FPGA-based Robotic Effect Voice Changer

A Design Project Report
Presented to the School of Electrical and Computer Engineering of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering, Electrical and Computer Engineering

Submitted by
Shuzhe Liu, Jiacheng Tu, Xiangzhou Wei
MEng Field Advisor: Van Hunter Adams
Degree Date: December 2024

Contents

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: FPGA-based Robotic Effect Voice Changer

Authors: Shuzhe Liu, Jiacheng Tu, Xiangzhou Wei

Abstract:

We developed a real-time voice-changing system on a Zybo-Z7 board, featuring an ARM Cortex-A9 processor paired with Xilinx 7-series FPGA logic. This system, designed to transform human voices into robotic tones, includes an audio codec driver, an ARM to FPGA communication module, and an FPGA-based voice changer. The process involves the ARM processor collecting audio data, which it sends to the FPGA via AXI BRAM. Within the FPGA, the audio is filtered through 32 second-order IIR band-pass filters, followed by a low-pass filter to modulate the data rate. Direct Digital Synthesis is then employed for sound re-synthesis. Post-processing, the data returns to the ARM processor through AXI BRAM and is outputted via the audio codec. The project aims to safeguard vocal identities from AI misuse by changing and compressing audio information, thereby enhancing privacy. Utilizing FPGA's parallel computing capabilities ensures efficient, scalable, and customizable voice processing, offering robust protection against potential data leaks or hacking.

Executive Summary

Over the past year, our group has successfully developed a real-time voice-changing system utilizing a Zybo-Z7 board, which combines a dual-core ARM Cortex-A9 processor with Xilinx 7-series FPGA logic. This innovative system is capable of altering human voices into robotic tones, addressing concerns about potential misuse of AI in stealing vocal identities.

The voice changer consists of three main components: an audio codec driver, a communication interface between the ARM processor and the FPGA, and the voice transformation system within the FPGA itself. The process begins with the ARM processor capturing audio data through the audio codec and transmitting it to the FPGA via AXI BRAM. The FPGA then processes this data through 32 IIR band-pass filters, isolating sound into various frequency bands, followed by a low-pass filter to control the data's rate of change. The final output is achieved through Direct Digital Synthesis, which reconstructs the sound into a comprehensible robotic voice. The processed audio is subsequently sent back to the ARM processor and output through the audio codec for external transmission.

The final prototype of our voice changer works as expected: whenever we speak into the microphone, the speaker outputs a robotic voice in real-time, processed by the board. By adjusting the switches on the FPGA, we could alter the pitch of the output, making the voice sound higher or lower.

The choice of FPGA over other microcontrollers or software solutions significantly enhances the system’s security, making the audio data less prone to leaks or hacks. Moreover, the FPGA’s robust parallel computing capabilities allow for highly efficient and energy-saving operations. In addition, Our system offers scalability and customizability, facilitating straightforward adjustments to our RTL modules to incorporate additional layers of effects or extra filters, tailoring it to specific user requirements.

This project significantly enhances user privacy by transforming and compressing vocal data. Through this development, we have provided a potent tool against the exploitation of vocal data, positioning our system at the forefront of secure voice processing technologies.

Introduction

In recent years, concerns about the misuse of artificial intelligence (AI) technologies, particularly in the realm of voice manipulation, have led to a growing interest in developing systems that can protect individual privacy. Voice-changing technology, which can alter a person's voice to sound different, is one potential solution to safeguard personal voice identity from being exploited. The objective of this project is to develop a voice-changing system using a Zybo-Z7 board, which has an ARM processor and FPGA, that can transform human voices into robotic tones in real-time, potentially lowering risks associated with voice identity theft.

The choice of FPGA over other traditional microcontrollers or software implementations is motivated by the enhanced security it offers; the closed, hardware-based nature of FPGAs makes them less vulnerable to external hacks and leaks. Additionally, the inherent scalability and customizability of FPGAs allow developers to tailor the system to meet specific requirements or adapt to new challenges, further solidifying the role of this technology in future privacy-oriented applications.

Key outcomes of this project include detailed input and output traces captured from an oscilloscope, along with video clips that demonstrate the transformation of a human voice to a robotic tone and the accompanying switch-enabled pitch shifts achieved by our system. These results underline the effectiveness of the FPGA-based solution in real-time audio processing systems.

Design

Design Overview

The project is based on a Zybo-Z7(shown in figure 1), integrating a dual-core ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA) logic. The ARM processor was used to receive and send audio data. The FPGA was mainly responsible for processing the audio data. Audio codec on this board was used to encode analog audio as digital signals and decode digital back into analog, block ram (BRAM) was used as shared memory between FPGA and the processor. The switches on this board could help us add user-interactable features to the system. Most code is written in Verilog, programmed and wrapped on Vivado Design Suite. Using the Software Development Kit provided by Xilinx, we wrote C++ programs that interact with the codec chip through the processor 's interface to the FPGA by AXI. To assist in designing our voice changing system, MATLAB scripts were written to automate writing repetitive code and calculate the coefficients required for our filters.

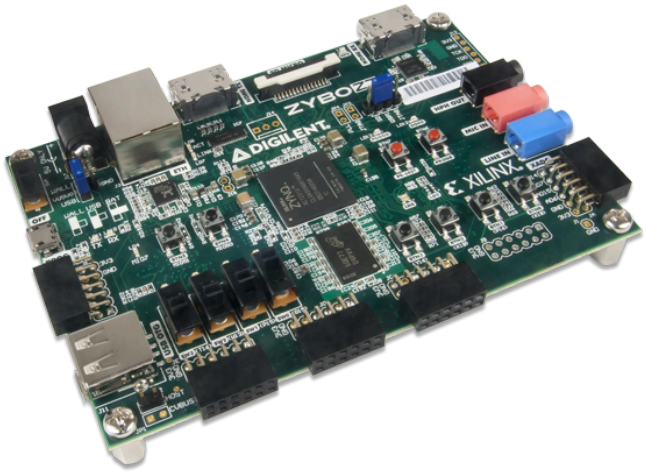


Figure 1. Zybo Z7 [1]

Hardware Design

The overall hardware system framework shown in figure 2 consists of two submodules:

- Communication Bridge between ARM & FPGA
- Voice Changing System

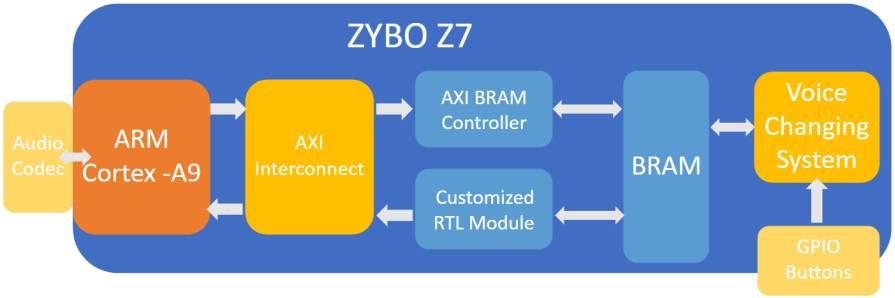


Figure 2. System Framework

The Audio Codec is driven by the ARM processor to get audio input and it is interacted with the FPGA through a bridge of AXI Block RAM and AXI GPIO. The processing task of FPGA is realized by building customized RTL modules to achieve successful writing and reading operation on AXI Block RAM and to develop the Voice changing System where we generate the voice changing algorithm function blocks.

As can be seen in figure 3, the Voice changing system consists of three parts: Initiate Impulse Response bandpass filters, Low pass filters and Direct Digital synthesizers.

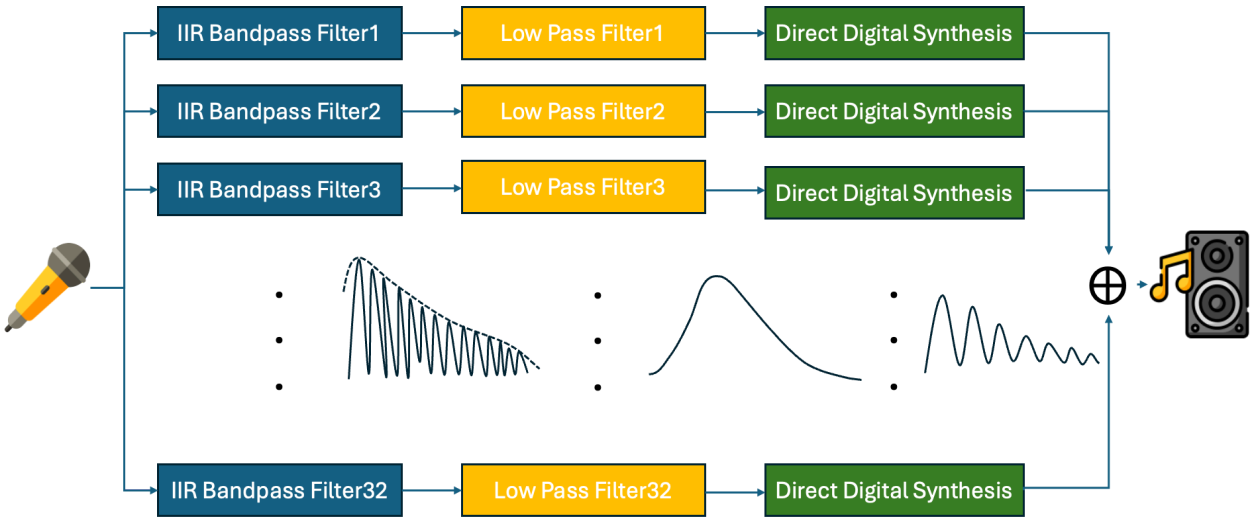


Figure 3. Voice changing system

The audio input first enters a bank of IIR bandpass filters, which decomposes the sound signal into different frequency bands. Then, each IIR bandpass filter is followed by a low pass filter to limit the rate of change of the sound content. Finally, a Direct Digital Synthesis module is used to reinsert the extracted sound information back into the original frequency bands, recreating an audible sound signal. This can also be adjusted to different frequency bands to achieve a pitch-shifting effect.

