FPGAudioSphere: Spatial Audio Implementation on an FPGA

Christopher Espitia-Alvarez
Department of Electrical Engineering and Computer Science
Cambridge, MA, USA
cespalv@mit.edu

Sarah Lov
Department of Electrical Engineering and Computer Science
Cambridge, MA, USA
smyl362@mit.edu

***Abstract*—We investigate spatial audio implementations on FPGA hardware as a means of accelerating digital signal processing computations that convert real-time audio inputs into the spatial domain at a high resolution. The head creates a natural filter when processing aural stimuli, and spatial audio mimics this filter by manipulating the interaural intensity and time displacements (i.e. volume and delay) onto corresponding left and right audio channel outputs. This is accomplished by streaming and transmitting analog audio inputs via the Inter-IC Sound (I2S) communication protocol and convolving input audio samples with Head Related Impulse Response (HRIR) coefficients via a finite impulse response (FIR) filter. This FPGA project explores digital-analog conversion circuitry, I2S protocol implementations, and an original efficiently pipelined FIR filter design to achieve spatial audio effects. The project also implements a graphic user interface (GUI) that visualizes the "direction" of the audio, and the effect of the spatial audio computation on the actual audio waves.**

I. Spatial Audio Intuition

Central to creating the spatial audio illusion is HRTF (Head-Related Transfer Function) values, an empirically derived mathematical model of how sound is perceived by a person depending on the position and size of their ears, head, shoulder, and torso. We obtained coefficients from a UC Davis study database containing the sound perception of 45 participants, which have been converted from HRTF (frequency-domain) values to HRIR (time-domain) values. The HRIR values will be convolved with the real-time audio samples using a window size of 200. The HRIR values come as a 25 x 50 x 200 dimension matrix, representing 25 azimuthal (horizontal plane) angles, 50 elevational (vertical plane) angles, and 200 values at each spatial coordinate. However we will be slicing along the second dimension to only consider the 0° elevation plane, so our resulting HRIR matrix is 25 x 200 values. These 200 values per spatial coordinate again correspond to a 200-audio sample convolution window that lasts roughly 4.5 ms long. This corresponds to an audio input frequency that is 44.1 kHz, which we accounted for when choosing the ADC I2S transmitter module.

II. Input Audio Pipeline Design

The input audio pipeline is as follows: an analog audio input is fed from a device with a headphone jack, which roughly has a 1V peak-to-peak voltage. We then amplify this signal using a MAX4410 to be a 3V peak-to-peak voltage. This range includes negative voltages due to its sinusoidal wave nature; however, the PCM1802 ADC that we feed the audio into only accepts nonnegative values, so we shift this analog voltage range to be positive with a DC offset of 1.5V. Using the LM358 operational amplifier (rated to handle a 20kHz input bandwidth), we create an adder that sums the 1.5V DC signal with the amplified audio signals, ultimately shifting the entire audio signal within a positive range. Upon testing this amplified and shifted signal, we can see that the signal maintains its resolution and timing. Lastly, the downstream PCM1802 ADC module also acts as the I2S controller, sending serial data to the FPGA at 44.1 kilosamples per second for either the left or right channel and 24-bit precision. One note is that because the FPGA has a 100MHz system clock while the audio modules run at 44.1 MHz, we have roughly 2200 cycles to break up the audio streaming and convolution computation to create the effect. We call each 2200 cycle a "window", and at the beginning of each of these windows, a one-cycle high "start" signal is fired.

III. Generating Clocks with Clock Wizards

In order to properly interface with our ADC/DAC hardware using the I2S protocol, we must generate two clocks, one running at 2.1168 MHz for the I2S transmit module, and one running at 11.2896 MHz for the I2S receive module. We do so using Vivado's Clocking Wizard IP, transferring over the .xci and Verilog header files into our project.

