which encodes the standard VGA control signals into the 10-bit transition-minimized differential signaling format. These signals are driven by the onboard TI TPD12S521 HDMI transmitter. To support serialization, the FPGA must provide a 125 MHz clock. This specific frequency is necessary since sending the 10-bit pixel data requires a clock five times the speed of the 25 MHz pixel clock. Most of the graphical user interface was implemented entirely in hardware logic for simplicity and to ensure zero-latency render times. The system is then allowed to render UI elements like the “Play” triangle, “Record” circle, and effect label text. We optimized the size of the design by using distributed RAM for the Font ROM, since we were running out of BRAM space due to the audio storage.

While the static UI and pixel rendering are hardcoded in SystemVerilog, the dynamic behavior of the dual volume bars is driven by the MicroBlaze soft processor. The C code reads the audio amplitude data and applies a gravity decay algorithm, essentially acting as a physics engine. The processor computes the smoothed bar height and writes this value to the hardware with the AXI-4 Lite connection. The hardware then uses this data to draw the box of the volume bar, along with the color of the bar (green on the quieter side, red on the louder side, and yellow in the middle) on the UI screen. This UI bar dynamically updates based on live audio received from the microphone and chord synthesizer, as well as the looped audio once the system is in the playback phase.
This separation allows the hardware to handle the high-speed 25 MHz pixel stream while the software manages the fluid, physics-based effect, giving the amplitude meter bars an analog depiction.

III. Module Descriptions:

A. SV Modules

1. Module: mb_audio_looper.sv-
   - Inputs: clk, rstn, record_en, pcm_valid, pcm_in [15:0], pitch_val [11:0], pitch_enable
   - Outputs: pcm_out [15:0]
   - Description: It implements an audio looper with pitch-controlled playback. PCM samples are smoothed by averaging consecutive samples to reduce static. The output switches between live audio passthrough during recording.
   - Purpose: It enables real-time audio looping and pitch manipulation on the FPGA using storage-based BRAM and phase-accumulator-driven playback.
2. Module: vga_controller-
   - Inputs: pixel_clk, reset
   - Outputs: hs, vs, active_nblank, sync, drawX [9:0], drawY [9:0]
   - Description: It generates timing signals for a standard 640×480 VGA display using a 25 MHz pixel clock. It implements the horizontal and vertical counters hc, vc that track the current pixel and line positions.

- Purpose: The module provides the low-level timing control necessary to interface with a VGA DAC or HDMI encoder, ensuring synchronization for video output.

3. Module: color_mapper.sv-
    - Inputs: BallX [9:0], BallY [9:0], DrawX [9:0], DrawY [9:0], Ball_size [9:0]
    - Outputs: Red [3:0], Green [3:0], Blue [3:0]
    - Description: The module determines the RGB color for each display pixel. It checks if the pixel is part of the ball by doing a calculation. Pixels within the ball are assigned an orange color, while the background pixels are shaded in a grayscale gradient.
    - Purpose: The module maps object position data and pixel coordinates to visible colors. It is the last rendering stage in the pipeline.
4. Module: mb_audio_visualizer_top.sv
    - Inputs: clk_100m, rstn, btn [3:1], sw_record, sw_bitcrush, sw_distortion, sw_pitch_en, sw_ringmod, vp_in, vn_in, mic_pdm, uart_rtl_0_rxd
    - Outputs: mic_clk, audio_left, audio_right, hdmi_tmds_clk_n, hdmi_tmds_clk_p, hdmi_tmds_data_n [2:0], hdmi_tmds_data_p [2:0], uart_rtl_0_txd, LED [15:0]

- Description: This is the top-level module that integrates the complete audio processing and visualization system. It manages clock generation, microphone PDM to PCM conversion, audio looping with pitch control, audio effects, and synthesizer generation.
- Purpose: It serves as the top-level FPGA system that ties together real-time audio synthesis, with effects, visualization, and external I/O.

5. Module: mb_bitcrush.sv
    - Inputs: clk, enable, pcm_in [15:0]
    - Outputs: pcm_out [15:0]
    - Description: This module applies a bitcrushing audio effect by reducing the bit depth of the input PCM signal. When enabled, the lower 12 bits of the PCM sample are masked off, leaving only the upper 4 bits.
    - Purpose: Provides a simple hardware-based bitcrushing effect for real-time audio processing pipelines, allowing for dynamic degradation of audio resolution.
6. Module: mb_distortion.sv
    - Inputs: clk, enable, pcm_in [15:0]
    - Outputs: pcm_out [15:0]
    - Description: This module implements a distortion effect by amplifying the PCM audio signal and applying hard

clipping. The input sample is sign-extended and boosted to increase gain, then clipped to a fixed threshold to introduce harmonic distortion.

