ECE532: Digital Systems Design
Final Project Group Report
April 12, 2024

Audio Vocoder

Group 11:
Khalil Damouni
Ililli Terefe
Hamas Waqar

1. Overview

This project creates an audio manipulation tool, called a vocoder, using a Nexys Video Board and a number of peripherals. The user uses a wired microphone to speak into the system, and wired speakers output the modified sound. The graphical user interface (GUI) displays the audio waveforms in the time domain and the frequency domain and offers a list of settings that the user can change to modify their voice. This GUI is powered using VGA Pmod and display. These settings can be controlled using a Bluetooth keyboard that is connected using two BT2 Pmods.

1.1 Background

Audio vocoders are a popular audio manipulation tool that have been used in pop culture and songs to create artificial sounds. The user speaks into an input, and the tool outputs their modified voice, making it sound, for example, robotic. We aim to create this tool on an FPGA, and interface with a number of peripherals.

1.2 Motivation

Our group decided to produce an audio-based project because we were all interested in learning more about how audio is represented digitally. Therefore, we found that an audio vocoder would be a great way to familiarize ourselves with digital audio, while creating an interactive project. We also decided to add a GUI to visualize the ways in which the audio has been manipulated, and allow the user to compare the soundwaves before and after audio manipulation.

1.3 Goals

The main goal of this project is to create a real-time audio application that modifies the user's voice. This is accomplished through utilizing phase vocoder techniques that are discussed in the later sections. Furthermore, we aimed to provide the user with a visualization of the audio manipulation and control over the system. This would be done using the GUI shown on a VGA display. The audio data would be used to create waveforms, showing the inputted audio before and after manipulation, as well as several settings the user can change to control the sound of the output. These settings would be controlled using a keyboard input.

1.4 Block Diagram

Figure 1 illustrates the block diagram for the audio vocoder project. The project consists of two main components: the audio data path and the soft processor. The data path components handle the transmission of audio signals from the microphone to the speaker, and perform the audio manipulation on the audio data signal to generate a vocoder effect. The window length for a frame in the audio data path was set to 1024 samples. This window length provides a detailed frequency resolution of the signal while not significantly consuming the on-chip FPGA board memory. Additionally, the soft processor component was responsible for interacting with all

project peripherals, including the Bluetooth keyboard and the audio codec of the microphone/speaker ports. Moreover, it generated a GUI to visually represent the current control settings for the project. A detailed explanation of the functionality of each IP can be found in the later sections.

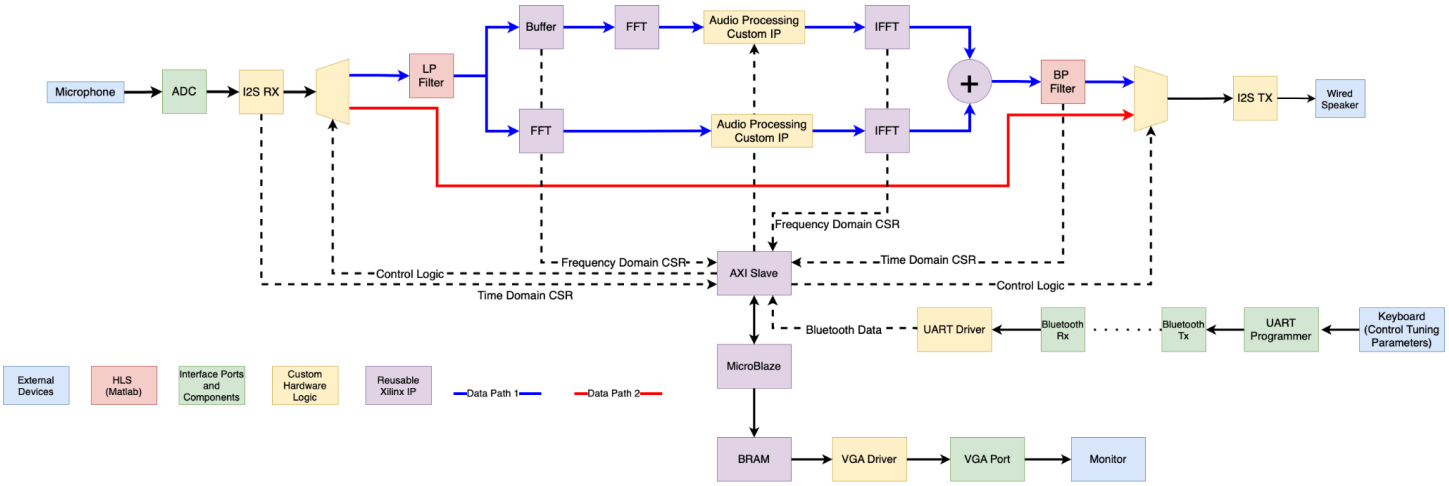

Figure 1. Block diagram of the audio vocoder project

1.5 System Changes

Figure 2 illustrates the initial block diagram outlined in our project proposal.

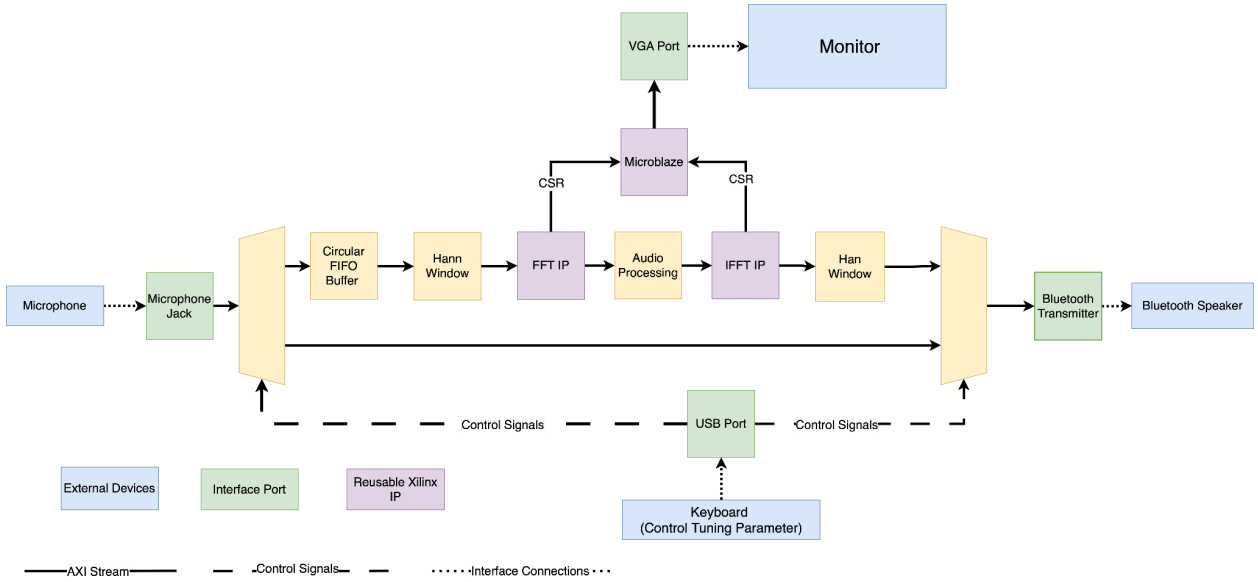

Figure 2. Block diagram of the audio vocoder project proposed in the project proposal

As we gained further insights into the project and its constraints, we made necessary adjustments to the block diagram as described below:

1. Initially, we planned to route the audio output through a Bluetooth speaker. However, upon discovering that the Bluetooth Pmod lacked support for audio transmission, we pivoted to using a wired speaker instead. We opted to utilize the Bluetooth Pmod for wireless communication with the keyboard. This allows us to maintain the complexity of our project.

2. During testing, we discovered that our audio data path introduced noise, which posed a significant challenge for the project. Further experimentation revealed that the FFT IP inherently introduced noise, especially in the low-frequency bins where the audio frequency resides. Consequently, we implemented a bandpass filter in post-audio processing to eliminate unwanted high and low audio frequencies. Additionally, the Hann window function was integrated into the filter design. For precaution, we also implemented a lowpass filter into the pre-audio processing to attenuate any high-frequency noise generated from the microphone. Additionally, we introduced a secondary audio processing path in the design. This design implementation allowed us to overlap the processed samples between the two paths, as seen in Figure 3, which amplifies the desired processed audio to make it audible.