IV. I2S Implementation

I2S is a three-wire communication protocol typically used for transmitting and receiving audio samples, consisting of system clock (SCK), word select (WS), and serial data (SDA) lines. In particular, the system clock runs at 2.1 MHz and

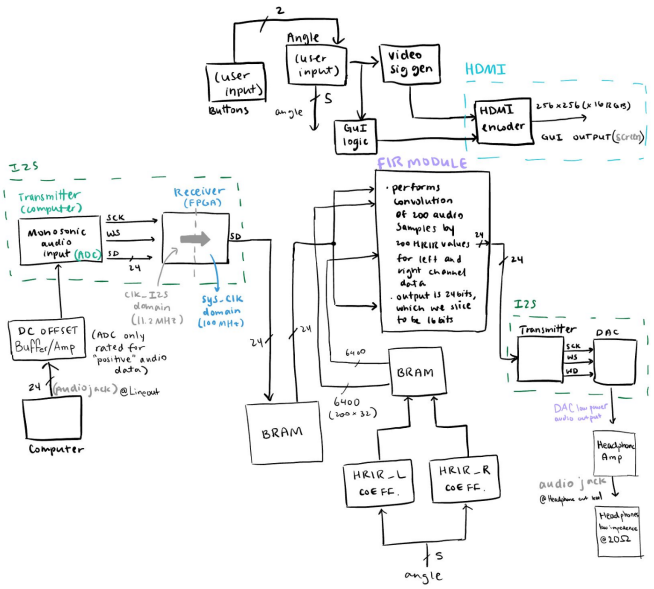

Fig. 1: Block diagram overview of the spatial audio pipeline, from raw audio input to spatialized audio output.

accommodates our 44.1 kHz audio sampling frequency. Word select indicates left versus right channel audio when pulled low or high respectively. Serial data is fed back-to-back in 24-bit data frames, MSB first (no start or stop signals are transmitted on this line).

We set our transmitter as the controller and the receiver as the peripheral and use this protocol to bookend the stream of audio into and out of our pipeline.

A. I2S - Audio In

In our first instance of I2S communication, the PCM1802 ADC module is the I2S controller, setting the system clock, and sending the word select and serial data to the I2S receiver, which is the FPGA in our case.

B. I2S - Audio Out

At the end of our entire pipeline is another instance of I2S, with the FPGA as the transmitter and the UDA1334A DAC as the receiver. The 24-bit audio data is transmitted to another MAX4410 headphone amplifier before being sent out via 3.5mm audio jack to a pair of low-impedance headphones.

C. I2S & Hardware - Testing

In order to ensure the isolated I2S modules functioned as a communication protocol, we wired up all of the hardware components, including the ADC, DAC, Headphone Amplifiers, DC offset circuitry, via breadboard along with the FPGA and sent known data via the transmitters and receivers. Important to note is that we wired the SCL, WS, and SDA lines of the ADC and DAC to the PMODA and PMODB ports of the FPGA respectively. We verified the SCL line frequencies as well as bit-by-bit reception of data via oscilloscope readings.

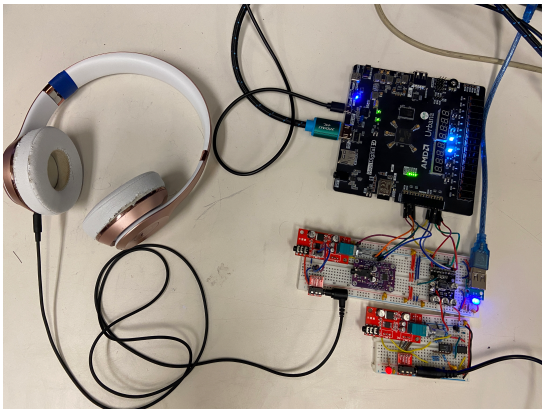

Fig. 2: Hardware setup of audio pipeline, including ADCs, DACs, FPGA, headphones, and audio source.

