ARTIFICIAL REVERBERATION PROJECT DOCUMENTATION

Landon Bates

Table of Contents

[0.0 – Introduction 4](#_Toc164281952)

[1.0 – Scope 4](#_Toc164281953)

[2.0 – Design Overview 4](#_Toc164281954)

[2.1 – Requirements 4](#_Toc164281955)

[2.2 – Constraints 4](#_Toc164281956)

[2.3 – Applicable Standards 4](#_Toc164281957)

[2.4 – Dependencies 5](#_Toc164281958)

[2.5 – Theory of Operations 5](#_Toc164281959)

[3.0 – Design Details 5](#_Toc164281960)

[3.1 – Hardware 5](#_Toc164281961)

[3.1.0 – Zynq-7000 SoC 5](#_Toc164281962)

[3.1.1 – ADAU1761 Audio Codec 6](#_Toc164281963)

[3.2 – FPGA Code 6](#_Toc164281964)

[3.2.0 – FPGA Code Overview 6](#_Toc164281965)

[3.2.1 – FIR Compiler IP 7](#_Toc164281966)

[3.2.2 – Block Memory Generator IP 8](#_Toc164281967)

[3.2.3 – Input Buffering 8](#_Toc164281968)

[3.2.4 – Input Data Sampling 9](#_Toc164281969)

[3.2.5 – Output Data Calculation 10](#_Toc164281970)

[3.2.6 – Edited Base Overlay 10](#_Toc164281971)

[3.3 – Software 11](#_Toc164281972)

[3.3.0 – Loading FPGA overlay 11](#_Toc164281973)

[3.3.1 – Configuring Audio Stream 11](#_Toc164281974)

[3.3.2 – RIR Filter Design 11](#_Toc164281975)

[4.0 – Testing 12](#_Toc164281976)

[4.1 – System Latency 12](#_Toc164281977)

[4.2 – LPF Verification 13](#_Toc164281978)

[4.3 – Audio Test 14](#_Toc164281979)

[5.0 – Conclusion 15](#_Toc164281980)

[6.0 – Appendix 15](#_Toc164281981)

[6.1 – FPGA Code 15](#_Toc164281982)

[6.2 – Other Code 20](#_Toc164281983)

List of Figures

[Figure 1: Top Level Diagram 5](#_Toc164281984)

[Figure 2: Zynq-7000 SoC Features 6](#_Toc164281985)

[Figure 3: FPGA Convolution Block Diagram 7](#_Toc164281986)

[Figure 4: FIR Compiler Convolution Block Diagram 7](#_Toc164281987)

[Figure 5: Read Address Equation 8](#_Toc164281988)

[Figure 6: Input Buffering 9](#_Toc164281989)

[Figure 7: Input Sampling 9](#_Toc164281990)

[Figure 8: Output Sum 10](#_Toc164281991)

[Figure 9: Base Overlay Modification 10](#_Toc164281992)

[Figure 10: Load FPGA Overlay 11](#_Toc164281993)

[Figure 11: Audio Stream Configuration 11](#_Toc164281994)

[Figure 12: RIR Filter Generation 12](#_Toc164281995)

[Figure 13: Filter .coe File Generation 12](#_Toc164281996)

[Figure 14: System Latency Measurement 12](#_Toc164281997)

[Figure 15: ADAU1761 Latency 13](#_Toc164281998)

[Figure 16: LPF Frequency Response Magnitude 13](#_Toc164281999)

[Figure 17: LPF Frequency Response Phase 14](#_Toc164282000)

# 0.0 – Introduction

Artificial reverberation methods have existed for over fifty years. One of these methods involves producing a loud, sharp noise (like a balloon pop) in a space and recording the audio response. This recording can be used to find the space’s room impulse response, or RIR. Convolving an audio signal with this RIR will output audio with an artificial reverberation effect that sounds like the input was played in the space the RIR was recorded in.

This project is a continuation of the “Audio Characterization” project that was completed in the summer of 2023. The completed system from that project is written in python and can record an audio input, convolve the input with an RIR, and save the resulting audio to a .wav file. The system described in this document expands on that system by implementing all the audio processing as real-time FPGA code.

# 1.0 – Scope

This document describes the audio convolution system implemented on the AUP PYNQ-Z2 development board including system requirements, design details, and testing results. This document does not include the specifications for the PYNQ-Z2 board or documentation for the PYNQ code libraries.