- Purpose: It provides a hardware-based audio distortion effect for real-time sound processing, enabling gain-based clipping to create aggressive audio tones.

7. Module: mb_pdm_to_pcm.sv
   - Inputs: pdm_clk, rstn, pdm_in
   - Outputs: pcm_out [15:0], pcm_valid
   - Description: It converts a 1-bit PDM microphone input stream into signed 16-bit PCM audio using decimation and integration. Incoming PDM bits are accumulated over a fixed decimation window scaled to PCM amplitude.
   - Purpose: It provides a hardware PDM-to-PCM conversion stage for microphone input, enabling clean, low-noise audio capture and visualization in the audio system.
8. Module: simple_synth.sv
   - Inputs: clk, buttons [3:0]
   - Outputs: pcm_out [15:0]
   - Description: This module implements a simple square wave audio synthesizer using button inputs. Each button has a different musical note, which is generated by toggling a square wave using a clock-driven counter.

- Purpose: It generates musical tones, enabling piano-style interaction and audio mixing within an FPGA-based audio processing system.

9. Module: ring_modulator.sv
    - Inputs: clk, enable, pcm_in [15:0]
    - Outputs: pcm_out [15:0]
    - Description: It implements a modulation effect by multiplying the incoming PCM audio signal with a low-frequency square wave carrier. It generates a 30 Hz carrier that alternates the signal between normal and its inverted polarity.
    - Purpose: It provides a hardware-based ring modulation effect for real-time audio processing, enabling robotic sound modulation in the FPGA audio pipeline.
10. Module: mb_pcm_to_pwm.sv
    - Inputs: pwm_clk, rstn, pcm_in [15:0], pcm_valid
    - Outputs: pwm_out
    - Description: It converts signed 16-bit PCM audio samples into a PWM signal, which is suitable for audio output. A free-running counter and comparator generate the PWM waveform with a duty cycle proportional to the audio amplitude.

- Purpose: It provides a PCM to PWM conversion stage for driving audio output from digital audio samples, enabling analog filtering.

B. Block Diagram Modules

1. Microblaze (microblaze_0):
   - Description: It acts like a 32-bit RISC processor designed by Xilinx. It acts as the central processing unit that executes firmware controlling USB input, HDMI display, and peripheral operations. It interacts with memory, AXI peripherals, and interrupt controllers.
2. MicroBlaze Debug Module (mdm_1):
   - Description: It provides a JTAG-based debug interface for the MicroBlaze processor. It allows the user to download code and debug the processor through the Vitis IDE.
3. AXI Interconnect (microblaze_0_axi_periph):
   - Description: It is a multi-channel communication that connects AXI masters and slaves. It routes read/write data and control signals between MicroBlaze and peripherals. Collects interrupt signals from devices and prioritizes them before sending them to the interrupt line.
4. Local Memory (microblaze_0_local_memory):
   - Description: This is a BRAM controller providing instruction and data memory for the MicroBlaze. It stores

the program and data used by the Microblaze during execution. Both the instruction ILMB and DLMB connect to this block.

5. AXI UARTLite (axi_uartlite_0):
   - Description: It is a lightweight UART controller connected to the AXI bus. It provides serial communication for debugging and data exchange. It is used for printing messages or debugging output from the MicroBlaze program.
6. Processor System Reset (rst_clk_wiz_1_100M):
   - Description: Centralized reset controller that synchronizes reset signals across multiple clock domains, resetting the processor and AXI interfaces simultaneously upon system reset or clock instability.
7. AXI Interrupt Controller (microblaze_0_axi_intc):
   - Description: It aggregates and manages interrupt signals from multiple peripherals. It routes and prioritizes interrupts from the timer and SPI modules before passing them to the Microblaze's interrupt port.
8. Clocking Wizard (clk_wiz_1):
   - Description: It generates the required system clock frequencies from a single input oscillator. It produces the

appropriate clock frequencies required by the microblaze core and AXI peripherals.

