

Pitch Perfect: A Hardware-Accelerated Real-Time Phase Vocoder for Pitch Scaling

Embedded Systems Design (CSEE4840)
Spring 2024

Sanjay Rajasekharan (sr3764), Maria Rice (mhr2154), Steven Winnick (shw2139)

Abstract

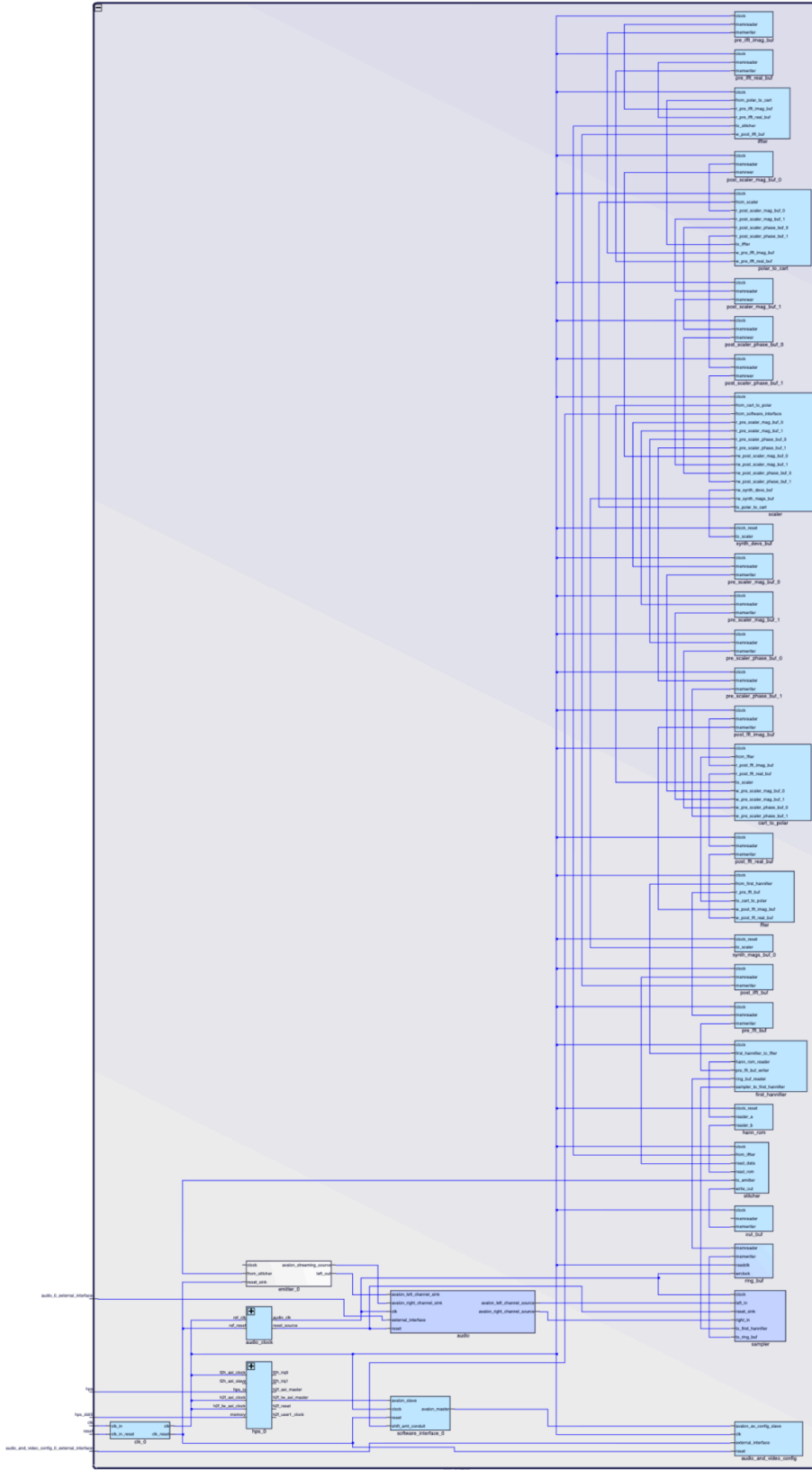
Our project aims to build a phase vocoder for real time pitch scaling primarily through hardware. We utilize an FFT block from the Intel IP Core to perform a Short Term Fourier Transform on the input audio stream. Combined with a series of other hardware-based transformations, further described below, the input audio stream will be pitched up or pitched down by a configurable amount.

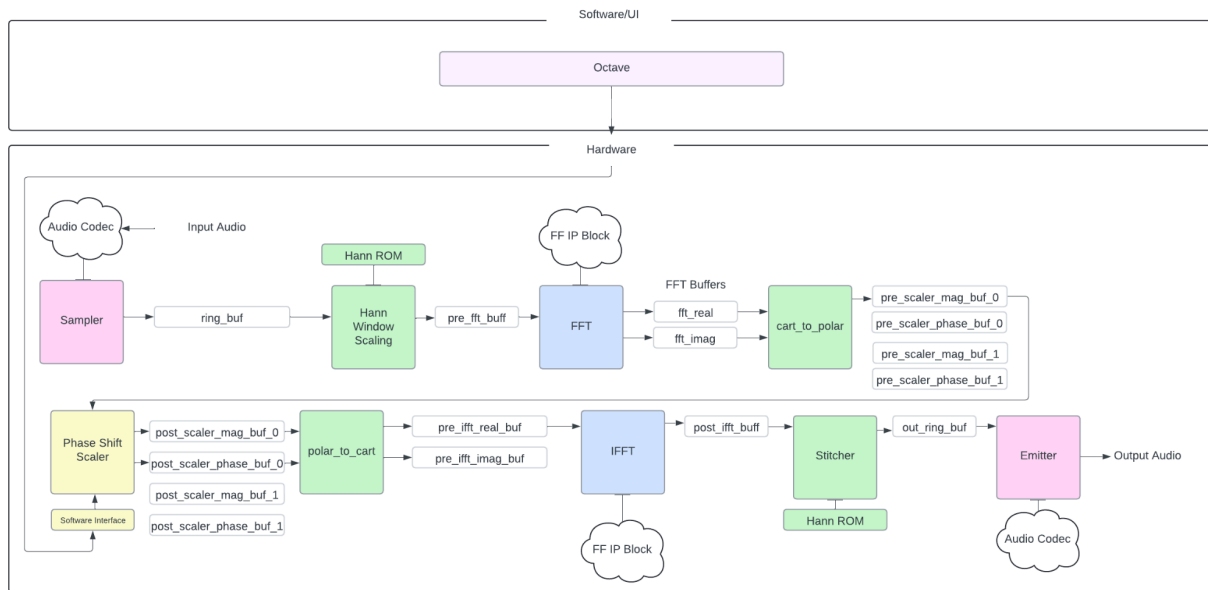
Revised Proposal

Our original inspiration for this project was to create a karaoke companion that allows users to hit high notes through live pitch shifting. Once we more deeply understood the complexities of creating a real-time phase vocoder, our focus shifted to doing just so, removing emphasis from both audio input and output.

In our final output, each of the components of the pitch shifter have been written and compiled to hardware, with the intention to test and execute the linked system on the DE1-SOC board. However, due to a variety of constraints including significant time spent debugging, the system does not work as a whole.

System Block Diagram





Algorithm Description: The Phase Vocoder Algorithm for Pitch Scaling

Input

The algorithm takes as input a stream of audio. In our implementation, these samples are signed 16-bit integer values at a sample rate of 48kHz.

Windowing

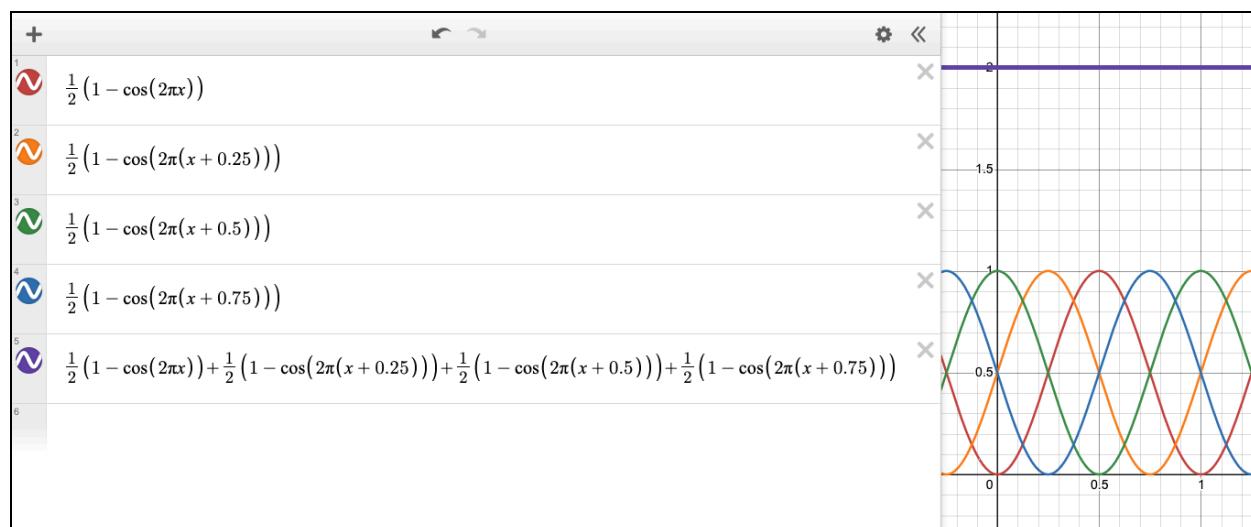
The input audio stream is then split into overlapping “windows,” or groups of samples. Each window contains 4096 samples, and is offset by a “hop length” of 1024 samples from the previous window.

First Hann Window Scaling

The samples in each window are then scaled according to the Hann Function, so the zero-indexed n^{th} sample in a window will be scaled by a factor of $\sin^2(2\pi n/4095)$. This does not cause a loss of information about the original audio stream. Because each window is offset by $\frac{1}{4}$ of the window length and for all n ,

$$\sin^2(2\pi n) + \sin^2(2\pi(n + 0.25)) + \sin^2(2\pi(n + 0.5)) + \sin^2(2\pi(n + 0.75)) = 2$$

we are able to reconstruct the original audio by adding half of each sample’s windowed value for all 4 windows it appears in.



Short-Time Fourier Transform

A Fourier Transform is then applied to each scaled window. In our implementation, we use the polar coordinate representation of the complex values for more space-efficient computation in hardware.

Phase Shift Scaling

This is the point in the algorithm where the pitch scaling actually occurs. For each frequency bin in a given window, we compare the difference in phase between the current and previous windows to the difference we'd expect from a pure tone (sine wave) at that bin's center frequency of n cycles per window, which gives it an expected phase difference of

$$2\pi n / (1024/4096) = \pi n / 2$$

The gap between the expected and observed phase differences, wrapped to a value between $-\pi$ and π , gives us a fractional "bin deviation" indicating how high up or down the real pitch captured by the bin was relative to the bin's center. We then scale the sum of this value and n by a constant factor, the pitch scaling amount, to determine the output "synthesis" bin for this sound. The amount that this differs from the whole-number bin number it is rounded to again gives us a bin deviation, but this time for the synthesized output bins. When multiple input bins end up scaled to the same output bin, we sum their bin deviation contributions. In a process that exactly mirrors the one on the input side, we compute a "phase remainder" using the fractional bin deviation, which tells us how much the actual phase difference of the bin from the previous window to the current one ought to differ from our expectation based on the bin's center frequency. We then add this, the expected phase difference, and the phase of the same bin in the prior window to get the new phase for the bin. For the magnitude, we use the sum of all the input bin magnitudes that ended up in the same synthesis bin.

Inverse Fourier Transform

After being converted back from polar to imaginary values, an inverse Fourier Transform will be done to each window to return it to an array of samples.

Second Hann Window Scaling

To minimize artifacts in the synthesized sound, we repeat the same Hann Function scaling a second time. This allows for a smoother blend between windows whose time-domain waveforms may now be discontinuous from one another.

Window Stitching

The de-transformed phase-adjusted windows of samples are then stitched back together into a main audio stream by adding half of each sample's windowed value for all 4 windows it appears in.

Output

The algorithm outputs a stream of audio. Like the input, in our implementation these will be signed 16-bit integer values at a sample rate of 48kHz.

Preliminary Software Implementation and Simulation

Python File Converters

Utilizing another

Python Algorithm Simulation

Initially, we attempted to implement our pitch scaler in C without any code reference that implemented our exact pitch-scaling algorithm. We had reference code for a [similar algorithm](#) we found from a [YouTube video](#) that claimed to work with pitch scaling but only included code for time stretching. We attempted to implement this in C, adapting the code based on the logic in the video to perform pitch scaling instead of time stretching, but despite much time debugging, couldn't get it to work.

We then found another [video](#) on a related algorithm from a course at Queen Mary University of London, with [skeleton code](#) from a corresponding homework assignment to implement the algorithm in C++. This time, we decided to attempt the implementation in Python first, allowing us to limit variables relating to reading an input file and performing the Fourier Transform, which we were still in the process of testing in C, using SciPy and Librosa. We also used NumPy to abstract away details of the matrix operations done in the pitch scaling algorithm. Having this more abstract Python simulation allowed us to easily fine-tune details of our algorithm, such as the window size and hop length, before implementing them in C.

C Real-Time Streaming Algorithm

The next step was to create a slightly modified version of our outer-level C program which could be used to test our algorithm in “real time,” allowing it to process a stream of input audio of any length and output a stream of pitch-scaled audio as it ran. Our simulation worked by reading samples from the standard input and emitting the scaled stream to the standard output, which would allow it to connect to programs to stream live audio to it and playback its output as live audio. For our testing, however, we would simply pipe in a long sequence of samples from a file and output them to another file, which we would then convert to a listenable format.

C Fourier Transform

Our Fourier Transform functions in C were created based off of various implementations found online. The transform consists of 3 main functions:

```
void rearrange(float real[], float imaginary[], const unsigned int N);
```

This function is crucial for preparing data before applying the FFT algorithm. Its purpose is to reorder the real and imaginary parts of the complex numbers so that the FFT computation may be applied more efficiently. It does so using a bit-reversal method, reordering the elements of an array so that their indices are reversed in binary representation.

```
void compute(float real[], float imaginary[], const unsigned int N);
```

This function performs the core FFT computation on the inputted complex numbers. It iteratively computes the FFT by dividing the input data into smaller groups and applying twiddle factors to combine them, following the Cooley-Tukey FFT algorithm. [The Cooley-Tukey FFT algorithm](#) is a commonly used method which breaks down FFT computation into smaller subproblems, applying a divide-and-conquer approach, to efficiently compute the FFT.

```
void inverseCompute(float real[], float imaginary[], const unsigned int N);
```

This function computes the inverse FFT of the given complex numbers. It first conjugates the data before calling both `rearrange()` and `compute()` in order to perform a forward FFT on the data. Lastly, it conjugates the results again to receive the inverse transform before scaling the result by $1/N$ in order to get the correct inverse FFT output.

Both the forward and inverse FFT were tested on a variety of data points, including a window size of 4096 which is its ultimate application within the phase vocoder algorithm.

Shell Script

A simple shell script that creates an end to end pipeline for the software simulation. Sets up a python environment with the necessary requirements for WAV file processing. It then runs the

python converter script and stores the WAV samples in a temporary file. It then iteratively pipes these samples into the C vocoder algorithm, storing this next step of the output in a new temporary file. Finally, the TXT to WAV python script is run on the output of the vocoder, generating a pitched audio file while removing all intermediate files.