# 2.0 – Design Overview

## 2.1 – Requirements

The requirements (deliverables) for the system are as follows:

1. Record and process a book drop recording into a normalized RIR

2. Convolve an audio input with the RIR in real time

3. Minimize output latency to no more than 7 milliseconds, which is indistinguishable to the human ear

4. Perform the convolution on the FPGA side of the PYNQ board as VHDL code

5. Output audio that sounds like the input was played in the room the RIR is modeled after

## 2.2 – Constraints

The FPGA contains 630 kB of block ram. The design described in this document utilized all the block RAM on the FPGA, and expanding the design will require either an FPGA with more block RAM or the utilization of RAM outside of the FPGA.

## 2.3 – Applicable Standards

The I2C communication standard is used to interface with the ADAU1761 audio codec.

## 2.4 – Dependencies

The PYNQ-Z2 board requires a 12-volt power supply.

## 2.5 – Theory of Operations

The design is split into two sections. The bigger of the two sections consists of FPGA code which is responsible for reading audio data from the ADAU1761 audio codec, convolving that data with an RIR filter, and writing the resulting audio data back to the audio codec. The second section of code is python code that is used to configure the audio codec, start the audio stream, and generate RIR filter coefficients.

A black and white diagram of a computer

Description automatically generated with medium confidence

Figure 1: Top Level Diagram

After startup, various ADAU1761 registers need to be configured. These registers are configured by sending data from the system’s python module to the audio codec by utilizing the I2C transmitter located on the PYNQ-Z2’s FPGA. Next, audio data is read from the audio codec by the FPGA’s I2C receiver and sent into the convolution module. The convolved audio samples are then written to the audio codec using the I2C transmitter.

# 3.0 – Design Details

## 3.1 – Hardware

### 3.1.0 – Zynq-7000 SoC

The processor in the PYNQ-Z2 board is a Zynq-7000 SoC. The Zynq processor features an architecture that is part ARM processor and part FPGA. A list of features for this processor is given on the following page:

A white rectangular object with black text

Description automatically generated

Figure 2: Zynq-7000 SoC Features

### 3.1.1 – ADAU1761 Audio Codec

The ADAU1761 audio codec comes built-in on the PYNQ-Z2 board. For this project, the codec is configured to sample audio at 48 kHz. The codec features a configurable high-pass filter on the audio-in path set to 2 Hz by default. This filter gets rid of DC offsets. The codec also features a configurable low-pass filter on the audio-out path. This filter is set to cut-off at 30 kHz by default. There is also a set of amplifiers on the audio out-path that are configurable using the “set\_volume” command given in the PYNQ audio library. Registers on the audio codec are read and written using I2C communication. These registers are used to configure different sections of the codec as well as read and write audio input and output data.

The audio data on the codec is stored as a 24-bit fixed point value. There are 23 fractional bits. Before audio data is played from the DAC on the codec, an additional four bits are added to the front of the data to determine the volume the data is played at. This is important to know when performing audio processing.

The codec also adds a group delay of 1 ms split between its input and output path filters (see figure 15 for more details).

## 3.2 – FPGA Code

### 3.2.0 – FPGA Code Overview

To convolve an audio input with a 10,000+ tap RIR filter in real-time with minimal latency, the 10,000+ tap filter is divided into N sections of 2048 taps. Each filter section is then loaded into its own FIR Compiler IP core. These FIR IP’s run in parallel. The first FIR reads audio samples directly from the audio input with no buffering. Each subsequent FIR requires a sample delay of 2048 \* (n – 1). Because of this, the system uses an input buffer of size N\*2048. The input buffer takes in a new audio sample once every sampling period. This input buffer is implemented as a circular buffer using the Block Memory IP. Once every sampling period, each FIR block reads an audio sample from the buffer at position 2048 \* (n – 1). Once every clock cycle, the outputs of each FIR block are summed together. This sum is then outputted from the convolution block.

A screenshot of a computer

Description automatically generated

Figure 3: FPGA Convolution Block Diagram

### 3.2.1 – FIR Compiler IP

Xilinx’s FIR Compiler IP core is utilized in this project because it is efficient in its FPGA resource utilization. The IP core calculates a real-time convolution with a filter of max length 2048. The filter coefficients are configured to have 17 fractional bits. Since the audio input samples from the audio codec have 23 fractional bits, the output from the IP has 40 fractional bits. Each FIR IP instance requires 1 DSP48 resource and 5 BRAM resources.