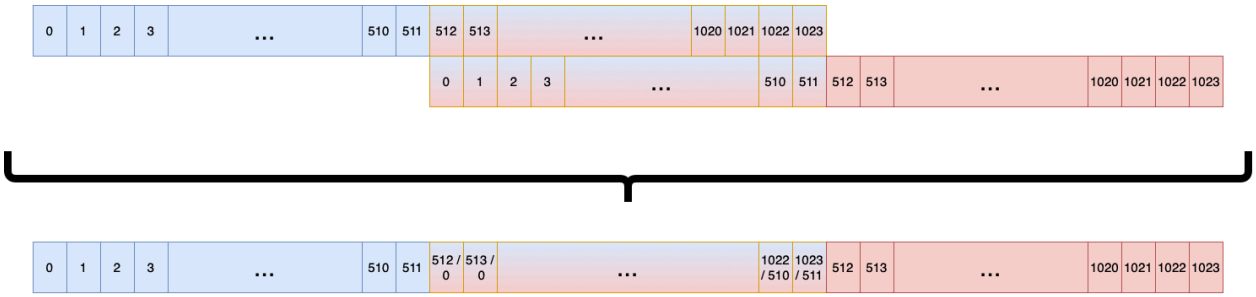

Figure 3. Depiction of overlapping two window frames to improve SNR

3. Initially, we planned to implement a circular FIFO to control the pitch of the audio data. However, we chose to remove this feature to enable team members to focus their efforts on reducing the noise present in the audio data.

1.6 Brief Descriptions of the IP

1.6.1 Audio Data Path

The audio data path consists of ten key blocks responsible for transmitting and modulating the audio data throughout the FPGA. The I2S Rx and I2S Tx IPs are custom blocks facilitating the conversion of audio data to/from the respective audio jacks using the audio codec. Once the audio data is converted by the I2S Rx IP, it is then routed to a lowpass FIR filter, which attenuates any high-frequency noise potentially accumulated from the microphone or the audio jacks. The data is subsequently transferred to two FFT IPs, which convert the audio data from the time domain to the frequency domain. One of the FFTs will receive the data after a delay of

some time samples as it passes through the Utility Buffer IP. Once the audio samples are in the frequency domain, they undergo modification in the Audio Processor IP to introduce the vocoder sound effects. Subsequently, the processed samples are transferred to the IFFT IP, converting the data from the frequency domain back to the time domain. Since we had two processing paths, an Adder IP was used to add the two samples. The FFT, in this process, introduces noise into the system, which is filtered using a bandpass FIR filter before being passed to the I2S Tx IP.
The I2S Tx, I2S Rx, and the Audio Processor IPs were custom-designed. The FFT, xFFT, Adder and Utility Buffer were all IPs that we incorporated from the Vivado IP library. The bandpass and lowpass filters were designed using Matlab HLS.

1.6.2 GUI Interface and Control Signals

The GUI is shown on a VGA display and driven using a custom VGA driver and a MicroBlaze. The MicroBlaze was instantiated along with the following related blocks: a MicroBlaze Debug Module (MDM), a MicroBlaze Local Memory, an AXI Interconnect, an AXI Uartlite, and an AXI Interrupt Controller. All the MicroBlaze related blocks are available in the Vivado library. The MicroBlaze provided the pixel values of the display by writing to a true dual-port BRAM. The custom VGA driver would read from this BRAM, and produce the correct signals to drive the VGA display, using a VGA Pmod.
The control signals were updated by the MicroBlaze using a modified AXI Slave block. This AXI Slave also held system data, such as the microphones values, and was also read by the MicroBlaze to create the soundwaves on the screen. Furthermore, an AXI IIC block, available in the Vivado library, was used to control the audio codec onboard the FPGA.
Lastly, control input to the FPGA was provided through a Bluetooth keyboard. A custom UART driver interfaces with the Bluetooth BT2 Pmod, which is paired to another BT2 Pmod. The second Pmod is the master, and is wired to a keyboard. This allows us to send keyboard values to the FPGA wirelessly, using Bluetooth.

2. Outcome

Overall, the project successfully routes input audio data from a microphone, to a speaker, performing manipulations on the audio quality. A GUI is used to provide the user with information about the system, and allows them to control audio settings.

2.1 Results

The BT2 Pmods did not support audio, and therefore could not be used to drive a Bluetooth speaker, which was the original idea for the BT2 Pmod. Therefore, we pivoted to use a Bluetooth keyboard instead, and wired speakers. This turned out to be a positive change, as the audio system was very sensitive to feedback, and holding the microphone close to the speaker created a lot of noise. By providing support for a Bluetooth keyboard instead of a USB keyboard, the user is able to change system settings at a distance from the FPGA board. This allows the user to

avoid creating noise through feedback, since they can now operate the microphone while at a distance from the speakers, and not be limited by the length of the keyboard's wire.
With regards to the GUI, the initial plan of having sound visualization and settings was successful. The left half of the screen is split into two, with the top half showing the waveform of the sound before the vocoder, and the bottom half showing the waveform after the vocoder and audio manipulation. These windows can visualize the information in both the time domain and frequency domain, which was the initial goal. A recreation of the final GUI interface can be seen in Figure 4.

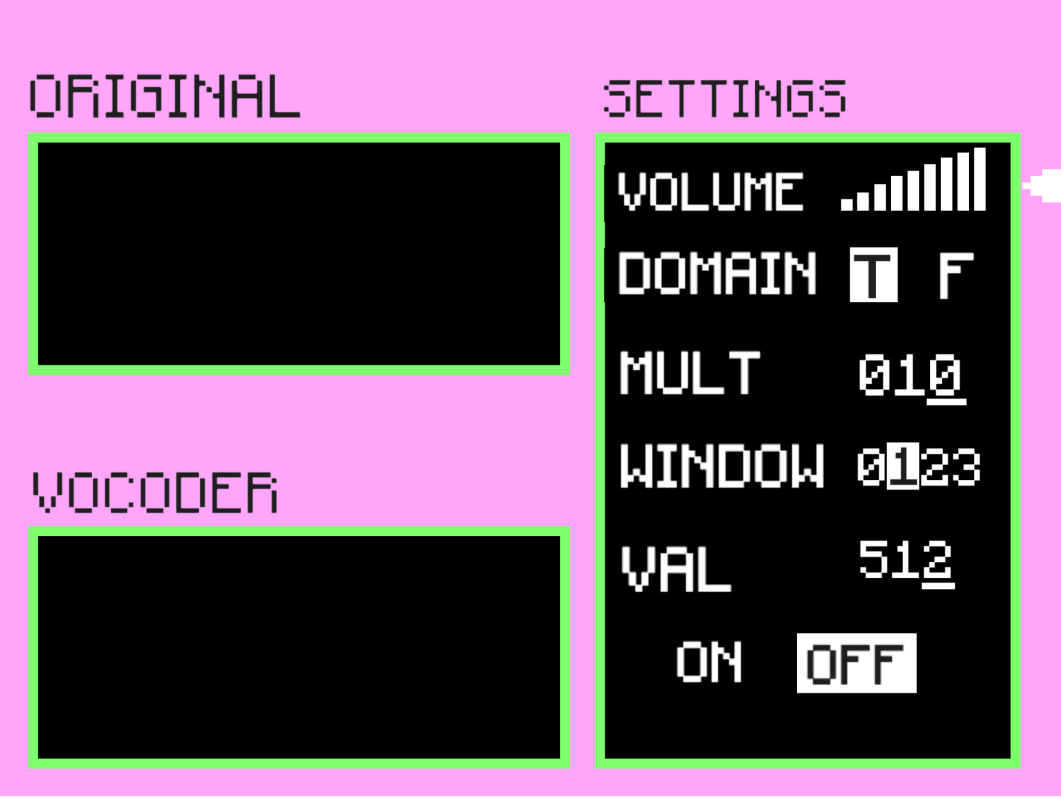

Figure 4. GUI Interface for the Vocoder Project

The right side of the display is a list of settings that the user can control, such as the volume, the domain of the visualization, various settings specific to the data path, and an on/off switch. The initial plan was to provide the user with audio manipulation settings, but the specific were not decided yet as we were not familiar with the IPs we would use. Overall, the final list of settings provided the user with enough control over the sound of the output:

- **Volume**: Allows the user to control the volume of the output.
- **Domain**: Allows the user to choose which domain, time or frequency, to display in the audio visualization windows.
- **Mult**: Allows the user to choose the constant that multiplies the imaginary part of the audio data in the frequency domain. This supports values from 0 to 999.