V. HRIR Coefficients - Storing and Processing

We will be using a 25 x 200 HRIR coefficient matrix (set at the 0° elevational angle) to convolve our audio samples with to achieve the spatial audio effect. In particular, this HRIR matrix encodes 25 azimuthal angles (-80° to 80° with a step size of 14.4°) which are convolved with a 200 audio sample window. Because the HRIR coefficient values come in the form of signed doubles (64-bit floating-point values), we will shift and slice them to become 32-bit integer values and discard the rest of the floating-point precision (to avoid length computations) that we will multiply the audio samples by. To do so, we first find the minimum HRIR coefficient value to be on the order of 1E-12, and multiply all of the coefficients by 2E+30 (i.e. left-shifting the 64-bit value by 30), and before taking the top 32 bits of these results. We wish to deal with integer multiplications (the largest of the coefficients pos-processing being on the order of 1E+9), so shifting and then slicing largely preserves the precision of the coefficients while decreasing the complexity of calculations. Cutting the bit-width in half this way will also not result in a substantial difference in output quality since only the relative values of audio amplitude affect the spatial audio illusion, as opposed to the actual magnitude of audio data.

After this coefficient processing, we convert the matrix values back into a .mat file, then convert it to a .mem file to be readily stored in a BRAM upon module instantiation. All of this processing is completed using a Python script we wrote that also preserves the signs of the coefficients.

VI. Memory Considerations and BRAM Usage

We instantiate two types of BRAMs for two main purposes. The first BRAM is used to store 24-bit incoming audio inputs in a cyclical fashion such that at each rising edge of the 44.1 MHz clock controlled by the PCM1802, the oldest audio sample is replaced with a fresh one. We will be averaging audio samples from the left and right channels such that they become a mono-channel audio source stored in the BRAM

(this enables our program to function on both stereo and mono audio sources). Again, it is the convolution with the HRIR coefficients that gives this spatial effect. The BRAMs are read from a signal generated from within the downstream convolution module (FIR filter). We maintain a write and read address pointer for the audio BRAM, with the write address to store real-time audio samples. The read address pointer will always be roughly 200 cycles behind the write address pointer (205 cycles behind in our implementation for a small buffer). This ensures that the audio samples can be properly pipelined into the FIR filter, since that module requires 200 audio samples per window.

Additionally, we have two BRAMs that will be used to store all of the left and right HRIR values separately, and we stack them by azimuthal angle such that the addresses of these 200-coefficient sets can be calculated easily by azimuth selection. These BRAMs will only have a read port, as the values are all loaded from the pre-processed .mem coefficient file.

VII. Azimuth Selection

To set the azimuth angle from which the audio sounds like it is coming from, the user can choose between two options: an azimuthal auto-sweep, whereby the azimuth changes roughly every half second in a cyclical fashion, or a manual selection, which can be cycled through using the five rightmost switches. The azimuth auto-sweep works by incrementing the angle every quarter- to half-second (translating to roughly 14000 cycles) and propagating that to offset the values read by the HRIR coefficient BRAM. On the other hand, the switch logic is LEDs are lit according to azimuth selection, for another complementary visual to users.

VIII. FIR Filter

The proper and timely convolution of HRIR coefficients with the audio samples is essential in creating the spatial audio effect. We divided this process into six stages: INITIAL, LOAD, MULTIPLY, and three tree accumulation stages (TREE_ACCUMA, TREE_ACCUMB, TREE_ACCUMC) within an overall FIR filter module. We instantiate two FIR filters to simultaneously compute left and right audio outputs that are pipelined downstream to audio out.

A. INITIAL Stage

This inaugural state simply detects when the start signal is high, which indicates that a new window of convolution should begin. If the start signal is true, it will directly move to the LOAD state on the next style. The INITIAL stage is also where the FIR filter remains for the rest of a window after computing a valid output.