The System block design diagram is shown in figure 4, the RTL module “bram_pl_wr” contains the logic for implementing the voice changing system and data flow control. The block “processing_*system7*_0” indicates the ARM processor system. The block “zybo_audio_ctrl” exposes the ports which are used to drive the audio codec on the AXI bus, this allows us to drive the audio codec from ARM side in C program. The block “axi_bram_ctrl” is the AXI BRAM controller which allows us to manipulate the memory from the ARM side. The block ”ps7_0_axi_periph” is generated automatically by Vivado for arranging AXI bus connections. The block “blk_mem_gen” is the true dual port BRAM.

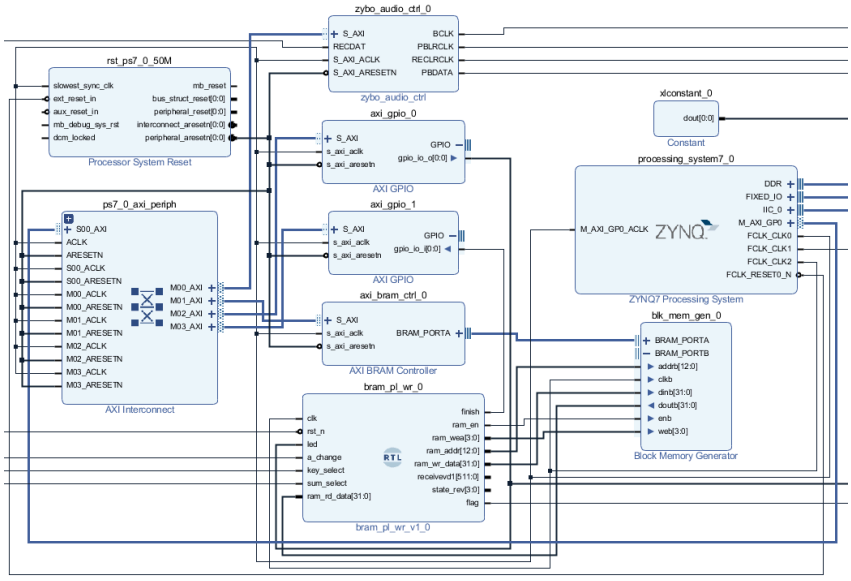


Figure 4. System Block Diagram

Finite State Machine

The main task for FPGA to execute is constantly running the finite state machine to read memory, implement voice changing effect algorithms and write memory. The Finite State

Machine shown in figure 5 demonstrates the simplified audio processing procedure for better understanding the overall working principle of the system.

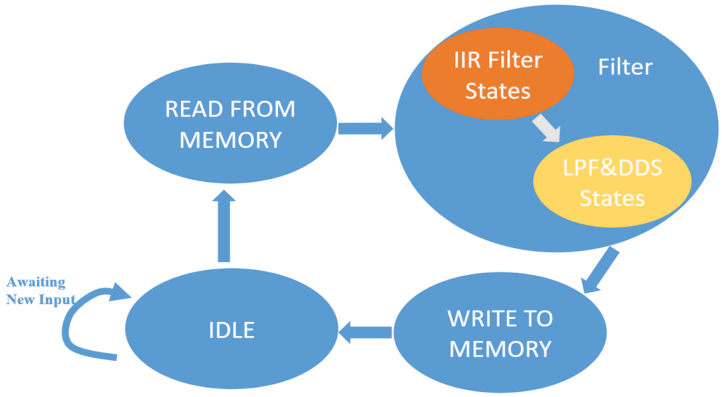


Figure 5. System Control Flow

Initially, the state machine starts with IDLE state, waiting for the new input that comes from the ARM processor side. To be specific, every time when the ARM processor receives a new audio input data, it writes it into AXI BRAM and pulls up the "valid" signal which is connected to an AXI GPIO output from ARM to FPGA. When the FPGA detects the "Val" signal which indicates that there is a new input stored into the BRAM, it moves to "READ FROM MEMORY" state where it reads out the new input data. Then it proceeds to the filter state (refers to the voice changing system), which consists of two smaller state machines for IIR filters, Low pass filters & DDS, respectively. In the next state "WRITE TO MEMORY", the processed audio data is written into the BRAM and after it completes the write operation, it will pull up the "Ack" signal which is also connected to an AXI GPIO output from FPGA to ARM and goes back to the IDLE state to wait for the next audio input.

IIR bandpass filter

Second-order Infinite Impulse Response(IIR) filters are used for the bandpass function as they provide sufficiently good bandpass effects without consuming excessive computational resources. Specifically, the implementation uses a multiply-and-accumulate (MAC) scheme to compute each term on the right side of equation 1.[2] (x[n] is the input signal at time n, and this is the data being fed into the filter. y[n] is the output signal at time n, and this is the data coming out of the filter after processing. a(1), a(2), a(3) are coefficients of the filter, often called filter tap

values for the output signal y. b(1), b(2), b(3) are coefficients of the filter, often called filter tap values for the input signal x. The term a(1) is always assumed to be one.) Moreover, it is important to note that for computational ease on FPGA, the filter uses 18-bit fixed-point numbers, with 2's complement notation and 16 bits of fraction for both the signal and the coefficients.[3] The filter tap values are computed using MATLAB by solving for the coefficients that meet the desired filter specifications.

$$a(1) * y[n] = b(1) * x[n] + b(2) * x[n-1] + b(3) * x[n-2] - a(2) * y[n-1] - a(3)y[n-2] \quad \text{(eq.1)}$$

We decided to use a total of 32 bandpass filters to provide enough sound components to ensure that the extracted sound features have sufficient granularity. The center frequencies of these 32 bandpass filters are not evenly distributed across the frequency band to better match the human auditory system. Since human ears are more sensitive to low-frequency sound information, we placed more filters in the lower frequency ranges. Specifically, we used the Mel scale.[4][5] The Mel scale is a perceptual scale of pitches judged by listeners to be equal in distance from one another. It is designed to reflect how the human ear perceives sound, particularly in the frequency range of speech. The scale is non-linear, with a higher resolution at lower frequencies and a lower resolution at higher frequencies, mimicking human auditory perception. We distributed the center frequencies evenly across the Mel spectrum, then converted the Mel scale to conventional frequencies. Equation 2 shows the transformation between mel scale and normal frequency (m represents mel scale and f represents normal frequency scale). This resulted in a filter system that is dense in the low frequencies and sparser in the high frequencies. The edges of each bandpass filter were selected to be the frequency at which the amplitude dropped down to about 50% of the peak at the center frequency.

$$f = 700(e^{\frac{m}{1125}} - 1) \quad \text{(eq.2)}$$

We used MATLAB to calculate the coefficients for each filter, which is very convenient. Below is an amplitude graph of the 32 generated IIR bandpass filters. It is evident that the center frequencies range between 300 Hz and 3500 Hz, which aligns with our expectations and is similar to the typical frequency range of the human voice.

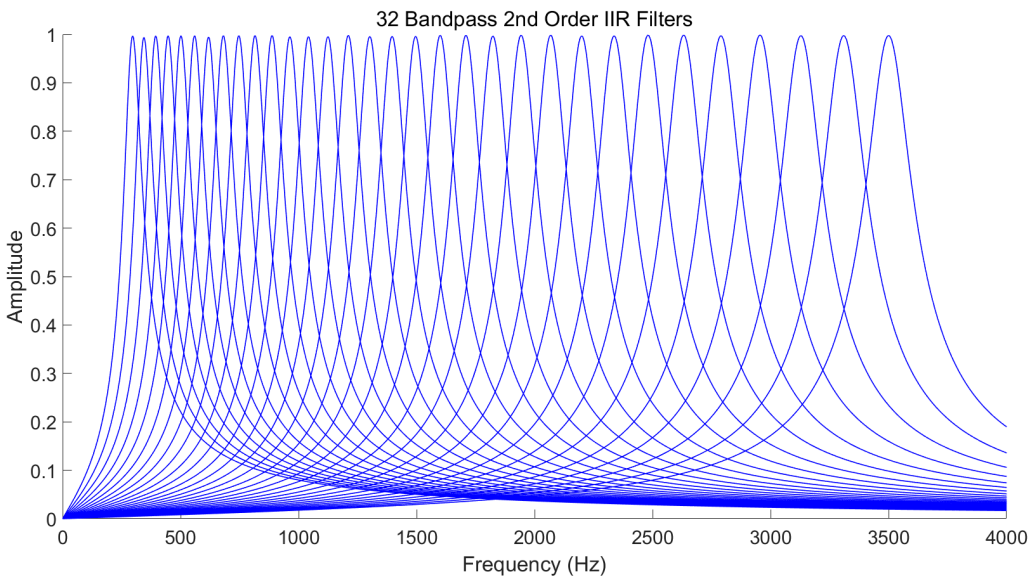


Figure 6. IIR bandpass filters

Lowpass filter

Next, we feed the data from the IIR bandpass filters into a low-pass filter. At this stage, since we are not inputting the original sound signal, the purpose of this low-pass filter is not to remove high-frequency noise, but rather to limit the rate of change of information in the sound signal within this frequency band. Due to the limitations of the human vocal apparatus, the rate at which the content of sounds produced by humans can change has an upper limit, typically around 100 Hz. It is this upper limit of content that represents the sound information, or sound features, we are interested in. Therefore, we use a low-pass filter to extract information within this limit.