- **(Configuration) Window**: Allows the user to choose between four different configuration settings for the audio processor.
  0: The first configuration is an audio bypass that creates a direct connection between the microphone to the speaker. In this configuration window, the audio bypass does not go through the FFT and IFFT IPs.
  1: The second configuration applies the vocoder modulation effect without zeroing any samples in the window frame.
  2: The third configuration applies the deep vocal effect to the audio by zeroing the first *'Val'* samples of the window frame in the frequency domain.
  3: The fourth configuration applies the high pitch vocal effect to the audio data by zeroing the last *'Val'* samples of the window frame in the frequency domain.
  A more comprehensive explanation of these effects is provided in Section 4.1.5.
- **Val**: Allows the user to decide the number of samples of the window to zero in the frequency domain. This supports values from 0 to 999.
- **ON / OFF**: Allows the user to turn off the vocoder, letting the unedited audio input come out of the speakers.

The overall result was a VGA-driven GUI that provided the user control over the audio manipulation and visualization, using a keyboard connected through Bluetooth.

2.2 Future Works and Improvements

With regard to the GUI, the audio visualization, in both the time and frequency domain, was not as informative as we hoped. In the time domain, the audio input was in pulse-density modulation (PDM) form. Translating this format into an amplitude that could be visualized proved challenging, and in the future, more time should be spent on this to ensure the sound wave is as informative as possible. The final product successfully visualized the input data in the time domain, but it was slightly inaccurate at times and stuttery.

In the frequency domain, the input to the visualiser was taken by probing the output of the FFT and IFFT. This output was then translated into an amplitude and displayed on the screen. Unfortunately, the FFT and IFFT outputs were not the most informative given the amount of noise they introduced to the system. In future works, this visualizer should use a different source for its data, possibly a simpler implementation of the FFT and IFFT, that would calculate only the bins that were visualized.

We realized that most audio projects developed on Xilinx boards do not use the xFFT IP due to their noise; instead they use HLS to design the FFT and IFFT modules. For the future, we would like to replace the xFFT IPs with HLS implementations of the FFT algorithm. This will remove the noise that has been generated from the xFFT IP, resulting in a clearer audio output.

3. Project Schedule

Below is a table of our initial milestones at the time of writing the project proposal, the final milestones for each week.

Table 1. Description of the weekly milestones for this project

Original Milestone	**Final Milestone**
Milestone 1	
● Research Bluetooth protocols and Pmod integration. ● Research USB protocols for keyboard input. ● Research audio filtering (FFT, IFFT) implementation in software and hardware. ● Researching audio codecs protocols. ● Research GUI implementations for performance monitoring.	● Researched Bluetooth protocols and PMOD integration. ● Researched USB protocols for keyboard input. ● Researched the Xilinx xFFT IP and selected the best configuration for the block that meets our design. ● Researched the window implementation that best meets the needs of our project. ● Researched the audio codec on the Nexys Video.
Milestone 2	
● Get onboard microphone input waveform. ● Get USB keyboard input to register (either as a waveform or on the 7-segment display). ● Begin Bluetooth Pmod integration. ● Begin working on the GUI display using VGA. ● Decide on architecture for hardware implementation.	● Pivoted away from the onboard microphone. ● Got USB keyboard input to register as planned. ● Got VGA working without MicroBlaze involvement. ● Tried to implement and simulate the xFFT IP on a small project which was unsuccessful. ● Tried to use the Xilinx I2S receiver and transmitter IPs as well as DMA with MicroBlaze to store audio and play it back. ● Encountered issue with sending interrupt signals from DMA.
Milestone 3	
● Pair a device to FPGA using Bluetooth. ● Display hard-coded text on GUI using VGA. ● Get off-board microphone input waveform. ● Design queue custom IP and implement FFT and IFFT filters. ● Get baseline design for the hardware audio processing logic.	● Integrated BRAM with MicroBlaze, working on getting it working with VGA Driver. ● Buffer week for Khalil. ● Got the xFFT IP simulated on the TOFC server. ● Implemented a simple I2S Receiver and Transmitter connection to perform loopback. ● Outputted noise but could not pick up mic data. ● Investigated issue using ILA.
Milestone 4	
● Display text from keyboard on GUI using	● Finished integrating the BRAM with the VGA

VGA. ● Output sound from Bluetooth speaker. ● Get most of the design for the hardware audio processing logic design with some verification.	driver and MicroBlaze. ● Pivoted from using Bluetooth speaker to Bluetooth keyboard. ● Finished designing the Hann window module. We had to be careful in our implementation to handle overflow and rounding caused by the multiplication operation. Without it, the Hann window module may produce incorrect results. ● Finished designing the Audio Processing Custom IP. We verified this IP through simulations. ● Started working on implementing the circular FIFO IP for pitch manipulation but had to discard this block so we could focus on integrating our design for the demo. ● Pivoted to the on-board Nexys DDR mic. ● Implemented an audio loopback to support real time audio.
Milestone 5	
● Integrate Bluetooth speaker with off-board microphone. Output the sound registered by microphone. ● Integrate keyboard input with audio filters. ● Finish designing the hardware audio processing logic design with full verification. ● Finish designing the software microblaze implementation of the audio processing logic.	● Integrated GUI block diagram with audio loop-back block diagram. ● Made arrangements to get another Bluetooth Pmod, since two are needed. ● Started designing the FIR filter module by using the FIR Compiler IP to mitigate the noise. However, this IP, when implemented, added to the undesirable noise. We suspect that we did not correctly configure the FIR Compiler IP, although we did not pursue this further as we found a MATLAB HLS extension which was more promising. We chose the MATLAB HLS for its versatility and flexibility which made implementation simpler with the lack of time. ● Discarded the Hann Window IP as we incorporated the windowing function into the FIR Filter coefficients. ● Pivoted back to Nexys Video board due to memory resource concerns for final integration. ● Implemented and tested custom I2S IP to receive and transmit audio.
Milestone 6	
● Integrate all components. ● Use keyboard input to change audio	● Got a Bluetooth keyboard to work using 2 BT2 Pmods and a custom UART driver.

processing. ● Route audio input from off-board microphone through audio processing. ● Output resulting sound through Bluetooth speaker.	● Started adding settings that are controlled using the GUI. ● Integrated audio processing IPs with the new I2S IP and implemented it on the Nexys Video Board. ● Designed the FIR filter using MATLAB HLS which was integrated into our project. ● Discovered that most of the noise was being generated from the xFFT IP, hence we experimented with different cutoff frequencies to determine which cutoff frequency would attenuate most of the low-frequency noise while still providing audible audio output.
Milestone 7	
● Buffer week.	● Finished integrating the GUI with the audio manipulation block diagram. ● Tried to implement different design techniques to remove the noise. One of which is the parallel audio path to overlap frames which did help improve the SNR. Also, tried to modify the scaling schedule to reduce the noise which was unsuccessful.

3.1 Discussion

Initially the milestones were reasonably similar to what was planned, especially with getting the VGA to work. However, as we learned more about our components and began to pivot, we diverged from the specifics of the plan, but continued to use it as a point of reference for how complete our project should be each week. For example, although we had pivoted from using a Bluetooth speaker by milestone 5, we were on track to have the Bluetooth Pmods working, accounting for the buffer week.

Most of the data path implementation also followed our predicted milestones, with some changes. Firstly, the FFT took longer than expected to implement due to lack of support available and a particular bug encountered in Hamas' Vivado installation. Similarly, the I2S IP used to interact with the audio codec was deprecated. Hence, we had to design our own I2S Rx/Tx IP which took longer than expected. In parallel with its design, we used the on-board mic so that we could start testing our datapath with real audio samples. From this work, we began discovering noise in our system. In our initial milestone, we did not consider noise playing a big role, this was a shock to us. Due to this hiccup, we had to pivot and modify our design to focus on mitigating the noise.

4. Detail Description of the IP Blocks

Our IP blocks are split into two distinct categories, the first dealing with the audio data path, and the second being focused on the soft processor and control signals.

4.1 Audio Data Path

The following list of IPs that were used for the audio datapath. As discussed earlier, the audio datapath handles the transmission of audio signals from the microphone to the speaker while applying audio effects to modulate the data.

4.1.1 I2S RX/TX IP

This IP implements the I2S protocol, facilitating the transfer of audio data to and from the audio codec. The programming of the audio codec, which includes tasks such as setting the ADC/DAC sample rate and audio jack selection, is managed within the MicroBlaze via the IIC controller (further details provided in section 4.2.1). A high level implementation of the IP is depicted in Figure 5. The top-level file of this IP is named **i2s_rx_tx**, which instantiates a custom module, i2s_ctrl, and a Xilinx ODDR IP.

i2s_ctrl contains the logic responsible for implementing the I2S protocol and performs three primary tasks: parallelizing incoming data from the audio codec, serializing the modified audio received from the data path, and generating the left/right clock (LRCLK) and bit clocks (BCLK). Please refer to Appendix A1 for more details.