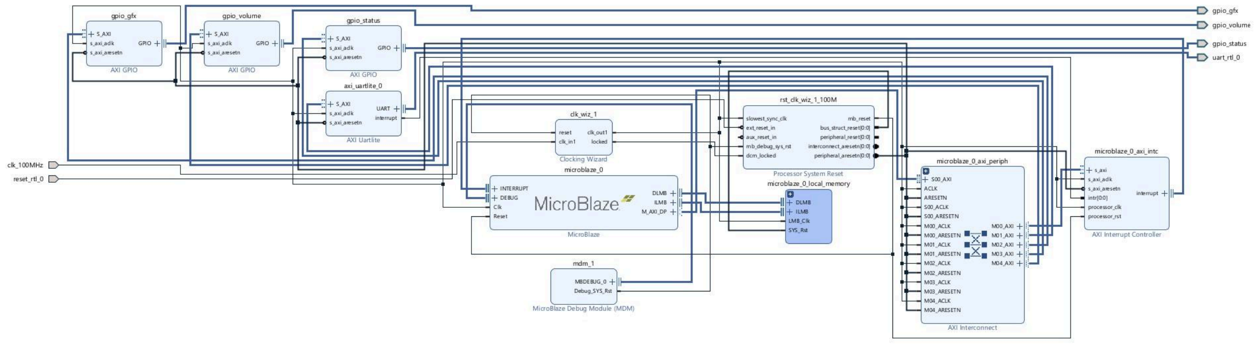
9. AXI GPIO (gpio_status):
    - Description: The AXI GPIO module communicates with the system and affects states from the FPGA to the MicroBlaze. It encodes the current position of the FPGA to the MicroBlaze, like record, bitcrush, distortion, etc. These flags allow the processor to change the UI behavior accordingly.
10. AXI GPIO (gpio_volume):
    - Description: This module provides the MicroBlaze with the audio magnitude of the FPGA. Through this, the processor can understand the scale and magnitude of how loud the sound is playing.
11. AXI GPIO (gpio_gfx):
    - Description: This AXI GPIO module is used by the MicroBlaze to send graphics and visualization data to the FPGA HDMI rendering logic. It outputs values like bar height or visual parameters.
12. HDMI Text Controller (hdmi_tx_0):
    - Description: It converts RGB pixel data into HDMI-compatible output. Handles synchronization (Hsync, Vsync, DE/VDE). It generates the HDMI signal

that can be sent to a monitor or display. It’s the final stage in your video pipeline.

13. Clocking Wizard (clk_wiz_hdmi):
    - Description: It generates multiple clock frequencies from a single reference clock input. It is used to provide the necessary timing signals for various modules in the design. The 25 MHz clock drives the HDMI TX Controller module for pixel synchronization. The 125 MHz clock supports the MicroBlaze processor.
14. Clocking Wizard (clk_wiz_0):
    - Description: It generates the precise audio-related clocks required by the design from the 100 MHz system clock. It produces around a 4.81 MHz clock for the PDM audio, and a 12.288 MHz clock for the PWM/audio timing.
15. XADC Wizard (xadc_wiz_0):
    - Description: This module configures with the FPGA’s internal XADC to sample analog signals. It operates in continuous, single-channel mode and outputs 12-bit ADC data with a strobe.
16. Block Memory Generator (blk_mem_gen_0):
    - Description: The Block Memory Generator (BRAM) is the VRAM for the text, where 16-bit character and color data for the HDMI Text Controller are located. It has the

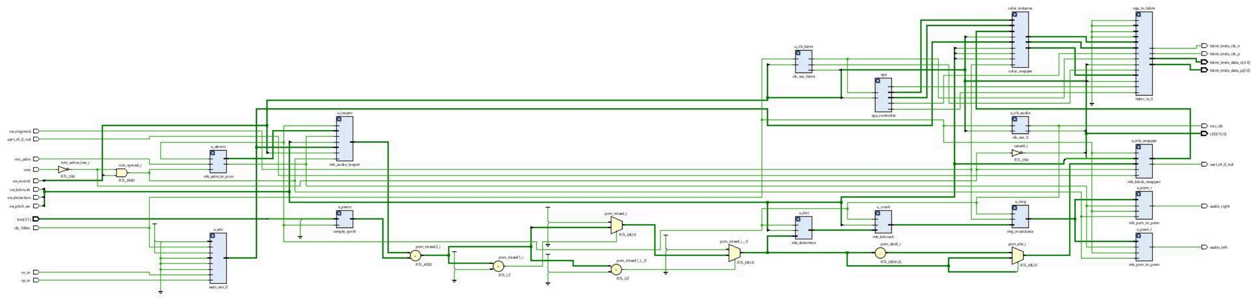
configuration of a True Dual Port RAM, which permits the processor to concomitantly write on one port while the text controller reads on the opposite port; furthermore, efficient updates are made possible due to byte-write enable. The dual-port configuration furnishes the CPU and video hardware with an uncomplicated, non-conflicting framebuffer.