The low-pass filter is implemented through equation 3, where the α parameter is calculated based on the filter's cutoff frequency and the system's sampling rate. (Similar to eq.1, x[n] is the input signal at time n. y[n] is the output signal at time n. α is the coefficient of the filter. α' is the coefficient of the filter.)

$$y[n] = \alpha * x[n] + (1 - \alpha) * y[n-1] = \alpha * (x[n] - y[n-1]) + y[n-1] \quad \text{(eq.3)}$$

$$= (x[n] - y[n-1]) >>> a' + y[n-1]$$

$$a = \frac{f}{F_S} = \frac{f}{24000} \qquad a' \simeq (log_2 \frac{1}{\alpha})$$

Since computational resources on the board are limited, we conserve resources by converting the multiplication with α and other parts of the formula into shifts. Additionally, we place these calculations in the previously used IIR bandpass filter module, following the IIR computations. This reuses the multipliers in the module to optimize resource utilization.

Direct digital synthesis

Finally, to regenerate understandable sound from the information processed by the low-pass filter, we need to carry the sound back to the frequency bands it occupied before the bandpass filtering. To accomplish this, we use Direct Digital Synthesis (DDS) by creating a sine wave lookup table.[6] By using accumulators with different increments to search the table, we can generate sine wave outputs at various frequencies.. The increment for each DDS is calculated based on the center frequency of the corresponding sound signal before bandpass filtering to produce sine waves at that frequency. The calculation of the increment is shown in equation 4. (We use a 32-bit accumulator with a total of 2^{32} states. However, to conserve hardware resources, we use only an 8-bit look-up table, using the top 8 bits of the accumulator for look-up. The sampling frequency is 24kHz. When generating the original signal, the pitch parameter is set to 1.)

$$increment = \frac{f*total\ accumulator\ states*pitch\ shift}{F_S} = \frac{f*2^{32}*pitch\ shift}{24000} \quad \text{(eq.4)}$$

By multiplying the data from the low-pass filter output with these sine waves, thereby adjusting the amplitude of the sine waves, we reintroduce the sound information back to its original frequencies. In this process, by changing the increments for all DDS units, we can achieve pitch shifting of the output sound signal—increasing the increment results in a higher pitch, while decreasing it makes the sound deeper.

User interface

To allow the user to adjust the voice changing effect, switches that connect to the FPGA are utilized as the user input to control the pitch shift level and parameter α'.

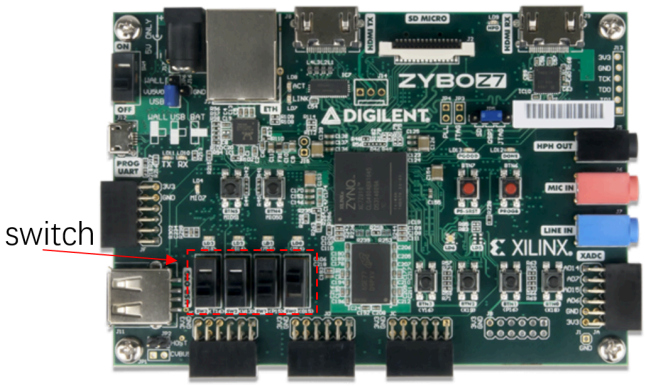


Figure 7. On board switches

We detect the rising edge from the signal caused by the switch. Two counters are utilized to control the value of the pitch shift level and parameter a'. Every time it turns on we can adjust the pitch shift from 0-4 and parameter a' from 7-13. Every time the corresponding switch turns on the pitch shift level will be incremented from 0 to 4 and a' will be incremented from 7 to 13.

Software Description

Audio Codec Driver

To drive the audio codec, we made the connection between the audio codec and the I2C ports by utilizing the integrated IP core named “zybo_audio_ctrl” and configured by C program on the ARM processor.

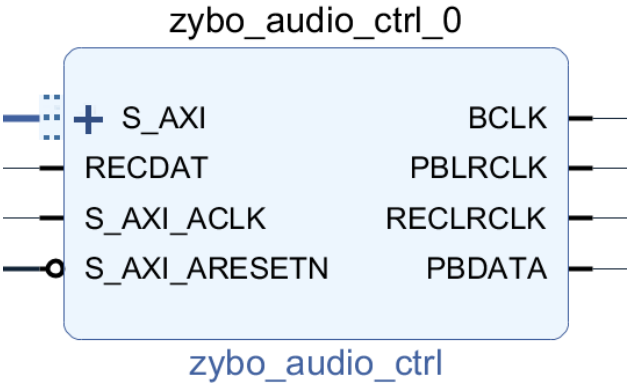


Figure 8. Audio control IP

The SSM2603 audio codec chip enables the Zynq-7000 AP SoC to handle integrated digital audio processing, supporting stereo recording and playback at sample rates ranging from 8kHz to 96kHz[7] . On the analog front, it connects to three standard 3.5mm audio jacks. The

digital interface of the SSM2603 is connected to the programmable logic section of the Zynq. It uses the I2S protocol to transfer audio data, and configuration is managed through an I2C bus. The relevant pins of the device are shown below.

SSM 2603 Pin	Function	Direction from FPGA
BCLK	I2S(Serial Clock)	Output
PBDAT	I2S(Playback Data)	Output
PBLRC	I2S(Playback Channel Clock)	Output
REDCAT	I2S(Record Data)	Input
RECLRC	I2S(Record Channel Clock)	Output
SDIN	I2C(Clock)	Input/Outputs
SCLK	I2C(Clock)	Output
MUTE	Digital Enable	Output
MCLK	Master Clock	Output

Table 1. Audio Codec Configuration Pins

As our input is based on human voice, we don't have to use a very high sampling rate, here we set the ADC and DAC sampling rate to be 24KHz

MCLK (CLKDIV2 = 0)	MCLK (CLKDIV2 = 1)	ADC Sampling Rate (RECLRC)	DAC Sampling Rate (PBLRC)	USB	SR[3:0]	BOSR	BCLK (MS = 1)[1]
12.288 MHz	24.576 MHz	8 kHz (MCLK/1536)	8 kHz (MCLK/1536)	0	0011	0	MCLK/4
		8 kHz (MCLK/1536)	48 kHz (MCLK/256)	0	0010	0	MCLK/4
		12 kHz (MCLK/1024)	12 kHz (MCLK/1024)	0	0100	0	MCLK/4
		16 kHz (MCLK/768)	16 kHz (MCLK/768)	0	0101	0	MCLK/4
		24 kHz (MCLK/512)	24 kHz (MCLK/512)	0	1110	0	MCLK/4
		32 kHz (MCLK/384)	32 kHz (MCLK/384)	0	0110	0	MCLK/4
		48 kHz (MCLK/256)	8 kHz (MCLK/1536)	0	0001	0	MCLK/4
		48 kHz (MCLK/256)	48 kHz (MCLK/256)	0	0000	0	MCLK/4
		96 kHz (MCLK/128)	96 kHz (MCLK/128)	0	0111	0	MCLK/2

Figure 9. Sampling Frequency Chosen

To initialize the audio codec, some provided functions are useful for setting up the configuration registers. Here we choose the audio data format to be 32 bit, I2S audio data transmission mode under 24 KHz sampling frequency as well as other default working mode configurations.

```
void AudioPllConfig() {
    printf("start config\r\n");
    AudioWriteToReg(R15_SOFTWARE_RESET,               0b000000000); //Perform Reset
    usleep(75000);
    AudioWriteToReg(R6_POWER_MANAGEMENT,              0b000110000); //Power Up
    AudioWriteToReg(R0_LEFT_CHANNEL_ADC_INPUT_VOLUME,  0b000010111); //Default Volume
    AudioWriteToReg(R1_RIGHT_CHANNEL_ADC_INPUT_VOLUME, 0b000010111); //Default Volume
    AudioWriteToReg(R2_LEFT_CHANNEL_DAC_VOLUME,        0b001111001);
    AudioWriteToReg(R3_RIGHT_CHANNEL_DAC_VOLUME,       0b001111001);
    AudioWriteToReg(R4_ANALOG_AUDIO_PATH,             0b000010010); //Allow Mixed DAC, Mute MIC
    AudioWriteToReg(R5_DIGITAL_AUDIO_PATH,            0b000000111);
    AudioWriteToReg(R7_DIGITAL_AUDIO_I_F,             0b000001110); //I2S Mode, set-up 32 bits
    AudioWriteToReg(R8_SAMPLING_RATE,                 0b000111000); //24 kHz Sampling Rate emphasis, no high pass
    usleep(75000);
    AudioWriteToReg(R9_ACTIVE,                        0b000000001); //Activate digital core
    AudioWriteToReg(R6_POWER_MANAGEMENT,              0b000100010); //Output Power Up
}
```

Figure 10. Audio Codec Configuration

To read out the input audio data and send data back to the audio codec, we utilize the function “Xil_In32(UINTPTR Addr)” and “Xil_Out32(UINTPTR Addr, u32 Value)” provided by the Xilinx library, which enables the data transformation between.