The incoming serial data from the microphone (SDATA_I) is captured and stored in a shift register triggered by the rising edge of the Bit Clock (BCLK). Subsequently, this serialized data is converted into a 24-bit parallel format and forwarded to the filter for processing within the audio pipeline.

Upon completion of the processing stage, the processed sample is returned to the IP and stored in one of two registers, depending on whether it pertains to the left or right channel. This determination is made by monitoring the transitions of the LRCLK signal. Specifically, at the falling edge of LRCLK, the data is stored as left channel data, whereas at the rising edge, it is stored as right channel data.

Concurrently, the data stored in one of the registers is shifted out in correspondence with the LRCLK signal. Specifically, left channel data is shifted out at the falling edge of LRCLK, while right channel data is shifted out at the rising edge, ensuring synchronized data transmission.

The ODDR IP (Output Double Data Rate Output Register) serves the purpose of forwarding the 12.288MHz clock entering the **i2s_rx_tx** IP to the MCLK pin of the audio codec. This ensures that the forwarded clock is not routed through any interconnects.

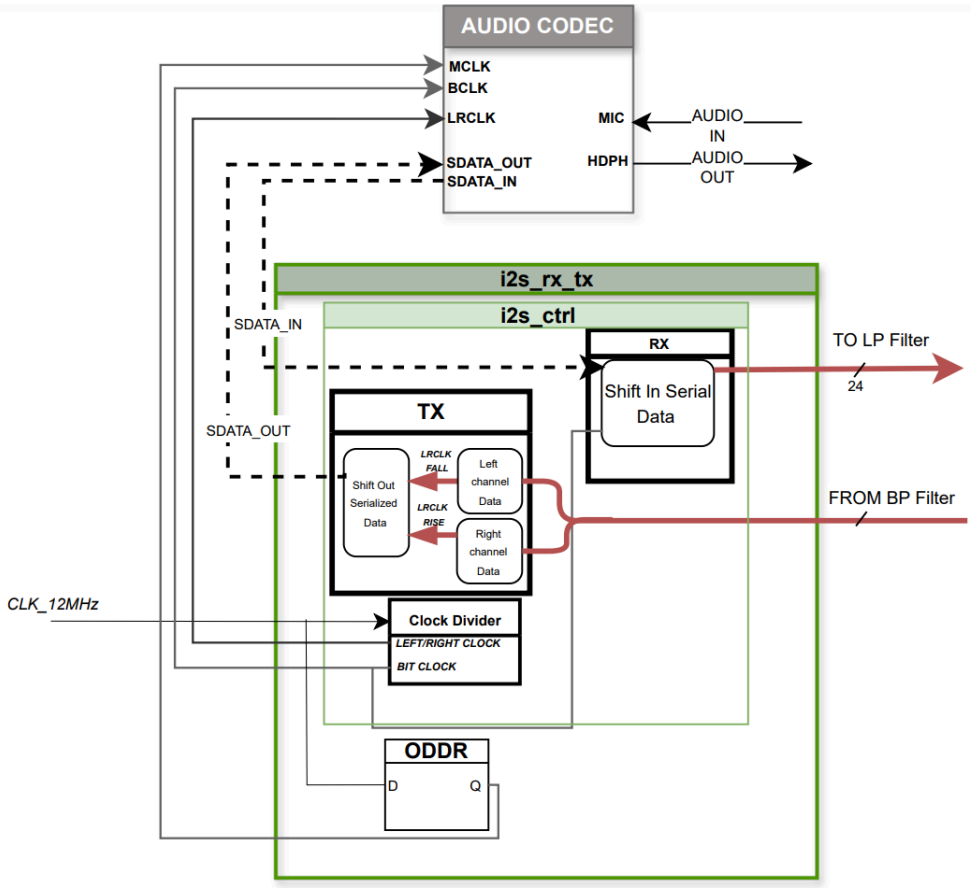

Figure 5. High level diagram of the i2s_rx_tx IP implementation

4.1.2 Lowpass (LP) and Bandpass (BP) FIR Filter IP

FIR filters were introduced towards the latter stages of the project to address the noise generated within the system. Two distinct custom FIR filters were integrated into the pre-audio processing and post-audio processing stages of the data path.

The initial addition to the design was a bandpass filter which aimed at mitigating noise generated by the xFFT IPs post-audio processing. As discussed in Section 4.1.4, xFFT IPs inherently introduce noise, especially in the low-frequency bins where audible bands are present. While the audible voice range for most humans spans from 20 Hz to 20 KHz, the more prominent frequencies of human voice typically range between 100 Hz and 8 KHz [1].

During our simulations, we observed that the majority of static noise generated by the xFFT IP predominantly occurred in frequencies below 400 Hz. However, noise still existed in frequencies above 400 Hz, although it did not significantly contribute to the noise. Human voice consists of multiple overlaid sine waves at different frequencies. Setting a cut-off frequency of 400 Hz would attenuate most of the noise but render the audio inaudible. Essentially, this cut-off frequency would remove low-frequency overlapping sine waves, making some syllables unintelligible and introducing gaps in the audio output. Therefore, after experimentation, a lower bound cut-off frequency of 150 Hz was chosen, effectively mitigating most of the noise while

maintaining audible audio output. As a precaution, the second cut-off frequency for the bandpass filter was set to 20 KHz to attenuate any audio output beyond the audible range. Figure 6 illustrates the complete transfer function for the bandpass filter.

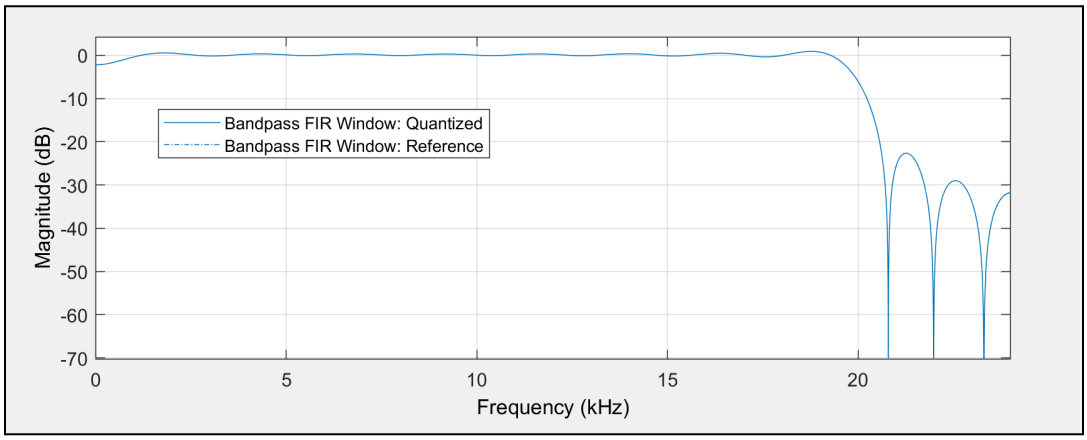

Figure 6. Transfer function for the bandpass filter

The bandpass filter was designed with an order of 40, as shown in Figure 7, as it rapidly attenuated the signal. We observed that a lower-order filter would attenuate the signal slowly, leading to audio smearing. Conversely, a higher-order filter could not be used as it would significantly decrease the magnitude of the audio filter. Additionally, the Hann window was incorporated into the filter design to ensure a smooth transition between sequential window frames.