Hardware Implementation

Audio Interface

Pitch Perfect is intended to work using the Line In and Line Out ports on the Cyclone V DE1-SOC board. To configure the board to allow us to interface with these ports in a way that made sense for our project, we set up the following components:

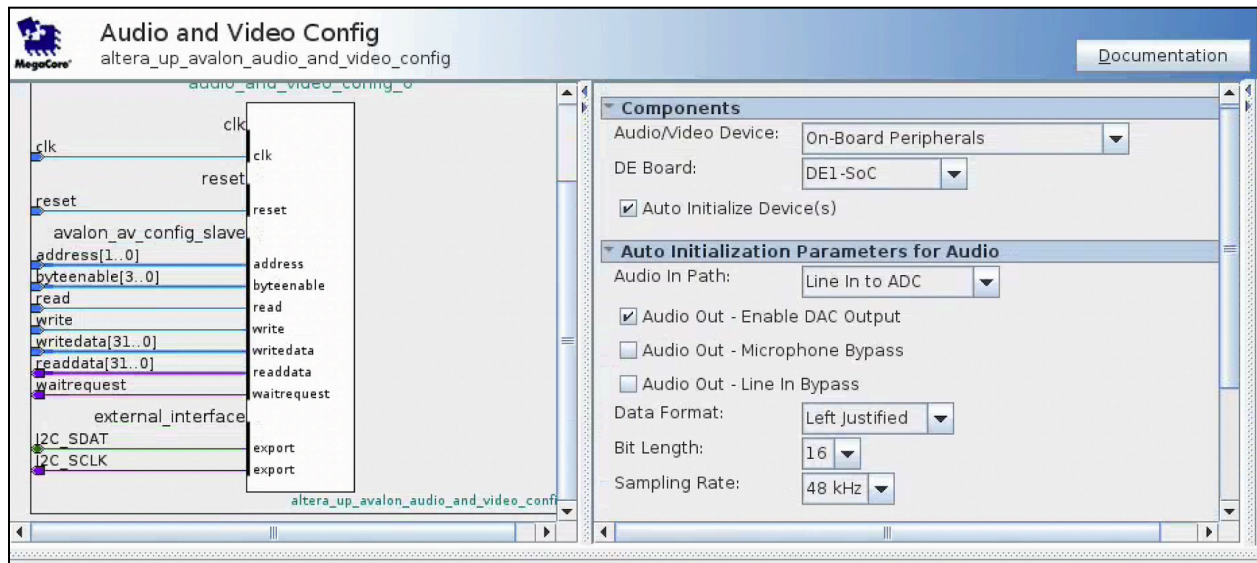
1. Audio Clock for DE-series Boards

Configured to a frequency of 12.288 MHz to work with a 48kHz sampling rate of both the ADC and DAC according to the following specification from page 39 of the [Wolfson WM8731/WM8731L CODEC manual](#) to achieve a Base Oversampling Rate of 0. Output `clk_out` serves as the clock input for the audio core and sampler components, described below.

SAMPLING RATE		MCLK FREQUENCY	SAMPLE RATE REGISTER SETTINGS					DIGITAL FILTER TYPE
ADC	DAC		BOSR	SR3	SR2	SR1	SR0	
kHz	kHz	MHz						
48	48	12.288	0 (256fs)	0	0	0	0	1
		18.432	1 (384fs)	0	0	0	0	
48	8	12.288	0 (256fs)	0	0	0	1	1
		18.432	1 (384fs)	0	0	0	1	
8	48	12.288	0 (256fs)	0	0	1	0	1
		18.432	1 (384fs)	0	0	1	0	
8	8	12.288	0 (256fs)	0	0	1	1	1
		18.432	1 (384fs)	0	0	1	1	
32	32	12.288	0 (256fs)	0	1	1	0	1
		18.432	1 (384fs)	0	1	1	0	
96	96	12.288	0 (128fs)	0	1	1	1	2
		18.432	1 (192fs)	0	1	1	1	
44.1	44.1	11.2896	0 (256fs)	1	0	0	0	1
		16.9344	1 (384fs)	1	0	0	0	
44.1 (Note 1)	8	11.2896	0 (256fs)	1	0	0	1	1
		16.9344	1 (384fs)	1	0	0	1	
8 (Note 1)	44.1	11.2896	0 (256fs)	1	0	1	0	1
		16.9344	1 (384fs)	1	0	1	0	
8 (Note 1)	8 (Note 1)	11.2896	0 (256fs)	1	0	1	1	1
		16.9344	1 (384fs)	1	0	1	1	
88.2	88.2	11.2896	0 (128fs)	1	1	1	1	2
		16.9344	1 (192fs)	1	1	1	1	

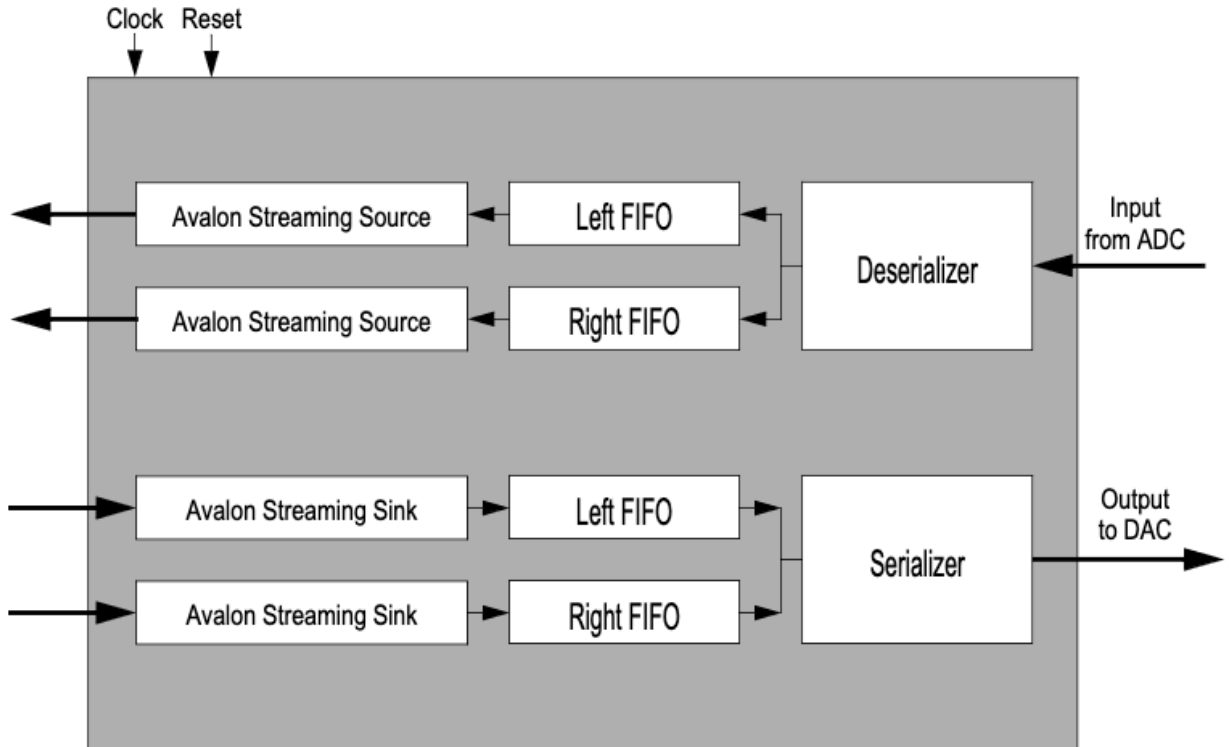
2. Audio and Video Config Core

Provides a way to initialize and reconfigure the Wolfson Audio CODEC. It communicates with the CODEC through the 2-wire I2C serial bus via its I2C_SDAT and I2C_SCLK external interface, which connect to the board's FPGA_I2C_SDAT and FPGA_I2C_SCLK wires, respectively. It has been configured to auto-initialize the CODEC to use the Line In to ADC audio input path, the DAC to Line Out audio output path, using 16-bit left-justified samples at a sampling rate of 48kHz. Its Avalon Slave config interface is connected to the software_interface component, but unused, as we don't want to allow the core to be reconfigured after its initial configuration.



3. Audio Core

Provides an interface for audio input/output to the FPGA's Audio CODEC. Configured in "streaming mode" to allow us to stream values from other hardware components, and otherwise auto-initialized by the Audio and Video Config Core.



software_interface

Communicates with software via the Avalon Bus, connected to the processor's

`h2f_lw_axi_master` output, to receive an 8-bit fixed-point "shift amount" with 6 fractional bits, then indicates to `scaler` how much to scale the pitch by.

```
// Input from the Avalon Bus
input logic [7:0]   writedata,
input logic        write,
input             chipselect,
input logic [2:0]  address,

// Tell the rest of the system how much to shift by (controlled by software)
output logic [7:0] shift_amt = 0,

// Fill the AV Config component's avalon slave module to quiet warnings
output logic [1:0]  av_config_slave_address = 0,
output logic [3:0]  av_config_slave_byteenable = 0,
output logic        av_config_slave_read = 0,
output logic        av_config_slave_write = 0,
output logic [31:0] av_config_slave_writedata = 0,
input logic [31:0]  av_config_slave_readdata,
```

```
input logic          av_config_slave_waitrequest
```

sampler

Avalon streaming component to read samples of left `from_adc` channel of the Wolfson Audio CODEC to a ring buffer. Connects to the `avalon_left_channel_source` and `avalon_right_channel_source` of the Audio Core according to the [Avalon Streaming Sink specification](#), flashing the `in_ready` inputs for one clock cycle and expecting the `in_valid` and `in_data` values to be set accordingly on the next cycle. Every 1024 samples read, a window is ready to be processed by the rest of the pitch scaling algorithm, so the `go_out` wire is flashed high for one cycle to indicate to `first_hannifier` that it should begin processing samples from the `ring_buf`, as will happen for each of the following components. The 3-bit `window_start` output indicates to `hann_buf` whether the current window starts at index 0, 1024, 2048, 3072, or 4096 of `ring_buf`. The ring buffer, `ring_buf`, is a 2-port RAM component of M10K memory holding $4096 + 1024 = 5120$ 16-bit samples, with 2 clocks, one for a write-only input from sampler at the audio clock speed, and one for a read-only output to `first_hannifier` at the main FPGA clock speed of 50MHz.

```
// Read from avalon_left_channel_source from audio codec
input logic [15:0] left_in_data,
input logic      left_in_valid,
output logic     left_in_ready = 0,

// Read from avalon_right_channel_source from audio codec
input logic [15:0] right_in_data,
input logic      right_in_valid,
output logic     right_in_ready = 0,

// Write to ring buffer
output logic [15:0] ring_buf_data,
output logic [12:0] ring_buf_addr = 0,
output logic      ring_buf_wren = 0,

// Communicate with first_hannifier
output logic [2:0] window_start,
output logic      go_out = 0
```

first_hannifier

Applies the Hann Windowing function to a window of samples from `ring_buf`. All 4096 Hann Window scaling values have been stored in ROM as 16-bit fixed-point values with all 16 bits fractional. Hann Windowed values are written to `pre_fft_buf`, a 2-port single-clock RAM

component of M10K memory holding 4096 16-bit fixed-point words with 8 fractional bits, as are all following RAM blocks throughout the implementation.

```
// Communicate with sampler
input logic [1:0] window_start,
input logic      go_in,

// Read from ring buffer
input logic [15:0] ring_buf_data,
output logic [12:0] ring_buf_addr = 0,

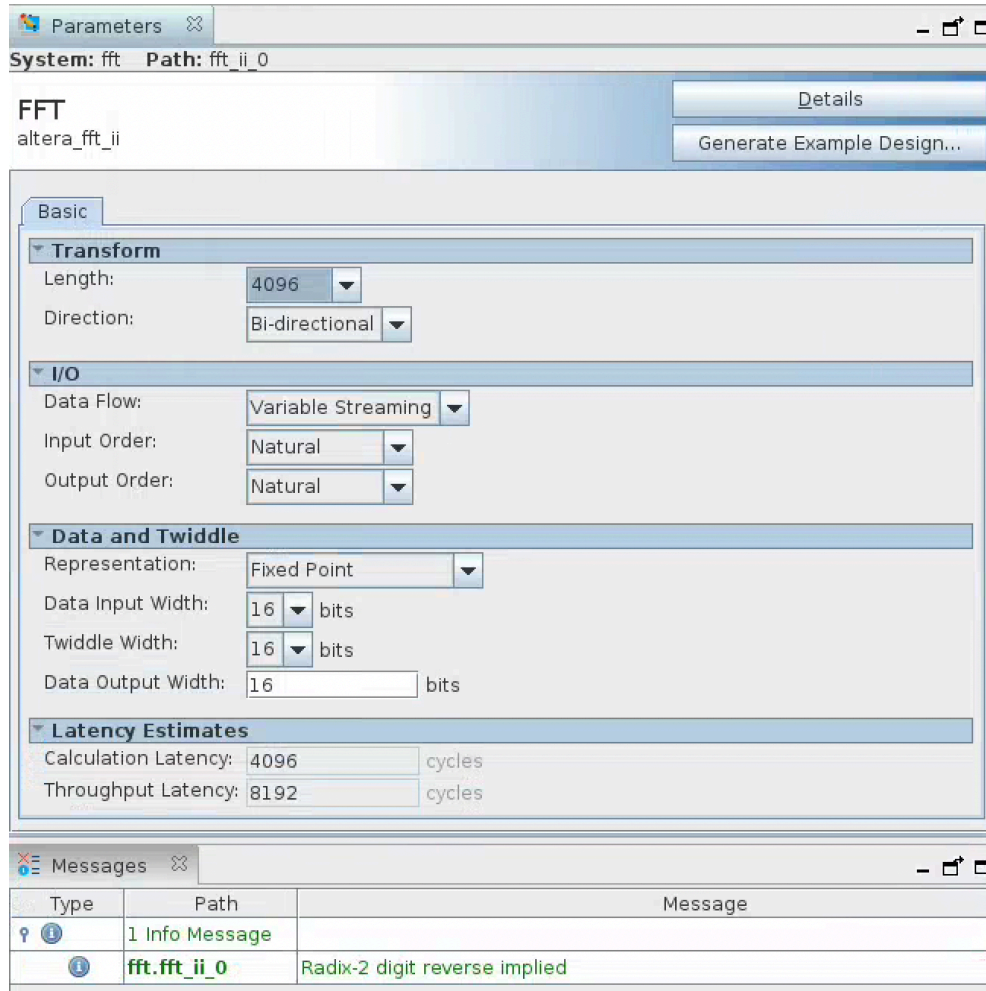
// Read from Hann Window ROM
input logic [15:0] hann_rom_data,
output logic [11:0] hann_rom_addr = 0,

// Write to pre-FFT buffer
output logic [15:0] out_buf_data = 0,
output logic [11:0] out_buf_addr = 0,
output logic      out_buf_wren = 0,

// Communicate with ffter
output logic      go_out = 0
```

FFT-er

At the core of our implementation are two Fourier Transform hardware blocks. For these, we've decided to use Intel's FFT Engine rather than attempting to implement our own Fast Fourier Transform module, as we have faith that the good people at Intel were able to implement an efficient FFT module. Both our FFT and IFFT IPs were created with the following parameters:



Window length = 4096 samples (~ 92 ms)

Hop length = 1024 samples (~ 23 ms)

The window length was selected to be 4096 based on the convention of using this window length in audio processing. The hop length was calculated to be 1024 based on testing the C-version of our implementation.

