Digital Systems: Final Report

Audio Signal Processing & Adaptive Noise Cancellation Using Digital Filters on an FPGA

Team members : Jiya Desai | Mumuksh Jain | Nishant Kumar | Nishi Shah

Date: 23/04/2024

Problem Objective

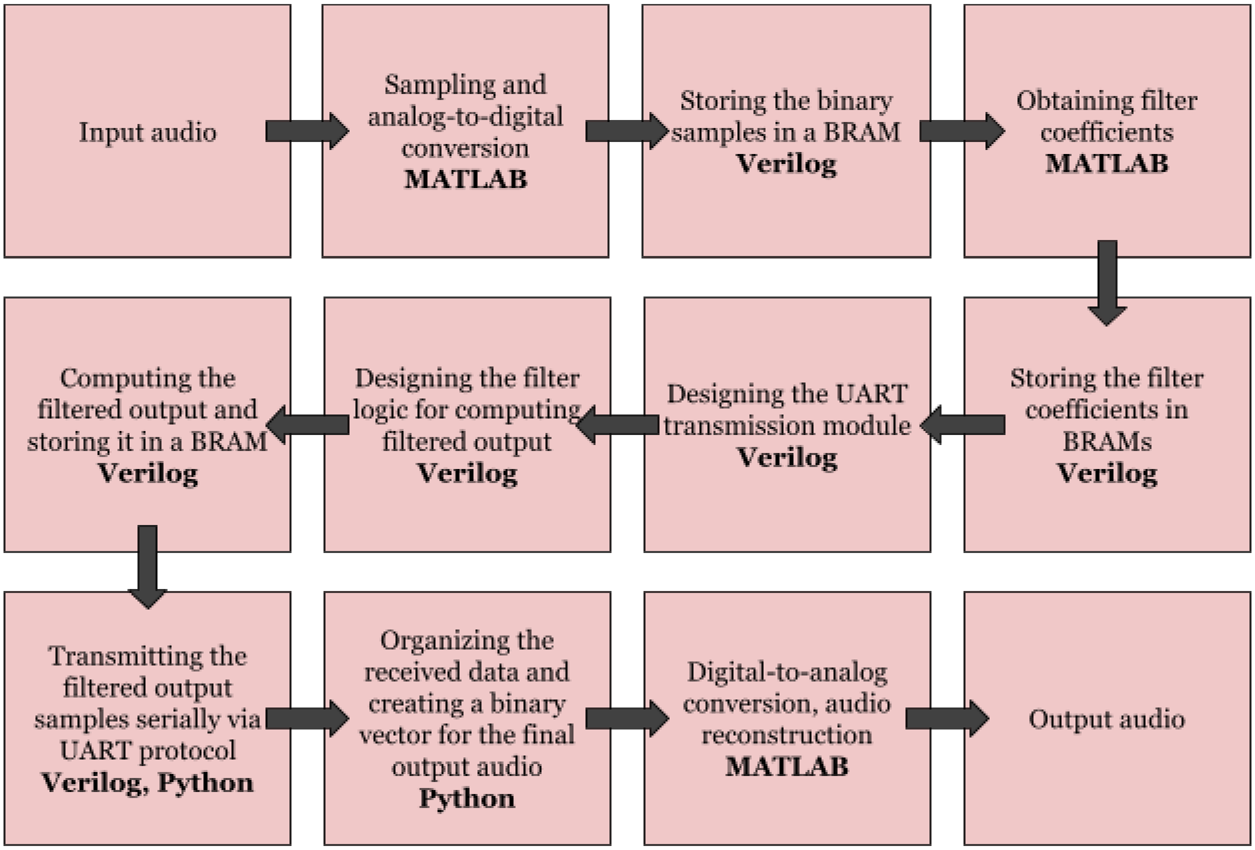
Signals obtained from sensors commonly contain significant noise. To eliminate this noise, one can either utilize improved sensors/probes or implement post-acquisition noise filtering.