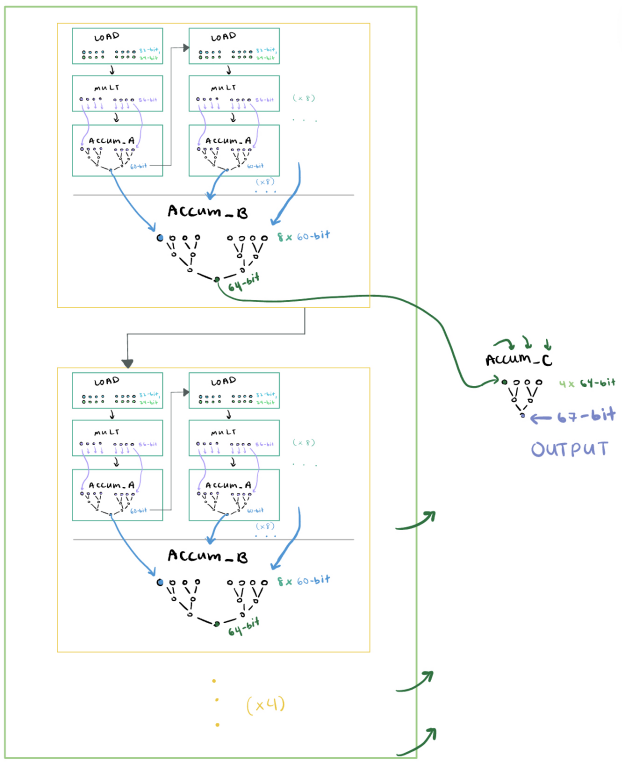

Fig. 3: Diagram showing main FIR computation stages (designed to be cyclical for efficiency and minimized register memory usage), data flow within the intermediate tree accumulation structures, and associated data widths.

B. LOAD and MULTIPLY Stages

In the LOAD stage, we issue BRAM read requests (via a "load_ready" output signal) to begin obtaining both the audio samples and HRIR coefficients. After inducing a two-cycle delay to the FIR Filter to account for the BRAM latency, eight 24-bit audio samples and 32-bit HRIR coefficients are loaded into respective registers across eight clock cycles. also pull the "load_ready" signal low two cycles before the streaming ends (i.e. on the 6th set of streamed values). We chose to load eight values each at a time to minimize the number of registers used to store these values in preparation for the MULTIPLY stage, thus requiring a more cyclical nature of our FSM. Once the values have been loaded, the filter moves to the MULTIPLY stage. In the MULTIPLY stage, the eight audio samples and HRIR coefficients are multiplied in parallel and stored in 56-bit registers. Following this, the FIR Filter moves onto the TREE_ACCUMA stage.

C. TREE_ACCUMX Stages

The tree accumulation stages are established to do parallel pairwise additions, with TREE_ACCUMA and TREE_ACCUMB starting with 8 leaves each (56-bits and 60-bits respectively), and TREE_ACCUMC starting with 4 64-bit leaves. At each layer of the tree, there are two times fewer leaves, which are stored in registers as intermediary sums that

trickle down to a singular final sum. In doing so, the sum bits only grow (maximally) by one bit at a time, and given there are only two or three layers in the tree, saves on register space usage overall. An additional bit is added to the output register to preserve its signed bit.

After the tree summation in the TREE_ACCUMA stage is completed, the FSM redirects back to the LOAD → MULTIPLY → TREE_ACCUMA cycle seven more times until there are enough intermediary sum values (that are once again, saved in registers) that can be fed into the leaves of the TREE_ACCUMB tree. Once TREE_ACCUMB completes its tree summation, the entire cycle starting from LOAD begins again until there are enough inputs for summation to begin the TREE_ACCUMC stage. This last summation stage happens only once, and the 67-bit output from that tree is the final FIR filter output sent downstream for further data manipulation before being outputted as audio.

Again, a central design choice we made in this FSM flow is the division of data loading and processing into smaller subcycles to minimize register data usage.