To initially test the FFT module, we created the IP as outlined above and generated an example file which we then analyzed using QuestaSim.

After doing so, we focused on creating a more developed wrapper module to further test and understand the functionalities of the FFT IP. This consisted of the following three files, and was developed with help from [this video](#), with modifications made to the specifications of our project:

`fft_wrapper.v`

This module serves as a wrapper around the FFT core. It handles input/output signals, controls, and instantiates the FFT module. It also manages signal synchronization and data flow between the FFT core and external components.

`control_for_fft.v`

This module generates control signals required for the FFT computation. It manages signals such as validity, start of packet (SOP), end of packet (EOP), and error indications. It also handles configuration options like inverse FFT and FFT point size.

`testbench.v`

This testbench is used for verifying the functionality of the FFT wrapper and control modules. It generates clock signals, drives input signals to the FFT wrapper, and monitors the output signals. It instantiates the FFT wrapper and the NCO IP to provide input to validate its behavior under different conditions.

Although these files were successfully compiled in Quartus, we faced FFT core errors when attempting to simulate them in QuestaSim. Therefore, we moved on to creating a wrapper that is focused on interfacing the FFT IP within the necessary functionalities of the phase vocoder.

[cart_to_polar](#)

Converts imaginary values to polar coordinates. Converted values are written to `pre_scaler_mag_buf_0` and `pre_scaler_phase_buf_0` or `pre_scaler_mag_buf_1` and `pre_scaler_phase_buf_1`, alternating each cycle. When a window is completed, `cur_buf` indicates to `scaler` whether the current set of values were written to buffers 0 or 1. This module utilizes an open source CORDIC module. CORDIC works as an approximation for trigonometric functions by iteratively rotating the input point by a predetermined amount that provides a convenient update rule at each step, only requiring addition and bit shifting. The update rule is as follows:

$$X_{(k+1)} = X_k - Y \ll k$$

$$Y_{(k+1)} = X_k + Y \ll k$$

Once the point is shifted to a 0 degree angle, it is clear what the angle and magnitude of the original point is. As the value you left on `x_k = magnitude * scale_factor` and `angle = sum(r_i)` from 0 to k, where `r_i` represents the rotation factor.

```
// Communicate with ffter
input logic      go_in,
```

```

// Read from post_fft_buf_real
input logic [15:0]  real_buf_data,
output logic [12:0] real_buf_addr = 0,

// Read from post_fft_buf_imag
input logic [15:0]  imag_buf_data,
output logic [12:0] imag_buf_addr = 0,

// Write to scaler_mag_buf_0
output logic [15:0] mag_buf_0_data = 0,
output logic [11:0] mag_buf_0_addr = 0,
output logic          mag_buf_0_wren = 0,

// Write to scaler_phase_buf_0
output logic [15:0] phase_buf_0_data = 0,
output logic [11:0] phase_buf_0_addr = 0,
output logic          phase_buf_0_wren = 0,

// Write to scaler_mag_buf_1
output logic [15:0] mag_buf_1_data = 0,
output logic [11:0] mag_buf_1_addr = 0,
output logic          mag_buf_1_wren = 0,

// Write to scaler_phase_buf_1
output logic [15:0] phase_buf_1_data = 0,
output logic [11:0] phase_buf_1_addr = 0,
output logic          phase_buf_1_wren = 0,

// Communicate with scaler
output logic          cur_buf = 0,
output logic          go_out = 0

```

scaler

Performs the pitch scaling on polar values of FFT bins according to the algorithm described earlier. Reads `scale_amt` from `software_interface` to know how much to scale by. Requires both write and read access to output `post_scaler` buffers to read values from previous cycles for comparison (see algorithm description section above). Also utilizes its own `synth_mags` and `synth_devs` buffers to hold intermediate values from the computations for output frequency bins. When a window is completed, `cur_buf` indicates to `polar_to_cart` whether the current set of values were written to buffers 0 or 1.

```
// Communicate with cart_to_polar
input logic      go_in,
input logic      cur_window,

// Communicate with software_interface
input logic [15:0] scale_amt,

// Read from pre_scaler_mag_buf_0
input logic [15:0] mag_in_buf_0_data,
output logic [11:0] mag_in_buf_0_addr = 0,

// Read from pre_scaler_phase_buf_0
input logic [15:0] phase_in_buf_0_data,
output logic [11:0] phase_in_buf_0_addr = 0,

// Read from pre_scaler_mag_buf_1
input logic [15:0] mag_in_buf_1_data,
output logic [11:0] mag_in_buf_1_addr = 0,

// Read from pre_scaler_phase_buf_1
input logic [15:0] phase_in_buf_1_data,
output logic [11:0] phase_in_buf_1_addr = 0,

// Read from and write to post_scaler_mag_buf_0
input logic [15:0] mag_out_buf_0_rdata,
output logic [11:0] mag_out_buf_0_raddr = 0,
output logic [15:0] mag_out_buf_0_wrdata = 0,
output logic [11:0] mag_out_buf_0_wraddr = 0,
output logic      mag_out_buf_0_wren = 0,

// Read from and write to post_scaler_phase_buf_0
input logic [15:0] phase_out_buf_0_rdata,
output logic [11:0] phase_out_buf_0_raddr = 0,
output logic [15:0] phase_out_buf_0_wrdata = 0,
output logic [11:0] phase_out_buf_0_wraddr = 0,
output logic      phase_out_buf_0_wren = 0,

// Read from and write to post_scaler_mag_buf_1
input logic [15:0] mag_out_buf_1_rdata,
output logic [11:0] mag_out_buf_1_raddr = 0,
output logic [15:0] mag_out_buf_1_wrdata = 0,
output logic [11:0] mag_out_buf_1_wraddr = 0,
```

```

output logic    mag_out_buf_1_wren = 0,

// Read from and write to post_scaler_phase_buf_1
input logic [15:0] phase_out_buf_1_rdata,
output logic [11:0] phase_out_buf_1_raddr = 0,
output logic [15:0] phase_out_buf_1_wrdata = 0,
output logic [11:0] phase_out_buf_1_wraddr = 0,
output logic    phase_out_buf_1_wren = 0,

// Read from and write to scaler_synth_mags_buf
input logic [15:0] synth_mags_rdata,
output logic [11:0] synth_mags_raddr = 0,
output logic [15:0] synth_mags_wrdata = 0,
output logic [11:0] synth_mags_wraddr = 0,
output logic    synth_mags_wren = 0,

// Read from and write to scaler_synth_devs_buf
input logic [15:0] synth_devs_rdata,
output logic [11:0] synth_devs_raddr = 0,
output logic [15:0] synth_devs_wrdata = 0,
output logic [11:0] synth_devs_wraddr = 0,
output logic    synth_devs_wren = 0,

// Communicate with polar_to_cart
output logic    cur_buf = 0,
output logic    go_out = 0

```

polar_to_cart

Converts polar coordinates back to imaginary values. Utilizes a CORDIC approach to computing sine and cosine values. Converted values are written to `pre_ifft_real_buf` and `pre_ifft_imag_buf`.

```

// Communicate with scaler
input logic    go_in,
input logic    cur_window,

// Read from scaler_mag_buf_0
input logic [15:0] mag_buf_0_data,
output logic [11:0] mag_buf_0_addr = 0,

// Read from scaler_phase_buf_0
input logic [15:0] phase_buf_0_data,
output logic [11:0] phase_buf_0_addr = 0,

```



```

// Read from scaler_mag_buf_1
input logic [15:0] mag_buf_1_data,
output logic [11:0] mag_buf_1_addr = 0,

// Read from scaler_phase_buf_1
input logic [15:0] phase_buf_1_data,
output logic [11:0] phase_buf_1_addr = 0,

// Write to pre_ifft_real_buf
output logic [15:0] real_buf_data,
output logic [12:0] real_buf_addr = 0,
output logic      real_buf_wren = 0,

// Write to pre_ifft_imag_buf
output logic [15:0] imag_buf_data,
output logic [12:0] imag_buf_addr = 0,
output logic      imag_buf_wren = 0,

// Communicate with iffter
output logic      go_out = 0

```

IFFT-er

The inverse FFT follows the same wrapper module as the FFT, however, the input of “inverse” is set to 1, as seen below:

stitcher

Applies the Hann Windowing function to a window of samples from `post_ifft_buf`. Values are either overwritten or added to those in `stitched_buf`. The 2-bit `window_start` output indicates to emitter whether the current window starts at index 0, 1024, 2048, or 3072.

```

// Communicate with iffter
input logic      go_in,

// Read from post_ifft_buf
input logic [15:0] in_buf_data,
output logic [11:0] in_buf_addr = 0,

// Read from Hann Window ROM
input logic [15:0] hann_rom_data,
output logic [11:0] hann_rom_addr = 0,

```

```

// Write to stitched_buf
output logic [15:0] out_buf_data = 0,
output logic [11:0] out_buf_addr = 0,
output logic          out_buf_wren = 0,

// Communicate with emitter
output logic [1:0] window_start,
output logic          go_out = 0

```

emitter

Avalon streaming component to write samples to both left and right `to_dac` channels of the Wolfson Audio CODEC. Connects to the `avalon_left_channel_sink` and `avalon_right_channel_sink` of the Audio Core according to the [Avalon Streaming Sink specification](#), flashing the `out_valid` for one clock cycle with the next sample of data on `out_data` after seeing `out_ready` flashed. Per the Audio Core specification, both `avalon_left_channel_sink` and `avalon_right_channel_sink` need to be written to for the Audio CODEC to not block and the sample to be played.

```

// Communicate with stitcher
input logic [1:0] window_start,
input logic          go_in,

// Write to avalon_left_channel_sink of audio codec
output logic [15:0] left_out_data = 0,
output logic          left_out_valid = 0,
input logic          left_out_ready,

// Write to avalon_right_channel_sink of audio codec
output logic [15:0] right_out_data = 0,
output logic          right_out_valid = 0,
input logic          right_out_ready

```

Resource Budgets: Memory Requirement Estimation

From the table entry above, we see that our chosen FFT engine uses 14 of our 397 available M10K memory blocks.

In our implementation, as seen in the system block diagram utilizes 14 intermediary buffers of length 4096. Each sample is 16 bits.

Additionally, we utilize two ring buffers which contain 5120 16-bit samples

Finally, we utilize one 4096 16-bit ROM.

The memory consumption of our buffers is thus 112 M10K Blocks.

The FFT Blocks utilize the following amounts of memory based on our specifications.

Cyclone V	Variable Streaming Floating Point	4,096	—	13,945	60	138	--	22,615	701	132
-----------	-----------------------------------	-------	---	--------	----	-----	----	--------	-----	-----

Because we utilize two FFT Blocks, one for the forward and reverse processes we utilize a total of 279 M10K Blocks.

Together with the buffers and ROMs the total memory consumption of this project is 391 M10K Blocks.

Who Did What

Maria

- Researched pitch shifting algorithms
- Researched various implementations of the fast fourier transform for pitch shifting
- Implemented the Cooley-Tukey FFT algorithm in C
 - Tested with a variety of inputs, including applying FFT and IFFT on a 4096 size window to ensure that the final output was equivalent to the input
- Helped with debugging the C vocoder algorithm
- Created converters in python for testing purposes
- Researched the FFT intel core module
 - Identified the best options of the module to use within our phase vocoder
 - Tested a generated example using QuestaSim
- Created a FFT wrapper and testbench for testing purposes
 - Compiled the files successfully in Quartus
 - Ran into errors with protected files while compiling in Questasim
- Implemented `fft` and `ifft` hardware components

Sanjay

- Researched pitch shifting algorithms
- Wrote C algorithm to simulate input buffer, Hann window scaling, and audio outputting
- Wrote bash script to integrate Python wav convertors and C simulation

- Debugged and contributed to C vocoder algorithm to remove noise
- Converted C vocoder algorithm to operate on polar coordinates instead from cartesian
- Attempted to set up ROM components for Hann Window coefficients and CORDIC polar conversion
- Researched and implemented fixed point calculations, computing necessary coefficients and writing code for computing moduli
- Developed polar conversion hardware components
- Contributed to scaler hardware component

Steven

- Researched various pitch scaling algorithms
- Implemented two different pitch scaling algorithms in Python
- Implemented C vocoder algorithm
- Created real-time C implementation wrapper
- Created file type converters in Python
- Attempted to configure and connect to DE1-SOC Audio module
- Configured top level hardware system in Platform Designer
- Designed interfaces of all hardware components
- Created skeletons of all hardware components with necessary I/O to connect in Platform Designer
- Implemented `sampler`, `first_hannifier`, and `emitter` hardware components, and contributed to `scaler`

Lessons Learned

How to Perform Fixed Point Computation

In this project, we learned about fixed point representation, something none of us had heard of before. We utilized fixed point representation throughout our project such as in our FFT implementation as well as our polar/cartesian convertors.

Hardware Function Approximations and “Cutting Corners”

In this project, we learned about fixed point representation, something none of us had heard of before. We utilized fixed point representation throughout our project such as in our FFT implementation as well as our polar/cartesian convertors. One of the biggest lessons of this project was learning how difficult it is to compute certain functions that we often take for granted in hardware. Hey to our algorithm were functions such as `fmodf`, `atan2`, and sine and cosine. We learned about approximation methods for computing trigonometric functions. We utilized the CORDIC algorithm for this. Additionally, we were able to factor out `fmodf` by carefully inspecting the data that ran through our algorithm. We noticed that the values were bounded. Thus, we could

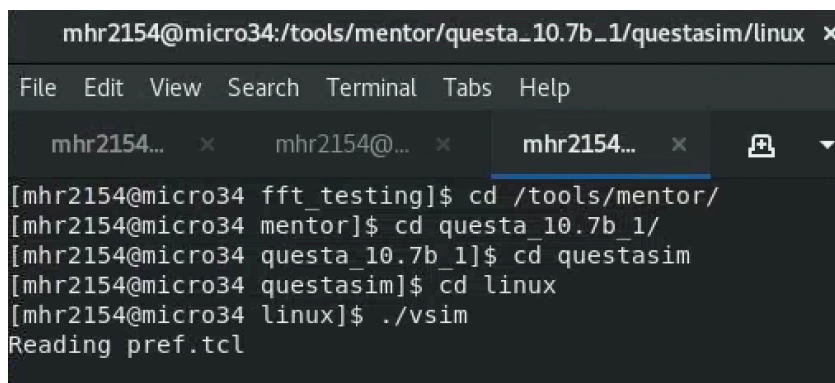
avoid utilizing a costly fixed point mod algorithm, achieving the same effect through addition and division.

General Lessons

- Simulate even the most simple algorithms with low-level C software to confirm that they work before building them in hardware
- Make your software simulation match the hardware as closely as possible - it is easier to think about things in terms of software first
- Give yourself more time to implement hardware
- It takes a serious amount of time to figure out how to interface with the board

How to open up Questasim

Although a very trivial part of a much more complicated project, we ran into errors with a broken pathway for “vsim” and had to dig into many different folders to find the correct pathway and open up Questasim for testing. Additionally, we learned that you can directly link Questasim to Quartus through settings, so that when you run “RTL simulation” it opens up and simulates the top-level module in Questasim.



```
mhr2154@micro34:/tools/mentor/questa_10.7b_1/questasim/linux x
File Edit View Search Terminal Tabs Help
mhr2154... x mhr2154@... x mhr2154... x
[mhr2154@micro34 fft_testing]$ cd /tools/mentor/
[mhr2154@micro34 mentor]$ cd questa_10.7b_1/
[mhr2154@micro34 questa_10.7b_1]$ cd questasim
[mhr2154@micro34 questasim]$ cd linux
[mhr2154@micro34 linux]$ ./vsim
Reading pref.tcl
```

How to work on hardware components

We found there to be a steep learning curve in learning the various steps needed for working on the hardware implementation of our project. Many specific commands need to be run in a specific order for things to work properly, and we didn't love scrolling through the entire Lab 3 document each time to remember them. To that end, we created the following guide for ourselves to follow to simplify the process, which we are providing here in the hopes that it will be useful for some future group. To those groups however, please note that we did not actually get our project working on the

FPGA as we intended, so this could be completely wrong, and you may want to check with Professor Edwards or a TA before following our instructions.