IV. Block Diagram



Above, we have our block diagram consisting of the MicroBlaze portion of our circuit. It contains 3 GPIO modules, which are responsible for communicating with the hardware so that the microcontroller can interact with the HDMI output to calculate and display elements on the screen.

V. Elaborated Circuit Diagram

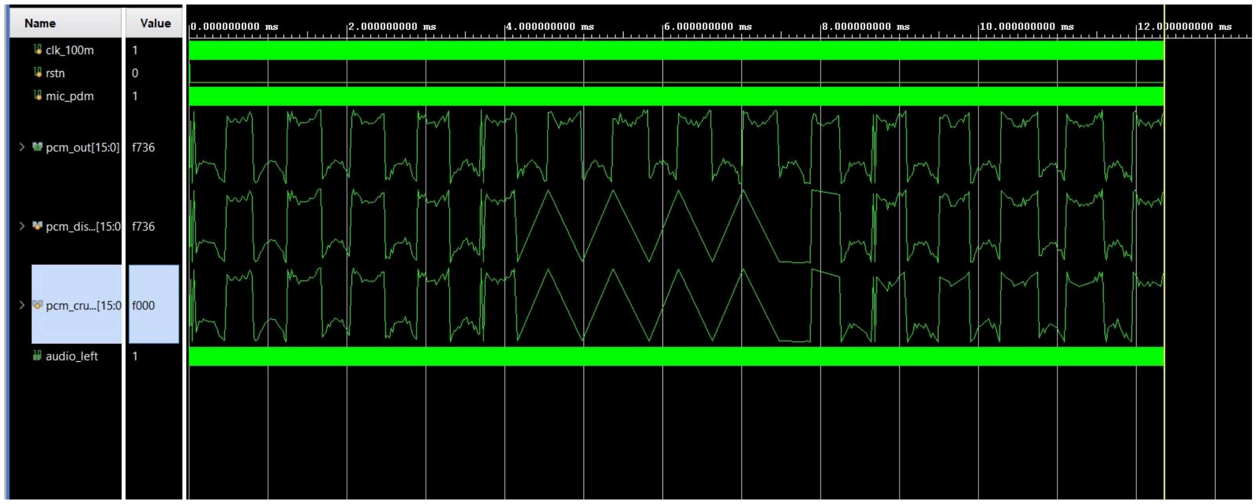


Above, our elaborated circuit diagram is a detailed description of the logical components within the circuit that make the decisions within our pipeline using combinatorial logic elements and LUTs.

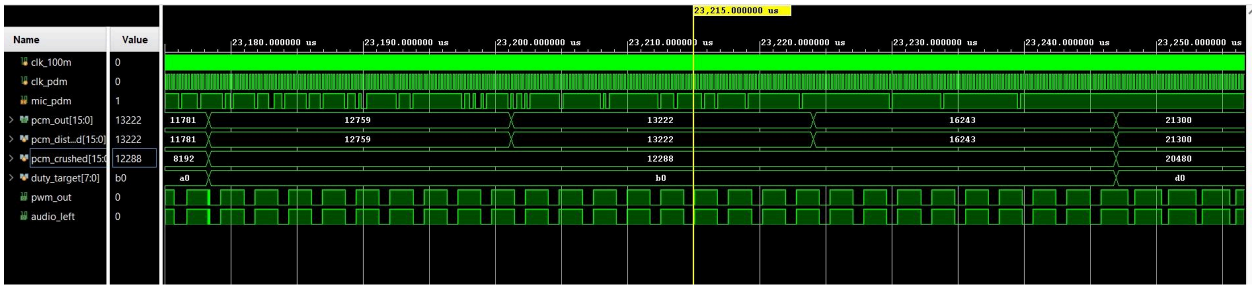
VI. Design Resources and Statistics

LUT	**2948**
DSP	**9**
Memory (BRAM)	**68**
Flip-Flop	**2019**
Frequency	**32.9 MHz**
Latches	**0**
Static Power	**0.076 W**
Dynamic Power	**0.471 W**
Total Power	**0.547 W**