The objective of this project is to employ digital filters on an FPGA to effectively eliminate the noise from the signal, and try to selectively trim the audio signal based on particular frequencies (i.e., allowing only certain frequencies to be present in the filtered signal).

In particular, we used audio signals as our input signal. We implemented basic digital filters - low pass, high pass, band pass, all pass and band stop - on an FPGA, before designing an adaptive noise cancellation filter (Weiner filter). We used a BASYS-3 FPGA board, Xilinx Vivado as an interface for Verilog, MATLAB and Python for the implementation of this project.

[Adaptive Noise Cancellation (ANC) reduces unwanted background noise from audio in real-time. It uses microphones to capture desired signals and noise, adjusting an adaptive filter to predict and cancel noise. The error signal refines the filter, enhancing the desired audio signal in noisy environments.]

We divided the project into specific weekly tasks and followed this breakdown.


Input audio
Sampling and analog-to-digital conversion
MATLAB
Storing the binary samples in a BRAM
Verilog
Obtaining filter coefficients
MATLAB
Storing the filter coefficients in BRAMs
Verilog
Designing the UART transmission module
Verilog
Designing the filter logic for computing filtered output
Verilog
Computing the filtered output and storing it in a BRAM
Verilog
Transmitting the filtered output samples serially via UART protocol
Verilog, Python
Organizing the received data and creating a binary vector for the final output audio
Python
Digital-to-analog conversion, audio reconstruction
MATLAB
Output audio

Our workflow for the project

Link to the GitHub repository:
https://github.com/Nishant-2307/DS_Project_ANC_Digital_Filters

Week 1 overview

Working in the domain of digital systems, the very first step is to design a way to convert our analog signals into their digital counterparts, and then to convert the filtered digital signals back to analog to produce the final output. For this task, we relied on MATLAB for Analog-to-Digital and Digital-to-Analog conversions. Along with this, it was also important to understand the memory and storage mechanism of our FPGA board at the very beginning. Hence, at the end of the first week, we were able to accomplish the following two tasks:

1. Conversion of analog audio signals to digital using MATLAB, changing each sample to an 8-bit binary number. The audio signal has amplitude values in the range [-1,1]. We first scaled up these values in the range [-127, 127] (corresponding to the range of numbers we can store using 8 bits) and then converted each sample to its binary equivalent. As a result, we had a bitstream of $8 \times No.\ of\ samples$ number of bits corresponding to our audio signal.

   Video demonstration link : Analog_to_Digital_binaryoutput_Analog.mp4

2. FPGAs use Block RAM (BRAM) as a storage element. We need to instantiate these memory elements on the FPGA board using Verilog. We were able to write data into BRAMs and read from them. In order to store pre-determined data into a BRAM, we first had to write the data into a COE (coefficient) file (which is used to create memory initialization files) and then instantiate the BRAM with the file. We can also overwrite previously stored data in BRAMs.

   Video demonstration link : BRAM_processing.mp4

Week 2 overview

1. We had to add noise through a natural source or generate it artificially using any known method.

   We recorded audio and added it to our signal for our noise through a natural source. These are the kinds of sounds we may encounter in real-world applications. However, they are generally trickier to deal with.

   Meanwhile, for the noise generated by artificial sources, we chose Additive White Gaussian Noise. This kind of noise can be added (arithmetic element-wise addition) to the signal. Also, its mean value is zero (randomly sampled from a Gaussian distribution with a mean value of zero. standard deviation can vary). It contains all the frequency components in an equal manner (hence “white” noise). AWGN is important because it is easier to model for analytics and generate. However, it may not represent realistic noise conditions for some applications.

   Signal-to-noise ratio (SNR) can be defined as follows :

   $$SNR = 10log(\frac{RMS^2_{signal}}{RMS^2_{noise}})$$

   where RMS_signal is the RMS value of signal and RMS_noise is that of noise.

   A greater SNR implies a ‘better’ audio signal where the noise offers lesser distortion to our original audio.