- To create a new component
 - To define the component's inputs/outputs, copy the sv file for any other component (*basic_component.sv*) and edit the copy to have the inputs/outputs desired
 - To figure out what inputs/outputs are needed, look at what it connects to
 - If it connects to a standard library component, right click that component and click edit, then check "show signals" to see which wires each interface (ex. Avalon Memory Mapped Slave) uses. You will set the interfaces in the next step
 - To add the new component to our system, do
 - *qsys-edit soc_system.qsys*
 - File > New Component
 - Give it a name and display name
 - Files > Synthesis Files > Add File > new_component.sv
 - Click Analyze Synthesis Files to have qsys create a component from the System Verilog file
 - Probably will need to fix errors in the .sv file
 - Set Top-Level Module to this file
 - Signals & Interfaces tab
 - Create interfaces (ex. Avalon Memory Mapped Slave) as needed to set the input/output logic to match those
 - Click *Finish* to create a _hw.tcl file for the new component
 - ONLY if it is a software interfacing component (will also have to edit kernel module so really just stick with one of these) in a separate terminal tab, edit component_name_hw.tcl to add the following under the *module component_name* section
 - set_module_assignment embeddedsw.dts.vendor "csee4840"
 - set_module_assignment embeddedsw.dts.name "full_name"
 - set_module_assignment embeddedsw.dts.group "name"
 - In IP Catalog > Project (to the left), click the new component, then click [+Add...] to add it to our system
- To use pre-made Altera components, you cannot just add them in Platform Designer. You must create files for them using Quartus.
- Right click the various parts of a component to connect them to other parts of other components
- Edit the logic inside a component by editing it's .sv file
- To test on the board
 - Click Generate HDL in the Platform Designer to generate the Verilog for the project

- Run *make quartus* to generate *output_files/soc_system.sof*
- Run *make rbf* to generate *output_files/soc_system.rbf*
- Run *embedded_command_shell.sh*
- Run *make dtb* to generate *soc_system.dtb*
- Plug the board in to power, ethernet, and micro-USB (black cable to the PC)
- Run *screen /dev/ttyUSB0 115200*
- Power on the board
- From the screened board terminal
 - Login with root, CSee4840!
 - Run *ifup eth0* to connect to the internet
 - Run *mount /dev/mmcblk0p1 /mnt*
 - Run *scp*
`<uni>@micro<XX>.ee.columbia.edu:~/pitch-perfect/hardware/hw/output_files/soc_system.rbf /mnt`
 - *scp*
`<uni>@micro<XX>.ee.columbia.edu:~/pitch-perfect/hardware/hw/soc_system.dtb /mnt` ← not in output_files!
 - Run *sync*
 - Run *reboot*

Complete File Listing

Software Simulation Files

software-simulation/python-vocoder/vocoder.py

```
"""
vocoder.py

Pitch scaling algorithm following
https://github.com/BelaPlatform/bela-online-course/blob/master/lectures/lecture-20/code-examples/fft-pitchshift.zip
using Python based on
https://github.com/JentGent/pitch-shift/blob/main/audios.ipynb
"""

import librosa
import numpy as np
import soundfile as sf
```

```

import sys

WINDOW_SIZE = 4096 # win_len, n_fft
HOP_LEN = 1024 # hop_len
PHASE_SHIFT_AMOUNT = 2 ** (5 / 12) # scaling

waveform, samp_rate = librosa.load(sys.argv[1], sr=None, mono=True) #
waveform, sr # used to not be mono
num_samples = waveform.shape[0] # og_len

stft_result = librosa.stft(waveform, n_fft=WINDOW_SIZE,
hop_length=HOP_LEN, win_length=WINDOW_SIZE) # anls_stft
n_fft_bins, n_fft_frames = stft_result.shape # n_anls_freqs,
n_anls_frames, used to also have "channels" count

stft_result = np.transpose(stft_result)

stft_result_scaled = []
prev_anal_phases = np.zeros(n_fft_bins)
prev_synth_phases = np.zeros(n_fft_bins)

for idx, frame in enumerate(stft_result):

    if idx == 0:
        pass

    mags = np.abs(frame)
    phases = np.angle(frame)

    dphases_from_prev = phases - prev_anal_phases

    bin_center_freqs = np.arange(n_fft_bins) * 2 * np.pi / WINDOW_SIZE #
how much each bin should move per sample
    dphases_from_expected = dphases_from_prev - (bin_center_freqs *
HOP_LEN)

```



```

dphases_from_expected = np.mod(dphases_from_expected + (3 * np.pi), 2 *
np.pi) - np.pi # wrap to [-pi, pi]

bin_deviations = (dphases_from_expected * WINDOW_SIZE) / (2 * np.pi *
HOP_LEN) # how many bins we should change (over a whole bin length)
new_bins = (np.arange(n_fft_bins) + bin_deviations) *
PHASE_SHIFT_AMOUNT
new_bin_nums = np rint(new_bins)

synth_mags = np.zeros(n_fft_bins)
synth_deviations = np.zeros(n_fft_bins)
for old_idx, new_bin_num in enumerate(new_bin_nums):
    if new_bin_num >= 0 and new_bin_num < n_fft_bins:
        synth_mags[int(new_bin_num)] += mags[old_idx]
        synth_deviations[int(new_bin_num)] += new_bins[old_idx] -
new_bin_nums[old_idx]

phase_remainders = ((synth_deviations) * 2 * np.pi * HOP_LEN) /
WINDOW_SIZE
synth_phases = prev_synth_phases + phase_remainders + (2 * np.pi *
bin_center_freqs * HOP_LEN)
synth_phases = np.mod(synth_phases + (3 * np.pi), 2 * np.pi) - np.pi #
wrap to [-pi, pi]

stft_result_scaled.append(synth_mags * np.exp(synth_phases * 1j))

prev_anal_phases = phases
prev_synth_phases = synth_phases

stft_result_scaled = np.array(stft_result_scaled)
stft_result_scaled = np.transpose(stft_result_scaled)

new_waveform = librosa.istft(stft_result_scaled, n_fft=WINDOW_SIZE,
hop_length=HOP_LEN, win_length=WINDOW_SIZE)

sf.write(sys.argv[2], new_waveform, samp_rate, 'PCM_24')
```

software-simulation/realtime/main.c

```
// #include <stdlib.h>
// #include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <errno.h>

#include "../fft/fft.h"
#include "../scaling/scaling.h"

#define WINDOW_SIZE 4096
#define HOP_LENGTH 1024
#define PHASE_SHIFT_AMOUNT(x) pow(2.0, (x / 12.0))

void hannify(float* inputSamples, int startIdx, float* output) {
    for (size_t i = 0; i < WINDOW_SIZE; ++i) {
        float w = 0.5 * (1 - cos(2 * M_PI * i / WINDOW_SIZE));
        output[i] = inputSamples[(i + startIdx) % (WINDOW_SIZE +
HOP_LENGTH)] * w;
    }
}

float inputSamples[WINDOW_SIZE + HOP_LENGTH];
int inputWindowStart = 0;
int inputCurIdx = 0;
float hanned1[WINDOW_SIZE];
float hanned2[WINDOW_SIZE];
float *fftRealBufs[2];
float *fftImagBufs[2];
float *shiftRealBufs[2];
float *shiftImagBufs[2];
int fftBufIdx = 0;
float fftReal1[WINDOW_SIZE];
```

```
float fftImag1[WINDOW_SIZE];
float fftReal2[WINDOW_SIZE];
float fftImag2[WINDOW_SIZE];
float shiftReal1[WINDOW_SIZE];
float shiftImag1[WINDOW_SIZE];
float shiftReal2[WINDOW_SIZE];
float shiftImag2[WINDOW_SIZE];
float ifftReal[WINDOW_SIZE];
float ifftImag[WINDOW_SIZE];
float stitcher[WINDOW_SIZE];
int stitcherPtr = 0;
char *curLine;
size_t curLineLen = 0;

int main(int argc, char** argv)
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <semitone-shift> \n", argv[0]);
        return 1;
    }

    int semitoneShift = strtol(argv[1], NULL, 10);

    fftRealBufs[0] = fftReal1;
    fftImagBufs[0] = fftImag1;
    fftRealBufs[1] = fftReal2;
    fftImagBufs[1] = fftImag2;
    shiftRealBufs[0] = shiftReal1;
    shiftImagBufs[0] = shiftImag1;
    shiftRealBufs[1] = shiftReal2;
    shiftImagBufs[1] = shiftImag2;

    for (int i = 0; i < WINDOW_SIZE; i++) {
        fftReal1[i] = 0;
        fftImag1[i] = 0;
    }
}
```

```

    fftReal2[i] = 0;
    fftImag2[i] = 0;
    shiftReal1[i] = 1;
    shiftImag1[i] = 1;
    shiftReal2[i] = 1;
    shiftImag2[i] = 1;
    ifftReal[i] = 0;
    ifftImag[i] = 0;
    stitcher[i] = 0;
}

while (-1 != getline(&curLine, &curLineLen, stdin)) {

    // Convert curLine to float
    inputSamples[inputCurIdx] = strtod(curLine, NULL) * 0.8; // to
prevent clipping

    // Abstraction: assume all processing takes place in span of 1
sample
    if (inputCurIdx == (inputWindowStart + WINDOW_SIZE) %
        (WINDOW_SIZE + HOP_LENGTH)) {

        // Initial Hann Windowing
        hannify(inputSamples, inputWindowStart, hanned1);

        // Perform FFT
        for (int i = 0; i < WINDOW_SIZE; i++) {
            fftRealBufs[fftBufIdx][i] = hanned1[i];
            fftImagBufs[fftBufIdx][i] = 0;
        }
        rearrange(fftRealBufs[fftBufIdx], fftImagBufs[fftBufIdx],
WINDOW_SIZE);
        compute(fftRealBufs[fftBufIdx], fftImagBufs[fftBufIdx],
WINDOW_SIZE);

        // Shift phase

```

```

    processTransformed(fftRealBufs[(fftBufIdx + 1) % 2],
                      fftImagBufs[(fftBufIdx + 1) % 2],
                      fftRealBufs[fftBufIdx],
                      fftImagBufs[fftBufIdx],
                      shiftRealBufs[(fftBufIdx + 1) % 2],
                      shiftImagBufs[(fftBufIdx + 1) % 2],
                      shiftRealBufs[fftBufIdx],
                      shiftImagBufs[fftBufIdx],
    PHASE_SHIFT_AMOUNT(semitoneShift));

    // Perform IFFT
    for (int i = 0; i < WINDOW_SIZE; i++) {
        ifftReal[i] = shiftRealBufs[fftBufIdx][i];
        ifftImag[i] = shiftImagBufs[fftBufIdx][i];
    }
    inverseCompute(ifftReal, ifftImag, WINDOW_SIZE);

    // Second Hann Windowing to prevent popping
    hannify(ifftReal, 0, hanned2);

    // Add outputs to stitcher
    for (int i = 0; i < WINDOW_SIZE; i++) {
        if (i < WINDOW_SIZE - HOP_LENGTH)
            stitcher[(stitcherPtr + i) % WINDOW_SIZE] +=
                (hanned2[i] / 2.0);
        else
            stitcher[(stitcherPtr + i) % WINDOW_SIZE] =
                (hanned2[i] / 2.0);
    }

    // Yield completed stiches to stdout
    for (int i = 0; i < HOP_LENGTH; i++) {
        printf("%f\n", stitcher[stitcherPtr + i]);
    }

    // Increment position-tracking pointers

```

```

        inputWindowStart = (inputWindowStart + HOP_LENGTH) %
                            (WINDOW_SIZE + HOP_LENGTH);
        fftBufIdx = (fftBufIdx + 1) % 2;
        stitcherPtr = (stitcherPtr + HOP_LENGTH) % WINDOW_SIZE;

    }

    inputCurIdx = (inputCurIdx + 1) % (WINDOW_SIZE + HOP_LENGTH);

}
}

```

software-simulation/fft/fft.c

```

#include <math.h>
#include <stdio.h>
#include "fft.h"

void rearrange(float real[], float imaginary[], const unsigned int N)
{
    unsigned int target = 0;
    for (unsigned int position = 0; position < N; position++)
    {
        if (target > position)
        {
            const float temp_real = real[target];
            const float temp_imaginary = imaginary[target];
            real[target] = real[position];
            imaginary[target] = imaginary[position];
            real[position] = temp_real;
            imaginary[position] = temp_imaginary;
        }
        unsigned int mask = N;
        while (target & (mask >>= 1))
            target &= ~mask;
    }
}

```

```
    target |= mask;
}
}

void compute(float real[], float imaginary[], const unsigned int N)
{
    const float pi = -3.14159265358979323846;

    for (unsigned int step = 1; step < N; step <<= 1)
    {
        const unsigned int jump = step << 1;
        const float step_d = (float)step;
        float twiddle_real = 1.0;
        float twiddle_imaginary = 0.0;
        for (unsigned int group = 0; group < step; group++)
        {
            for (unsigned int pair = group; pair < N; pair += jump)
            {
                const unsigned int match = pair + step;
                const float product_real = twiddle_real * real[match] -
twiddle_imaginary * imaginary[match];
                const float product_imaginary = twiddle_imaginary *
real[match] + twiddle_real * imaginary[match];
                real[match] = real[pair] - product_real;
                imaginary[match] = imaginary[pair] - product_imaginary;
                real[pair] += product_real;
                imaginary[pair] += product_imaginary;
            }

            // we need the factors below for the next iteration
            // if we don't iterate then don't compute
            if (group + 1 == step)
            {
                continue;
            }
        }
    }
}
```

```
        float angle = pi * ((float)group + 1) / step_d;
        twiddle_real = cos(angle);
        twiddle_imaginary = sin(angle);
    }
}

void inverseCompute(float real[], float imaginary[], const unsigned int N)
{
    // Take conjugate of the complex numbers
    for (unsigned int i = 0; i < N; ++i) {
        imaginary[i] = -imaginary[i];
    }

    // Perform forward FFT on the conjugate complex numbers
    rearrange(real, imaginary, N);
    compute(real, imaginary, N);

    // Take conjugate of the result (to get the inverse FFT)
    for (unsigned int i = 0; i < N; ++i) {
        imaginary[i] = -imaginary[i];
    }

    // Scale by 1/N to get the correct result
    const float scale = 1.0f / N;
    for (unsigned int i = 0; i < N; ++i) {
        real[i] *= scale;
        imaginary[i] *= scale;
    }
}
```



```
software-simulation/fft/fft.h
```

```
#ifndef FFT_H
#define FFT_H

#include <stddef.h> // for size_t

struct complex_num {
    float real;
    float imsag;
};

void rearrange(float real[], float imaginary[], const unsigned int N);
void compute(float real[], float imaginary[], const unsigned int N);
void inverseCompute(float real[], float imaginary[], const unsigned int
N);

#endif
```

```
software-simulation/fft/main.c
```

```
#include <stdio.h>
#include "fft.h"

int main() {
    // Test case 1 for FFT and IFFT
    const unsigned int N1 = 8;
    float input_real1[] = {1.5, 0.0, 2.3, 0.0, 3.4, 0.0, 4.2, 0.0};
    float input_imaginary1[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
    rearrange(input_real1, input_imaginary1, N1);
    compute(input_real1, input_imaginary1, N1);
    printf("FFT Output:\n");
    for (unsigned int i = 0; i < N1; ++i) {
        printf("%f + %fi\n", input_real1[i], input_imaginary1[i]);
    }
}
```

```

}

// Use the output of FFT as input for IFFT
inverseCompute(input_real1, input_imaginary1, N1);
printf("\nIFFT Output:\n");
for (unsigned int i = 0; i < N1; ++i) {
    printf("%f + %fi\n", input_real1[i], input_imaginary1[i]);
}

// Test case 2 for FFT and IFFT
const unsigned int N2 = 4;
float input_real2[] = {1.0, 2.0, 3.0, 4.0};
float input_imaginary2[] = {0.0, 0.0, 0.0, 0.0};
rearrange(input_real2, input_imaginary2, N2);
compute(input_real2, input_imaginary2, N2);
printf("\nFFT Output:\n");
for (unsigned int i = 0; i < N2; ++i) {
    printf("%f + %fi\n", input_real2[i], input_imaginary2[i]);
}

// Use the output of FFT as input for IFFT
inverseCompute(input_real2, input_imaginary2, N2);
printf("\nIFFT Output:\n");
for (unsigned int i = 0; i < N2; ++i) {
    printf("%f + %fi\n", input_real2[i], input_imaginary2[i]);
}

// Test case 3 for FFT and IFFT
const unsigned int N3 = 8;
float input_real3[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0};
float input_imaginary3[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
rearrange(input_real3, input_imaginary3, N3);
compute(input_real3, input_imaginary3, N3);
printf("\nFFT Output:\n");
for (unsigned int i = 0; i < N3; ++i) {
    printf("%f + %fi\n", input_real3[i], input_imaginary3[i]);
}