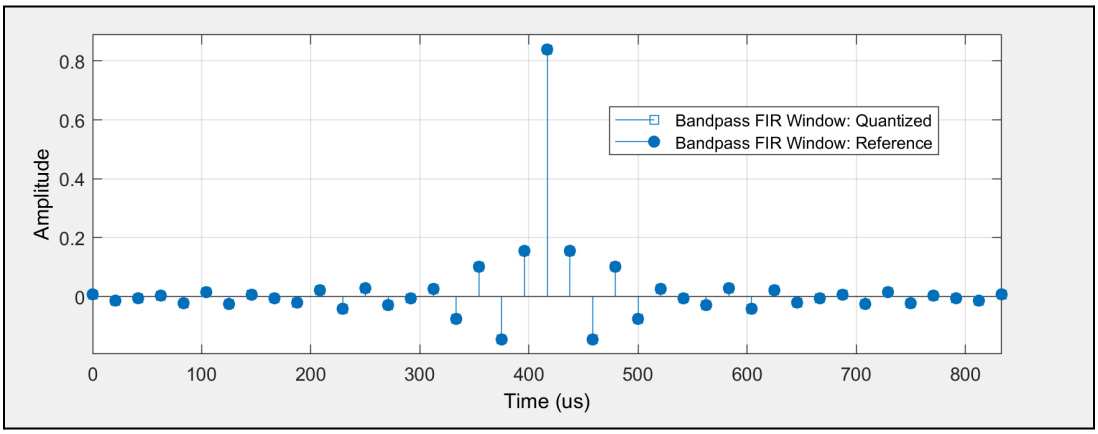

Figure 7. Impulse response for the bandpass filter

Furthermore, we introduced a second FIR filter into the system design at the pre-audio processing stage. During experimentation, we observed instances where the audio jacks and microphone generated white noise above 20 KHz. To prevent introducing bias into the data samples during audio processing, we implemented a low-pass filter with a cut-off frequency of

20 KHz. This effectively eliminated the high-frequency noise also audible through the speaker. The transfer function for the low-pass filter is depicted in Figure 8. The order of the low-pass filter was maintained at 40, primarily for the same reasons as the bandpass filter described earlier. Figure 9 illustrates the impulse response of the low-pass FIR Filter. Similar to the bandpass filter, the Hann window was incorporated into the filter design.

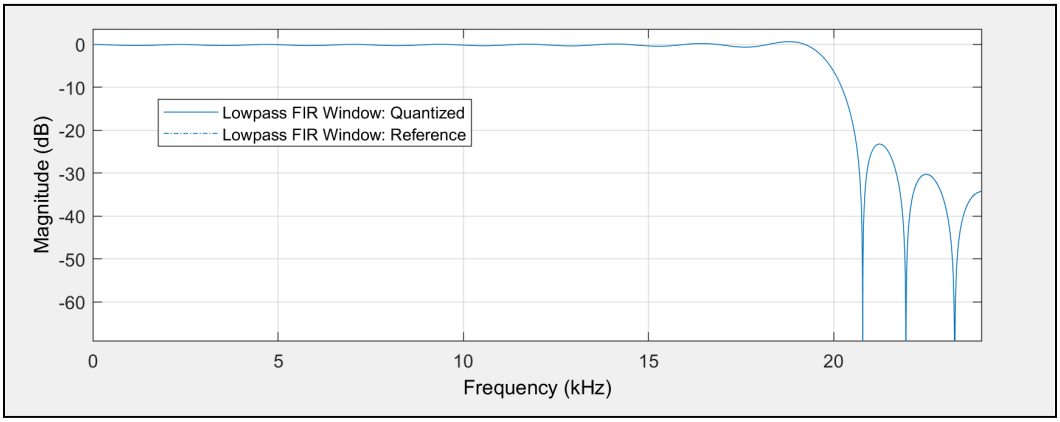

Figure 8. Transfer function for the low pass filter

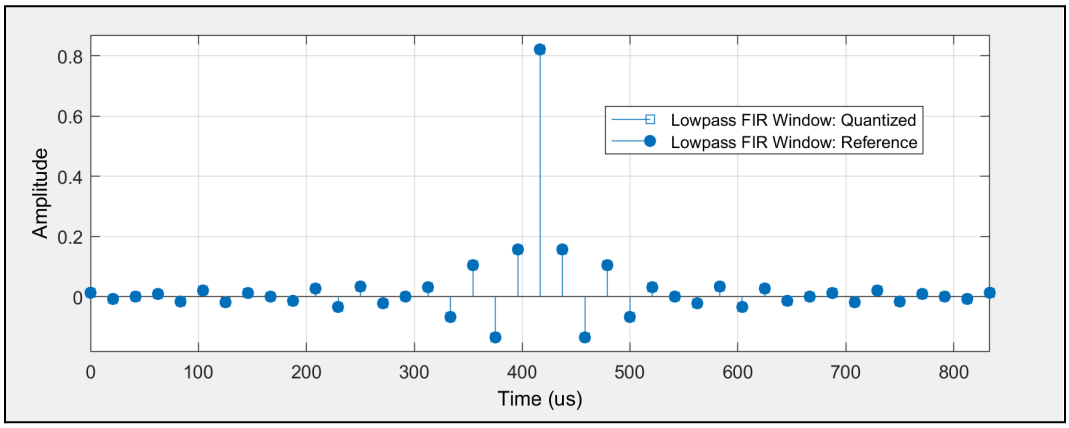

Figure 9. Impulse response for the low pass filter

Both the bandpass and low-pass filters were designed using MATLAB, a widely used mathematical software in academia and industry. One of its many uses is for effectively designing and simulating DSP applications. Moreover, MATLAB offers a powerful and free built-in High-Level Synthesis (HLS) generator [2] that is capable of converting MATLAB syntax into Verilog code. Leveraging this feature, we converted our MATLAB filter design code into synthesizable Verilog modules to create custom IPs. While the generated Verilog code may not be fully optimized, it still yields accurate results which is still beneficial for our vocoder design.

4.1.3 Buffer IP

Data Path 1, as labeled in the block diagram comprises two subpaths, each containing instantiations of the FFT, IFFT, and the Audio Processing IP. Filtered audio data is routed to a buffer on one path and directly to an FFT on the second path. The buffer in the former case was realized using Xilinx's FIFO Generator IP.

To achieve the desired 512-sample delay in the first subpath, a small FIFO control module was developed to manage the FIFO's write enable signal. This ensures that the write operation begins only after the FIFO has completed reading 512 samples. The controller module receives the FIFO Generator's *full* status flag as an input and sets its write enable signal to high when the flag is raised. Figure 10 illustrates the connection of this control module and the FIFO Generator.

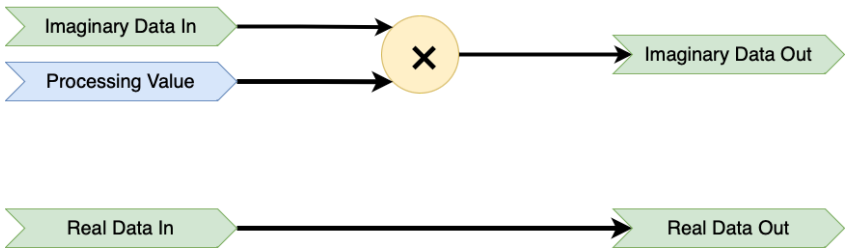

Figure 10. Audio processing design for the vocoder effect

4.1.4 xFFT IP

The FFT and IFFT blocks were implemented using the xFFT IP provided by Xilinx. This IP is a configurable module designed to efficiently execute FFT computations on the FPGA without imposing significant resource demands. However, it does have certain limitations, which will be discussed in further detail later in this subsection. Like all IPs, the xFFT IP was tailored to meet our design specifications, as outlined below:

- The first customization applied to this IP involved configuring the window length to 1024 samples. The window length of the FFT plays a critical role in determining the resolution of the frequency bins in the frequency domain. Ideally, we aim to select a larger window length to enable high resolution in the frequency domain; however, this choice means increased resource utilization, particularly for the DSPs. Additionally, it's important to note that the resolution of the frequency domain is also influenced by the sampling frequency of the audio data. In this project, the ADC in the audio codec was set to sample at a frequency of 48 kHz. With a window length of 1024 samples, the xFFT IP achieved a frequency resolution of 46 Hz. We opted not to select a window length of 2048 or higher due to the limited DSP available. Also, we refrained from choosing a smaller window length to achieve a lower resource utilization because this lowers the frequency resolution and amplifies the noise generated by the IP.
- The second customization implemented for the IP involved configuring the input and phase bits to be 24 bits wide. The xFFT IP, due to its generality, represents data as complex

numbers for both inputs and outputs. In this context, the input parameter of the IP denotes the real component of the data, while the phase parameter signifies the imaginary component. As the audio codec supplies fixed-point numbers that are 24 bits wide, the sizes of both the input and phase parameters were adjusted accordingly to match this width.