Control Flow

The control flow logic on the ARM side is shown in figure 11, when the ARM processor boots up, it starts with initialization and configuration of the audio codec and timer interrupt. It will read out a new input audio data in 24KHz and write it into the BRAM which indicates that the audio data stored in the BRAM is valid now for the FPGA side to do signal processing through the state machine. Therefore, it pulls up the “Val" signal and waits for the “Ack” signal to come back. Receiving the Ack signal means that the processing of audio data on the FPGA side has completed and the processed data has been stored in BRAM. Then it reads out the processed audio data and sends it to the audio codec for playing sound.

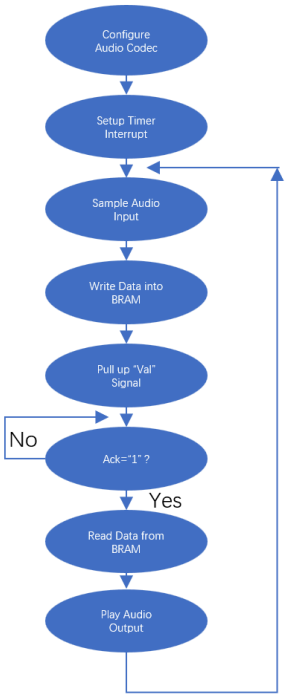


Figure 11. Control Flow

Besides, to guarantee that we always get the new input data after a specific time period 1/24KHz for synchronization between the processing of audio data between ARM and FPGA. We utilize the timer interrupt for counting that period of time to achieve the synchronization.

Communication between ARM & FPGA

The communication bridge between ARM and FPGA is implemented under the AXI bus protocol. AXI is part of ARM AMBA (Advanced Microcontroller Bus Architecture)[8], it supports the following three types of interfaces:

1. AXI4(-Full): high-performance storage mapping interface.
2. AXI4-Lite: A simplified version of the AXI4 interface, used for storage mapping communication with a smaller amount of data.
3. AXI4-Stream: used for high-speed data stream transmission, non-storage mapping interface.

The communication bridge between the ARM processor and programmable logic part contains five channels. The five channels of the AXI protocol have their own VALID/READY handshake signal pairs. The name of each channel handshake signal pair is shown in the table below:

Transaction channel	Handshake pair
Write address channel	**AWVALID, AWREADY**
Write data channel	**WVALID, WREADY**
Write response channel	**BVALID, BREADY**
Read address channel	**ARVALID, ARREADY**
Read data channel	**RVALID, RREADY**

Table 2. Channel handshake signal pairs

Here we use the AXI4 (full) bus. For the devices that hung on the AXI bus, we will assign a specific memory address range for it. On the FPGA, we just connect the corresponding ports of the device to our customized RTL module to control it, though the connection to the processor via the AXI interconnect is abstract for programmers, we can use the assigned address to point to the device that hangs on the bus we want to control.

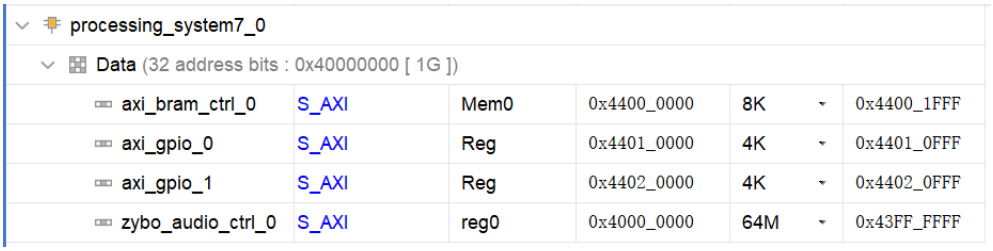


Figure 13. Address Assignment for Devices on AXI Bus

BRAM (Block RAM) is a type of memory array integrated on the FPGA. ARM and FPGA use BRAM to read and write to realize data interaction. In FPGA, the BRAM is read and written through the output clock, address, read and write control and other signals, while on the ARM side, the processor does not need to directly drive the BRAM port, but through the AXI BRAM controller to read and write the BRAM. On ARM processor, we utilize the two functions shown below to do the write and read operations

```
void psWriteBram( Xint32 send_data, u32 offset_add)
{
        XBram_WriteReg(XPAR_BRAM_0_BASEADDR, offset_add*4,send_data);
}

Xint32 psReadBram(u32 offset_add)
{
    Xint32 data_rec;
     data_rec= XBram_ReadReg(XPAR_BRAM_0_BASEADDR, offset_add*4);
     return data_rec;
}
```

Figure 14. BRAM Write & Read functions

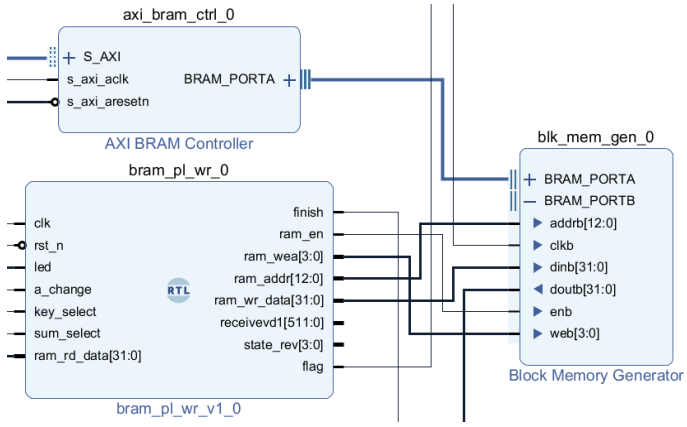


Figure 15. Connection for AXI BRAM

In the block design, we utilize the IP core "Block Memory Generator" to create a true dual port ram where two ports connect to the AXI BRAM controller and the customized RTL module respectively. We set up the BRAM works under default mode (Write First mode). The write and read operation timing sequence is shown in figure x. Each Ram port contains five inputs (ram_address, write_data, enable, write_enable and clock) and one output (read_data). The read and write operations share the address port, and the write_enable [3:0] indicates the type of operation (read or write). The reason why the width of it is 4 bits is that it supports byte operations, but we don't utilize this function in our design. Everytime our operation is based on the whole word (32 bit), so for write operation, we specify it to be 4'b1111 and for read it's all zeros.

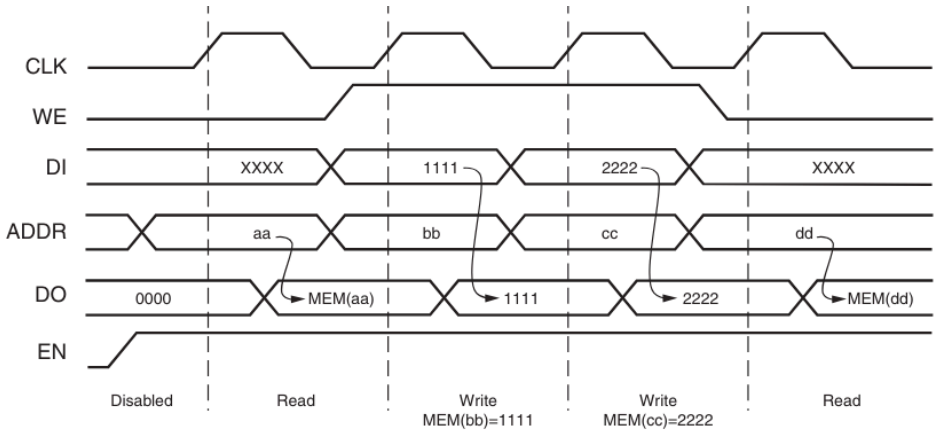


Figure 16. BRAM Read/Write Sequence Example

In WRITE_FIRST mode, the input data is simultaneously written into memory and stored in the data output (transparent write). For read, In latch mode, the operation uses one clock edge. The read address is registered on the read port, and the stored data is loaded into the output latches after the RAM access time.

Except for using AXI BRAM to achieve communication bridge between ARM processor and FPGA, we also utilize the AXI GPIO which is more convenient to configure for transmitting small amounts of data.

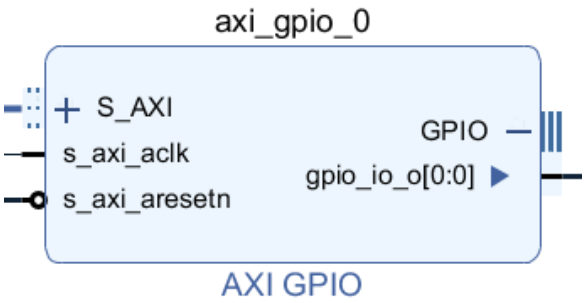


Figure 17. AXI GPIO Block

The handshake protocol signals (Val and Ack) on ARM processor and FPGA regarding whether the new input audio data has been stored in the BRAM is implemented through two dual-port AXI GPIOs.

Testing

Our testing follows a hierarchical and modular principle, as our entire system is composed of layers built from modules designed according to different functions. To ensure that all parts are correct and complete without causing issues for other components, we conducted extensive and thorough testing on each part. Initially, we tested the communication between the ARM processor and FPGA by performing read and write operations on each and observing the waveform from Integrated Logic Analyzer (a signal monitoring tool built into VIVADO), which is the foundational layer of our system. This step ensured that data transmission between the two was correct, and both could properly perform read and write operations on the shared BRAM.