```

```

}

// Use the output of FFT as input for IFFT
inverseCompute(input_real3, input_imaginary3, N3);
printf("\nIFFT Output:\n");
for (unsigned int i = 0; i < N3; ++i) {
    printf("%f + %fi\n", input_real3[i], input_imaginary3[i]);
}

// Test case 4 for FFT and IFFT
const unsigned int N4 = 8;
float input_real4[] = {0.34785901, 0.71862502, 0.11248643, 0.59830784,
0.90372152, 0.44128975, 0.29513480, 0.87653024};
float input_imaginary4[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
rearrange(input_real4, input_imaginary4, N4);
compute(input_real4, input_imaginary4, N4);
printf("FFT Output:\n");
for (unsigned int i = 0; i < N4; ++i) {
    printf("%f + %fi\n", input_real4[i], input_imaginary4[i]);
}

// Use the output of FFT as input for IFFT
inverseCompute(input_real4, input_imaginary4, N4);
printf("\nIFFT Output:\n");
for (unsigned int i = 0; i < N4; ++i) {
    printf("%f + %fi\n", input_real4[i], input_imaginary4[i]);
}

// Additional test case for reading 4096 numbers from nums.txt
const unsigned int N5 = 4096;
float input_real5[N5];
float input_imaginary5[N5] = {0}; // Initialize with zeros
FILE *file = fopen("nums.txt", "r");
if (file != NULL) {
    for (unsigned int i = 0; i < N5; ++i) {
        fscanf(file, "%f", &input_real5[i]);
    }
}

```

```

    }

    fclose(file);

    printf("length of real: %lu",
sizeof(input_real5)/sizeof(input_real5[0]));

    rearrange(input_real5, input_imaginary5, N5);
    compute(input_real5, input_imaginary5, N5);

    // Output FFT results to fft.txt
    FILE *fft_file = fopen("fft.txt", "w");
    if (fft_file != NULL) {
        for (unsigned int i = 0; i < N5; ++i) {
            fprintf(fft_file, "%f + %fi\n", input_real5[i],
input_imaginary5[i]);
            //fprintf(fft_file, "%f\n", input_real5[i]);
        }
        fclose(fft_file);
    } else {
        printf("Error opening fft.txt for writing.\n");
    }

    // Use the output of FFT as input for IFFT
    inverseCompute(input_real5, input_imaginary5, N5);

    // Output IFFT results to ifft.txt
    FILE *ifft_file = fopen("ifft.txt", "w");
    if (ifft_file != NULL) {
        for (unsigned int i = 0; i < N5; ++i) {
            fprintf(ifft_file, "%f + %fi\n", input_real5[i],
input_imaginary5[i]);
        }
        fclose(ifft_file);
    } else {
        printf("Error opening ifft.txt for writing.\n");
    }
}

```

```

    } else {
        printf("Error opening nums.txt for reading.\n");
    }

    return 0;
}

```

software-simulation/scaling/scaling.c

```

#include <stdio.h>
#include <math.h>
#include "scaling.h"

#define WINDOW_SIZE 4096
#define HOP_LENGTH 1024

float synthMags[WINDOW_SIZE / 2];
float synthBinDeviations[WINDOW_SIZE / 2];

double phaseDifference(float real1, float imag1, float real2, float imag2)
{
    return atan2(imag2, real2) - atan2(imag1, real1);
}

float wrapPhase(float phaseIn)
{
    if (phaseIn >= 0)
        return fmodf(phaseIn + M_PI, 2.0 * M_PI) - M_PI;
    else
        return fmodf(phaseIn - M_PI, -2.0 * M_PI) + M_PI;
}

void processTransformed(float* realPrev, float* imagPrev, float* realNew,
                       float* imagNew, float* realOutPrev, float*
imagOutPrev,

```

```

float* realOutNew, float* imagOutNew, double
phaseScaleAmount) {

    for (int i = 0; i < WINDOW_SIZE; i++) {
        synthMags[i] = 0;
        synthBinDeviations[i] = 0;
    }

    for (int i = 0; i < WINDOW_SIZE / 2; i++) {
        float dPhase;
        if (realNew[i] == 0 && imagNew[i] == 0) {
            continue;
        }
        else if (realPrev[i] == 0 && imagPrev[i] == 0) {
            dPhase = i * 2 * M_PI * HOP_LENGTH / WINDOW_SIZE;
        }
        else {
            dPhase = phaseDifference(realPrev[i], imagPrev[i], realNew[i],
imagNew[i]);
        }

        float expectedDPhase = i * 2 * M_PI * HOP_LENGTH / WINDOW_SIZE;
        float dPhaseFromExpected = dPhase - expectedDPhase; // compute
phase remainder
        // dPhaseFromExpected = fmodf(fmodf(dPhaseFromExpected, 2 * M_PI) +
(2 * M_PI), 2 * M_PI) - M_PI;
        float binDeviation = wrapPhase(dPhaseFromExpected) * WINDOW_SIZE /
(2 * M_PI * HOP_LENGTH);
        float newBin = (i + binDeviation) * phaseScaleAmount;
        int newBinNum = fmax(fmin(round(newBin), WINDOW_SIZE - 1), 0); //
round cap at max bin
        float newBinDeviation = newBin - newBinNum;

        synthMags[newBinNum] += sqrt((realNew[i] * realNew[i]) +
(imagNew[i] * imagNew[i]));
        synthBinDeviations[newBinNum] += newBinDeviation;

```

```

}

for (int i = 0; i < WINDOW_SIZE / 2; i++) {
    float newPhase;
    if (synthMags[i] == 0) {
        newPhase = 0;
    }
    else if (realOutPrev[i] == 0 && imagOutPrev[i] == 0) {
        newPhase = synthBinDeviations[i];
    }
    else {
        float phaseRemainder = synthBinDeviations[i] * 2 * M_PI *
HOP_LENGTH / WINDOW_SIZE;
        newPhase = atan2(imagOutPrev[i], realOutPrev[i]) +
phaseRemainder + (i * 2 * M_PI * HOP_LENGTH / WINDOW_SIZE);
    }
    realOutNew[i] = cos(newPhase) * synthMags[i];
    imagOutNew[i] = sin(newPhase) * synthMags[i];
    if (i != 0) {
        realOutNew[WINDOW_SIZE - i] = cos(-newPhase) * synthMags[i];
        imagOutNew[WINDOW_SIZE - i] = sin(-newPhase) * synthMags[i];
    }
}
}

void simpleTransform(float* realPrev, float* imagPrev, float* realNew,
                    float* imagNew, float* realOutPrev, float*
imagOutPrev,
                    float* realOutNew, float* imagOutNew, double
phaseScaleAmount) {

    for (int i = 0; i < WINDOW_SIZE; i++) {
        synthMags[i] = 0;
    }

    for (int i = 0; i < WINDOW_SIZE / 2; i++) {
        if (realNew[i] == 0 && imagNew[i] == 0) {

```

```

        continue;
    }
    int newBinNum = round(i * phaseScaleAmount);
    synthMags[newBinNum] += sqrt((realNew[i] * realNew[i]) +
(imagNew[i] * imagNew[i]));
}

for (int i = 0; i < WINDOW_SIZE / 2; i++) {
    float newPhase;
    if (synthMags[i] == 0) {
        newPhase = 0;
    }
    else if (realOutPrev[i] == 0 && imagOutPrev[i] == 0) {
        newPhase = 0;
    }
    else {
        newPhase = atan2(imagOutPrev[i], realOutPrev[i]) + (i * 2 *
M_PI * HOP_LENGTH / WINDOW_SIZE);
    }
    realOutNew[i] = cos(newPhase) * synthMags[i];
    imagOutNew[i] = sin(newPhase) * synthMags[i];
    if (i != 0) {
        realOutNew[WINDOW_SIZE - i] = cos(-newPhase) * synthMags[i];
        imagOutNew[WINDOW_SIZE - i] = sin(-newPhase) * synthMags[i];
    }
}
}
}

```

software-simulation/scaling/scaling.h

```

#ifndef SCALING_H
#define SCALING_H

double phaseDifference(float real1, float imag1, float real2, float
imag2);
void processTransformed(float* realPrev, float* imagPrev, float* realNew,

```



```

        float* imagNew, float* realOutPrev, float*
imagOutPrev,
        float* realOutNew, float* imagOutNew,
        double phaseScaleAmount);
void simpleTransform(float* realPrev, float* imagPrev, float* realNew,
        float* imagNew, float* realOutPrev, float*
imagOutPrev,
        float* realOutNew, float* imagOutNew,
        double phaseScaleAmount);

#endif

```

software-simulation/scaling/main.c

```

#include <stdio.h>
#include <math.h>
#include "scaling.h"

#define WINDOW_SIZE 4096

int approx(float a, float b) {
    if (!(fabs(a - b) < 0.001)) {
        printf("%f neq %f", a, b);
    }
    return fabs(a - b) < 0.001;
}

int main(int argc, char** argv) {

    float testReal1[WINDOW_SIZE]; // previous FFT transform
    float testImag1[WINDOW_SIZE];
    float testReal2[WINDOW_SIZE]; // current FFT transform
    float testImag2[WINDOW_SIZE];
    float testPrevReal[WINDOW_SIZE]; // previous pitch scaled
    float testPrevImag[WINDOW_SIZE];
    float testOutReal[WINDOW_SIZE]; // current output
    float testOutImag[WINDOW_SIZE];

```

```
printf("Test 1: No change\n");
int passed = 1;
for (int i = 0; i < WINDOW_SIZE; i++) {
    testReal1[i] = 1.0;
    testImag1[i] = 0.0;
    testReal2[i] = 1.0;
    testImag2[i] = 0.0;
    testPrevReal[i] = 1.0;
    testPrevImag[i] = 0.0;
}
processTransformed(testReal1, testImag1, testReal2, testImag2,
                  testPrevReal, testPrevImag, testOutReal,
testOutImag,
                  pow(2.0, 0.0));
for (int i = 0; i < WINDOW_SIZE; i++) {
    if (!approx(testOutReal[i], 1.0) || !approx(testOutImag[i], 0.0)) {
        printf(" for i = %d\n", i);
        passed = 0;
        break;
    }
}
if (passed) {
    printf("Passed\n");
} else {
    printf("Failed\n");
}

printf("Test 2: Simple rotation\n");
passed = 1;
for (int i = 0; i < WINDOW_SIZE; i++) {
    testReal1[i] = 1.0;
    testImag1[i] = 0.0;
    testReal2[i] = 0.0;
    testImag2[i] = 1.0;
    testPrevReal[i] = 0.0;
```

```

    testPrevImag[i] = 1.0;
}
processTransformed(testReal1, testImag1, testReal2, testImag2,
                  testPrevReal, testPrevImag, testOutReal,
testOutImag,
                  pow(2.0, 0.0));
for (int i = 0; i < WINDOW_SIZE; i++) {
    if (!approx(testOutReal[i], -1.0) || !approx(testOutImag[i], 0.0))
{
        printf(" for i = %d\n", i);
        passed = 0;
        break;
    }
}
if (passed) {
    printf("Passed\n");
} else {
    printf("Failed\n");
}

printf("Test 3: Shifted start\n");
passed = 1;
for (int i = 0; i < WINDOW_SIZE; i++) {
    testReal1[i] = 1.0;
    testImag1[i] = 0.0;
    testReal2[i] = 0.0;
    testImag2[i] = 1.0;
    testPrevReal[i] = -1.0;
    testPrevImag[i] = 0.0;
}
processTransformed(testReal1, testImag1, testReal2, testImag2,
                  testPrevReal, testPrevImag, testOutReal,
testOutImag,
                  pow(2.0, 0.0));
for (int i = 0; i < WINDOW_SIZE; i++) {

```

```

        if (!approx(testOutReal[i], 0.0) || !approx(testOutImag[i], -1.0))
    {
        printf(" for i = %d\n", i);
        passed = 0;
        break;
    }
    }
    if (passed) {
        printf("Passed\n");
    } else {
        printf("Failed\n");
    }

    printf("Test 4: Different Magnitudes\n");
    passed = 1;
    for (int i = 0; i < WINDOW_SIZE; i++) {
        testReal1[i] = 1.0;
        testImag1[i] = 0.0;
        testReal2[i] = 0.0;
        testImag2[i] = 2.0;
        testPrevReal[i] = -1.0;
        testPrevImag[i] = 0.0;
    }
    processTransformed(testReal1, testImag1, testReal2, testImag2,
        testPrevReal, testPrevImag, testOutReal,
testOutImag,
        pow(2.0, 0.0));
    for (int i = 0; i < WINDOW_SIZE; i++) {
        if (!approx(testOutReal[i], 0.0) || !approx(testOutImag[i], -2.0))
    {
        printf(" for i = %d\n", i);
        passed = 0;
        break;
    }
    }
    if (passed) {

