ECE532

Final Report

Prepared by:

Ruben Tjhie **Farhad Yusufali** **Pranali Rathi**

Professor: **Professor Paul Chow**

TA: **Fernando Martin del Campo**

Table Of Contents

1.0 Overview

1.1 Background & Motivation

The recent introduction of various new products in the music industry has transformed the experience of listening to music significantly; these products include things like Spotify, music streaming, Apple AirPods, wireless headphones, Sonos and smart speakers. However, they provide limited real time audio processing, a deficit that serves as the primary motivation for this project.

1.2 Goals

The goal of the project is to create an interconnected music streaming system that explores the intricacies of building a rich listening experience for users. Specifically, we are modelling a system where music is streaming from an ethernet connected server to a computer, while a Bluetooth device is being used to control the playback of music. An analogous system would be Spotify's servers sending an audio file to a smart speaker, while a phone is being used to control playback via Bluetooth. We additionally store the audio from the server in DDR memory, effectively treating it like a cache to allow playback even without an available internet connection. We introduce several real-time audio processing features in order provide an unique advantage over existing systems. Specifically, we implement two filters:

- An Echo Filter: As the name implies, the filter adds an echo to the song. The output signal $y[n]$ is defined in terms of the input signal $x[n]$ using the equation below:

  $$y[n] = \alpha x[n] + \beta x[n - \tau]$$

  Here, τ represents an arbitrary constant.

- A Chorus Filter : This filter creates a "chorus" like effect; the output signal $y[n]$ is defined in terms of the input signal $x[n]$ using the equation below:

  $$y[n] = \alpha x[n] + \beta x[n - \tau(n)]$$

  Here, $\tau(n)$ represents a sinusoidal signal whose value is retrieved from a look-up table. This filter can also behave as a *flange* filter if desired by simply modifying the frequency of the sinusoidal in the lookup table (we provide a script to generate look-up table entries for an arbitrary sinusoidal)