IX. Graphical User Interface (GUI)

Lastly, we wish for the user to have a complementary visual experience so that the user can view the audio waves coming from the original signal compared to the streamed-in left and right channel audio waves. Additionally, we will graphically display the azimuthal angle chosen by the user, which updates as the user presses the left and right buttons on the FPGA to change the angle. The video graphics must take in original data point as well as the output of the FIR filter for both left and right. The video graphics takes in these new values on the last cycle of a 2200 cycle window. In order to map the 24 bit values into a visual offset, the top 8 bits are chosen as the offset in pixels. Therefore, each audio signal has a peak to peak max amplitude of 256 pixels. Additionally, this 8 bit value is stacked for original, left, and right data into one 24 bit value. Offset values are stored and read of out a BRAM, which will now be elaborated on further.

The system clock and audio filter logic occurs on the 100 MHz "sys_clk". The video logic occurs on the "clk_pixel" which is a slower clock signal at 75.25 MHz. In order to cross the clock domain, audio samples go through the BRAM, ergo they are written on "sys_clk" and read out on "clk_pixel". When computing the output of a frame, the BRAM that is being read from must not be written to. This is done by having two BRAMs, one is frozen and is being read on the "clk_pixel" domain, while the other is fed up the currrent streaming data on the 100 MHz domain. On the "nf_hdmi" signal, the BRAM switchs. This control logic is dictated by the slower "clk_pixel" side and directly controls the write enable ports on the 100 MHz side. In order to cross the time domain from slower to faster clock, a pulse synchronizer macro is employed on the "nf_hdmi" signal so that the 100 MHz clock domain can update which port it is writing to. The Video Signal Generator runs 1280x720 HDMI at 60 frames per second. Therefore, the max number of audio sample offsets that need to be stored in the BRAM is how many samples the 44.1 kHz sampling outputs within 1/60th of a second. This number turns out to be 735. So, both BRAMS have a depth 735 with a width of 24 bits.

For the video graphics, we split the graphics into three regions: original input wave, left and right output wave, and GUI for spatial representation of which direction audio is coming from. For the visual of audio direction, we fan out squares representing audio sources by their respective angle relative to a center square that represents the user. Azimuths values are coming from the 100 Mhz clock domain, so they must go through a BRAM where they are read out in the clk pixel domain.

Finally, the graphics driver successfully visualizes the audio signals and the direction of the audio source. The video is sent to a screen via HDMI.

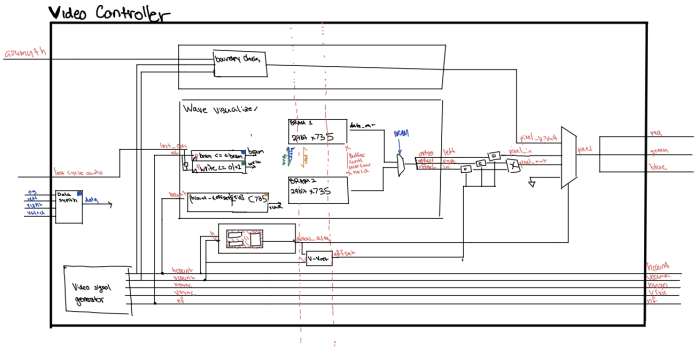

Fig. 4: Input signals into video controller and video controller diagram.