```

```

    printf("Passed\n");
} else {
    printf("Failed\n");
}

printf("Test 5: Scaled shifts up\n");
passed = 1;
for (int i = 0; i < WINDOW_SIZE; i++) {
    testReal1[i] = 1.5;
    testImag1[i] = 1.5;
    testReal2[i] = 0.0;
    testImag2[i] = 2.0;
    testPrevReal[i] = -3.5;
    testPrevImag[i] = -3.5;
}
processTransformed(testReal1, testImag1, testReal2, testImag2,
                  testPrevReal, testPrevImag, testOutReal,
testOutImag,
                  pow(2.0, 1.0));
for (int i = 0; i < WINDOW_SIZE; i++) {
    if (!approx(testOutReal[i], sqrt(2)) || !approx(testOutImag[i],
-sqrt(2))) {
        printf(" for i = %d\n", i);
        passed = 0;
        break;
    }
}
if (passed) {
    printf("Passed\n");
} else {
    printf("Failed\n");
}

printf("Test 6: Scaled shifts down\n");
passed = 1;
for (int i = 0; i < WINDOW_SIZE; i++) {

```

```

    testReal1[i] = 0.4;
    testImag1[i] = 0.4;
    testReal2[i] = -0.6;
    testImag2[i] = -0.6;
    testPrevReal[i] = -1.0;
    testPrevImag[i] = 0.0;
}
processTransformed(testReal1, testImag1, testReal2, testImag2,
                  testPrevReal, testPrevImag, testOutReal,
testOutImag,
                  pow(2.0, -1.0));
for (int i = 0; i < WINDOW_SIZE; i++) {
    if (!approx(testOutReal[i], 0.0) || !approx(testOutImag[i],
sqrt(0.72))) {
        printf(" for i = %d\n", i);
        passed = 0;
        break;
    }
}
if (passed) {
    printf("Passed\n");
} else {
    printf("Failed\n");
}

printf("Test 7: Shifting backwards\n");
passed = 1;
for (int i = 0; i < WINDOW_SIZE; i++) {
    testReal1[i] = 0.0;
    testImag1[i] = 1.0;
    testReal2[i] = 1.0;
    testImag2[i] = 0.0;
    testPrevReal[i] = 0.0;
    testPrevImag[i] = -1.0;
}
processTransformed(testReal1, testImag1, testReal2, testImag2,

```

```

        testPrevReal, testPrevImag, testOutReal,
testOutImag,
        pow(2.0, 0.0));
    for (int i = 0; i < WINDOW_SIZE; i++) {
        if (!approx(testOutReal[i], -1.0) || !approx(testOutImag[i], 0.0))
    {
        printf(" for i = %d\n", i);
        passed = 0;
        break;
    }
    }
    if (passed) {
        printf("Passed\n");
    } else {
        printf("Failed\n");
    }
}

printf("Test 8: Prev FFT 0\n");
passed = 1;
for (int i = 0; i < WINDOW_SIZE; i++) {
    testReal1[i] = 0.0;
    testImag1[i] = 0.0;
    testReal2[i] = 1.2;
    testImag2[i] = -3.4;
    testPrevReal[i] = -5.6;
    testPrevImag[i] = 7.8;
}
processTransformed(testReal1, testImag1, testReal2, testImag2,
        testPrevReal, testPrevImag, testOutReal,
testOutImag,
        pow(2.0, 1.2));
    for (int i = 0; i < WINDOW_SIZE; i++) {
        if (!approx(testOutReal[i], 1.2) || !approx(testOutImag[i], -3.4))
    {
        printf(" for i = %d\n", i);
        passed = 0;
    }
}

```

```
        break;
    }
}
if (passed) {
    printf("Passed\n");
} else {
    printf("Failed\n");
}

printf("Test 9: Prev output 0\n");
passed = 1;
for (int i = 0; i < WINDOW_SIZE; i++) {
    testReal1[i] = 1.2;
    testImag1[i] = 3.4;
    testReal2[i] = -5.6;
    testImag2[i] = -7.8;
    testPrevReal[i] = 0.0;
    testPrevImag[i] = 0.0;
}
processTransformed(testReal1, testImag1, testReal2, testImag2,
                  testPrevReal, testPrevImag, testOutReal,
testOutImag,
                  pow(2.0, 0.0));
for (int i = 0; i < WINDOW_SIZE; i++) {
    if (!approx(testOutReal[i], -5.6) || !approx(testOutImag[i], -7.8))
{
        printf(" for i = %d\n", i);
        passed = 0;
        break;
    }
}
if (passed) {
    printf("Passed\n");
} else {
    printf("Failed\n");
}
}
```



```
}

```

software-simulation/converters/txtter.py

```
import sys
from scipy.io import wavfile
import numpy as np

samplerate, data = wavfile.read(sys.argv[1])

# print line count
if len(sys.argv) > 2:
    with open(sys.argv[2], 'w') as file:
        for sample in data:
            # if sample is a list, write the first element
            if isinstance(sample, np.ndarray):
                file.write(str(sample[0]) + '\n')
            else:
                file.write(str(sample) + '\n')
else:
    for sample in data:
        if isinstance(sample, np.ndarray):
            print(sample[0])
        else:
            print(sample)
```

software-simulation/converters/waver.py

```
import numpy as np
import wave
import sys

def read_samples_from_file(filename):
    samples = []
    if filename is not None:
        with open(filename, 'r') as file:
```

```
        samples = [float(sample) for sample in file.readlines()]

    else:
        samples = [float(sample) for sample in sys.stdin.readlines()]
    return np.array(samples)

def write_wav_file(samples, filename, sample_rate=48000, amplitude=1):
    wav_file = wave.open(filename, 'w')
    wav_file.setparams((1, 2, sample_rate, len(samples), 'NONE', 'not
compressed'))

    # Scale samples to fit within amplitude range
    scaled_samples = np.int16(samples * amplitude)

    # Convert the samples to bytes
    samples_bytes = scaled_samples.tobytes()

    # Write the bytes to the wav file
    wav_file.writeframes(samples_bytes)

    # Close the wav file
    wav_file.close()

if __name__ == "__main__":
    # Change this to the path of your input file
    input_filename = None
    output_filename = None
    if len(sys.argv) == 2:
        output_filename = sys.argv[1]
    elif len(sys.argv) == 3:
        input_filename = sys.argv[1]
        output_filename = sys.argv[2]

    else :
        # print two usage options: one for reading from stdin and one for
reading from a file
```

```
sys.exit(1)

# Read samples from input file
samples = read_samples_from_file(input_filename)

# Write samples to output wav file
write_wav_file(samples, output_filename)

print("WAV file generated successfully.")
```

software-simulation/setup.sh

```
#!/bin/bash

# Check for the presence of the requirements.txt file
if [ ! -f "requirements.txt" ]; then
    echo "requirements.txt file not found!"
    exit 1
fi

# Set the name of the virtual environment
VENV_NAME="venv"

# Function to activate virtual environment
activate_venv() {
    echo "Activating the virtual environment..."
    source $VENV_NAME/bin/activate
}

# Function to install dependencies
install_deps() {
    echo "Installing dependencies from requirements.txt..."
    pip install -r requirements.txt || { echo "Installation failed"; exit
1; }
}

# Check if the virtual environment already exists
```

```
if [ -d "$VENV_NAME" ]; then
    echo "Virtual environment already exists."
    activate_venv

    # Check for mismatched dependencies
    trap 'rm -f .current_requirements.txt' EXIT
    pip freeze > .current_requirements.txt
    DIFF=$(diff -u .current_requirements.txt requirements.txt | grep '^+'
| wc -l)
    if [ "$DIFF" -ne 0 ]; then
        echo "Dependencies in the virtual environment do not match
requirements.txt."
        install_deps
    else
        echo "All dependencies are up to date."
    fi
else
    # Create the virtual environment using python3
    echo "Creating virtual environment..."
    python3 -m venv $VENV_NAME

    if [ ! -d "$VENV_NAME" ]; then
        echo "Failed to create virtual environment."
        exit 1
    fi

    activate_venv
    install_deps
fi

# Confirm the environment is ready
echo "To activate the virtual environment in the future, use the command:
source $VENV_NAME/bin/activate"
echo "To deactivate an active virtual environment, use the command:
deactivate"
```

```
# Make realtime
echo "Building the realtime executable..."
make -C realtime

# Check for the successful build of the realtime executable
if [ ! -x "realtime/main" ]; then
    echo "Failed to build the realtime executable."
    exit 1
fi

# Exiting script
exit 0
```

software-simulation/shift.sh

```
#!/bin/bash

# Check for arguments

if [ "$#" -ne 3 ]; then
    echo "Usage: $0 <shift-amount> <input_file> <output_file>"
    exit 1
fi

find_python() {
    if command -v python3 &>/dev/null; then
        echo "python3"
    elif command -v python &>/dev/null; then
        echo "python"
    else
        echo "No suitable Python interpreter found." >&2
        exit 1
    fi
}

check_scaling_main() {
```

```
if ! [ -x "realtime/main" ]; then
    echo "realtime/main is not built. Run setup.sh" >&2
    exit 1
fi
}

PYTHON=$(find_python)

check_scaling_main

samples_temp="./.samples"
samples_scaled_temp="./.samples_scaled"

# trap 'rm -f "$samples_temp" "$samples_scaled_temp"' EXIT

"$PYTHON" converters/txtter.py "$2" "$samples_temp"
if [ $? -ne 0 ]; then
    echo "Failed to generate samples, exiting." >&2
    exit 1
fi

if ! cat "$samples_temp" | realtime/main "$1" > "$samples_scaled_temp";
then
    echo "Scaling failed, exiting." >&2
    exit 1
fi

"$PYTHON" converters/waver.py "$samples_scaled_temp" "$3"
if [ $? -ne 0 ]; then
    echo "Failed to process scaled samples, exiting." >&2
    exit 1
fi

echo "Processing complete."
```

Hardware Implementation Files

implementation/hw/components/audio_piper

```
/*
 * Avalon streaming component to pipe left from_adc chanel of Wolfson Audio
 * CODEC to both left and right channels of output
 *
 * Steven Winnick
 * Columbia University
 */

module audio_piper(
    input logic      clk,
    input logic      reset,

    // from avalon_left_channel_source from audio codec
    input logic [15:0] left_in_data,
    input logic      left_in_valid,
    output logic      left_in_ready = 0,

    // from avalon_right_channel_source from audio codec
    input logic [15:0] right_in_data,
    input logic      right_in_valid,
    output logic      right_in_ready = 0,

    // to avalon_left_channel_sink in audio codec
    output logic [15:0] left_out_data = 0,
    output logic      left_out_valid = 0,
    input logic      left_out_ready,

    // to avalon_right_channel_sink in audio codec
    output logic [15:0] right_out_data = 0,
    output logic      right_out_valid = 0,
    input logic      right_out_ready,
```

```
        output logic [9:0] lights = 0
    );

    logic [15:0] data = 0;
    logic [9:0] ctr1 = 0;
    logic [9:0] ctr2 = 0;
    logic [9:0] ctr3 = 0;
    logic [9:0] ctr4 = 0;
    logic [9:0] ctr5 = 0;

    always_ff @(posedge clk) begin
        if (left_in_valid) begin
            data <= left_in_data;
            left_in_ready <= 1;
        end

        // Per Avalon Interface Specification, ready only flashes for one
cycle
        if (left_in_ready) begin
            left_in_ready <= 0;
        end

        if (left_out_ready) begin
            left_out_data <= data;
            left_out_valid <= 1;
        end

        if (right_out_ready) begin
            right_out_data <= data;
            right_out_valid <= 1;
        end

        if (left_out_valid) begin
            left_out_valid <= 0;
        end
    end
```



```
    if (right_out_valid) begin
        right_out_valid <= 0;
    end

    if (left_in_valid)
    if (ctr1 == 0) begin
        lights[0] <= 1;
    ctr1 <= ctr1 + 1;
end
    else
    ctr1 <= ctr1 + 1;
    else
    if (ctr2 == 0) begin
        lights[0] <= 0;
    ctr2 <= ctr2 + 1;
end
    else
    ctr2 <= ctr2 + 1;

    if (right_in_valid)
        lights[1] <= 1;
    else
        lights[1] <= 0;

    if (left_out_ready)
        lights[2] <= 1;
    else
        lights[2] <= 0;

    if (right_out_ready)
        lights[3] <= 1;
    else
        lights[3] <= 0;

    if (ctr3 == 0) begin
```

```

        ctr3 <= ctr3 + 1;
        if (lights[8])
            lights[8] <= 0;
        else
            lights[8] <= 1;
    end
    else
        ctr3 <= ctr3 + 1;
    end
endmodule

```

implementation/hw/components/cart_to_polar

```

/*
 * Convert the imaginary coordinates from the FFT to polar coordinates
 */

module cart_to_polar(
    input logic      clk,

    // Communicate with ffter
    input logic      go_in,

    // Read from post_fft_buf_real
    input logic [15:0] real_buf_data,
    output logic [11:0] real_buf_addr = 0,

    // Read from post_fft_buf_imag
    input logic [15:0] imag_buf_data,
    output logic [11:0] imag_buf_addr = 0,

    // Write to scaler_mag_buf_0
    output logic [15:0] mag_buf_0_data = 0,
    output logic [11:0] mag_buf_0_addr = 0,
    output logic      mag_buf_0_wren = 0,

```

```

    // Write to scaler_phase_buf_0
    output logic [15:0] phase_buf_0_data = 0,
    output logic [11:0] phase_buf_0_addr = 0,
    output logic      phase_buf_0_wren = 0,

    // Write to scaler_mag_buf_1
    output logic [15:0] mag_buf_1_data = 0,
    output logic [11:0] mag_buf_1_addr = 0,
    output logic      mag_buf_1_wren = 0,

    // Write to scaler_phase_buf_1
    output logic [15:0] phase_buf_1_data = 0,
    output logic [11:0] phase_buf_1_addr = 0,
    output logic      phase_buf_1_wren = 0,

    // Communicate with scaler
    output logic      cur_buf = 0,
    output logic      go_out = 0
);

logic [15:0] magnitude;
logic [15:0] phase;
logic [11:0] n_samples = 0;
logic aux, i_ce, done;

// Instance of the polar_converter
topolar polar_converter (
    .i_clk(clk),
    .i_reset(!go_in),
    .i_ce(i_ce),
    .i_xval(real_buf_data[15:0]),
    .i_yval(imag_buf_data[15:0]),
    .i_aux(aux),
    .o_mag(magnitude),
    .o_phase(phase),
    .o_aux(aux)

```

```
.o_done(done)
);

always_ff @(posedge clk) begin
    if (!going) begin
        if (go_in) begin
            // Start of processing
            going <=1;
            go_out <=0;
            i_ce <= 1;
            curr_buf <= !curr_buf;

            end
        end else begin
            // Perform address increment and data processing
            if (done) begin
                real_buf_addr <= real_buf_addr + 1;
                imag_buf_addr <= imag_buf_addr + 1;
                n_samples <= n_samples + 1;

                if (!curr_buf) begin

                    mag_buf_0_data <= magnitude;
                    mag_buf_0_addr <= real_buf_addr - 1;
                    mag_buf_0_wren <= 1;
                    phase_buf_0_data <= phase[15:0];
                    phase_buf_0_addr <= real_buf_addr - 1;
                    phase_buf_0_wren <= 1;

                end
            else begin

                mag_buf_1_data <= magnitude;
                mag_buf_1_addr <= real_buf_addr - 1;
                mag_buf_1_wren <= 1;
                phase_buf_1_data <= phase[15:0];
                phase_buf_1_addr <= real_buf_addr - 1;
                phase_buf_1_wren <= 1;

            end
        end
    end
end
```

```

        end

        if (n_samples == 12'd4095) begin
            going <= 0;
            go_out <= 1;
        end
    end
end

endmodule

`default_nettype    none
//
module  topolar(i_clk, i_reset, i_ce, i_xval, i_yval, i_aux,
               o_mag, o_phase, o_aux);
    localparam  IW=16, // The number of bits in our inputs
               OW=16, // The number of output bits to produce
               NSTAGES=16,
               XTRA= 3, // Extra bits for internal precision
               WW=18, // Our working bit-width
               PW=16; // Bits in our phase variables

    input          i_clk, i_reset, i_ce;
    input  wire    signed [(IW-1):0] i_xval, i_yval;
    output reg signed [(OW-1):0] o_mag;
    output reg     [(PW-1):0] o_phase;
    input  wire    i_aux;
    output reg     o_aux;
    output wire o_done;

    // First step: expand our input to our working width.
    // This is going to involve extending our input by one
    // (or more) bits in addition to adding any xtra bits on
    // bits on the right. The one bit extra on the left is to
    // allow for any accumulation due to the cordic gain
    // within the algorithm.
    //