- The third customization applied to the FFT IP involved configuring the scaling option to block floating point. The IP design inherently introduces bit growth which degrades the accuracy of the output samples. Xilinx, in its datasheet, recommends using a static scaling factor to mitigate this bit growth. However, their recommended settings for the static scaling factor tend to be conservative, resulting in a significant reduction in the amplitude of the output data. Discussions on Xilinx forums have mentioned that the magnitude of the output signal can be less than 100th of the input signal with their recommended settings. The block floating point scaling factor automatically adjusts the scaling factor based on the degree of bit growth. Although this approach utilizes more logic, it prevents unnecessary reduction in the magnitude of the signal caused by excessive scaling.
- The final customization applied to the FFT IP involved configuring the output ordering to natural order. The FFT IP utilizes a butterfly algorithm in its design, which inherently scrambles the bit orders. This can complicate the analysis of the output data. Selecting natural order ensures that the output data is presented in the sequential order that most designs are accustomed to.

The remaining settings in the IP customization window have been left at their default values. Due to the IP's generality, the xFFT IP can be initialized for use in both FFT and IFFT applications by adjusting the configuration bit. Specifically, setting the configuration bit to 1 forces the IP to operate as an FFT, while setting it to 0 forces the IP to operate as an IFFT. During experimentation, we discovered a limitation with the xFFT IP: it exhibits significant noise in the low-frequency bins. This issue likely arises from the fact that the FFT IP is optimized for signal processing applications that primarily utilize high-frequency bands for data transmission. However, audio processing predominantly involves low-frequency bands. As a result, the IP inherently introduces noise into the audio data, which became noticeable during testing and demo.

4.1.5 Audio Processing IP

The Audio Processing IP stands out as one of the few custom IPs developed for this project. This IP seamlessly integrates sound effects in real-time into the audio data. Initially, our primary objective was to incorporate a vocoder effect to impart a robotic sound to the audio voice. However, we expanded the scope of the project by integrating additional effects such as a deep vocal effect and a high-pitch effect.

The vocoder is an audio effect that modifies the frequency bands of an audio signal within the frequency domain, resulting in various effects such as robotic or whispering sounds depending on the alteration applied. In our project, the frequency bands were adjusted by multiplying the imaginary component of the data output from the FFT, as depicted in Figure 10 [3]. For example,

setting all the imaginary components of the audio data to zero would produce a robotic effect, while multiplying the imaginary component by a factor of 20 would create an alien effect as seen in our experimentation.

The Audio Processing IP also includes two additional real-time effects: a deep vocal effect and a high-pitched vocal effect. These effects are achieved by zeroing certain samples, in both the real and imaginary domains, within each frequency domain window. This process effectively removes specific frequency bands from the voice, altering the tone of the audio. For example, zeroing the first *n* samples, as illustrated in Figure 11, eliminates the high-frequency bands from the voice, resulting in a deepening of the vocal tone. Conversely, zeroing the last *n* samples of the window in the frequency domain creates a high-pitched vocal effect that is similar to a cartoon chipmunk. It's important to note that the user has control over the number of samples to be zeroed, allowing for adjustments to the tone of the sound. During the demo, we found that this effect worked well when the number of samples within a window frame to be zeroed was set to 512. The vocoder effect can still be applied on top of these effects.

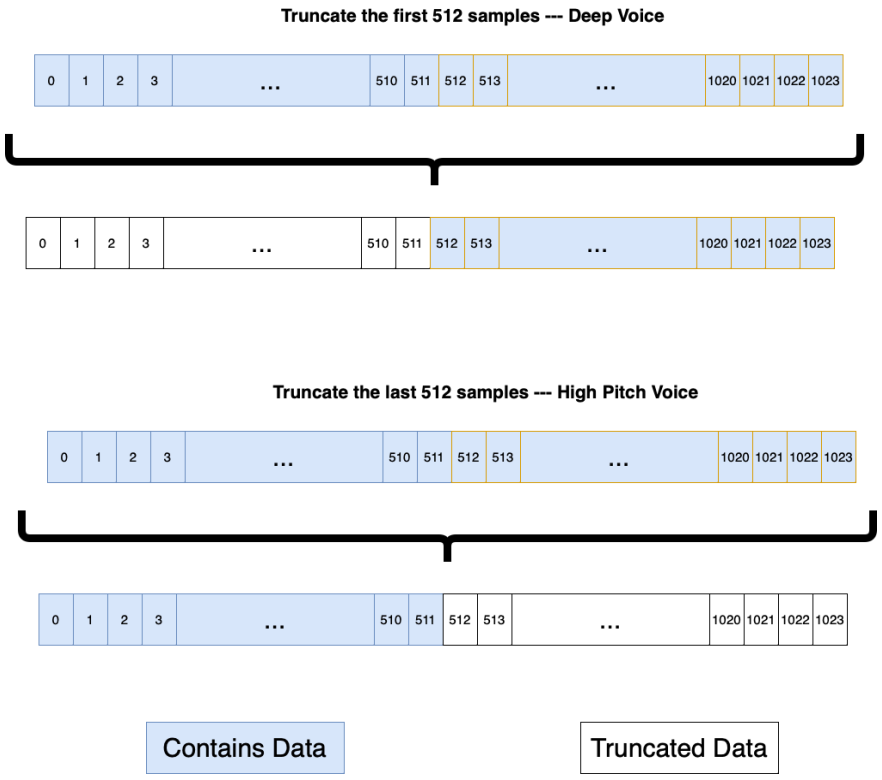

Figure 11. Design concept to control the tone of the voice

4.1.6 Adder/Subtracter IP

This IP was used to overlap the outputs of the subpaths in Data Path 1 in Figure 1. The IP was customized to perform the addition of 24-bit signed numbers, and the resulting output was routed to the BP filter, from which it enters the I2S IP to be sent to the audio codec.

4.1.7 AXI IIC IP

The IIC IP was used to interface with the Nexys Video Board's audio codec, and write to its registers to control settings such as sampling rate, audio jack selection and volume control.

4.2 Soft Processor Interface

The following list of blocks support the MicroBlaze, the GUI, the Bluetooth keyboard, and all the control signals needed to operate the program. It makes use of a VGA Pmod, to allow the Nexys Video Board to connect to a VGA display, and two Bluetooth Pmods, that support the use of a keyboard connected using Bluetooth.

4.2.1 Microblaze IP (and associated blocks)

The use of the MicroBlaze IP includes the MicroBlaze Debug Module (MDM), the MicroBlaze Local Memory, an AXI Interconnect, an AXI Uartlite, and an AXI Interrupt Controller. All these IPs are available in the Vivado library.
The MicroBlaze IP was set to 32 bit, with its implementation optimization set to 'Performance'. The AXI Interconnect was set to have 1 slave port, and 5 master ports. This allowed the MicroBlaze to connect with the AXI Interrupt Controller, the AXI Uartlite, the AXI BRAM Controller, the 512-Register AXI Slave, and the AXI IIC IP. The MicroBlaze, and all connected blocks, are clocked by a 100MHz clock source. The MicroBlaze IP was programmed using the provided SDK software.

4.2.1.1 MicroBlaze Software Files

The MicroBlaze's software consists of 4 pairs of files:

1. helloworld.c / helloworld.h
2. audio.c / audio.h
3. demo.c / demo.h
4. iic.c / iic.h

'helloworld' contains the main function, and the superloop. It first instantiates all the global variables required, including the pointers required for accessing the peripherals, such as the AXI Slave. It then creates the GUI by writing a pixel value to the associated BRAM location, and continues to update it in the superloop. At the beginning of the loop, it polls all the required registers from the AXI Slave: The keyboard value, the microphone value, the vocoder output, and other values required for creating the soundwaves. It then analyzes these values, and updates the GUI accordingly.

‘audio’ serves as the audio driver facilitating communication with the audio controller via I2C. It includes functions to write and read data from the audio codec’s registers, configure its clock settings, initialize audio settings such as the sample rate and volume, and select the audio input/output path.
‘iic’ contains the initialization functions for an I2C controller.
‘demo’ initializes the I2C controller, then calls the audio initialization functions defined in ‘audio’ to configure the audio codec.
These files were made by using [4] as a reference.

4.2.2 AXI Slave IP

This IP is defined in the ‘axi_child_512.v’ file. This modified 512 register AXI Slave IP was created to facilitate communication between the MicroBlaze and other IPs. This was created in Vivado using the ‘Create and Package New IP’ tool, and it contains 512 32-bit registers. The value of 512 was chosen to make sure there are enough registers to hold as many control, status, and data values as needed. The top 11 registers (0 to 10) are used as inputs to the MicroBlaze. 11 input wires are added to the module’s declaration, and the ‘read’ logic is edited to return to the reader the value of the associated wire, instead of the register. This allows the MicroBlaze to communicate with the AXI Slave as normal, but the AXI Slave now returns the value of the input wires instead of registers, bypassing the AXI protocol.
Similarly, the bottom 5 registers (511 to 508) are allocated as outputs. 5 output wires are added to the module’s declaration, and ‘assign’ statements are used to connect them to a register. Therefore, as those registers are written to by the MicroBlaze, the outgoing wires are instantaneously updated with their value, allowing the MicroBlaze to control wire values by simply writing to an address.
This method also allowed for easy debugging, as we can connect any wire we chose to the AXI Slave, and poll it using the MicroBlaze, similar to the ILA. Therefore, remnants of debugging code can be seen in the Verilog file.