Next, we tested the reception and transmission of audio signals by using an ARM processor to interact with an audio codec. We built an audio loopback system on the ARM processor. By inputting a sine wave or voice signal and then listening to the audio output through a speaker and observing the waveform on an oscilloscope, we verified the functionality of receiving and transmitting audio data. Then, we combined these two components; the ARM processor received audio data and passed it to the FPGA, which then returned the data to the ARM processor, establishing another audio loopback system with the same test method as before. This setup served as the basis for subsequent sound processing, as all sound processing occurred within the FPGA, between receiving audio data and transmitting it back to the ARM processor.

Following that, we tested the voice changing system. We started with testing the bandpass filtering effect of the IIR bandpass filters, then the low-pass effect of the low-pass filters, and the sine wave generation capability of the DDS. To verify the band-pass and low-pass effects, we input a series of sine waves at different frequencies into filters with specific center frequencies. We then used an oscilloscope to observe the output waveforms to determine if the filtering meets our expectations: signals within the filter's range should pass through normally, while signals outside the range should gradually attenuate to silence. For the DDS's sine wave generation capability, we directly output a sine wave of a specific frequency to the oscilloscope and checked if the frequency was correct. Finally, we combine the three components to test the initial robotic voice effect. We used an oscilloscope to compare the input and output waveforms for similar characteristics and directly listened to the output through a speaker to evaluate the result.

User control testing followed, where we used switches on the board to adjust parameters within the voice changer system, thus controlling the voice changing functions. The effectiveness was judged by directly listening to the voice output to see if it meets our desired effect. Finally, we tested the real-time microphone input system to observe the final outcome. The results were very satisfactory; our voice changer could process the incoming sound very quickly and output the modified voice effectively.

Results

DMA audio tutorial

The DMA audio demo provided by Digilent is useful for understanding the onboard resources and giving us some insights on the design structure of our overall system.[9] And this is also a teaching material that allows us to quickly become familiar with how to program and debug the board.

The hardware that the demo leverages includes four buttons, the embedded audio codec chip and DDR3 Memory. The audio demo can record a sample over a period of time(default 5s, can change by revising parameter) from microphone input or line input port and then plays it back with a headphone output port.

Audio playback and recording are controlled by push buttons as below.

Button	Function
BTN0	Play default audio
BTN1	Record audio from mic input port
BTN2	Play back audio on hph output port
BTN3	Record audio from line input port

Table 3. Button functions of DMA audio tutorial

The system module diagram is shown in figure 18, it mainly contains 5 functional modules. The ZYNQ7 Processing System module(based on ARM dual cortex core) is the central control unit which is responsible for controlling the audio codec chip including driving the chip, storing the audio data to memory, and sending audio data from memory to audio codec chip. In this process, for the chip driving part, it executes software programs to implement the I2C and I2S protocol (Using the SDK provided by Xilinx, developers can write C or C++ programs that interact with the codec chip through the processor's interface to the FPGA by AXI and EMIO). For doing load and store data operation with DDR memory and communicating with the FPGA part, it leverages the AXI communication protocol and DMA technique. Other four functional

modules (AXI, DMA, I2C and UART) are just defining the hardware connection pins and setting the protocol configuration.

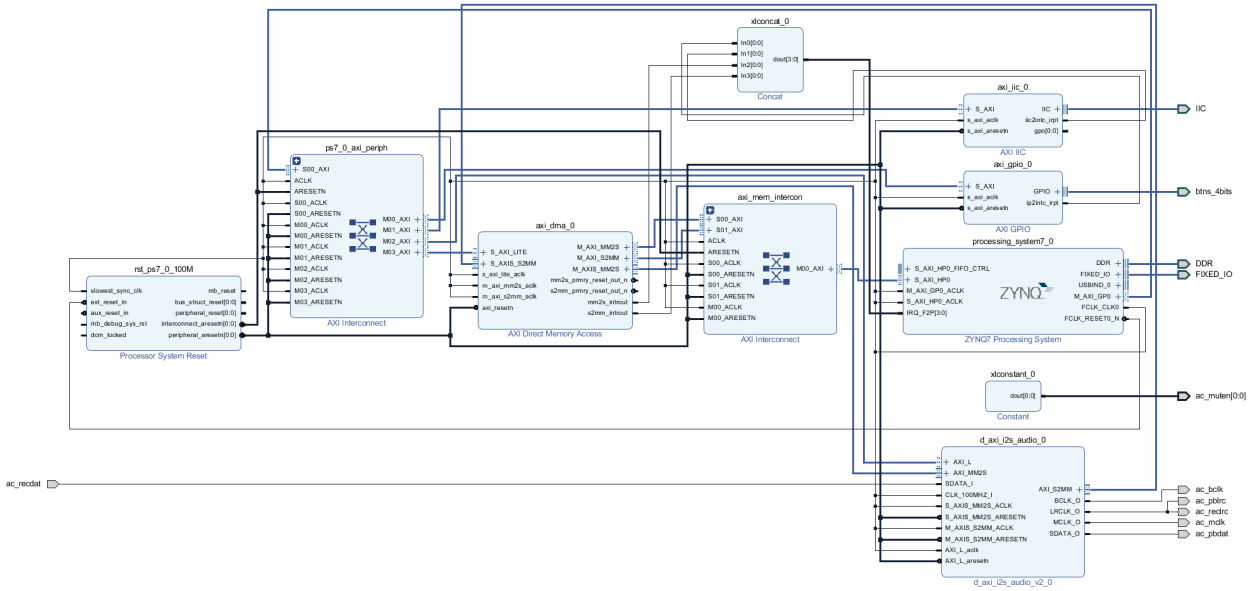


Figure 18. System block diagram

After loading the program on zybo-7000, we connect the HPH output port to a speaker and the Line input port to the laptop. We use the putty to communicate with the board with Uart communication protocol.

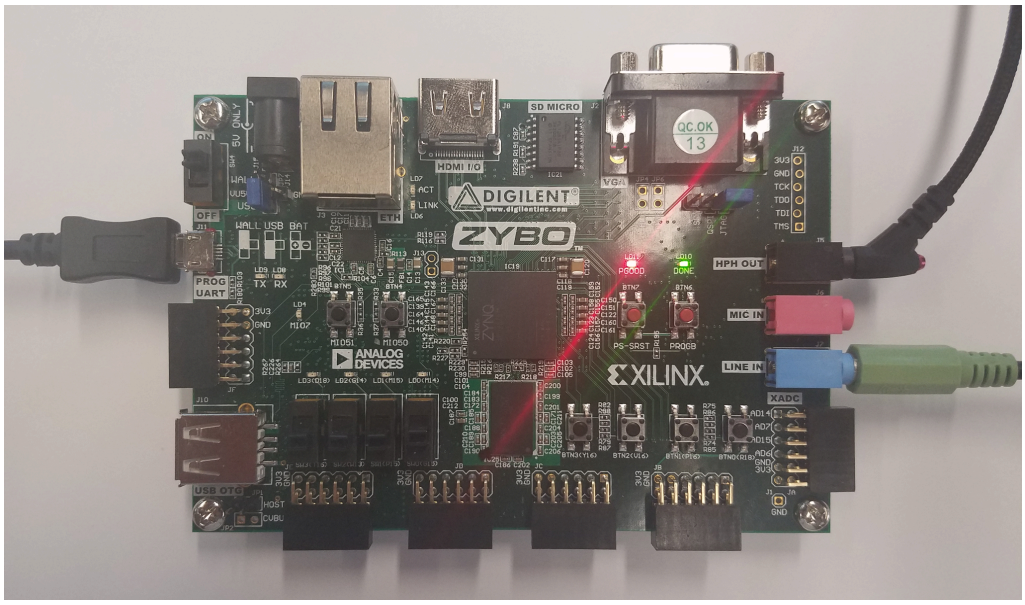


Figure 19. Hardware connection

We push the button BTN3 and play music on the laptop, it will display “Start recording” on the terminal and after 5 seconds, it prints out “Recording Done”. During recording, pushing any buttons will not affect it and it prints “Still Recording”. Then we push the button BTN, the speaker plays back the music we just played on the laptop which means it works well.