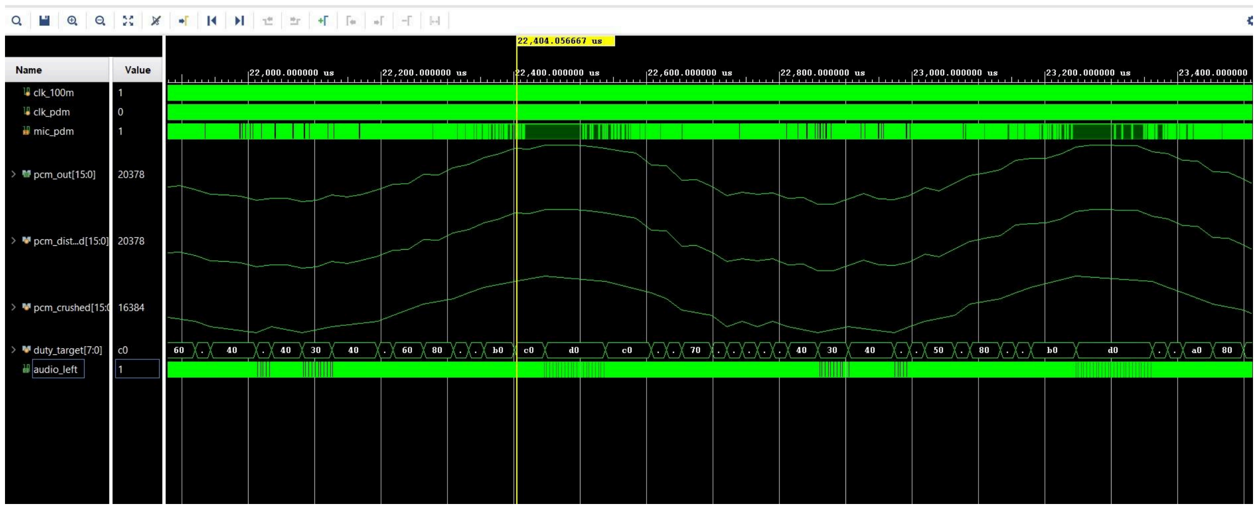
VII. Simulation Waveforms



This is a macroscopic or overall view capturing the system operation over 12 milliseconds, showing the stability of the microphone-to-speaker pipeline over several thousand clock cycles. The waveform shows the system handling dynamic inputs with different amplitudes and frequencies, sort of like the different ways that humans talk. All this information is taken in without triggering overflow or underflow issues. The pcm_dist signal is consistently tracking the pcm_out signal, even during peak amplitudes, which means that the distortion effects are working and don't produce unexpected results. In addition, the simulation validates that the clk_pdm (the clock synchronizing the mic) is fully synced with the system clock (100 MHz) throughout the duration.



The waveform above provides a single sample level verification that the internal data pipeline actually works, demonstrating the demodulation of the raw input pulse density modulated signal coming in from the mic. A verification of the bitcrushing algorithm is visible by comparing pcm_out with the pcm_crushed. While the highly detailed input is updating at a smooth rate (we can see that it is transitioning from values such as 11781 to 12759), the pcm_crushed signal is highly quantized, holding constant values despite there being differences in the input. In addition, the duty_target signal is shown to update steadily and synchronously with the processed audio, which then allows the pwsm_out modulator to generate the right pulse widths for the analog speaker output to properly output the sound.