   Here’s the link to our notebook used for Gaussian noise generation:
   https://github.com/Nishant-2307/DS_Project_ANC_Digital_Filters/blob/main/Week_2/Noise_Addition-2.ipynb

   Here’s the link to the Github subfolder containing the audio signals with noise addition:
   https://github.com/Nishant-2307/DS_Project_ANC_Digital_Filters/tree/main/Week_2

2. Once we could read and write data in the FPGA memory, it was necessary to figure out how this data could be transmitted back to the computer. For this, we had to use the Universal Asynchronous Receiver/Transmitter (UART) protocol. In our case, we needed only the transmitter part, because we already had a way to store data in the BRAM through COE files as explained above. The task was

getting the data (generated through FPGA computations after filtering) back on the computer. The transmitter module of our UART is essentially a state-machine, and is able to transfer 8 bits (1 byte of data) at a time. The byte to be transmitted is enclosed within a start bit and a stop bit, which are states to recognise when the transmission starts and when it gets completed. It can also have an optional parity bit for error detection. It employs serial transmission.

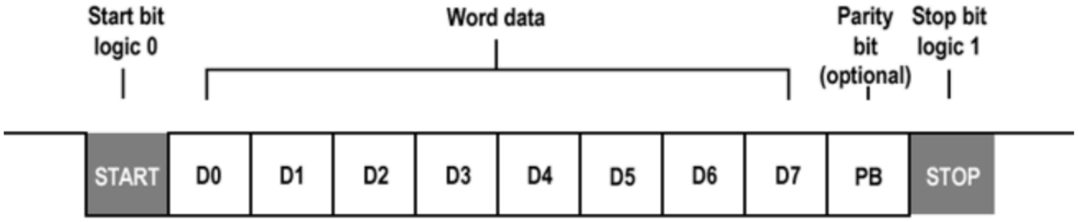


Fig. 1: UART logic
Image source: https://developer.electricimp.com/resources/uart

The transmitter module written in Verilog had an input clock, input data (8-bits), input reset, input transmit mode (on/off) signal (analogous to enable signal), output flag and an output data bit (the bit to be transmitted). It follows the standard transmission baud rate of 9600, which is the number of bits transmitted per second. The clock frequency of our FPGA board is 100 MHz, so the number of clock cycles required for the transmission of 1 bit came out to be $100,000,000 \div 9600 \approx 10417$.

UART is a serial communication protocol, which means it communicates with the computer using serial data communication ports. So for this, once we channeled the output transmission bit of the transmitter module to the designated port on the FPGA board, the data would be received by the serial port of our laptop to which we had connected our FPGA cable. In order to receive this data in the laptop, we used a Python wrapper code, which utilizes the PySerial library for reading serially received data at the serial communication ports, and used file handling to write the received data to a text file. This way, we were able to generate files of the data we had transmitted from the BRAM of the FPGA.

Video demonstration link :

UARTimplementation1.MOV

UARTimplementation2.MOV

Week 3 overview

1. Once we had the bitstream of our input audio signal as well as a way to store it on the FPGA and get the computed data back from the FPGA ready, it was time to design the filters. We started with the design of the 5 basic filters - low pass (allowing frequencies only below a specific frequency), high pass (allowing frequencies only above a specific frequency), band pass (allowing frequencies within a specified range), band stop (removing frequencies within a specified range) and all-pass (allowing all frequencies).

   All filter designs have a set of coefficients. The number of coefficients depends on the order of the filter we are using. The higher the order, the sharper is the roll-off gain of the filter (i.e., better sensitivity to the bandwidth).

   The filtered output is essentially the 1-D convolution of the audio samples and the filter coefficients. Different filters have different sets of corresponding coefficients, but the same 1-D convolution is followed for each filter. These filter coefficients were generated using MATLAB, and the 1-D convolution was done in Verilog. The input samples were stored in a BRAM instance, the coefficients in another, and the computed outputs were written in a third BRAM instance. This third BRAM instance was a simple dual port BRAM, different from a single-port BRAM, because we needed to read from the same BRAM that we had written into. The dual port BRAM facilitates writing and reading from the same BRAM instance at once.