1.2 Block Diagram

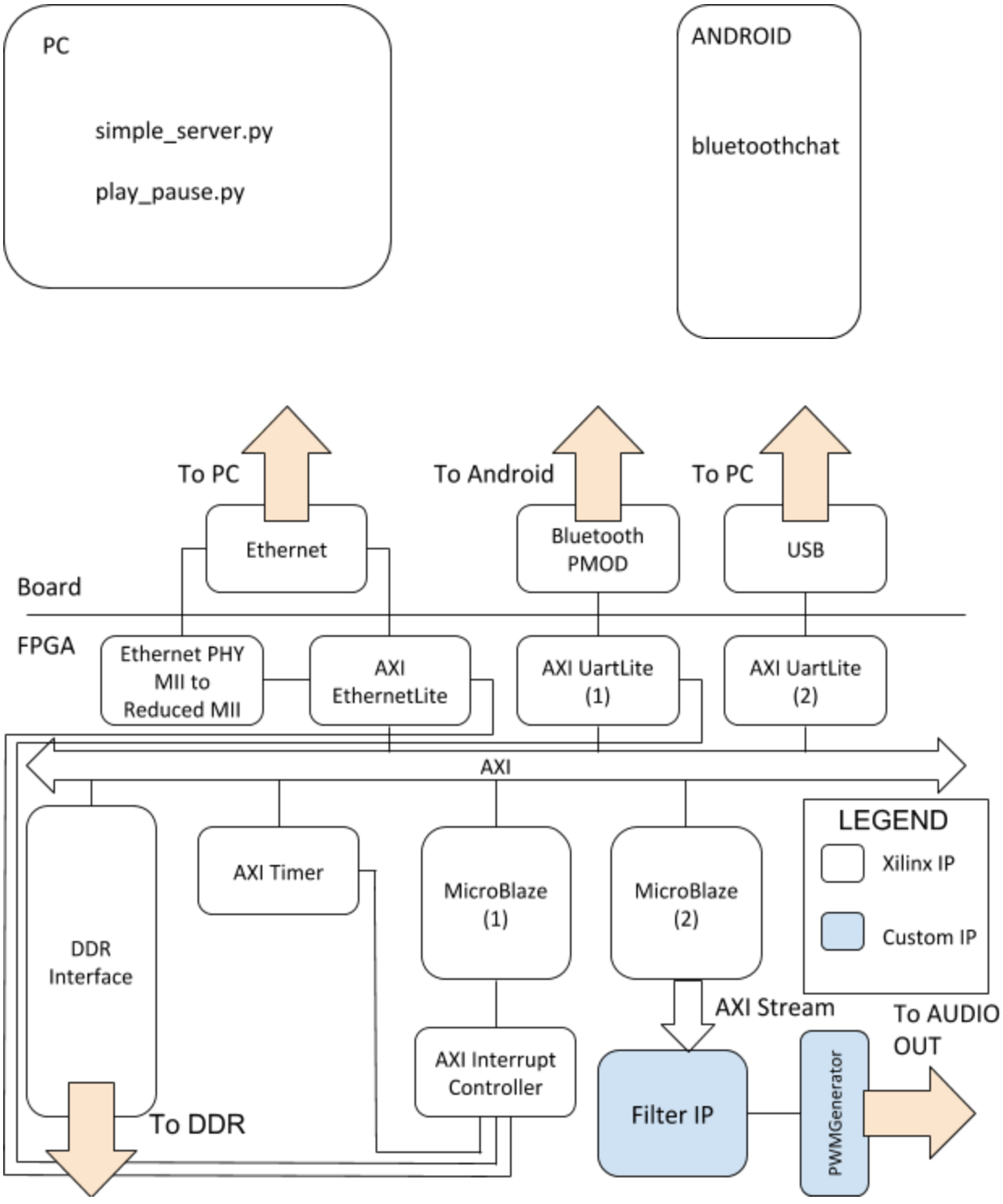

Figure 1 - *Complete System Block Diagram*

1.3 List of IPs

Brief descriptions of components used are as described in the table below.

Components	**Function**	**Source**
On Board		
Microblaze (1)	Processes all Ethernet and Bluetooth communication	Xilinx
Microblaze (2)	Streams music data from DDR memory to Filter IP	Xilinx
Ethernet to PHY MII to Reduced MII	Allows Ethernet communication	Xilinx
AXI EthernetLite	Allows Ethernet communication	Xilinx
AXI UartLite (1)	UART interface to Bluetooth PMOD	Xilinx
AXI UartLite (2)	UART interface to USB	Xilinx
MIG 7	DDR memory interface	Xilinx
AXI Timer	Network timeout	Xilinx
AXI Interrupt Controller	Generates interrupts for Microblaze from AXI components	Xilinx
Filter IP	Custom audio filters	Team
PWMGenerator	Converts PCM stream into PWM stream to output to board audio	Team
Off Board		
PmodBT2	Bluetooth module	Digilent
PC	Acts as music server, music player, and Bluetooth controller	Team
Android device	Bluetooth controller	Team
Software		

simple_server.py	Python module that received requests for music files OR receives music stream and outputs audio	Team
play_pause.py	Python module that sends data byte over Bluetooth to board	Team
Bluetoothchat	Android code that sets up and handles bluetooth connection. The original code allowed for two android devices to send text messages over bluetooth, but was modified for this project to include buttons to send specific commands over bluetooth to the FPGA.	Team/Google Android
Sine LUT Generator	Python script that generates sinusoidal samples to populate the LUT in the filters	Team
MicroBlaze SW	C code that processes network communication and saves music stream in DDR memory	Team

Table 1 - *Table listing the IPs and resources used in the project, and their sources*

2.0 Outcome

The overall goal of an interconnected music streaming system was achieved. The system requested music from an external server through a TCP connection using the Ethernet interface and stored it in DDR memory. At the same time, as music was received the system sent this music stream through a separate TCP connection to be played through remote speakers. The control of the music stream was activated through a remote Bluetooth controller to pause and play the music.

However, the integration of the custom filters and the Android controller were not completely successful. Although the components alone were functionally correct, when integrated into the system, they were met with interoperability issues. Additionally, in order to avoid having to work with the FFT module (due to a lack of time), an echo filter was implemented in place of the equalizer filter. An obvious improvement to our system would be to integrate these components. A considerable extension to our system would be to develop the capability to handle compressed audio files. Currently, the system works with uncompressed WAV files. These files strain the communication channels (Ethernet, Bluetooth) and use precious memory resources.

