Stealthoscope Final Report

Sophie Guo, Kevin Qian, and Maggie Zhang

***Abstract*—The stealthoscope is a point of care FPGA based medical device to help digitally record and process heart sounds from a stethoscope. Our system is comprised of three main components—audio sampling, data analysis, and data visualization. The combined system takes real-time audio recordings, performs an Adaptive Line Enhancer (ALE) algorithm to separate heart from lung sounds, and transforms this audio data into a visual display of the frequency spectrums for both sound types.**

Github: https://github.com/skeqiqevian/stealthoscope

I. Hardware Components

A. Stethoscope

The base of our stethoscope is a traditional analog stethoscope commonly used by doctors. Taking advantage of the tubing of a traditional stethoscope, we cut the tubing near the head of stethoscope and fit our own microphone and wiring into the tube.

B. Microphone and Microphone module

We use CUI Device's CMC-6018-44L100 Electret Condenser Microphone. This microphone is sufficient for our purposes as its frequency response curve is relatively stable until up to 10 kHz. The microphone is connected to an Adafruit Electret Microphone Amplifier - MAX4466 with an adjustable gain between 2.4 and 5.5 Volts. Tuning the amplifier via an oscilliscope, we set the mic to a 3.3 Volts gain, which is required of the Analog-to-Digital converter.

C. Analog to Digital Converter

We use Microchip Technology Inc.'s MCP3008 device which is a 10-bit Analog to-Digital converter (ADC). The analog input channels are configured as single-ended inputs. Communication with the Diligent Nexys 4 DDR FPGA is accomplished via SPI protocol.

D. Circuitry

In combining the ADC and Microphone module, we want to supply only AC power to our microphone at the voltage and gain that would achieve the highest spread of outputs for our module. Our circuit uses a $3.3\mu F$ capacitor to cancel our DC current from our FPGA. Once we have only AC current, we use a 100K Ohm potentiometer to divide our incoming voltage from the FPGA. The microphone module is capped at 3.3 Volts, so we divide the voltage in half to 1.65 Volts. By itself, the gain for heartbeats was around 0.5 Volts, which is about 1.15 Volts lower than optimal. To amplify the gain, we added an operational amplifier (op amp) paired with another 100K Ohm potentiometer to multiply our gain by 2.3. The voltage dividers and op amp create a voltage amplitude of 1.65 centered at 1.65 Volts, so that our heartbeat voltage ranges from 0 to 3.3 Volts.

II. Sampling and Data Processing (Sophie)

A. Data Sampling and Transmission with ADC

A majority of heart sounds emit within the range of 60 - 200 Hz, and lung sounds (i.e. breathing, coughing, wheezing) usually emit around 600 Hz, but can appear at frequencies as high as 2000 Hz. To sufficiently capture such frequencies, we aim for an ADC sampling rate of 40 ksps, allowing around 20 samples per period. Since data transmission from the ADC to our FPGA device occurs via SPI protocol, we implemented an SPI state machine module to handle this protocol. Because the SPI protocol requires 18 serial clock cycles to output one complete data sample, we must drive our serial clock (SCK) at 720 kHz to achieve our desired 40 ksps sampling rate. Since this module is driven by `pixel_clk` (65 MHz) we implemented our serial clock by waiting 45 cycles on the `pixel_clk` between driving edge changes in the serial clock.

The input channels are configured as single-ended inputs as part of the serial command through the serial data input pin. The results of the A/D conversion are outputted as 10 bits in MSB order.

B. FIR Low Pass Filter and Down Sampling

With an ADC output rate of 40 ksps that can represent audio frequencies up to 20 kHz, we intend to downsample our data to 10 ksps, so we can yield audio frequencies up to 5 kHz. In order to prevent aliasing during the downsampling process, we need to first remove frequencies of 5 kHz-20 kHz from the audio by passing it through an anti-aliasing filter. In addition, it is important to note that when data is read into the anti-aliasing module, the msb of our 10-bit data is flipped to convert our data from offset binary format, to a signed two's complement format.

The anti-aliasing filter implementation consists of two modules, one module that implements a 31 tap Finite Impulse Response Filter, and a combinational module that returns a signed 10 bit filter coefficient given a tap number between 0 and 30. These coefficients are obtained using the MATLAB `round(fir1(30, 0.25)*1024)` command to generate a lowpass filter with a cutoff frequency of 0.25. The coefficients are scaled by 2^{10} to produce integer tap coefficients.