4.2.3 BRAM IP

This IP was generated to support the GUI. It is a ‘True Dual Port RAM’, running in ‘Stand Alone’ mode, with ‘Generate address interface with 32 bits’ set to true, and ‘Common Clock’ set to false. It also supports ‘Byte Write’, where a byte is set to 8 bits. The write and read width are set to 32 bits, and the read depth is set to 80,000. This value was chosen because the VGA screen is 480x640 pixels, with each being 12 bits. By reducing the bit depth to 8, each 32 bit address could support 4 pixels, requiring in total 76,800 unique addresses to support the screen, with a slightly reduced pixel depth.
The read and write latency is provided by Vivado to be 2 cycles, which is fast enough to support a 60Hz refresh rate. Port A is connected to an AXI BRAM Controller, and is written to using the MicroBlaze. The MicroBlaze writes 8 bits values, each representing a single pixel on the screen.

Port B is connected to the VGA driver, and reads the associated 8 bit value to provide the correct VGA signals.

4.2.4 UART Driver

This custom IP was created to support the BT2 Pmod. It supports a Baud rate of 115200, and a configuration of 1 start bit, 8 data bits, 1 stop bit, and no parity bit, which is the default for the BT2 Pmod. It works by generating a clock signal at 115200 Hz, and polling the Rx pin of the Pmod header. This pin is set to high when no data is transmitted. Once the start bit is received (a low), the next 9 bits are captured and stored in a register (8 data bits, and a stop bit). Furthermore, a counter is incremented by 1 for each byte of data received, so the MicroBlaze knows whether the same letter was sent twice, or if it is polling an old value. Two output wires are assigned to the data register and the counter register, which are then connected to the AXI Slave to be read by the MicroBlaze. The pins for the BT2 Pmod are defined in the 'video.xdc' constraints file, for connection with Pmod header 'JA'. [5] was used as a reference for this design.

4.2.4.1 Programming the BT2 Pmods

The BT2 Pmods required programming before usage to configure their settings, such as their mode (master or slave), their paired devices, and more. This is done using a FT232RL USB to Serial UART programmer, shown in Figure 12, but it can be done using any USB to Serial UART programmer.

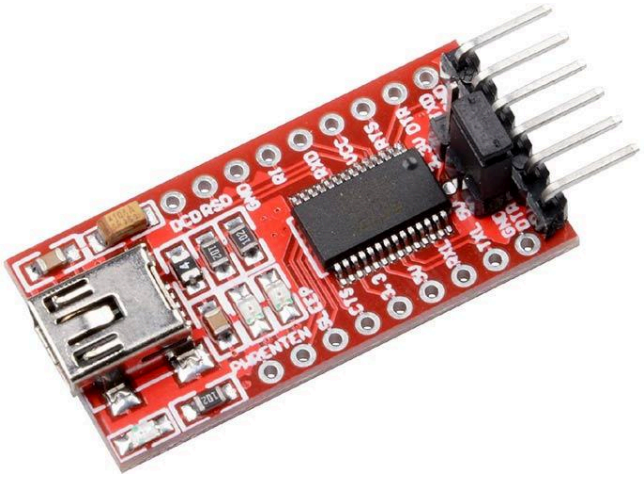

Figure 12. FT232RL USB to Serial UART Programmer [6]

The information in [7] was used as a reference for the pins of the BT2 Pmod, alongside the labels on the programmer, which were connected using female-to-female jumper wires. Firstly, connect

pins 11 and 12 of the BT2 Pmods to the ground pin and the 3.3V pin on the programmer, respectively. Then, connect pin 2 of the Pmod to the Tx pin of the programmer, and pin 3 to the Rx pin of the programmer.
To communicate with the Pmod, a serial port is needed. SerialTools for MacOS was used [8]. Commands for communication with the Pmod are defined in the reference manual here [9]. The instructions in [10] under the section 'Manual Connect' were used to connect the two Pmods, set the one connected to the FPGA as slave, and the one connected to the keyboard as master, with automatic pairing on powerup.

4.2.5 VGA Driver

This custom IP was created to drive the VGA display through the VGA Pmod, purchased here [11]. The required pins were defined in 'video.xdc', for connection with Pmod headers JB and JC. It uses one pin for the horizontal sync, one for the vertical sync, and 4 pins for each of the 3 colors, resulting in a total of 14 pins for the VGA. It also defined all the necessary wires for interfacing with the BRAM block, to retrieve the pixel color value. The information in [12] was used as a reference for this design.
Firstly, a 25 MHz clock signal was generated using a clock divider and the 100 MHz clock. This clock was then used to generate the vertical and horizontal counters. The horizontal counter would count from 0 to 799, then return to 0, and the vertical counter would count from 0 to 524, then return to 0. These two counters were then fed into the horizontal sync and vertical sync driver modules. These modules would set the horizontal sync to high when the horizontal counter was more than 0 and less than 96, and it would set the vertical sync to high when the vertical counter was more than 0 and less than 2.
These counters would also be used to index into the BRAM. If the horizontal and vertical counters were not in the addressable space (144 < horizontal counter < 784, and 35 < vertical counter < 515), the address was set to 0. Otherwise, the address was set to the following:

```
address = ((horizontal counter - 144) + 640*(vertical counter - 35))
```

This address value is used by the outgoing wire that interfaces with the BRAM block, and retrieves the associated pixel value on an input wire.
Lastly, 3 instances of a module are used to drive the RGB color wires. When the counters are in the addressable space, the wires are set to the value retrieved from the BRAM. Bits 0 to 3 are the blue wire value, bits 4 to 7 are the green wire value, and bits 8 to 11 are the red wire value. Otherwise, all three counters are set to 0. Due to space limitations, only 8 bits were used to define pixel colors, and the highest 4 bits of the color values were set to 0, causing the red wire to always be set to 0. This could be fixed by changing which bits each color wire was mapped to, but we found that this continued to give us enough color choice for the GUI.
To interface with the BRAM, this module has 6 output wires to drive the BRAM, and 1 input wire for the BRAM value. This connects to Port B on the BRAM, and reads the values set by the MicroBlaze.

5. Design Tree Description

The below tree in Figure 13 describes the location of the main files and folders of our project. Only the important and custom folders are described below. Our project has been uploaded to GitHub at the following address:

https://github.com/KhalilJDamouni/ECE532_Audio_Vocoder

Firstly, all MicroBlaze related files are in ‘audio_vocoder.sdk’. This includes the exported hardware files and C files required to support the GUI and control signals, including the audio codec. ‘audio_vocoder.srcs’ houses our custom Verilog code. This folder contains many redundant subfolders that were generated during integration. Lastly, the bottom 6 folders contain all the code for our custom IPs. ‘audio_i2s’ contains the I2S IP files. ‘datapath_ip’ includes a file for audio processing, manipulating frequency domain data to produce different audio effects. ‘fft_interface_ip’ and ‘ifft_interface_ip’ each contain a file that is responsible for managing handshaking and data transfer with the AXI stream interface of the FFT/IFFT blocks. The project can be opened by launching the ‘audio_vocoder.xpr’ file in Vivado.

```
ECE532_Audio_Vocoder
├── README.md
├── audio_vocoder.xpr //Project File
├── audio_vocoder.hw
├── audio_vocoder.sim
├── audio_vocoder.ip_user_files
├── audio_vocoder.sdk //MicroBlaze Files
│   ├── design_1_wrapper_hw_platform_0 //MicroBlaze Hardware Files
│   ├── microblaze_video_march_31_1
│   │   ├── Debug
│   │   └── src //MicroBlaze .c/.h Files
│   │       ├── audio.c
│   │       ├── audio.h
│   │       ├── demo.c
│   │       ├── demo.h
│   │       ├── helloworld.c
│   │       ├── helloworld.h
│   │       ├── iic.c
│   │       ├── iic.h
│   │       ├── lscript.ld
│   │       ├── platform.c
│   │       ├── platform.h
│   │       └── platform_config.h
│   └── microblaze_video_march_31_1_bsp //MicroBlaze Project File
├── audio_vocoder.srcs
│   ├── constrs_1/imports/new
│   │   └── video.xdc //Constraints File
│   └── sources_1
│       ├── ip
│       │   └── oddr_0 //Custom IP File
│       ├── bd
│       │   └── design_1 //Main Block Diagram
│       ├── imports //Custom Verilog Files
│       │   ├── sources_1
│       │   │   ├── imports/new
│       │   │   │   └── vga_driver.v
│       │   │   └── new
│       │   │       ├── axi_child_4.v
│       │   │       └── uart_reader.v
│       │   └── terefeil
│       │       ├── DataPath/src
│       │       │   └── processingPath.sv
│       │       └── audio_with_mb/audio_with_mb.srcs/sources_1/imports/new
│       │           └── i2s_ctrl.v
│       └── new //Custom Verilog Files
│           ├── audio_mux.v
│           ├── axi_child_512.v
│           ├── fft_output_format.v
│           └── fifo_ctrl.v
├── audio_i2s //Custom IP Files
├── datapath_ip //Custom IP Files
├── fft_interface_ip //Custom IP Files
├── firFilterBP //Custom IP Files
├── ifft_interface_ip //Custom IP Files
└── processing_path_ip //Custom IP Files
```