Additional extensions to the project could be to implement more sophisticated control features such as rewind, fast-forward, skip, song selection. More filters could be added to the system such as an equalizer, as well as volume control. These additions could be accessed through a menu system on the Android device.

Ultimately, if the team were to start over, more focus on earlier and iterative integration of components would be encouraged. A more accurate way to measure the progress of the project would be to see how many components have been successfully integrated. This is also a good way to verify that the interface protocols being used are appropriate.

2.1 Functional Requirements

Requirements	**Modified**	**Status**
Stream audio from server to FPGA via ethernet		Complete
Apply requested filter on audio files on FPGA		Incomplete
~~Send audio files from first FPGA to second FPGA via Bluetooth~~ Send audio files from FPGA to networked speaker	X	Complete
Output audio from FPGA board with desired filter		Incomplete
Play and pause music through Bluetooth	New	Complete
Cache music in DDR	New	Complete

Table 2 - *Project Functional Requirements*

2.2 Acceptance Criteria

Criteria	**Modified**	**Status**
~~An audio file from the laptop can be reproduced on the second FPGA~~ An audio file from laptop can be reproduced on networked speaker	X	Complete
The filtered audio from the (~~second FPGA)~~ custom IP matches the the output from the filter implemented in software	X	Complete

Table 3 - *Project Acceptance Criteria*

3.0 Project Schedule

3.1 Milestone 1

Milestone #1 - ORIGINAL	
Date:	Feb 5
Responsibilities	
Ruben	
Farhad	
Pranali	
Team	Research and design of the various components of our overall system in finer granularity

Table 4 - *Proposed Milestone 1*

Milestone #1 - ACTUAL	
Date:	Feb 5
Responsibilities	
Ruben	
Farhad	
Pranali	
Team	Research and design of the various components of our overall system in finer granularity

Table 5 - *Actual Milestone 1*

Milestone 1 was completed successfully, as the entire team participated in our system's design discussion.

3.2 Milestone 2

Milestone #2 - PROPOSED

Date:	Feb 12
Responsibilities	
Ruben	Set up server and establish TCP/IP connection between server and FPGA board and be able to transfer file
Farhad	Design detailed block diagrams for both filters
Pranali	Establish a Bluetooth connection between FPGA and another device
Team	

Table 6 - *Proposed Milestone 2*

Milestone #2 - ACTUAL	
Date:	Feb 12
Responsibilities	
Ruben	Built server software that transmits .wav file over TCP to client software on LAN using Python running on same machine. Ethernet connection to FPGA established and able to send and receive packets with Code for MicroBlaze client app.
Farhad	Block diagrams for both filters completed
Pranali	Bluetooth connection established between FPGA (using the pmodbt2) and Android device using a Bluetooth terminal app.
Team	

Table 7 - *Actual Milestone 2*

Milestone 2 was completed successfully, as we were able to set up the server and our network connection, design our filter blocks and setup a Bluetooth connection between the FPGA and an other device.

3.3 Milestone 3

Milestone #3 - PROPOSED	
Date:	Feb 26
Responsibilities	

Ruben	Output audio from FPGA
Farhad	Implement the chorus filter block
Pranali	Set up Bluetooth connection between 2 FPGA boards
Team	

Table 8 - *Proposed Milestone 3*

Milestone #3 - ACTUAL	
Date:	Feb 26
Responsibilities	
Ruben	Able to send audio file over TCP to board and echo the file back in real-time and output audio on laptop. Completed code for DDR2 SDRAM to save audio file on board
Farhad	Completed circular buffer, added in a sinusoidal LUT
Pranali	Can send commands and data from ready-made Android app and receive reply from Bluetooth module
Team	

Table 9 - *Actual Milestone 3*

Milestone 3 was modified from the originally proposed milestone due to changes in our design. Instead of using two FPGA boards connected over Bluetooth, it was decided that a single FPGA would be used with another device, for simplicity's sake.