The 31-location sample memory buffer required by the FIR module is implemented using a circular buffer and incoming samples are stored at an incrementing offset. Since our 65 MHz clock operates much faster than the 40 kHz sample rate of our incoming data, we are able to iteratively multiply each sample in the buffer with its respective coefficient and add to the running aggregate sum. Once 31 samples have been summed, this is our final filter output.

The filter calculation can be shown in the following summation:

$$\mathbf{y}[n] = \sum_{i=0}^{30} (c_i \cdot x[n-i])$$

where c_i is the coefficient supplied by the coefficients module and x is our circular buffer array of input samples.

After the FIR low pass filter module is applied, the data is forwarded to a downsampling module, which downsamples our data from 40 ksps to 10 ksps.

III. ALE (KEVIN)

In order to separate the heart and lung signals, we use an Adaptive Line Enhancer (ALE) algorithm described in [1]. At its core, the ALE algorithm learns a FIR filter which tries to predict the signal using historical data points. At each iteration, it updates the filter weights using least squares gradient descent. The algorithm is parameterized by

- L, the number of filter taps
- Δ, the prediction distance
- μ, the learning rate of the gradient descent

We selected values of $L = 50$, $\Delta = 32$, and $\mu = 2^{-23}$ based on Python simulations of the ALE algorithm on data collected by attaching an oscilloscope to our hardware setup.

To describe the algorithm mathematically, let $x[k]$ denote the input signal, $y[k]$ denote the output signal, and $\mathbf{w}[k]$ denote the vector of filter weights at time k. For notational convenience, let

$$\mathbf{X}[k] = [x[k-\Delta], x[k-\Delta-1], \ldots, x[k-\Delta-(L-1)]]$$

denote the historical data points. Then the ALE computes an output

$$y[k] = \mathbf{w}[k] \cdot \mathbf{X}[k].$$

Afterwards, the weights are updated according to the least squares method:

$$\mathbf{w}[k+1] = \mathbf{w}[k] + \mu(x[k] - y[k]) \cdot \mathbf{X}[k]$$

The ALE algorithm learns the autocorrelation of the audio signal at a fixed, predetermined shift Δ. Because the heart signal is narrowband and approximately periodic, it has strong autocorrelation, and thus the ALE will learn to predict it. On the other hand, the lung signal is broadband and thus has a weaker autocorrelation. Therefore, the ALE algorithm's output will approximate the heart signal. To derive the lung signal, we can subtract the ALE's output from the original signal.

Our testing shows that the ALE algorithm is capable of learning the periodic heart beats measured from the stethoscope. See the Results section for more details.

IV. FAST FOURIER TRANSFORM (KEVIN AND MAGGIE)

A. Fast Fourier Transform

We use Vivado to generate the Xilinx FFT IP (as a `*.xci` file). Our FFT module is configured to perform the calculation on 1024 input data points. The inputs are 10-bits wide and the output data is 32-bits wide.

B. Magnitude Calculations

After finishing the FFT calculations, we convert from the 32-bit output values into real magnitudes that represent intensities for given frequencies. For magnitude calculations, we use a `square_and_sum` module in addition to the `axis_data_fifo_0` and `cordic_0` IP's to accomplish this task. These modules in sequence transform our 32 bits of FFT data into a 24 bit output representing the intensity at each frequency.

V. FREQUENCY WATERFALL VIEW (MAGGIE)

A. Intensity Averages

Our FFT utilizes 1024 input values for each transformation. Because we have specified in our FFT module that we want natural bit ordering, our frequency values are output from smallest frequency to largest, which ranges from 0 to 10 kHz after downsampling. Though it would be optimal to output all 1024 different frequencies, our monitor caps our y-axis at 684 and since we aim to display both a heart and lung waterfall visualization, we chose to construct an averaging scheme of output values.

Initially, we strictly averaged every four FFT output values, but after additional testing on actual heartbeats, realized that the heartbeats span a small spectrum relative to the overall frequency values. Our final waterfall averaging scheme breaks up the FFT data into three sections. The lower 256 frequencies are averaged every two outputs, the next 256 values are averaged every four outputs, and the final 512 values are averaged every eight outputs. The new averaging scheme leads to a much more compelling waterfall plot that better highlights the heartbeats as they occur.

B. Magnitude to Color

After calculating average intensities of magnitudes from the FFT, we convert the 24 bit magnitude values into 8 bit colors. Our color palette consists of a 16 value subset within the classic Video Graphics Array (VGA) 256 color palette. The colors are bright, highly contrasted, and span the colors of the rainbow, lending itself to a visually compelling waterfall display.