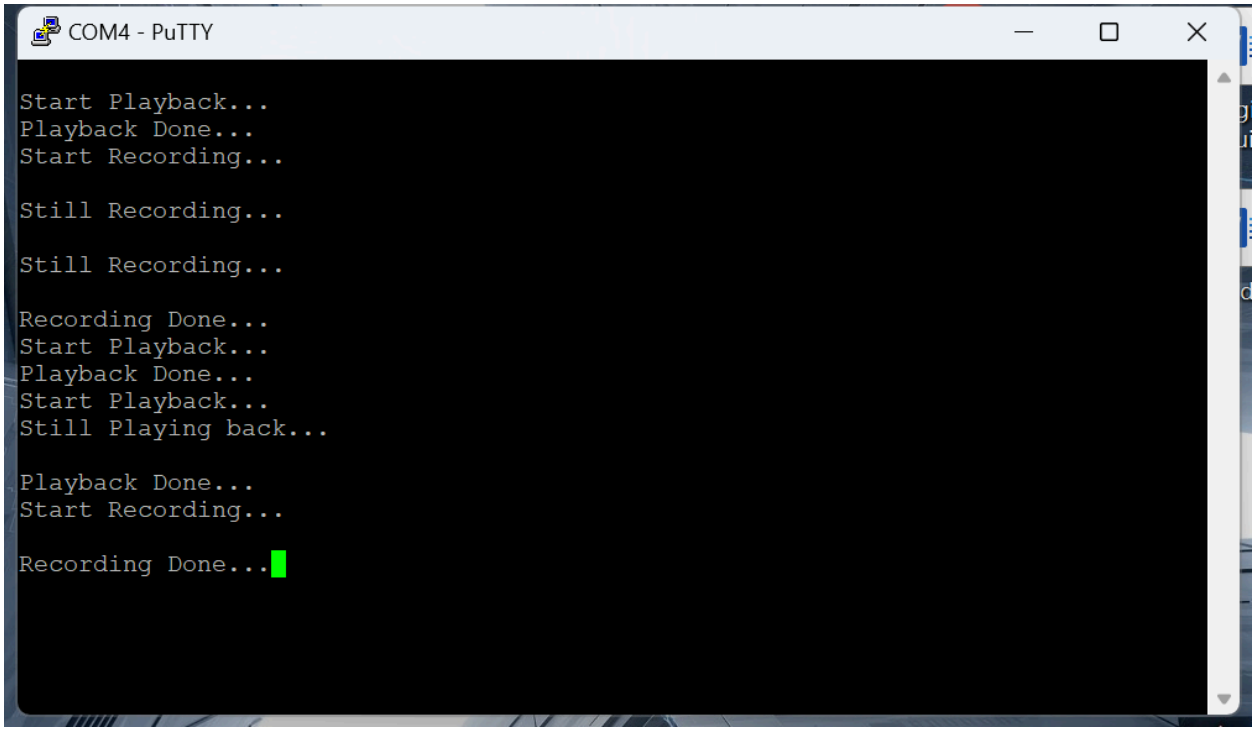


Figure 20. DMA Audio demo test

Petalinux system configuration

In 2023 Fall, we utilized PetaLinux to integrate Linux and FPGA technologies in our project, which can customize, build and deploy Embedded Linux solutions on our processing system. Specifically, we have built a PetaLinux project and generated the necessary boot image file. We then copied this boot image to the root directory of an SD card. Following this, we set the Zynq board's boot mode jumpers to the SD card mode, allowing us to utilize PetaLinux on this board. The figure below shows the use of a Serial Terminal application to receive messages printed by the PetaLinux system on the board.

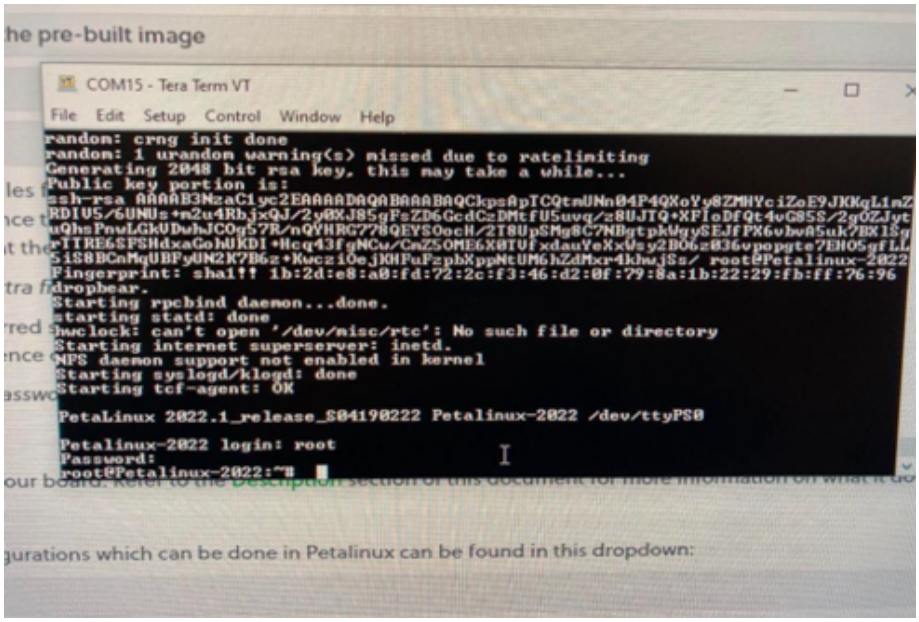


Figure 21. Received messages printed by the PetaLinux system on the board

As shown in figure 22, it can also display its command lines on a monitor through an HDMI connection.

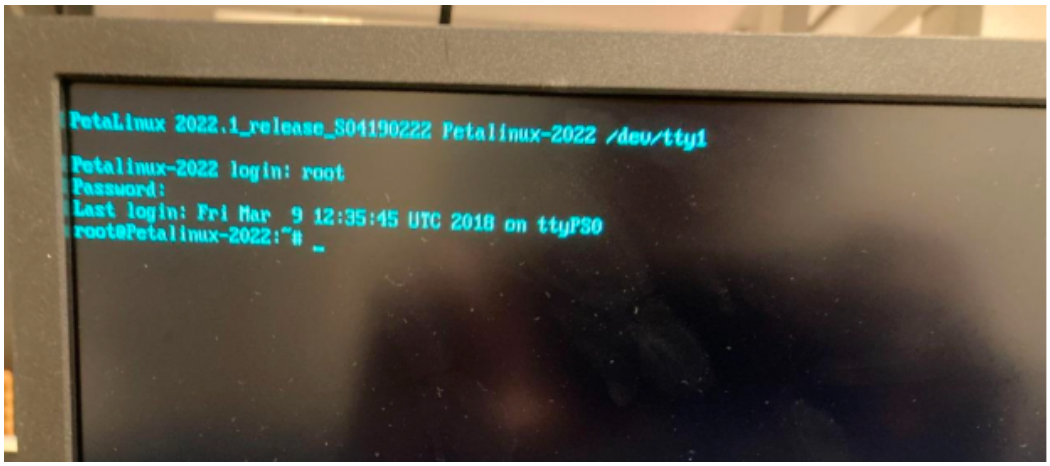


Figure 22. PetaLinux system being output to a monitor through an HDMI connection

We successfully wrote and ran a program that prints "hello world" within this system. However, at the same time, we encountered a serious issue: each time we started our system, all files and previous settings were lost. It seems that the PetaLinux system was not properly mounted on the SD card, even though we strictly followed the official steps provided. We spent

some time trying to resolve this issue, but it was beyond our capabilities, and we ultimately decided to abandon this method of controlling the board. Fortunately, we still have the option to use the SDK program provided by Xilinx for writing and running C code on the ARM processor.

Communication bridge using AXI BRAM[10]

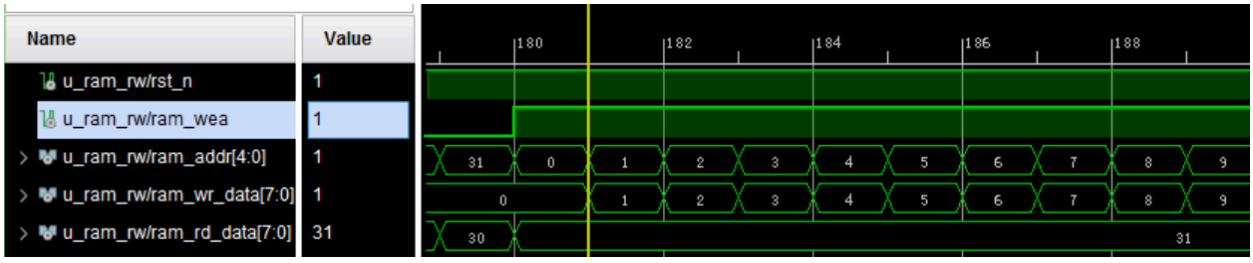


Figure 23. Write BRAM on FPGA

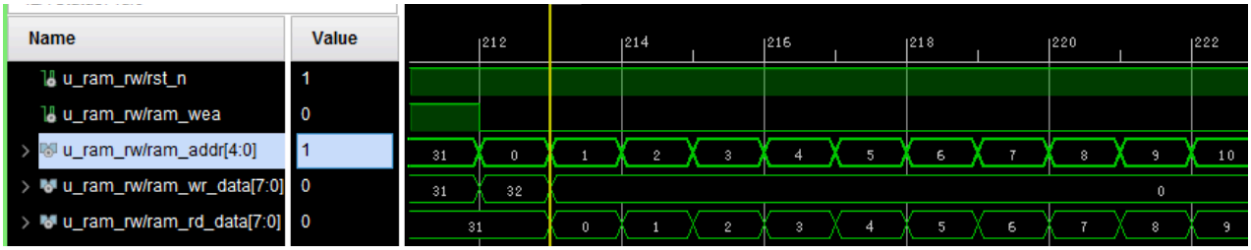


Figure 24. Read BRAM on FPGA

To verify the functionality of AXI BRAM implementation, for write operation we constantly write data value form 0-255 into address 0 - 255 both from FPGA side, in the logic analyzer it shows that the data is successfully written in the expected location. For read operation, we first wrote the same data array into the location of BRAM from the ARM side and then read from the FPGA. The waveform shows that we constantly read out new data one clock cycle after changing the address.

Audio loopback system

We constructed an audio loopback system to verify the correct input and output of sound. More specifically, we actually built two audio loopback systems. The first one was used to verify the ARM processor's functionality in reading audio data from the audio codec and writing it back, which, like the communication between the ARM and FPGA, is one of the most fundamental and crucial functions of our designed system. Figure 25 shows a waveform of an

audio displayed directly on an oscilloscope, and Figure 26 shows the waveform of this sound after passing through our audio loopback system. Although there are some differences due to digital sampling, the basic characteristics are similar. By connecting to a speaker instead of an oscilloscope, we could also directly hear that the output sound from both systems is nearly identical.