3.4 Milestone 4

Milestone #4 - PROPOSED	
Date:	March 5
Responsibilities	
Ruben	Receive audio via Bluetooth and output to audio
Farhad	Implement FFT of audio signal
Pranali	Set up and test DDR

Team	

Table 10 - *Proposed Milestone 4*

Milestone #4 - ACTUAL	
Date:	March 5
Responsibilities	
Ruben	Researched options for multiple threads/CPUs for testing multithreading on Microblaze
Farhad	Finished implementing chorus filter, read FFT block documentation
Pranali	Looked into Android documentation on how to write Bluetooth apps and created an Android test app
Team	

Table 11 - *Actual Milestone 4*

Due to previous modifications to our milestone and design, there is a discrepancy between the proposed and actual milestone 4. Since streaming music over Bluetooth seemed to take too long for this system to be real-time, it was decided only commands would be sent via Bluetooth. In addition, as DDR had already been set up the previous week, the Android app was developed instead.

3.5 Milestone 5

Milestone #5 (Mid-Project Demo) - PROPOSED	
Date:	March 12
Responsibilities	
Ruben	
Farhad	
Pranali	
Team	Integrate filter and ethernet to be able to stream audio file from server, apply filter and play audio. Be able to select filter options.

Table 12 - *Proposed Milestone 5*

Milestone #5 (Mid-Project Demo) - ACTUAL	
Date:	March 12
Responsibilities	
Ruben	
Farhad	
Pranali	
Team	Can successfully communicate across the entire end to end system. We choose to integrate DDR2 memory over the filter this week.

Table 13 - *Actual Milestone 5*

With the goal of starting to combine the entire system together, we faced challenges that caused delays in our workflow. Thus, it was decided that the focus for this milestone would be integrating the DDR2 memory with the filter, instead of integrating all parts at once.

3.6 Milestone 6

Milestone #6	
Date:	March 19
Responsibilities	
Ruben	
Farhad	
Pranali	
Team	Integrate filter, ethernet and Bluetooth to be able to stream audio file from server, apply filter, send audio file over Bluetooth and play audio. Be able to save and play audio file to and from DDR.

Table 14 - *Proposed Milestone 6*

Milestone #6	
Date:	March 19

Responsibilities	
Ruben	Implemented PCM to PWM hardware. Currently testing.
Farhad	Completed chorus filter, passed song through filter to test it, implemented in Matlab and compared output
Pranali	Implemented interrupt controller for Bluetooth and Android app. Finished Android App and able to send special characters from app to board
Team	

Table 15 - *Actual Milestone 6*

As a result of last milestone's integration challenges, this week focused on different tasks that would allow for easier integration. The PCM to PWM conversion was to be able to apply the filters to the audio files and the interrupts were developed in order to integrate the Bluetooth commands with the rest of the system.

3.7 Milestone 7

Milestone #7 - PROPOSED	
Date:	March 26
Responsibilities	
Ruben	
Farhad	
Pranali	
Team	Complete implementing and integrating the equalization filter.

Table 16 - *Proposed Milestone 7*

Milestone #7 - ACTUAL	
Date:	March 26
Responsibilities	
Ruben	
Farhad	

Pranali	
Team	Integrated the Bluetooth interrupt with the existing system containing the network connection protocol. End to end system completed, where audio is transmitted to FPGA, Bluetooth sends commands and audio is sent back to the PC to be played out.

Table 17 - *Actual Milestone 7*

The main discrepancy between the proposed final milestone and what was produced was being unable to integrate the filter IP blocks with our overall system. The Android app was also unable to be integrated. Additionally, as implementing the equalizer filter would require time learning about the FFT IP, we choose to implement the echo filter instead. While each individual component was complete, integration challenges prevented the completion of the entire system.

4.0 Blocks Description

4.1 Echo Filter

The echo filter outputs a weighted sum of the current value of the audio signal and a previous value (weights of ¾ and ¼ were used respectively), where the delay corresponding to the previous value is given by a constant.

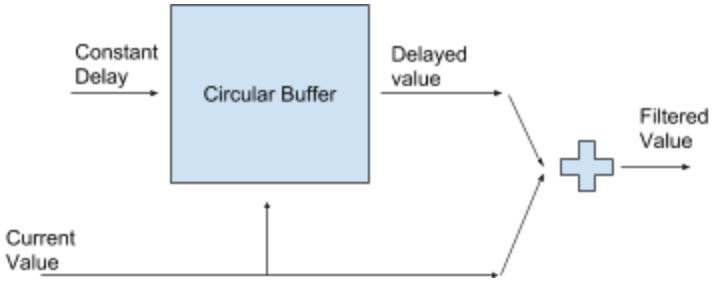

Figure 2 - *Diagram of Echo Filter*

Clearly, the echo filter's primary component is the circular buffer (described below).