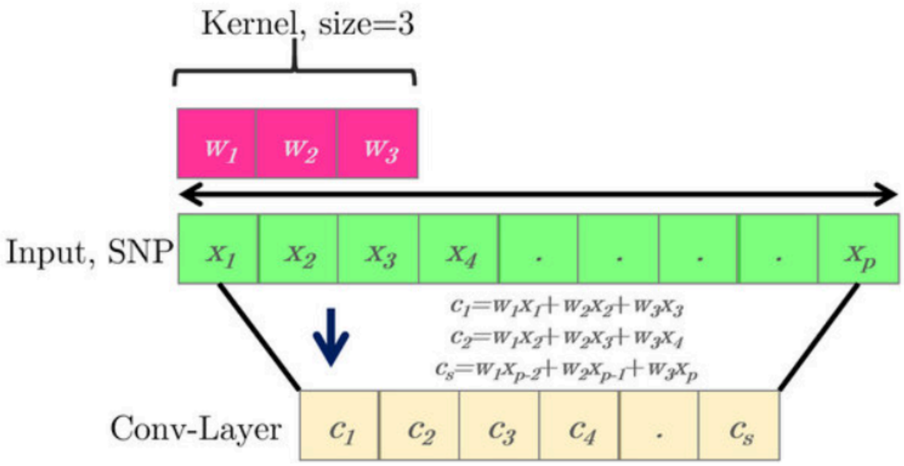
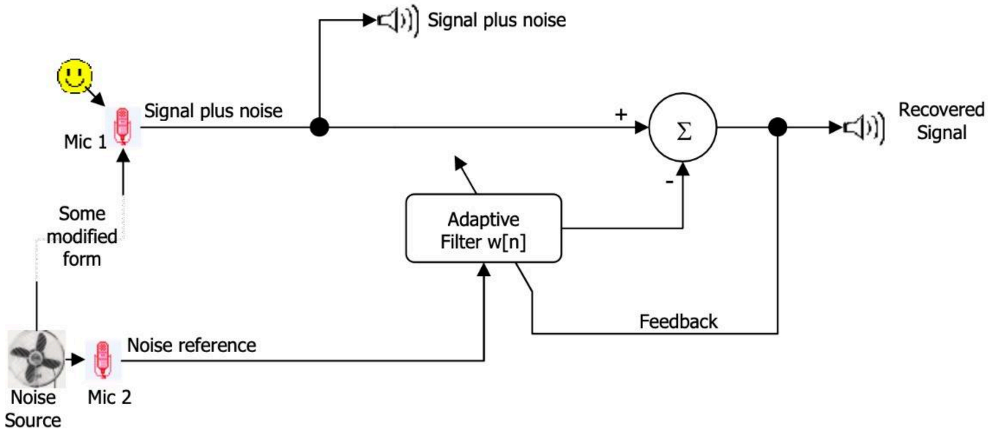


Fig. 2: Schematic representation of 1-D convolution
Image Source:
https://www.researchgate.net/figure/a-Simple-scheme-of-a-one-dimension-1D-convolutional-operation-b-Full_fig2_334609713