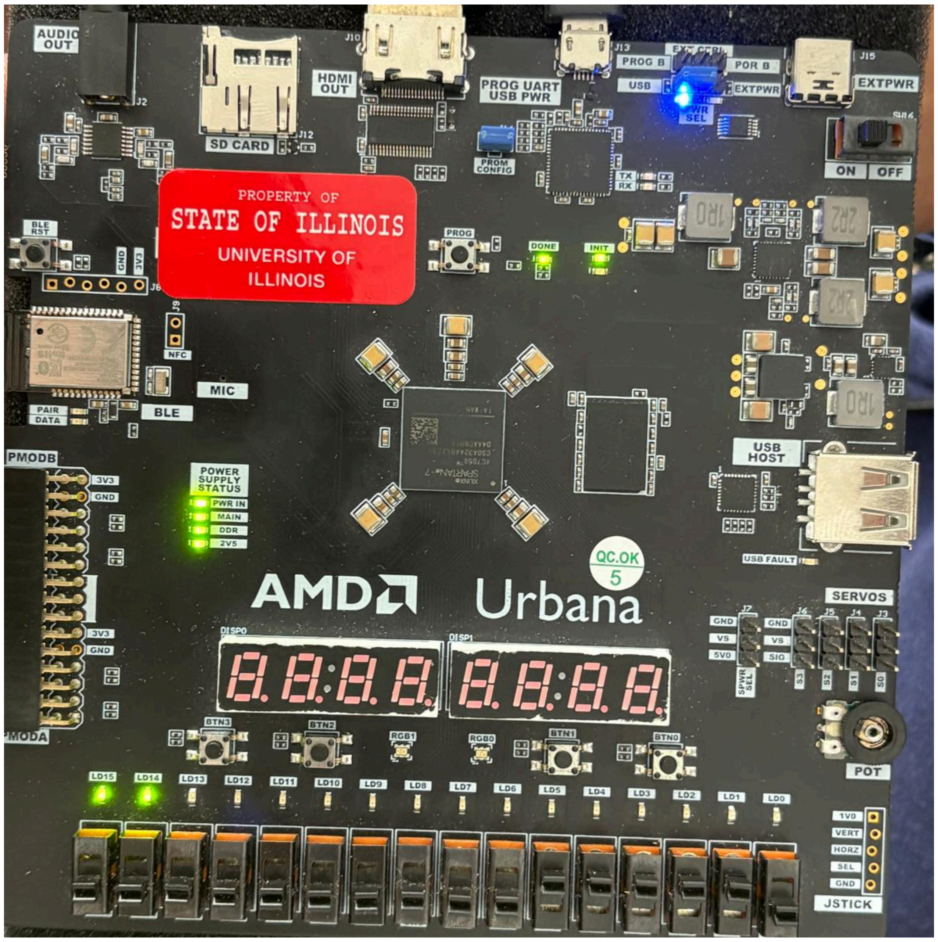


To verify how consistent or true the signal is, the simulator radix was set to signed, and the waveform display was switched to analog to visualize the numerical data as continuous waveforms. The top trace, pcm_out, shows a smooth reconstructed sine wave derived from the pulse-density modulated (PDM) microphone input. This shows the integrity of when the signal was converted from PDM to pulse code modulated (PCM) version. In contrast, however, the bottom waveform, pcm_crushed, demonstrated the stepping effects of the bitcrushing effect. The waveform does follow the general pattern of the input, but it isn’t as smooth or continuous, a key feature of bit crushing, since data is purposefully compressed with loss on the FPGA. The high-density waveform activity on the audio_left channel shows that the PDM output is successfully toggling at a 100 MHz clock rate for the speaker to work.

VIII. Conclusion

This project successfully brought together many of the core concepts taught throughout ECE 385, including digital logic design, clocking, and hardware-software integration. We also integrated things we learned from other classes' topics, like signal processing and filtering. By implementing a complete audio pipeline with effects and visualization, the project demonstrated how individual modules interact with each other to form a cohesive FPGA system. Overall, ECE 385 was a challenging but extremely rewarding course that significantly strengthened our understanding of digital hardware design and FPGA development. The hands-on labs and final project encouraged problem-solving, creativity, and a deeper appreciation for how hardware and software coexist in real systems. Below, we wrote the control scheme for our project.