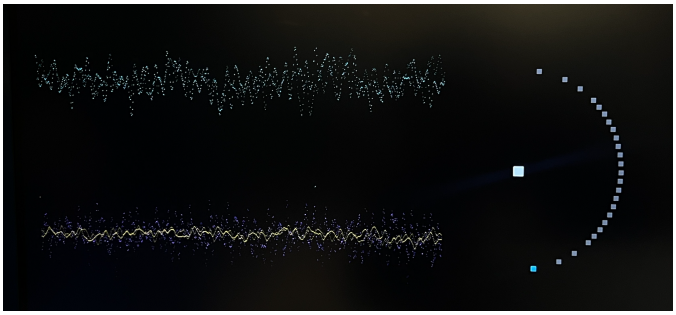

Fig. 5: Completed graphic user interface (GUI) is shown. On the left side, there are two waves. The cyan-colored wave on top is the buffered audio sample from an audio source. The two-colored bottom wave is a superposition of the resulting left (yellow) output audio and right (purple) output audio coming from the FIR filter. On the right, there is a diagram of the azimuth that the user has chosen; the gray square that is highlighted blue indicates the "direction" the audio is perceived to be coming from.

X. Testing and Results

To ensure that the individual components and modules work as intended, we performed incremental tests on the ADC/DAC hardware and I2S modules, FIR filter, and HRIR coefficient processing. We tested the audio pipeline incrementally, first ensuring that audio properly moves as input to output through the I2S transmitter and receiver modules. We ensured that the DC offset was maintained via oscilloscope testing and that there were no unwanted side effects. Next, we added the audio BRAM to that pipeline and tested that setup. Separate from this, we tested the FIR filter by comparing its output of convolving sets of known signed and unsigned numbers for which ground truth resulting values were computed. This required extensive test benching, parsing packed registers with binary, and revising logic to be more efficient via parallelization if possible. We then fed in the HRIR coefficients into the FIR filter, and once confirmed that the convolved output aligned, wired all of the components together.

A brief discussion on the efficiency our design. It takes 445 cycles for the FIR computation to complete, and two values are computed in parallel. That is a theoretical throughput of 4,500,000 samples per second. However, we only can compute as fast as we receive data so it is limited to 88,000 samples per second. In terms of resource utilization, we used 96 DSP Blocks, 38 RAMB36s, and 2 RAMB18s. For both BRAM and DSP, our resource utilization was 24%. This could be improved, since instead of having two parallel FIR filters, we could just have one that computes left then right. This would half the DSP usage. However, our current usage is still well below the max and such optimization is not necessary as we are bounded by audio sample rate regardless. In terms of BRAM, usage is currently well optimized

In terms of our goal and timing requirements, our design fully meets and exceed our requirements. We finish computing audio outputs early within the 2200 cycle window. Also, in terms of the video graphics pipeline, our design successfully never loses a sample when outputting HDMI at 1280 x 720 pixels. Finally, we reached our ideal goal of properly and quickly computing spatial audio and displaying our audio samples to a screen.

XI. Future Work

Upon building our completed project, we noticed we had a worst negative slack (WNS) of around 2.44, meaning we had potential computations left unused. Some future work we would like to explore is interpolation, whereby azimuth angles that were not specifically assigned sets of coefficients could be statistically generated. Additionally, we would like to have the full azimuthal range from 0 to 359 degrees, which would offer a complete angular range to experience.

XII. Member Contributions and Acknowledgments

Chris worked on setting up the hardware, I2S receiver and transmitter modules and generating the GUI and HDMI display. Sarah worked on pre-processing the HRIR coefficients into .mem files suitable for our project use and implementing and test benching the FIR filter.

Both members were present throughout the debugging process, collective integration of modules, and designing and revising the entire pipeline from audio-in to audio-out.

We would like to thank Professor Steinmeyer and TAs Kailas, Kiran, and Jan for giving us advice and helping us debug our project.

References

[1] V. R. Algazi, R. O. Duda and D. M. Thompson, "The CIPIC HRTF Database," U.C. Davis, October 2001.

[2] "Building a high speed Finite Impulse Response (FIR) Digital Filter," ZipCPU, Gisselquist Technology, LLC, September 2017:

[3] C. Cheng and W. Wakefield, "Introduction to Head-Related Transfer Functions (HRTF's): Representations of HRTF's in Time, Frequency, and Space," pp. 1-28.

[4]