To verify the correctness of the filter, the filter was implemented in MATLAB and the output of the hardware implementation was compared against this. The filter was verified to work successfully. As the rest of the system wasn't complete, in order to retrieve the output of the filter, the output from the IP block was generated in simulation. An excerpt from the testbench to generate the output is included below:

```
if(!$feof(musicFile))
begin
    scannedData = $fscanf(musicFile, "%h\n", sample);

    dataIn = sample;
    $fwrite(fMusicFile, "%h\n", dataOut[7:0]);
    resetn = 1;
    enable = 1;
end
else
begin
    $display("Finished!\n");
    $fclose(musicFile);
    $fclose(fMusicFile);
    $finish;
end
```

Figure 3 - *Excerpt from testbench to simulate filter*

```
for sample = 1:length(music)
    circularBuffer(writeAddress) = music(sample);
    filteredSample = int16(int16(bitshift(music(sample), -1)) + int16(bitshift(circularBuffer(readAddress), -1)));
    fprintf(filteredMusicFile, '%x\n', filteredSample);

    if writeAddress + 1 > length(circularBuffer)
        readAddress = 1 - delay;
    else
        readAddress = (writeAddress + 1 - delay);
    end

    if readAddress <= 0
        readAddress = readAddress + length(circularBuffer);
    end

    writeAddress = writeAddress + 1;

    if (writeAddress > length(circularBuffer))
        writeAddress = 1;
    end
end
```

Figure 4 - *Excerpt from MATLAB Implementation of Echo Filter*

The rest of the system interfaced with the echo filter using an AXI Stream Slave interface. In order to verify the functionality of the interface, the AXI Stream VIP module was used; an excerpt of the test bench is included below:

```
initial
begin
    //axi4stream_transaction wr_transaction;
    mst_agent = new("master vip agent",DUT.design_1_i.axi4stream_vip_0.inst.IF);

    mst_agent.vif_proxy.set_dummy_drive_type(XIL_AXI4STREAM_VIF_DRIVE_NONE);
    mst_agent.start_master();

    fork
       begin
          $display("Simple master to slave transfer example with randomization completes");
          for(int i = 0; i < 6;i++) begin
             mst_gen_transaction();
          end
       end
    join_any
end

task mst_gen_transaction();
  axi4stream_transaction                        wr_transaction;
  wr_transaction = mst_agent.driver.create_transaction("Master VIP write transaction");
  WR_TRANSACTION_FAIL: assert(wr_transaction.randomize());
  mst_agent.driver.send(wr_transaction);
endtask
```

Figure 5 - *AXI VIP Testbench*

4.2 Chorus Filter

The chorus filter outputs a weighted sum of the current value of the audio signal and a previous value (weights of ¾ and ¼ were used respectively), where the delay corresponding to the previous value is a sinusoidal signal. The sinusoidal values are retrieved from a LUT.

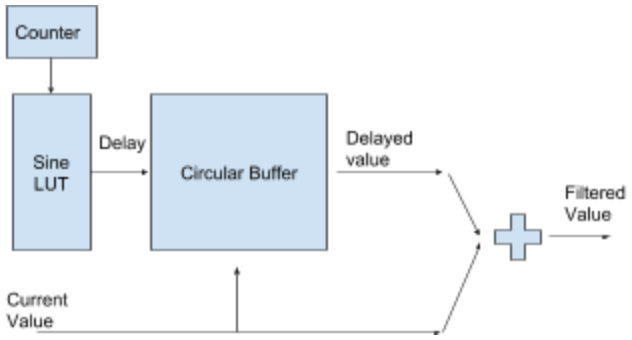

Figure 6 - *Diagram of Chorus Filter*

A Python script was written to automatically generate sample values for any sinusoidal signal (for the LUT). For the sake of the demo, a 1 Hz sinusoidal signal with an amplitude of 441 and offset of 1764 was used (while this isn't the exact sine parameters required for a "Chorus" effect, we slightly modified the parameters to make the effect more easily heard in the demo). Additionally, other effects, like the Flange effect, can be implemented using the same module by simply changing the parameters of the sine signal.