```

```

wire    signed [(WW-1):0]    e_xval, e_yval;
assign  e_xval = { {(2){i_xval[(IW-1)]}}, i_xval, {(WW-IW-2){1'b0}} };
assign  e_yval = { {(2){i_yval[(IW-1)]}}, i_yval, {(WW-IW-2){1'b0}} };

// Declare variables for all of the separate stages
reg signed [(WW-1):0]    xv    [0:NSTAGES];
reg signed [(WW-1):0]    yv    [0:NSTAGES];
reg        [(PW-1):0]    ph    [0:NSTAGES];

//
// Handle the auxilliary logic.
//
// The auxilliary bit is designed so that you can place a valid bit
into
// the CORDIC function, and see when it comes out. While the bit is
// allowed to be anything, the requirement of this bit is that it
*must*
// be aligned with the output when done. That is, if i_xval and i_yval
// are input together with i_aux, then when o_xval and o_yval are set
// to this value, o_aux *must* contain the value that was in i_aux.
//
reg        [(NSTAGES):0]    ax;

always @(posedge i_clk)
if (i_reset)
    ax <= {(NSTAGES+1){1'b0}};
else if (i_ce)
    ax <= { ax[(NSTAGES-1):0], i_aux };

// First stage, map to within +/- 45 degrees
always @(posedge i_clk)
if (i_reset)
begin
    xv[0] <= 0;
    yv[0] <= 0;
    ph[0] <= 0;

```

```

end else if (i_ce)
    case({i_xval[IW-1], i_yval[IW-1]})
    2'b01: begin // Rotate by -315 degrees
        xv[0] <= e_xval - e_yval;
        yv[0] <= e_xval + e_yval;
        ph[0] <= 19'h70000;
        end
    2'b10: begin // Rotate by -135 degrees
        xv[0] <= -e_xval + e_yval;
        yv[0] <= -e_xval - e_yval;
        ph[0] <= 19'h30000;
        end
    2'b11: begin // Rotate by -225 degrees
        xv[0] <= -e_xval - e_yval;
        yv[0] <= e_xval - e_yval;
        ph[0] <= 19'h50000;
        end
    // 2'b00:
    default: begin // Rotate by -45 degrees
        xv[0] <= e_xval + e_yval;
        yv[0] <= -e_xval + e_yval;
        ph[0] <= 19'h10000;
        end
    endcase
//
// In many ways, the key to this whole algorithm lies in the angles
// necessary to do this.  These angles are also our basic reason for
// building this CORDIC in C++: Verilog just can't parameterize this
// much.  Further, these angle's risk becoming unsupportable magic
// numbers, hence we define these and set them in C++, based upon
// the needs of our problem, specifically the number of stages and
// the number of bits required in our phase accumulator
//
wire    [18:0]  cordic_angle [0:(NSTAGES-1)];

assign  cordic_angle[ 0] = 19'h0_9720; // 26.565051 deg

```

```

assign cordic_angle[ 1] = 19'h0_4fd9; // 14.036243 deg
assign cordic_angle[ 2] = 19'h0_2888; //  7.125016 deg
assign cordic_angle[ 3] = 19'h0_1458; //  3.576334 deg
assign cordic_angle[ 4] = 19'h0_0a2e; //  1.789911 deg
assign cordic_angle[ 5] = 19'h0_0517; //  0.895174 deg
assign cordic_angle[ 6] = 19'h0_028b; //  0.447614 deg
assign cordic_angle[ 7] = 19'h0_0145; //  0.223811 deg
assign cordic_angle[ 8] = 19'h0_00a2; //  0.111906 deg
assign cordic_angle[ 9] = 19'h0_0051; //  0.055953 deg
assign cordic_angle[10] = 19'h0_0028; //  0.027976 deg
assign cordic_angle[11] = 19'h0_0014; //  0.013988 deg
assign cordic_angle[12] = 19'h0_000a; //  0.006994 deg
assign cordic_angle[13] = 19'h0_0005; //  0.003497 deg
assign cordic_angle[14] = 19'h0_0002; //  0.001749 deg
assign cordic_angle[15] = 19'h0_0001; //  0.000874 deg
// Std-Dev      : 0.00 (Units)
// Phase Quantization: 0.000030 (Radians)
// Gain is 1.164435
// You can annihilate this gain by multiplying by 32'hdbd95b16
// and right shifting by 32 bits.

genvar i;
generate for(i=0; i<NSTAGES; i=i+1) begin : TOPOLARloop
    always @(posedge i_clk)
        // Here's where we are going to put the actual CORDIC
        // rectangular to polar loop.  Everything up to this
        // point has simply been necessary preliminaries.
        if (i_reset)
            begin
                xv[i+1] <= 0;
                yv[i+1] <= 0;
                ph[i+1] <= 0;
            end else if (i_ce)
            begin
                if ((cordic_angle[i] == 0) || (i >= WW))
                    begin // Do nothing but move our vector

```



```

// forward one stage, since we have more
// stages than valid data
    xv[i+1] <= xv[i];
    yv[i+1] <= yv[i];
    ph[i+1] <= ph[i];
end else if (yv[i][ (WW-1) ]) // Below the axis
begin
    // If the vector is below the x-axis, rotate by
    // the CORDIC angle in a positive direction.
    xv[i+1] <= xv[i] - (yv[i]>>>(i+1));
    yv[i+1] <= yv[i] + (xv[i]>>>(i+1));
    ph[i+1] <= ph[i] - cordic_angle[i];
end else begin
    // On the other hand, if the vector is above the
    // x-axis, then rotate in the other direction
    xv[i+1] <= xv[i] + (yv[i]>>>(i+1));
    yv[i+1] <= yv[i] - (xv[i]>>>(i+1));
    ph[i+1] <= ph[i] + cordic_angle[i];
end

end

if (i == NSTAGES-1) begin
    o_done <= 1;
end else begin
    o_done <= 0;
end

end

end

end endgenerate

// Round our magnitude towards even
wire [ (WW-1):0 ] pre_mag;

assign pre_mag = xv[NSTAGES] + $signed({{ (OW){1'b0}},
    xv[NSTAGES] [ (WW-OW) ],
    { (WW-OW-1) {!xv[NSTAGES] [WW-OW]} }});

always @(posedge i_clk)

```

```

if (i_reset)
begin
    o_mag    <= 0;
    o_phase <= 0;
    o_aux   <= 0;
end else if (i_ce)
begin
    o_mag    <= pre_mag[(WW-1):(WW-OW)];
    o_phase <= ph[NSTAGES];
    o_aux   <= ax[NSTAGES];
end

wire    [(WW-OW):0] unused_val;
assign  unused_val = { pre_mag[WW-1], pre_mag[(WW-OW-1):0] };
endmodule

```

implementation/hw/components/emitter

```

/*
 * Avalon streaming component to write to both left and right to_dac
 channels of
 * the Wolfson Audio CODEC
 */

module emitter(
    input logic    clk,

    // Communicate with stitcher
    input logic [1:0]  window_start,
    input logic    go_in,

    // Write to avalon_left_channel_sink of audio codec
    output logic [15:0] left_out_data = 0,
    output logic    left_out_valid = 0,
    input logic    left_out_ready,

```

```

    // Write to avalon_right_channel_sink of audio codec
    output logic [15:0] right_out_data = 0,
    output logic      right_out_valid = 0,
    input logic      right_out_ready

);

logic going = 0;
logic just_finished = 0;
logic tmp = 0;

always_ff @(posedge clk) begin
    if (!going) begin
        if (go_in) begin
            going <= 1;
        end
    end
    else begin
        if (!just_finished) begin
            tmp <= 1;

            // Need to finish one cycle later so we can write last to
out buf

            if (tmp) begin
                just_finished <= 1;
            end
        end
        else begin // just finished
            // out_buf_wren <= 0;
            just_finished <= 0;
            going <= 0;
        end
    end
end

end

endmodule

```

implementation/hw/components/ffter

```
/*
 * Wrapper around the Altera FFT IP block to perform the FFT on samples in
 * one
 * buffer and send the results to another buffer
 */

module ffter(
    input logic    clk,

    // Communicate with first_hannifier
    input logic    go_in,

    // Read from pre_fft_buf
    input logic [15:0]  in_buf_data,
    output logic [11:0] in_buf_addr = 0,

    // Communicate with FFT IP block
    // [TODO]

    // Write to post_fft_buf_real
    output logic [15:0] real_buf_data = 0,
    output logic [11:0] real_buf_addr = 0,
    output logic      real_buf_wren = 0,

    // Write to post_fft_buf_imag
    output logic [15:0] imag_buf_data = 0,
    output logic [11:0] imag_buf_addr = 0,
    output logic      imag_buf_wren = 0,

    // Communicate with cart_to_polar
    output logic      go_out = 0
);

logic going = 0;
```

```

logic just_finished = 0;
logic tmp = 0;

always_ff @(posedge clk) begin
    if (!going) begin
        if (go_in) begin
            go_out <= 0;
            going <= 1;
        end
    end
else begin
    if (!just_finished) begin
        tmp <= 1;

        // Need to finish one cycle later so we can write last to
out buf

        if (tmp) begin
            just_finished <= 1;
        end
    end
else begin // just finished
    // out_buf_wren <= 0;
    just_finished <= 0;
    go_out <= 1;
    going <= 0;
end
end
end
endmodule

```

implementation/hw/components/first_hannifier

```

/*
* Apply Hann Windowing function to a window of samples from the input ring
* buffer

```

```

*/

module first_hannifier(
    input logic      clk,

    // Communicate with sampler
    input logic [2:0] window_start,
    input logic      go_in,

    // Read from ring buffer
    input logic [15:0] ring_buf_data,
    output logic [12:0] ring_buf_addr = 0, // larger than typical
buffer because ringbuf holds one extra window

    // Read from Hann Window ROM
    input logic [15:0] hann_rom_data,
    output logic [11:0] hann_rom_addr = 0,

    // Write to pre-FFT buffer
    output logic [15:0] out_buf_data = 0,
    output logic [11:0] out_buf_addr = 0,
    output logic      out_buf_wren = 0,

    // Communicate with ffter
    output logic      go_out = 0
);

logic going = 0;
logic just_finished = 0;

always_ff @(posedge clk) begin
    if (!going) begin
        if (go_in) begin
            ring_buf_addr <= window_start;
            hann_rom_addr <= 0;
            out_buf_addr <= 0;

```

```

        go_out <= 0;
        going <= 1;
    end
end
else begin
    if (!just_finished) begin
        out_buf_wren <= 1;
        out_buf_data <= ring_buf_data * hann_rom_data >> 16; //
hann_rom_data: Q0.16, I think output is Q16.16, and then take only first
16 most sig bits
        out_buf_addr <= hann_rom_addr;
        hann_rom_addr <= hann_rom_addr + 1;

        if (ring_buf_addr == 4095 + 1024) begin
            ring_buf_addr <= 0;
        end
        else begin
            ring_buf_addr <= ring_buf_addr + 1;
        end

        // Need to finish one cycle later so we can write last to
out buf
        if (hann_rom_addr == 4095) begin
            just_finished <= 1;
        end
    end
end
else begin // just finished
    out_buf_wren <= 0;
    just_finished <= 0;
    go_out <= 1;
    going <= 0;
end
end
end
end
endmodule

```

implementation/hw/components/iffter

```
/*
 * Wrapper around the Altera FFT IP block to perform the IFFT on samples in
 * one
 * buffer and send the results to another buffer
 */

module ffter(
    input logic    clk,

    // Communicate with first_hannifier
    input logic    go_in,

    // Read from pre_ifft_real_buf
    input logic [15:0]  real_buf_data,
    output logic [11:0] real_buf_addr = 0,

    // Read from pre_ifft_imag_buf
    input logic [15:0]  imag_buf_data,
    output logic [11:0] imag_buf_addr = 0,

    // Communicate with IFFT IP block
    // [TODO]

    // Write to post_ifft_buf
    output logic [15:0] out_buf_data = 0,
    output logic [11:0] out_buf_addr = 0,
    output logic        out_buf_wren = 0,

    // Communicate with cart_to_polar
    output logic        go_out = 0
);

logic going = 0;
```



```

logic just_finished = 0;
logic tmp = 0;

always_ff @(posedge clk) begin
    if (!going) begin
        if (go_in) begin
            go_out <= 0;
            going <= 1;
        end
    end
else begin
    if (!just_finished) begin
        tmp <= 1;

        // Need to finish one cycle later so we can write last to
out buf

        if (tmp) begin
            just_finished <= 1;
        end
    end
else begin // just finished
    // out_buf_wren <= 0;
    just_finished <= 0;
    go_out <= 1;
    going <= 0;
end
end
end
endmodule

```

implementation/hw/components/sampler

```

/*
* Avalon streaming component to read samples of left from_adc channel of
* the Wolfson Audio CODEC to a ring buffer

```

```

*/

module sampler(
    input logic      clk,
    input logic      reset,

    // Read from avalon_left_channel_source from audio codec
    input logic [15:0] left_in_data,
    input logic      left_in_valid,
    output logic      left_in_ready = 0,

    // Read from avalon_right_channel_source from audio codec
    input logic [15:0] right_in_data,
    input logic      right_in_valid,
    output logic      right_in_ready = 0,

    // Write to ring buffer
    output logic [15:0] ring_buf_data,
    output logic [12:0] ring_buf_addr = 0, // larger than typical
buffer because ringbuf holds one extra window
    output logic      ring_buf_wren = 0,

    // Communicate with first_hannifier
    output logic [2:0] window_start,
    output logic      go_out = 0
);

logic going = 0;
logic just_finished = 0;
logic tmp = 0;

always_ff @(posedge clk) begin
    if (!going) begin
        if (left_in_valid) begin
            go_out <= 0;
            going <= 1;

```

```

        end
    end
    else begin
        if (!just_finished) begin
            tmp <= 1;

            // Need to finish one cycle later so we can write last to
out buf

            if (tmp) begin
                just_finished <= 1;
            end
        end
        else begin // just finished
            // out_buf_wren <= 0;
            just_finished <= 0;
            go_out <= 1;
            going <= 0;
        end
    end
end
end
end

endmodule

```

implementation/hw/components/scaler

```

/*
* Component which performs pitch scaling
*/

module scaler(
    input logic    clk,

    // Communicate with cart_to_polar
    input logic    go_in,
    input logic    cur_window,