IX. General Control Scheme



Sw0 - ON: Records

Sw0 - OFF: Playback

Sw1 - BitCrush

Sw2 - Distortion

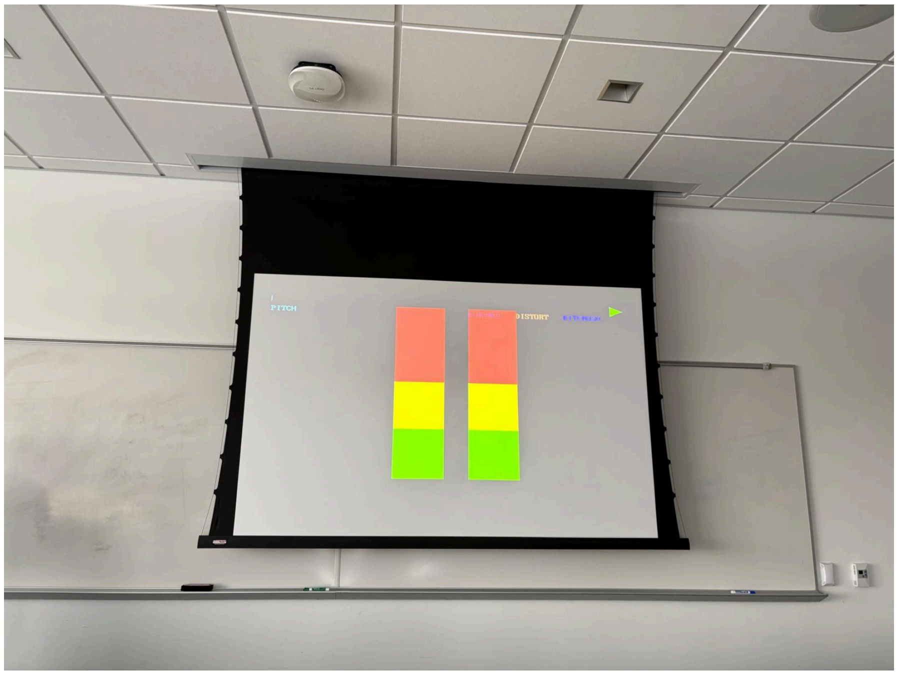
Sw3 - Adjust the Pitch with the Potentiometer

Sw4 - Turn on Ring Modulation (Robotic Mode)

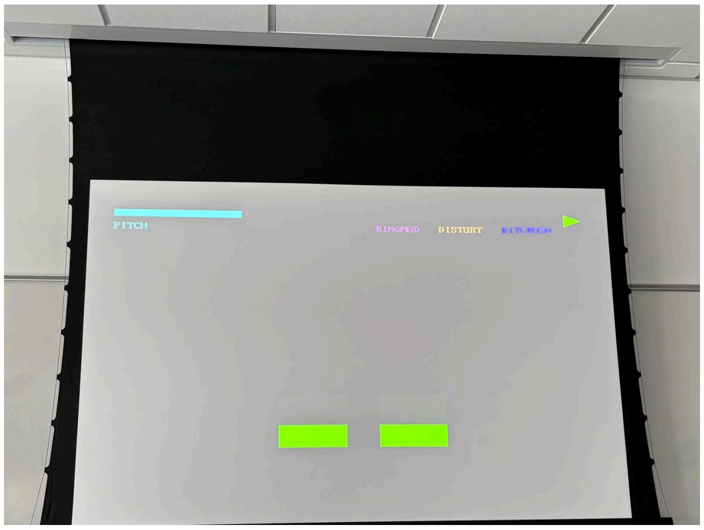
BTN0 - Reset

BTN1-BTN3 - Music Notes

X. Project Outcome



The image at the top displays the HDMI user interface of the project projected on a screen. The two bars show how loud the audio is for the left and right ears, respectively. The color green indicates low amplitude, yellow medium levels, and red high intensity. Text labels at the top indicate the active effects, such as Distortion, Bitcrush, and pitch. We can see that you can change the pitch level in the picture below, as the top picture has a way lower pitch than the bottom picture. Multiple effects can be enabled simultaneously, allowing them to be layered together and reflected in real time through both the audio output and the on-screen visualization.



Sources:

- Real Digital. (n.d.). Urbana Board Reference Manual: Audio Out Port. Retrieved December 16, 2025, from https://www.realdigital.org/doc/496fed57c6b275735fe24c85de5718c2#audio-out-port
- Google. (2025). Gemini (Dec 17 version) [Large Language Model]. https://gemini.google.com/
  Gemini Prompt: DSP mathematics for SystemVerilog
- Real Digital. (n.d.). Urbana Board Reference Manual. Real Digital. Retrieved December 16, 2025, from https://www.realdigital.org/doc/496fed57c6b275735fe24c85de5718c2
- AMD. (n.d.). Spartan™ 7 FPGAs. AMD. Retrieved December 17, 2025, from https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/spartan-7.html
- Real Digital. (n.d.). Pulse width modulation (PWM) and pulse density modulation (PDM). Real Digital. Retrieved December 17, 2025, from https://www.realdigital.org/doc/822e17a669a05f748c80af2274478bb5
- Khan, M. A. (2025, February 27). XADC on the Spartan 7 FPGA board. Hackster.io. Retrieved December 17, 2025, from https://www.hackster.io/the-confused-genius/xadc-on-the-spartan-7-fpga-board-82168e