As with the echo filter, the filter was implemented in MATLAB and verified by comparing both outputs.

```
for sample = 1:length(music)
    circularBuffer(writeAddress) = music(sample);

    delay = LUT(mod(sample, length(LUT)) + 1);

    filteredSample = int16(int16(bitshift(3*music(sample), -2)) + int16(bitshift(circularBuffer(readAddress), -2)));
    fprintf(filteredMusicFile, '%x\n', filteredSample);

    if writeAddress + 1 > length(circularBuffer)
        readAddress = 1 - delay;
    else
        readAddress = (writeAddress + 1 - delay);
    end

    if readAddress <= 0
        readAddress = readAddress + length(circularBuffer);
    end

    writeAddress = writeAddress + 1;

    if (writeAddress > length(circularBuffer))
        writeAddress = 1;
    end
end
```

Figure 7 - *Excerpt from MATLAB Implementation of Chorus Filter*

4.3 Circular Buffer

The circular buffer is a buffer where the address being written to increases by one every clock cycle; when the address is incremented to the maximum writable address, it “circles” back around to 0. Additionally, the buffer takes a “delay” value. The output of the circular buffer is the the value of the input signal at the corresponding delay (in other words, *read address = write address - delay*). There are several corner cases to be considered. These cases were verified using a testbench; specifically, the following test cases had to be considered:

a. **Arbitrary write pointer value and arbitrary delay value:** The most basic test case to ensure the circular buffer writes correctly and outputs the correct delayed value.

b. **Arbitrary write pointer and delay value of 0:** In this case, since the delay is zero, the read pointer is equal to the write pointer. However, since the value being pointed to by the read/write pointer is *about* to be overwritten by the value at the write data port, you must output the value at the *write data port,* not the value in the buffer.

c. **Write pointer equal to buffer size and arbitrary delay value:** This case simply tests if the write pointer “circles” around back to 0 correctly.

d. **Arbitrary write pointer and a delay value larger than the write pointer value:** In this case, the read pointer is negative (since *read pointer = write pointer - delay*), which needs to handled by using *read pointer = write pointer - delay + buffer size* instead.

e. **Write pointer equal to buffer size and a non-zero delay value:** In this case, the write pointer needs to wrap around to 0, so *any* non-zero delay would result in a negative read pointer value. Note that this requires additional logic to handle and is *not* identical to case (d).

4.4 PCM To PWM Converter

Custom IP: PWMGenerator

An IP block that converts a PCM signal to a PWM signal. The block design was based of [1].

4.5 Bluetooth UART and Interrupt

Xilinx IP: AXI Interrupt Controller, AXI UartLite
Digilent: PmodBT2

Bluetooth was used to send commands (such as pause and play) between the PC and the FPGA. In order to use Bluetooth with the board, we used a PmodBT2. This Pmod was able to be connected and used as a UART interface with 8 bits of data, no parity bit and a single stop bit and a baud rate of 115.2 kbps.