Though only 10 seconds worth of data needs to be stored at a time in each heart and lung waterfall Block RAM (BRAM), to accommodate for BRAM space required by our time series data visualization stretch goal, we opted for a smaller color palette. To convert our averaged magnitude values to colors, we used the bits from indices [4:1]. Though the overall magnitudes were 24 bits coming off the IP, we realized that the actual data from the ALE, due to the weak intensities of the heartbeat, were well within positive and negative 32 for values, and thus we only used the smaller bits. This allows us to "bin" the magnitudes into 16 different bins which correspond to our palette.

To select from the pre-set palette, we utilized a single port BRAM with depth 16 and width of 8. The bit shifted and truncated magnitudes are our palette read addresses. We obtain an output two clock cycles later, and other inputs including

valid_in and is_last are therefore pipelined by two clock cycles.

VI. Time Domain View (Maggie)

Alongside the primary waterfall view of our heartbeat data, it is also useful to observe how the predicted ALE heartbeat tracks with our time domain datapoints. The time domain view is constructed similarly to a waterfall, using a dual port BRAM. The writing side of the BRAM takes 1024 ALE data points, downsampled by 20 times to show 2 seconds worth of time domain data. On the read side, we take a new signed 16 bit data point at every new hcount value which corresponds to a vcount coordinate for the given sample value to plot as red on the time domain graph.

VII. Video Graphics Array Display (Maggie)

The Video Graphics Array (VGA) has three main functionalities controlled by on board switches—displaying the frequency waterfall, showing a rotating or static frequency waterfall, displaying the time domain, and pausing the data readout.

A. Waterfall

For the waterfall display, after determining the colors for each averaged magnitude, we need to efficiently store these values in a rolling buffer for the heart and lung waterfall that dynamically update with each new FFT transformation. The waterfall module takes advantage of the two port BRAM to allow for simultaneous reads and writes. Using a BRAM of depth 256 (total number of averaged samples per FFT) x 10 (total seconds of recordings) x 10 (recordings per second) with width 8 bits, we write and read the palette colors. While we are writing new color values for outputs as they roll off the mag_to_color module, the more frequently updating hcount and vcount for the VGA will be reading off the stored color values.

On the writing side, we are storing 128 cycles of 256 values. Each cycle refers to a completed output from the FFT and the 256 values are the averaged frequencies. To index into the waterfall buffer for writing, we increment a addr_count by 1 with each new output from mag_to_color and store this value at the index. After reaching 128 cycles of FFT outputs, at our final color_last_heart, we reset addr_count to 0.

Because we are reading into the buffer linearly going straight from output into BRAM, we do all the index manipulations from the read side. To calculate the actual index that we need to extract from the BRAM, it's not as straightforward as simply computing an index from vcount and hcount. The diagram below highlights the conversion into index.

The process of index conversion consists of a few steps. The first is realizing that because there are only 128 values on the x axis, but we have a screen width of 1024, every 8 adjacent pixels must be copies of each other. So, the hcount when indexing is right bit shifted by 5 to account for the 32 copies. Afterward, we need to flip the count values to start at 255, the top of the frequency spectrum. Performing this is easy by just using 255−vcount. Finally, we account for the jumps in 256 index values on the hcount side, creating a computed index value of the following.

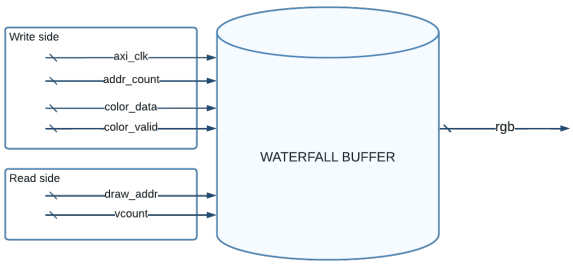

Fig. 1. Waterfall BRAM filling

$$\texttt{draw\_addr} \leftarrow (\texttt{hcount} \ll 5) + (255 - \texttt{vcount});$$

B. Rotating Waterfall

In addition to the sliding view of the waterfall that overrides old data, we have a second waterfall display that constantly feeds in new data from the left side of the monitor, showing a rotating display that dynamically shifts as we read in new inputs. To achieve the rotating effect, we utilize the sample_counter variable which keeps track of which sample is being written into the BRAM. From there, our drawing index will start at the location of the most up to date sample_counter rather than the start of the BRAM. The BRAM draw addressing is determined as follows.

$$(\texttt{sample\_counter} - (\texttt{hcount} \gg 3)) \ll 8 + (255 - \texttt{vcount})$$