A diagram of a block diagram

Description automatically generated

Figure 4: FIR Compiler Convolution Block Diagram

For this project, the IP is configured with a sample rate of 48 kHz and a clock rate of 100 MHz. Since the IP’s clock rate is significantly higher than the input sample rate, the IP can calculate a convolution output once every sampling period with no latency. This is achieved by performing one addition and multiplication each clock cycle. Since 100 MHz is 2083 times faster than 48 kHz, the IP core can calculate the 2048 additions and multiplications required to calculate 1 convolution output each sampling period without any latency.

More information about the FIR Compiler IP can be found in its product guide at https://www.xilinx.com/support/documents/ip\_documentation/fir\_compiler/v7\_2/pg149-fir-compiler.pdf.

### 3.2.2 – Block Memory Generator IP

The Block Memory Generator IP is used for the system’s input buffer. This IP allocates BRAM in the FPGA that can be read and written using the IP core. The amount of memory allocated is equal to the number of filter taps utilized in the system’s convolution. To write a sample to the allocated BRAM, the IP’s data in port is loaded with the new sample, the address port is loaded with the memory address to write to, and the write enable port is set to ‘1’ for one clock cycle. To read a sample from the allocated BRAM, the write enable port is set to ‘0’ and the address port is set to the address to read from. It takes two clock cycles for the IP to read the data stored at the given address and copy it to its data out port.

Generating the Block Memory IP with size 16,384 requires 11 BRAM resources. Doubling the size of the IP requires double the amount of BRAMs.

More information about the Block Memory Generator IP can be found in its product guide at https://docs.amd.com/v/u/en-US/pg058-blk-mem-gen.

### 3.2.3 – Input Buffering

To buffer the input samples with the Block Memory Generator IP, values are written to the IP circularly. When a value is written at the last available address of the IP, the next value will be written at the first address of the IP. Data for each FIR block is read at an address calculated in relation to the address of the last written value. This address is calculated with the equation:

Figure 5: Read Address Equation

N is the total number of filter blocks, and n is the FIR instance number. Since there is a two clock cycle delay in between telling the Block Memory IP which address to read from and receiving the data at that address, the next sample for each FIR block is read sequentially with a three clock cycle wait in between each read. This process is documented in figure 6.

A screenshot of a computer code

Description automatically generated

Figure 6: Input Buffering

### 3.2.4 – Input Data Sampling

The FPGA convolution block reads in a new sample once every sampling period. Once 2083 clock cycles have passed, a new audio sample is read from the data in port. This new sample is written to the Block Memory IP and to the first FIR block. This process does not rely on a valid signal since the ADAU1761 audio codec does not produce a valid signal.

A screenshot of a computer code

Description automatically generated

Figure 7: Input Sampling

### 3.2.5 – Output Data Calculation

Once every clock cycle, the output of each FIR block is summed together and written to the data out port of the FPGA convolution block.

A computer code with purple text

Description automatically generated with medium confidence

Figure 8: Output Sum

### 3.2.6 – Edited Base Overlay

The PYNQ code library includes a default FPGA overlay named “Base Overlay”. This system builds off that overlay. The audio codec controller block provided in the base overlay was edited to expose the audio receive and transmit paths that connect to the ADAU1761 audio codec. The FPGA convolution block was then added to the overlay, and the exposed audio receive and transmit ports of the modified audio codec controller were connected to the convolution block.

A computer screen shot of a computer

Description automatically generated

Figure 9: Base Overlay Modification

While building a new FPGA overlay from scratch would free up more resources to be used in the convolution block, it would also eliminate the ability to utilize some of the PYNQ board’s built-in code libraries. The main PYNQ library used in this system, the Audio library, is responsible for configuring the ADAU1761 audio codec registers. Creating a new overlay would require writing a new audio codec controller from scratch, a task that was out of the scope of this project.

## 3.3 – Software

### 3.3.0 – Loading FPGA overlay

To load the FPGA overlay onto the Zynq processor, the BaseOverlay function given in the PYNQ code library is used. This function loads a .bit file onto the FPGA.