While this UART interface could have been used as is, in order for this module to be integrated into the existing system containing the network connection setup, interrupts were also needed to determine when a Bluetooth command from the PC was issued. The follow code snippet shows the initialization, setup and usage of the interrupt for the UART interface.

```
// -- setup the Interrupt for UART interface
int SetupInterruptSystem(XUartLite *UartLitePtr){
    int Status;
    Status = XIntc_Initialize(&InterruptController, INTC_DEVICE_ID);
    Status = XIntc_Connect(&InterruptController, UARTLITE_INT_IRQ,
                           (XInterruptHandler)XUartLite_InterruptHandler,
                           (void *)UartLitePtr);
    Status = XIntc_Start(&InterruptController, XIN_REAL_MODE);
    XIntc_Enable(&InterruptController, UARTLITE_INT_IRQ);
    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                (Xil_ExceptionHandler)XIntc_InterruptHandler,
                &InterruptController);
    Xil_ExceptionEnable();
    return Status;

}
// -- the receive handler for when a UART interrupt is triggered
void RecvHandler(void *CallBackRef, unsigned int EventData){
    TotalReceivedCount = EventData;
    int gotint = *((int *)(0x40600000));
    /* Received Byte*/
     char got = gotint;
     return;
}
// -- initialize and watch for the UART interrupt
int UartLiteIntrExample(u16 DeviceId_1){
    int Status;
    Status = XUartLite_Initialize(&UartLite_1, DeviceId_1);
    Status = SetupInterruptSystem(&UartLite_1);
    XUartLite_SetRecvHandler(&UartLite_1, RecvHandler, &UartLite_1);
    XUartLite_EnableInterrupt(&UartLite_1);

    for(int Index = 0; Index < TEST_BUFFER_SIZE; Index++) {
            ReceiveBuffer[Index] = 0;
        }

    XUartLite_Recv(&UartLite_1, ReceiveBuffer, TEST_BUFFER_SIZE);
    while(1)
    {
        while(XUartLite_IsReceiveEmpty((&UartLite_1)->RegBaseAddress));

        XUartLite_Recv(&UartLite_1, ReceiveBuffer, TEST_BUFFER_SIZE);
    }
    return XST_SUCCESS;
}



int main()
{
    init_platform();
    int Status;
        Status = UartLiteIntrExample(BT_UARTLITE);
    cleanup_platform();
    return 0;
```

Figure 8 - *Initialization, setup and usage of the interrupt of the UART interface for Bluetooth*

4.6 Android App

Google Android Source Code: Bluetoothchat

Although the App was unable to be integrated into the overall system, the module itself was completed. The app consisted of buttons that, when pressed, sends special characters to the board over Bluetooth. Each special character corresponds to a specific command (i.e. filter type, play, pause etc.).

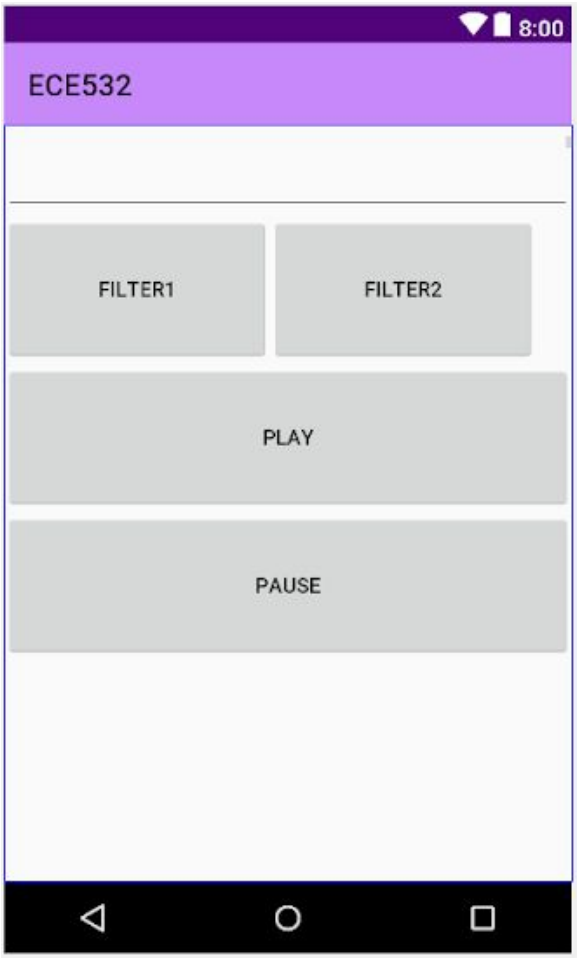

Figure 9 - *Screenshot of the Android App*

The app was based on the following Bluetooth tutorial provided by Google Android:
https://developer.Android.com/samples/BluetoothChat/index.html

4.7 Ethernet and LwIP

Xilinx IP: MicroBlaze, Ethernet MII to Reduced MII, AXI EthernetLite, AXI Timer, AXI Interrupt Controller, MIG 7
Software: MicroBlaze C code

To establish and maintain TCP connections, MicroBlaze running the LwIP Raw API was used. The API basics are outlined here: http://lwip.wikia.com/wiki/Raw/native_API

The design makes use of two separate TCP connections:

1. TCP connection that requests music from the server. This data is stored in DDR memory.
2. TCP connection that reads from DDR memory and sends data to a networked speaker.