C. Time Domain

Centered at 380 pixels down the monitor, the spread of the time domain coordinate across the center is plotted as discrete red lines, creating a continuous time domain display. For the coordinate_out data from the time domain BRAM, we determine which vcount values must be red to capture the size of the ALE output. All vcount values that fall between the outputted ALE data per sample hcount, we set the color to red, as represented by 1100_0000. The formula to determine the actual displayed RGB is as follows.

$$\texttt{rgb} = |\texttt{vcount}| < (\texttt{coordinate\_out} + 380) ? 1100\_0000 : 0;$$

D. Pausing

Switch one is used to pause the data read in and view the existing displays. When the display is paused, no more of the read in data is inserted into the waterfall or time domain BRAMS, creating to a static view that can be used for further analysis of the heartbeats. When un-paused, the data continues to be read in where the previous updates was stopped.

VIII. Audio Output (Sophie)

To keep the basic functionality of a stethescope, we added an audio output feature that allows the user to hear the heart, lung, and unfiltered audio recorded by the microphone.

A. Sound Selection

The user can control which audio data to output via the switches on the FPGA board. Keeping switches 11 and 12 low outputs the non-ale filtered data, switch 11 outputs lung audio, and switch 12 outputs heart audio.

B. Upsampling and Anti-Alias Reconstruction Filter

After the desired audio is selected, the data must be upsampled back to the original 40 ksps sampling rate, from its current 10 ksps sample rate. A zeroth order hold method is implemented in the upsampling module to read in a new sample every single time one is available, and output it 4 times in a row, before the next new sample is available. Next, the data is sent to the low-pass FIR filter again, which during upsampling, acts as a reconstruction filter.

To ensure that we still enable enough time for the sequential 31 tap filter operations in the following low-pass FIR filter, we make sure to output the data from the upsampling module with enough time between outputs. In our case, we waited 300 cycles of the 65 MHz `pixel_clk` between each `valid_out`.

C. Volume Control and PWM

Switches 13-15 enable the user to control the volume. When `sw[15:13]` is `3'b111`, volume is maximized and when it is `3'b000` volume is minimized. Next, the data is sent to a pulse width modulator module which converts the 2's complement signed data back to offset binary form and generates a PWM signal to drive the amplifier.

IX. Results

A. ALE Performance

The ALE algorithm is able to isolate the heartbeat audio and suppress the other noise/lung audio (Figure 2). The top spectrogram, which plots the output of the ALE algorithm, contains the heartbeats (the periodic blue amplitudes) and significantly less high frequency noise (green). For the derived lung signal (computed as the original signal minus the ALE's output signal), while we are able to suppress the heartbeat audio to some extent, unfortunately the heartbeat is still present (Figure 2).

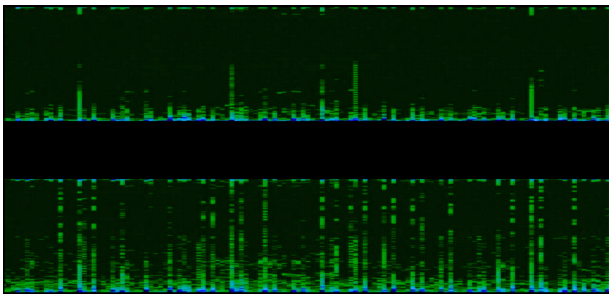

Fig. 2. Heart ALE filters out noise to isolate heartbeats (top) from lung sounds (below)

B. Memory Utilization

As the stealthoscope is constantly visualizing data on the screen as it takes in new audio, we were careful to minimize the amount of memory utilized to create the display. Overall in the whole stealthoscope module, we have 4 BRAMS—one for storing a set color palette, two for each of the heart and lung waterfalls, and one to display time domain data.

Overall, we utilized only 18.15% of the available FPGA BRAM. The palette BRAM utilizes 128 bits, the lung and heart waterfall each use 262 kbits, and the time domain data uses 16 kbits.

By selecting from a small color palette of eight bits and storing only small segments of input data, we leave room for future functionalities and potential storage of data without sacrificing the quality of the visual display.

C. Timing

In the stealthoscope project, we had to work around many different clocks and sampling rates to best capture the beats of the heart. Initially, we had negative build slack due to some complex computations between large positive and negative numbers occurring in both the ALE and Antialias. After maximally pipelining the operations and still being unable to meet timing, we realized that we could instead run the whole system on a slower clock. Because we had already slowed down the clock so much to sample at 40 ksps on the ALE, there was no legitimate need to run on a fast 100 kHz `axi_clk`. We ultimately standardized all of our components to use a 65 mHz `pixel_clock` which matched the clk used by the VGA display to avoid clock domain crossing and meet timing.