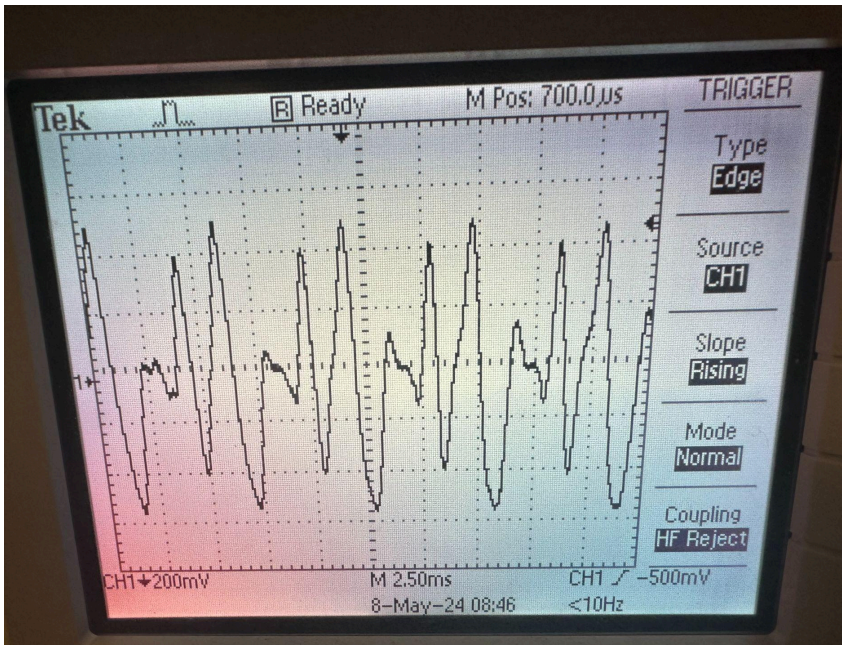


Figure 25. Waveform of audio input to loopback system

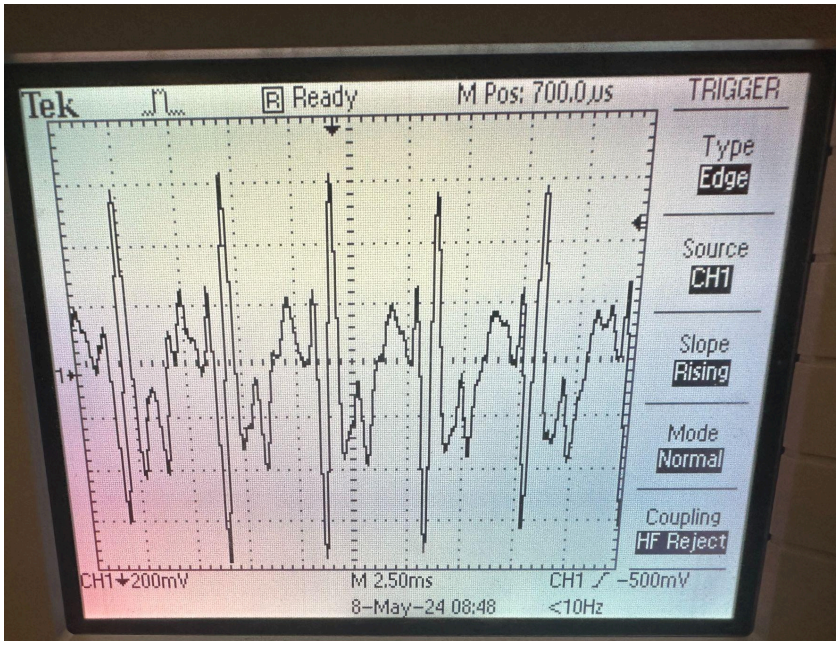


Figure 26. Waveform of audio output from loopback system

The second audio loopback system combined the sound read and write functions of the ARM processor with the communication function between ARM and FPGA. The ARM processor receives sound data, sends it to the FPGA by writing it to a specific address in BRAM, and then the FPGA writes the sound data to another BRAM address and sends it back to the ARM. The ARM then sends this sound data to the audio codec to be emitted externally. This setup is the basis for subsequent sound processing, as all operations to process sound are performed on the FPGA. This occurs between the ARM sending sound data to the FPGA and the FPGA sending it back to the ARM. The ability to perform this operation correctly is essential for us to implement further voice-changing functionality. Figure 27 shows the waveform of the same sound output after passing through the FPGA, similar to the direct output from the ARM, matching expectations, and the sound emitted through the speaker is also consistent.

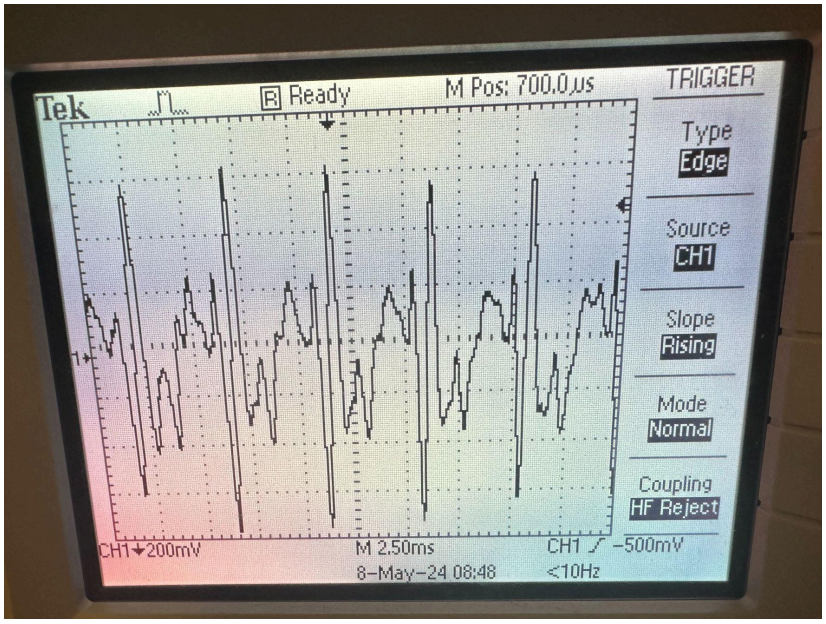


Figure 27. Waveform of audio output from FPGA loopback system

Filters

To achieve the voice-changing functionality, we constructed IIR bandpass filters and low pass filters to implement bandpass and lowpass filtering functions. We tested each filter by inputting sine waves of different frequencies to verify the normal operation of the filters. Specifically, we examined the filtering ranges of the 32 bandpass filters and the filtering range

corresponding to the specific parameter a of the low pass filter, all of which met our expectations. It's worth mentioning that we did not use the laboratory signal generator as its output signal waveform was not very stable, possibly due to incorrect configuration. Instead, we used a website capable of generating different sine wave audio outputs, and we fed these sounds into the board using a headphone cable connected to a laptop speaker.

Direct digital synthesis

We use Direct Digital Synthesis (DDS) for voice synthesis. Specifically, by reading a lookup table filled with sine wave data at a certain increment, we can construct sine waves of specific frequencies. Then, by multiplying the sound content by the values of this sine wave, we can carry the sound information to a specific frequency. We demonstrate the generated waveforms by directly outputting the specific frequency sine waves obtained from the lookup table at a set increment, connecting them to an oscilloscope to verify that their frequencies match the increments we have set.

Initial robot effect voice changer

By combining the earlier IIR bandpass filters, low pass filters, and DDS, we have created a preliminary robot effect voice changer. Figure 28 shows the input audio clip of "I am a robot", and Figure 29 shows the output audio clip. The characteristic values are almost identical. Additionally, when listening to the output sound clip from a speaker, you can hear a robotic effect of "I am a robot". The content is completely preserved, demonstrating the initial success of our design.

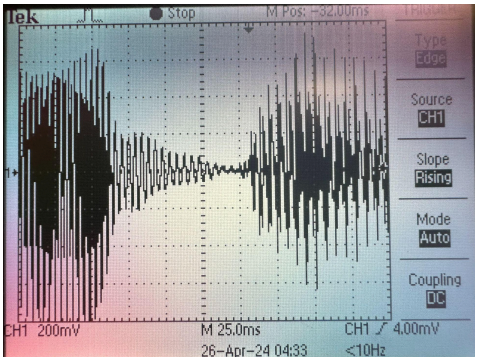


Figure 28. Waveform of "I am a robot"

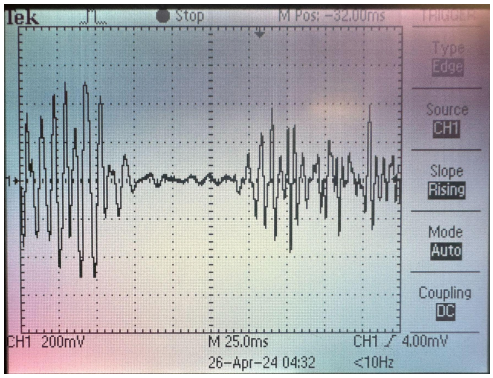


Figure 29. Waveform of robotic "I am a robot"

User interaction & Parameter test (alpha & pitch shift)

We use a switch to test different a' parameters in the low pass filter to find an appropriate value. After testing, the best effect occurs when a' is equal to 8, where the cutoff frequency is closest to 100 Hz, aligning with our initial assumptions.

Additionally, we use another switch to adjust the pitch level of the output sound, with five levels available: -2, -1, 0, 1, and 2. Negative numbers indicate a deeper sound, 0 represents the normal sound, and positive numbers correspond to higher pitches. Changes in the frequency can be clearly observed on the oscilloscope as we adjust the switch, and significant changes can also be heard through the speaker's output.