These connections run simultaneously, independent of each other and could be connected to the same machine as the team demonstrated or two unique machines.

4.8 DDR Memory

Xilinx IP: MIG 7

DDR memory is used by MicroBlaze (1) as local memory since LwIP is too large to fit in block RAM.

Additionally, as described in section 4.7, DDR memory is used to store the received payload from the server and read back to send to a networked speaker.

Store payload data from a TCP packet in memory using the following line of code:

```
//          dst,                src,                len
memcpy (    (void *)_memptr_,   (void *)p->payload, p->len);
```

4.9 USB Serial Port

Xilinx IP: AXI UartLite

UART communication is used to connect board to SDK through a serial COM port on the host machine.

4.10 Python Music Server/Playback

Using one Python module, depending on arguments given, the module will act as either a music server or a networked speaker/playback device.

Run simple_server.py using the command:

```
python simple_server.py [PORT] [PATH TO WAVE]
```

Key libraries used:

- Socket: establishes and maintains TCP connections
- Wave: handles .wav audio files
- Pyaudio: handles the machine's audio devices to play audio

4.11 Python Bluetooth Controller

This Python module sends signals over Bluetooth to the board which will interrupt the processor to play or pause the sending of music.

Run play_pause.py using the command:

```
python play_pause.py
```

Key libraries used:

- Serial: handles machines serial COM ports that communicate to remote devices using Bluetooth

5.0 Design Tree Description

The project can be found on GitHub: https://github.com/rtjhie/nexys4-ddr-music-streaming

The key directories are as follows:

- docs
    - Contains documentation associated with the project
- src
    - android
        - Bluetoothchat : directory that contains the code for the functionality of the app. These are the actual application files.
        - res : directory that contains the graphical layouts and individual panels of the app
    - python
        - play_pause.py : Bluetooth connection to board to play/pause music
        - simple_server.py : receives connection requests and services accordingly - either sends music or receives and outputs music through speakers
    - sdk
        - Bluetooth.c : xuartlite functions related to Bluetooth functions
        - callback.c : lwip callback functions when connecting to servers
        - echo.c : start_application() setups up TCP connections and transfer_data() is where data is sent
        - main.c : main function of the program
        - platform.c : functions to setup and initialize board components
    - vivado
        - Contains the main Vivado projects
    - ip_repo
        - ChorusFilter_1.0: Directory that contains IP package for Chorus Filter
            - src:
                - ChorusFilter.v: Top level module of chorus filter
                - CircularBuffer.v: Top level module of circular buffer
                - SineLUT.v: Top level module of sinusoidal LUT
                - LUT.txt: Text file containing sinusoidal values that are loaded into the LUT
                - filteredMusic.txt: Filtered output values of module in simulation
                - Music.txt: Input values to module in simulation
        - EchoFilter_1.0: Directory that contains IP package for Echo Filter
            - src:
                - EchoFilter.v: Top level module for of echo filter
                - CircularBuffer.v: Top level module for circular buffer

        - PWMGenerator
    - scripts
        - Chorus Filter MATLAB: Directory containing MATLAB implementation of Chorus Filter
            - ChorusFilter.m: MATLAB implementation of Chorus Filter
            - LUT.txt: LUT sinusoidal entries
            - Music.txt: Audio file samples
        - Echo Filter MATLAB: Directory containing MATLAB implementation of Echo Filter
            - EchoFilter.m: MATLAB implementation of Echo Filter
            - LUT.txt: LUT sinusoidal entries
            - Music.txt: Audio file samples
        - Misc Scripts: Directory containing other scripts used in the project
            - LUTGenerator.py: Python script that generates sinusoidal sample values for LUT
            - TextToWAV.py: Python script that generates a text file out of a WAV file
            - WAVToText.py: Python script that generates a WAV file out a text file
    - VIP
        - ChorusFilterTester: Vivado project to test VIP interface of filters

6.0 References

[1] “One-Bit DAC.” *FPGA 4 Fun*, www.fpga4fun.com/PWM_DAC_3.html.