A close-up of words

Description automatically generated

Figure 10: Load FPGA Overlay

### 3.3.1 – Configuring Audio Stream

The audio library given in the PYNQ code libraries is used to configure the ADAU1761 audio codec. The default sampling frequency of the codec is 48 kHz, but we need to configure the codec to sample audio from the line in port of the PYNQ-Z2 board. This is done using the “select line in” function. The gain for the codec is configured using the “set volume” function.

To start an audio stream, the Audio library’s “bypass” command is used. The bypass function has a maximum runtime of 10 seconds, so the command is called inside a while loop that runs until the user stops the program.

A screenshot of a computer program

Description automatically generated

Figure 11: Audio Stream Configuration

### 3.3.2 – RIR Filter Design

Generating an RIR filter from a recording of a book drop requires some processing. First, the recorded audio is loaded from a .wav file to python. Then, the left channel data is separated from the right channel data. The left audio channel is used to generate the RIR. The left channel data located before the book drop impulse occurs in the recording is trimmed off to avoid a delay in generated reverb from the RIR. The data is then normalized to the magnitude of the book drop impulse.

After the RIR has been created, N sections of 2048 data points are created from the RIR. Each section represents about 42.7 ms of reverb. If N is equal to 8, the first 341 ms of the RIR will be divided into 8 sections. Each section is then written to its own .coe file that will be used to generate a FIR block for the FPGA convolution module.

A screenshot of a computer code

Description automatically generated

Figure 12: RIR Filter Generation

A computer code with many colored text

Description automatically generated with medium confidence

Figure 13: Filter .coe File Generation

# 4.0 – Testing

## 4.1 – System Latency

To measure the total latency of system, an impulse signal was fed into the system and the resulting output was measured on an oscilloscope. The measured latency is 1 ms.

A graph with a line graph

Description automatically generated with medium confidence

Figure 14: System Latency Measurement

This measured latency is not from the FPGA convolution block. It is the sum of the group delay of the input and output path filters located in the ADAU1761 audio codec. The input path’s ADC decimation filter has a group delay of 0.479 ms, and the output path’s DAC interpolation filter has a group delay of 0.521 ms. When these two group delays are summed together, we get a total group delay of 1 ms. This matches what was measured on the oscilloscope. This means that the total latency for the FPGA convolution block is very close to 0.

A screenshot of a computer

Description automatically generated

Figure 15: ADAU1761 Latency

## 4.2 – LPF Verification

To verify the accuracy of the system, a LPF filter with 16,384 taps and a cutoff frequency at 3000 Hz was loaded into the FPGA convolution block. The frequency response of the filter was simulated with python and measured on an oscilloscope using the FPGA convolution system. The simulated and measured results were then compared.

A graph of a graph

Description automatically generated with medium confidence

Figure 16: LPF Frequency Response Magnitude

A graph of orange lines

Description automatically generated

Figure 17: LPF Frequency Response Phase

The frequency response magnitude measured from the FPGA has a higher pass-band gain than the simulated response. To decrease the SNR of the measured signal, the gain had to be increased on the ADAU1761 amplifier. This is most likely the cause of the pass-band gain not matching. The stop-band gain is also much higher on the measured response than on the simulated response. This could be because the FPGA system is stopping the gain from dropping below -60 dB. What causes this to happen is still unknown.

The frequency response phase from the simulation is zero, but the measured response phase is not. It appears to be linear in some places and non-linear in others. This is most likely caused by the combined group delay of the ADAU1761’s ADC and DAC filters. Since these filters do not have the same amount of group delay, their combined phase is non-linear.

## 4.3 – Audio Test

The final version of the system can add 0.334 seconds of reverberation to a real time audio input. Different music was played into the system and the music with reverberation was compared to the music without the reverberation. There was a clear difference between the two. This test did not provide a quantifiable result, but it does sound cool. The music processed with the system does sound like it was played in a stairwell which is where the RIR used in the system was recorded.

# 5.0 – Conclusion

The system described in this document met all the requirements. However, there are many aspects of the system that can be improved. The maximum reverberation time of the current system is limited to about a third of a second. This is because the system utilized all of the available BRAM resources located on the Zynq processor. To expand this system to replicate longer reverberation times, an FPGA with more block ram will need to be used. Alternatively, the system could be modified to utilize ram external to the Zynq processor. Other more efficient methods for generating reverberation artificial reverberation methods do exist, and one of them could be utilized to increase the reverberation time as well.

# 6.0 – Appendix

## 6.1 – FPGA Code

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated

## 6.2 – Other Code

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer code

Description automatically generated