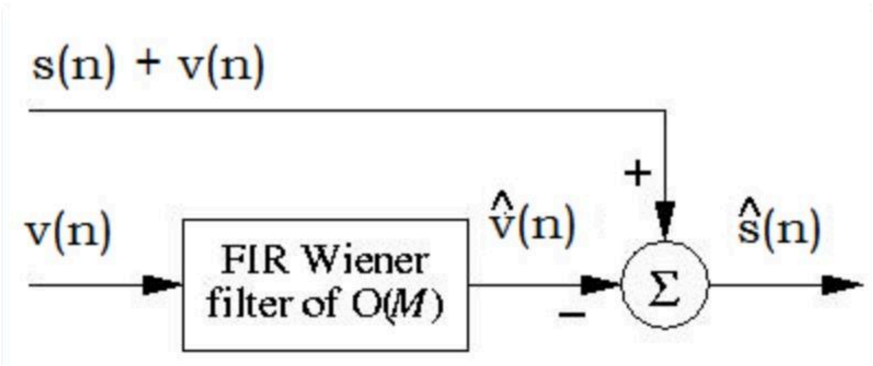
The number of output samples will be the same as the number of input samples. The size of the input samples is 8 bits, and the size of the coefficients we used is 16 bits. This would mean an output sample size of 8+16=24. As a result, we had as many output samples as the number of input samples, but each of size was 24 bits, not 8. Since the UART protocol only transfers one byte at a time, the 24-bit outputs posed a problem during the final transmission stage, as we will see later.

Video demonstration of 1-D convolution:
1Dconvolution.mp4

2. Designing the Wiener filter:
   For the active noise cancellation component of our project, we use the Wiener filter. The main objective of the Wiener filter is to estimate a desired signal (which might be corrupted by noise) from an observed signal that is a linear combination of the desired signal and noise. The flowchart below shows the MATLAB implementation of the wiener filter from where we obtain the coefficients for convolution.



The diagram below shows the final implementation on Verilog. This was further synthesized on the FPGA and the obtained output is processed similarly to our other five filters, transmitted by UART and then converted back into an analog signal.



Week 4 overview

Once we had all of our individual components ready, it was time for assembling the project together. We have the following set of files:

1. MATLAB code for ADC and DAC
2. Verilog code comprising the following modules:
   - UART transmitter module
   - Filter logic module with BRAM instances for filter coefficients
   - Top module with BRAM instances for input audio bitstream and filtered output (dual port)
3. Python code for receiving the transmitted output at the serial port
4. Python code for formatting the received output (removing 0b prefix from all the samples, zero padding to make each sample uniform and concatenating all the samples to a single vector)

The final binary vector we get can be heard as audio through MATLAB's digital-to-analog conversion.

The only major challenge we faced throughout the entire process was how to manage the 24-bit output samples, since UART can only transmit 8-bit samples. Potential solutions to this problem include:

1. Clipping the 24-bits samples to make them 8-bits. However, the reconstructed audio from these clipped samples does not match the expected output at all, since it causes the loss of information.
2. Breaking each 24-bits sample into 3 8-bit samples and then transmitting them. But this would require thrice the number of BRAM locations, since we need to store each 8-bits sub-sample at a different location for UART transmission. This would also require the modification of the Python and MATLAB code at the receiving end to re-organize the received information. But this approach will not result in any loss of information.

3. Modifying the size of the input samples and coefficients in such a way that the output would fit into 8-bits. However, it is extremely difficult to retain meaningful information within such small bit sizes.

Conclusion

With the proposed solutions to the problem we encountered, we were able to successfully meet the objectives of the project. There were learning opportunities at every step of the project, and a large majority of them were in coming up with solutions to the problems we faced. We explored the capabilities of Verilog and FPGA to a good extent, along with realizing the importance of tools such as MATLAB and Python in complementing the project.

Acknowledgements

This project wouldn’t have been possible without the guidance of our mentor, Prof. Joycee Mekie, who encouraged us to take it up and provided constant support and motivation throughout the process. We would also like to thank Haikoo Khandor for his invaluable input at every checkpoint of the project. Vrajesh Patel and Zaqi Momin also helped us in making this project a success.

References

1) Adding noise to audio clips by Lahiru Nuwan Wijayasingha. Online: https://medium.com/analytics-vidhya/adding-noise-to-audio-clips-5d8cee24ccb8

2) Real-time Audio Processing via FIR Filters on Basys-3 FPGA by Fahad Syed https://youtu.be/10Z_WsDO_uE?si=Wjb0NdubV4bbapij