Figure 13. Design tree for the project

6. Advice for Future Students

6.1 Utilization of the Bluetooth PMOD

Future students that wish to incorporate Bluetooth into their project should understand the strengths and the limitations of the BT2 Pmods. Programming the Pmods is very easy, and the reference manual in [9] is very straightforward and thorough. It can be difficult to figure out what modes to set the device to (SPP vs HID, and Slave vs Master), but once that is sorted, setting the device to the correct mode is simple. As discussed in the previous section, a USB to UART programmer allows for very easy interfacing with the BT2 Pmod from a laptop.
I recommend that students are aware of the limits of the Pmod, specifically when functioning in Master mode. They do not support connecting external peripherals to them, such as a mouse or keyboard, and receiving signals from those devices. If students wish to use those types of peripherals, two BT2 Pmods are required. One must be wired to the peripheral, and paired to the second, which would be connected to the FPGA. However, for applications where students are interfacing with a laptop or phone, the BT2 Pmod can pair with these devices, and receive and transmit data easily. Therefore, I recommend the use of the BT2 Pmods for students who wish to control the FPGA from their laptop or phone, similar to the project in [13].

6.2 Audio Project using xFFT IP2

Future students embarking on audio projects should be mindful of the limitations inherent in the Xilinx xFFT IP. Our understanding suggests that the xFFT IP is tailored for signal processing applications that operate primarily in high-frequency bands. For instance, its intended targets such as communication applications that employ either fiber optics, Bluetooth, or RF technologies. Consequently, the xFFT IP is not ideally suited for audio processing, as it tends to introduce significant noise in the low-frequency bands. This noise detrimentally affects the quality of the output audio, rendering it nearly inaudible—a less than ideal outcome for a course project.
Xilinx, in its datasheet, advises utilizing only the upper half (N/2+1 to N points) of the output data when performing a real-valued FFT. However, the term "points" was somewhat ambiguous, leading to confusion. Based on responses from various Xilinx forums, it appears that "points" refers to the window length, implying that only samples from N/2+1 to N are utilized for audio manipulation calculations. This approach did help mitigate noise in our project. However, I recently came across a document [14] indicating that "points" actually represent the bits. Therefore, frequency domain calculations should be based on the top half of the most significant bits (MSB) of the real and imaginary data. Unfortunately, we made this discovery too late for our project, but we hope it proves beneficial to future students. It's worth noting that most FFT implementations are designed using High-Level Synthesis (HLS) or ZipCPU FFT [15] methods. I would recommend students use these implementation methods for audio processing to avoid encountering noise issues.

The xFFT IP's algorithm inherently leads to bit growth. As a solution, Xilinx recommends using a scaling factor to downscale the data at each stage of the FFT pipeline. However, the scaling factor provided by Xilinx in their datasheet is quite conservative. We observed that this conservative scaling significantly reduces the magnitude of the output audio to the point of being inaudible, with the magnitude approximately one-hundredth of the original audio input. While Xilinx acknowledges this limitation and suggests experimenting with different scaling schedules, I advise against this as it can be time-consuming. Alternatively, the xFFT IP offers a block floating-point scaling factor setting that automatically adjusts the scaling factor based on the input data. In our project, this feature worked reasonably well and is worth considering, however, it comes at the expense of increased logic utilization.

References

[1] Sound Engineering Academy, "HUMAN VOICE FREQUENCY RANGE," Sound Engineering Academy, [Online]. Available: https://seaindia.in/blogs/human-voice-frequency-range/. [Accessed 12 April 2024].

[2] MathWorks, Inc., "What Is High-Level Synthesis?," MathWorks, [Online]. Available: https://www.mathworks.com/discovery/high-level-synthesis.html. [Accessed 12 April 2024].

[3] Bela Platform, "20: Phase Vocoder (part 3), C++ Real-Time Audio Programming with Bela," YouTube, 17 February 2021. [Online]. Available: https://www.youtube.com/watch?v=2p_-jbl6Dyc&t=3239s&ab_channel=BelaPlatform. [Accessed 12 April 2024].

[4] Digilent, "Nexys Video DMA Audio Demo," Digilent, [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-video/demos/dma-audio. [Accessed 12 April 2024].

[5] University of Wisconsin-Madison, "UART Basics," University of Wisconsin-Madison, [Online]. Available: https://ece353.engr.wisc.edu/serial-interfaces/uart-basics/#:~:text=The%20UART%20interface%20consists%20of,pin%20of%20the%20second%20device. [Accessed 12 April 2024].

[6] Components101, "FT232RL USB TO TTL 3.3V/5V FTDI Serial Adapter Module," Components101, 28 May 2021. [Online]. Available: https://components101.com/modules/ft232rl-usb-to-ttl-converter-pinout-features-datasheet-working-application-alternative. [Accessed 12 April 2024].

[7] Digilent, "Pmod BT2 Reference Manual," Digilent, [Online]. Available: https://digilent.com/reference/pmod/pmodbt2/reference-manual?redirect=1. [Accessed 12 April 2024].

[8] Apple Inc., "SerialTools," Apple Inc., 12 September 2021. [Online]. Available: https://apps.apple.com/us/app/serialtools/id611021963. [Accessed 12 April 2024].

[9] Roving Networks, "Bluetooth Data Module Command Reference & Advanced Information User's Guide," Roving Networks, 26 March 2013. [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/bluetooth_cr_UG-v1.0r.pdf. [Accessed 12 April 2024].

[10] S. Schmit, "Getting Started with RN42 Bluetooth Module," DigiKey, 1 September 2016. [Online]. Available: https://forum.digikey.com/t/getting-started-with-rn42-bluetooth-module/12859/1. [Accessed 12 April 2024].

[11] DigiKey, "410-345," DigiKey, [Online]. Available: https://www.digikey.ca/en/products/detail/digilent-inc/410-345/7560228. [Accessed 12 April 2024].

[12] M.-C. Yang, "Lecture 07: Driving VGA Display with ZedBoard," The Chinese University of Hong Kong, [Online]. Available: https://www.cse.cuhk.edu.hk/~mcyang/ceng3430/2020S/Lec07%20Driving%20VGA%20Display%20with%20ZedBoard.pdf. [Accessed 12 April 2024].

[13] alexwonglik1, "Bluetooth controlled FPGA Based Synthesizer," An Avnet Company, 31 July 2020. [Online]. Available: https://community.element14.com/products/manufacturers/digilent/b/blog/posts/bluetooth-controlled-fpga-based-synthesizer. [Accessed 12 April 2024].

[14] M. Gu and R. Berg, "Guitar Hero: Fast Fourier Edition," 9 December 2015. [Online]. Available: https://web.mit.edu/6.111/www/f2015/projects/mitchgu_Project_Final_Report.pdf. [Accessed 12 April 2024].

[15] ZipCPU, "fftdemo," GitHub, 19 January 2024. [Online]. Available: https://github.com/ZipCPU/fftdemo/tree/master/rtl/fft. [Accessed 12 April 2024].

Appendix A1

Left/Right Clock (LRCLK): Also known as the word clock, LRCLK determines whether the data being transmitted corresponds to the left or right audio channel. It typically alternates between high and low states to indicate the start of a new audio word.

Bit Clock (BCLK): The Bit Clock synchronizes the serial data transmission in the I2S protocol. BCLK generates a continuous stream of pulses, with each pulse representing a single bit of audio data.