Final robot effect voice changer

Ultimately, we want to implement a voice changer that captures audio input in real-time from a microphone connected to the LINE IN port on a board, processes the sound through an FPGA, and outputs the altered sound in real-time via the board's HPH OUT port. However, due to the lab's microphone being non-functional—possibly because the board's audio port does not supply the necessary voltage to power the microphone, and this particular microphone lacks alternative power sources—we resorted to another method of sound input. Previously, we tested sound by playing audio through a computer, which transmitted it to the board via a headphone cable. We continued with this approach for our final setup: users make a voice call through their mobile phone to a computer, which then forwards the received audio to the voice changer for

processing and outputs it to a speaker. The entire system is illustrated in Figure 30. The results are consistent with our expectations, although using voice calls introduced a slight delay, it did not affect functionality.

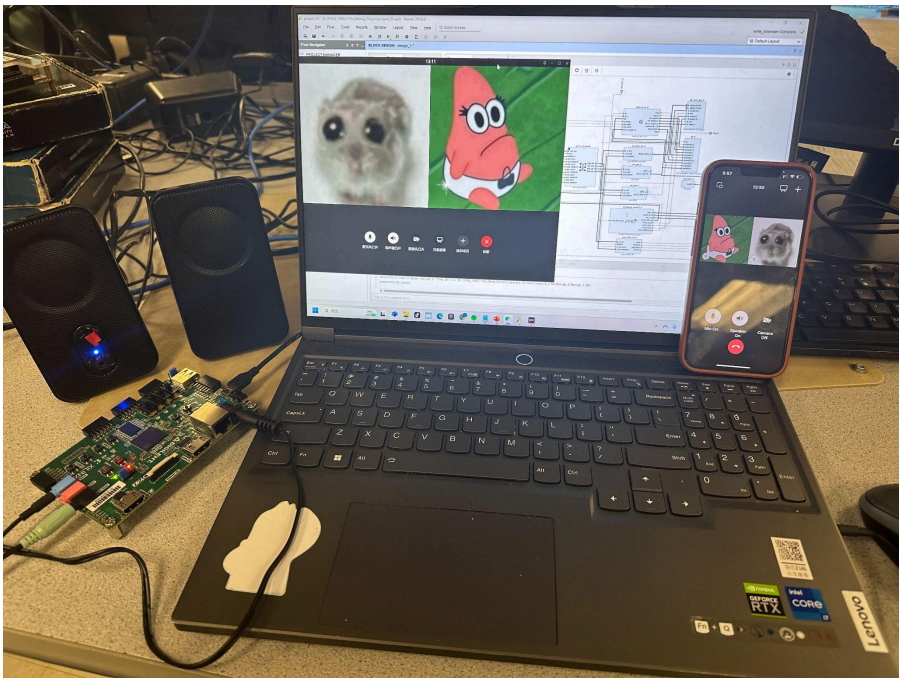


Figure 30. Final Voice Changer

Conclusion

Summary

We used the Zybo-Z7 board to build a robotic effect voice changer. The voice changer works as expected: whenever we speak into the microphone, the speaker outputs a robotic-sounding voice in real-time, processed by the board. Adjusting the switches on the FPGA changes the pitch of the output, making the voice sound higher or lower. In our system, the audio codec is driven by an ARM processor for audio input and output, and it connects to the FPGA via the AXI Block RAM and AXI GPIO bridge. The FPGA includes custom RTL modules to facilitate successful read and write operations on the AXI Block RAM. This setup lays the foundation for our voice changing system, where the incoming audio signal is processed through IIR band-pass filters, low-pass filters, and Direct Digital Synthesis implemented in the FPGA. Our system has three main advantages. First, being built on an FPGA, it leverages the parallel processing capabilities of the FPGA for efficient and real-time audio processing. Second, our system is scalable and customizable, allowing easy modifications of our RTL modules to introduce more layers of effects or additional filters to meet specific user needs. Third, the enclosed, entirely hardware-based nature of our system makes it less susceptible to hacking and data breaches, further enhancing user privacy.

Performance

Figure 31 shows the resources on the board used by our designed voice changer. The extensive use of LUT resources is primarily due to the total of 32 fixed-point multipliers within the 32 filters. As can be seen, there are still many idle resources available, which facilitates further functional expansion.

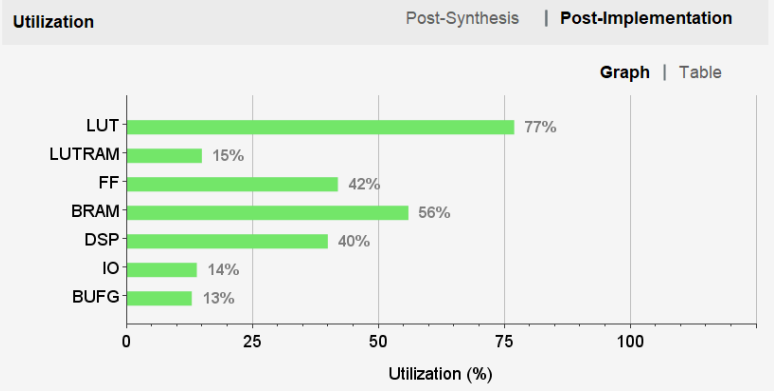
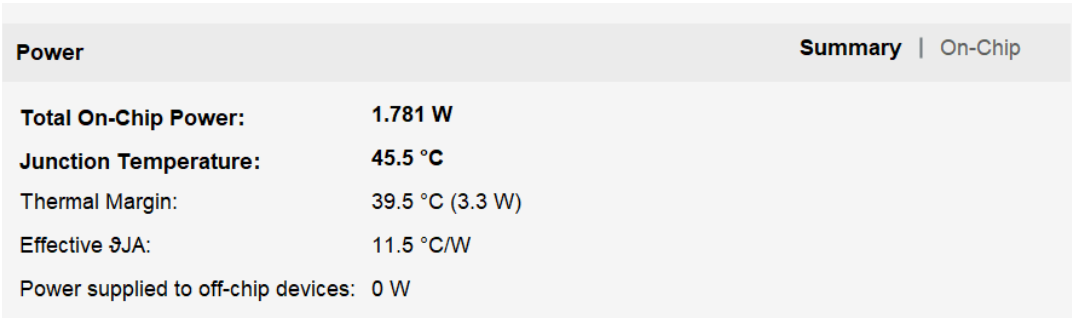


Figure 31. Resource utilization

Figure 32, on the other hand, displays the power consumption of our designed voice changer. The total on-chip power of our design is 1.781 W, which demonstrates that our voice changer performs well with very low power consumption.



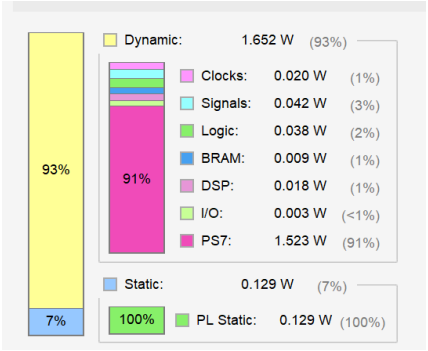


Figure 32. Power consumption

Potential Improvements

Our designed voice changer has many areas for improvement. First, by using a microphone that can be self-powered or connected to an external power source, our voice changer could obtain more real-time input, eliminating the propagation delays associated with voice calls. Secondly, we could explore adding more voice changing effects, not just altering the pitch. We could attempt to overlay voice data with sounds from other animals, like birds, to create an effect that mimics birds talking. Additionally, utilizing the voice information extracted during our sound processing, we could implement more complex features, such as voice recognition, which would enable voice control. For instance, when the system detects certain specific words spoken by the user, like 'higher,' the voice changer could automatically increase the pitch. More complex functionalities could also be explored, such as integrating machine learning. By extracting a person's voice characteristics to train a model, we could then modify others' voices to mimic that individual's voice. However, this would be highly complex and might encounter resource limitations on the board.

Reference

[1] Zybo Z7 Reference Manual

https://digilent.com/reference/programmable-logic/zybo-z7/reference-manual

[2] “DSP,” people.ece.cornell.edu.

https://people.ece.cornell.edu/land/courses/ece5760/DE2/fpgaDSP.html

[3] “Fixed-Point Arithmetic”, vanhunteradams.com,

https://vanhunteradams.com/FixedPoint/FixedPoint.html

[4] Mel-Frequency Cepstrum Theory, https://en.wikipedia.org/wiki/Mel-frequency_cepstrum

[5] Mel scale, https://en.wikipedia.org/wiki/Mel_scale

[6] “DDS”, vanhunteradams.com, https://vanhunteradams.com/DDS/DDS.html

[7] SSM2603 Datasheet, https://www.analog.com/en/products/ssm2603.html

[8] Vivado Design Suite: AXI Reference Guide (UG1037),

https://docs.amd.com/v/u/en-US/ug1037-vivado-axi-reference-guide

[9]“Zybo Z7 DMA Audio Demo - Digilent Reference,” digilent.com.

https://digilent.com/reference/programmable-logic/zybo-z7/demos/dma-audio (accessed Apr. 22, 2024).

[10] 7 Series FPGAs Memory Resources User Guide (UG473),

https://docs.amd.com/v/u/en-US/ug473_7Series_Memory_Resources

Appendix

Code:

https://github.com/Carl782051063/Cornell-Meng-Project.git