X. Takeaways and Next Steps

One potential area of improvement is in the resource utilization. We did not have substantial time to optimize this aspect of our project, but to make our stethoscope viable, we should try to minimize the resources necessary (logical units, DSP blocks, etc.) while still maintaining functionality.

In addition, the audio output would be another area of improvement that we would like to optimize. A significant amount of gain was added to the system via the circuitry to ensure enough signal to visualize the data. However, it is likely that this gain introduced a lot of noise into the system, therefore causing a bunch of static in the audio output. The next step would be to figure out a method to either amplify the data internally or find ways to filter out the noise in audio output.

Another interesting add one feature to the display would be automatic algorithms to zoom into heart beats and provide suggestions on whether or not any irregularities exist. This would likely require training some machine learning models, but would be useful in ultimately becoming a better tool for diagnoses.

References

[1] C. Chao, N. Maneetien, C. Wang, and J. Chiou, *Performance Evaluation of Heart Sound Cancellation in FPGA Hardware Implementation for Electronic Stethoscope*, https://www.hindawi.com/journals/tswj/2014/587238/.

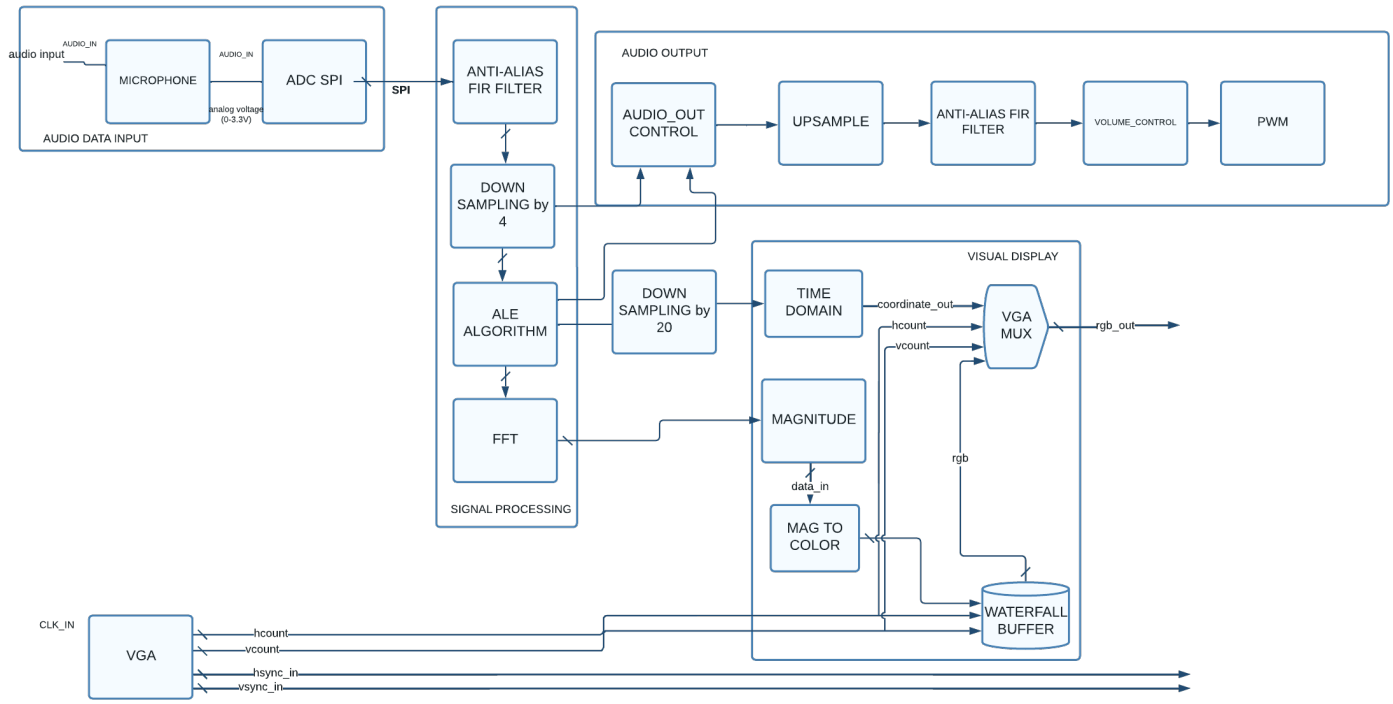

Fig. 3. Block diagram of entire system