```

```
// Communicate with software_interface
input logic [7:0]  scale_amt,

// Read from pre_scaler_mag_buf_0
input logic [15:0]  mag_in_buf_0_data,
output logic [11:0] mag_in_buf_0_addr = 0,

// Read from pre_scaler_phase_buf_0
input logic [15:0]  phase_in_buf_0_data,
output logic [11:0] phase_in_buf_0_addr = 0,

// Read from pre_scaler_mag_buf_1
input logic [15:0]  mag_in_buf_1_data,
output logic [11:0] mag_in_buf_1_addr = 0,

// Read from pre_scaler_phase_buf_1
input logic [15:0]  phase_in_buf_1_data,
output logic [11:0] phase_in_buf_1_addr = 0,

// Read from and write to post_scaler_mag_buf_0
input logic [15:0]  mag_out_buf_0_rdata,
output logic [11:0] mag_out_buf_0_raddr = 0,
output logic [15:0] mag_out_buf_0_wrddata = 0,
output logic [11:0] mag_out_buf_0_wraddr = 0,
output logic      mag_out_buf_0_wren = 0,

// Read from and write to post_scaler_phase_buf_0
input logic [15:0]  phase_out_buf_0_rdata,
output logic [11:0] phase_out_buf_0_raddr = 0,
output logic [15:0] phase_out_buf_0_wrddata = 0,
output logic [11:0] phase_out_buf_0_wraddr = 0,
output logic      phase_out_buf_0_wren = 0,

// Read from and write to post_scaler_mag_buf_1
input logic [15:0]  mag_out_buf_1_rdata,
output logic [11:0] mag_out_buf_1_raddr = 0,
```

```
output logic [15:0] mag_out_buf_1_wrdata = 0,
output logic [11:0] mag_out_buf_1_wraddr = 0,
output logic      mag_out_buf_1_wren = 0,

// Read from and write to post_scaler_phase_buf_1
input logic [15:0]  phase_out_buf_1_rdata,
output logic [11:0] phase_out_buf_1_raddr = 0,
output logic [15:0] phase_out_buf_1_wrdata = 0,
output logic [11:0] phase_out_buf_1_wraddr = 0,
output logic      phase_out_buf_1_wren = 0,

// Read from and write to scaler_synth_mags_buf
input logic [15:0]  synth_mags_rdata,
output logic [11:0] synth_mags_raddr = 0,
output logic [15:0] synth_mags_wrdata = 0,
output logic [11:0] synth_mags_wraddr = 0,
output logic      synth_mags_wren = 0,

// Read from and write to scaler_synth_devs_buf
input logic [15:0]  synth_devs_rdata,
output logic [11:0] synth_devs_raddr = 0,
output logic [15:0] synth_devs_wrdata = 0,
output logic [11:0] synth_devs_wraddr = 0,
output logic      synth_devs_wren = 0,

// Communicate with polar_to_cart
output logic      cur_buf = 0,
output logic      go_out = 0
);

logic going = 0;
enum logic [2:0] {awaiting, analysis, synthesis, just_finished} state =
awaiting;
logic anal_read = 1;
logic cur_buf_num;
```

```
logic [15:0] mag_in_data;
logic [15:0] mag_in_prev_data;
logic [15:0] phase_in_data;
logic [15:0] phase_in_prev_data;
logic [11:0] mag_in_addr;
logic [11:0] phase_in_addr;
logic [15:0] mag_out_rdata;
logic [15:0] mag_out_prev_rdata;
logic [11:0] mag_out_raddr;
logic [15:0] mag_out_wrddata;
logic [11:0] mag_out_wraddr;
logic      mag_out_wren;
logic [15:0] phase_out_rdata;
logic [15:0] phase_out_prev_rdata;
logic [11:0] phase_out_raddr;
logic [11:0] phase_out_prev_raddr;
logic [15:0] phase_out_wrddata;
logic [11:0] phase_out_wraddr;
logic      phase_out_wren;

logic [11:0] i = 0;
logic signed [15:0] d_phase;
logic signed [15:0] expected_d_phase = 0;
logic signed [15:0] d_phase_from_expected;
logic signed [15:0] bin_dev;
logic signed [15:0] new_bin_dev;
logic signed [15:0] fractional_bin;
logic signed [31:0] extended_mult;
logic signed [15:0] new_bin;
logic signed [15:0] new_bin_num;

logic signed [15:0] expected_d_phase_increment = 16'h192; // pi/2
logic signed [15:0] bin_dev_multiplier = 16'h0a3; // 2/pi
logic signed [15:0] pi = 16'h324;
logic signed [15:0] two_pi = 16'h648;
logic signed [15:0] neg_pi = 16'h9b8;
```

```
always_comb begin

    mag_in_data = cur_buf_num ? mag_in_buf_1_data : mag_in_buf_0_data;
    mag_in_prev_data = cur_buf_num ? mag_in_buf_0_data :
mag_in_buf_1_data;
    mag_in_buf_0_addr = mag_in_addr;
    mag_in_buf_1_addr = mag_in_addr;

    phase_in_data = cur_buf_num ? phase_in_buf_1_data :
phase_in_buf_0_data;
    phase_in_prev_data = cur_buf_num ? phase_in_buf_0_data :
phase_in_buf_1_data;
    phase_in_buf_0_addr = phase_in_addr;
    phase_in_buf_1_addr = phase_in_addr;

    mag_out_rdata = cur_buf_num ? mag_out_buf_1_rdata :
mag_out_buf_0_rdata;
    mag_out_prev_rdata = cur_buf_num ? mag_out_buf_0_rdata :
mag_out_buf_1_rdata;
    mag_out_buf_0_raddr = mag_out_raddr;
    mag_out_buf_1_raddr = mag_out_raddr;
    mag_out_buf_0_wraddr = mag_out_wraddr;
    mag_out_buf_1_wraddr = mag_out_wraddr;
    mag_out_buf_0_wrddata = mag_out_wrddata;
    mag_out_buf_1_wrddata = mag_out_wrddata;
    mag_out_buf_0_wren = mag_out_wren && !cur_buf_num;
    mag_out_buf_1_wren = mag_out_wren && cur_buf_num;

    phase_out_rdata = cur_buf_num ? phase_out_buf_1_rdata :
phase_out_buf_0_rdata;
    phase_out_prev_rdata = cur_buf_num ? phase_out_buf_0_rdata :
phase_out_buf_1_rdata;
    phase_out_buf_0_raddr = phase_out_raddr;
    phase_out_buf_1_raddr = phase_out_raddr;
    phase_out_buf_0_wraddr = phase_out_wraddr;
```

```

    phase_out_buf_1_wraddr = phase_out_wraddr;
    phase_out_buf_0_wrdata = phase_out_wrdata;
    phase_out_buf_1_wrdata = phase_out_wrdata;
    phase_out_buf_0_wren = phase_out_wren && !cur_buf_num;
    phase_out_buf_1_wren = phase_out_wren && cur_buf_num;

end

always_ff @(posedge clk) begin
    case (state)
        awaiting: begin
            if (go_in) begin
                go_out <= 0;
                cur_buf_num <= cur_window; // gets set at go, stays for
full run

                state <= analysis;
                expected_d_phase <= 0;
                anal_read <= 1;
            end
        end
    end

    analysis: begin
        if (anal_read) begin // do in two phases because we need to
read old data from memory, then add to it
            d_phase = phase_in_data - phase_in_prev_data;
            if (d_phase < 0) d_phase = d_phase + two_pi; // wrap from 0
to 2pi

            d_phase_from_expected = d_phase - expected_d_phase;
            if (d_phase_from_expected < neg_pi) d_phase_from_expected =
d_phase_from_expected + two_pi; // wrap from -pi to pi

            bin_dev = d_phase_from_expected * bin_dev_multiplier;
            fractional_bin = (i << 8) + bin_dev;
            extended_mult = fractional_bin * scale_amt;
            if (extended_mult > (2047 << 16)) // above Nyquist
frequency

                new_bin = 0;
        end
    end
end

```



```

else
    new_bin = extended_mult >> 16;
    new_bin_num = (new_bin >> 8) << 8;
    if (new_bin[7]) new_bin_num = new_bin_num + (2 ** 8); // if
most significant fractional bit is 1, round up
    new_bin_dev = new_bin - new_bin_num;

    synth_mags_raddr = new_bin_num >> 8;
    synth_devs_raddr = new_bin_num >> 8;
    synth_mags_wren = 0;
    synth_devs_wren = 0;
    anal_read = 0;
end
else begin
    synth_mags_wrddata <= synth_mags_rdata + mag_in_data;
    synth_devs_wrddata <= synth_devs_rdata + new_bin_dev;
    synth_mags_wraddr <= new_bin_num >> 8;
    synth_devs_wraddr <= new_bin_num >> 8;
    synth_mags_wren <= 1;
    synth_devs_wren <= 1;

    i <= (i == 2048) ? 0 : i + 1;
    expected_d_phase <= expected_d_phase +
expected_d_phase_increment;

    if (i == 2048) begin
        state <= synthesis;
        synth_mags_wren <= 0;
        synth_devs_wren <= 0;
        synth_mags_raddr <= 0;
        synth_devs_raddr <= 0;
        synth_mags_wrddata <= 0;
        synth_devs_wrddata <= 0;
        synth_mags_wraddr <= 0;
        phase_out_prev_raddr <= 0;
        expected_d_phase <= 0;
    end
end
end

```

```
        end

        anal_read <= 1;
    end
end

synthesis: begin

    phase_out_wrdata <= phase_out_prev_rdata + expected_d_phase;
    mag_out_wrdata <= synth_mags_rdata;
    phase_out_wraddr <= i;
    mag_out_wraddr <= i;
    phase_out_wren <= 1;
    mag_out_wren <= 1;

    synth_mags_raddr <= i + 1;
    phase_out_prev_raddr <= i + 1;
    i <= i + 1;
    expected_d_phase <= expected_d_phase +
expected_d_phase_increment;

    if (i == 2047) begin
        state <= just_finished;
    end
end

just_finished: begin
    expected_d_phase <= 0;
    synth_mags_wren <= 0;
    synth_devs_wren <= 0;
    synth_mags_raddr <= 0;
    synth_devs_raddr <= 0;
    synth_mags_wrdata <= 0;
    synth_devs_wrdata <= 0;
    expected_d_phase <= 0;
    i <= 0;
```

```

        go_out <= 1;
        state <= awaiting;
    end
    endcase
end
endmodule

```

implementation/hw/components/software_interface

```

/*
 * Software Interface for Pitch Perfect Project
 *
 * Steven Winnick
 * Columbia University
 */

module software_interface(
    input logic      clk,
    input logic      reset,

    // Input from the Avalon Bus
    input logic [7:0] writedata,
    input logic      write,
    input            chipselect,
    input logic [2:0] address,

    // Tell the rest of the system how much to shift by (controlled by
software)
    output logic [7:0] shift_amt = 0,

    // Fill the AV Config component's avalon slave module to quiet
warnings
    output logic [1:0] av_config_slave_address = 0,
    output logic [3:0] av_config_slave_byteenable = 0,

```

```

output logic      av_config_slave_read = 0,
output logic      av_config_slave_write = 0,
output logic [31:0] av_config_slave_writedata = 0,
input logic [31:0] av_config_slave_readdata,
input logic       av_config_slave_waitrequest
);

always_ff @(posedge clk) begin
    if (chipselct && write)
        shift_amt <= writedata;
end

endmodule

```

implementation/hw/components/stitcher

```

/*
 * Apply Hann Windowing function to a window of samples and insert them to
 the
 * output buffer
 */

module stitcher(
    input logic      clk,

    // Communicate with ifftr
    input logic      go_in,

    // Read from post_ifft_buf
    input logic [15:0] in_buf_data,
    output logic [11:0] in_buf_addr = 0,

    // Read from Hann Window ROM
    input logic [15:0] hann_rom_data,
    output logic [11:0] hann_rom_addr = 0,

```

```

        // Write to stitched_buf
        output logic [15:0] out_buf_data = 0,
        output logic [11:0] out_buf_addr = 0,
        output logic          out_buf_wren = 0,

        // Communicate with emitter
        output logic [1:0] window_start = 0,
        output logic          go_out = 0
    );

    always_ff @(posedge clk) begin
        window_start = 1;
    end
endmodule

```

implementation/hw/fft_testing/fft_sim/testbench.v

```

`timescale 10ns/100ps
module testbench;

    reg clk;

    wire [31:0] fsin_o, fcos_o;
    wire [31:0] real_power_sig, imag_power_sig;

    initial
    begin
        clk=0;
    end

    always
    begin
        #10 clk=!clk;
    end
endmodule

```

```
end

wire reset_n;

nco nco_inst(
    .clk      (clk),      // clk.clk
    .reset_n  (reset_n), // rst.reset_n
    .clken    (1'b1),    // in.clken
    .phi_inc_i (32'd41943040), // .phi_inc_i
    .fsin_o    (fsin_o), // out.fsin_o
    .fcos_o    (fcos_o), // .fcos_o
    .out_valid (out_valid) // .out_valid
);

fft_wrapper fft_wrapper_inst
(
    .clk(clk) , // input  clk_sig
    .in_signal(fsin_o) , // input [31:0] in_signal_sig
    .real_power(real_power_sig) , // output [31:0] real_power_sig
    .imag_power(imag_power_sig) , // output [31:0] imag_power_sig
    .fft_source_sop(fft_source_sop_sig) , // output  fft_source_sop_sig
    .sink_sop(sink_sop_sig) , // output  sink_sop_sig
    .sink_eop(sink_eop_sig) , // output  sink_eop_sig
    .sink_valid(sink_valid_sig) , // output  sink_valid_sig
    .reset_n(reset_n) // output  reset_n_sig
);

endmodule
```

```

`timescale 10ns/100ps
module testbench;

reg clk;

wire [31:0] fsin_o, fcos_o;
wire [31:0] real_power_sig, imag_power_sig;

initial
begin
clk=0;
end

mhr2154, yesterday • added fft testing files
always
begin
#10 clk=!clk;
end

wire reset_n;

nco nco_inst(
    .clk      (clk),          // clk.clk
    .reset_n  (reset_n),     // rst.reset_n
    .clken    (1'b1),        // in.clken
    .phi_inc_i (32'd41943040), // .phi_inc_i
    .fsin_o   (fsin_o),      // out.fsin_o
    .fcos_o   (fcos_o),      // .fcos_o
    .out_valid (out_valid)   // .out_valid
);

fft_wrapper fft_wrapper_inst
(
    .clk(clk) , // input  clk_sig
    .in_signal(fsin_o) , // input [31:0] in_signal_sig
    .real_power(real_power_sig) , // output [31:0] real_power_sig
    .imag_power(imag_power_sig) , // output [31:0] imag_power_sig
    .fft_source_sop(fft_source_sop_sig) , // output  fft_source_sop_sig
    .sink_sop(sink_sop_sig) , // output  sink_sop_sig
    .sink_eop(sink_eop_sig) , // output  sink_eop_sig
    .sink_valid(sink_valid_sig) , // output  sink_valid_sig
    .reset_n(reset_n) // output  reset_n_sig
);

endmodule

```

References

<https://www.youtube.com/watch?v=PjKIMXhxtTM>

<https://github.com/JentGent/pitch-shift/blob/main/audios.iptheynb>

https://en.wikipedia.org/wiki/Audio_time_stretching_and_pitch_scaling

https://en.wikipedia.org/wiki/Phase_vocoder

<https://www.guitarpitchshifter.com/algorithm.html>

https://cdrdv2-public.intel.com/667064/ug_fft-683374-667064.pdf&ved=2ahUKEwiR1YTHi5uFAxWCKlkEHUhhBCMqFnoECBMQAQ&usg=AOvVaw1FMRMwxHpeg2l39DPhM4wI

https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/18.1/University_Program_IP_Cores/Audio_Video/Audio.pdf

https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/18.1/University_Program_IP_Cores/Audio_Video/Audio_and_Video_Config.pdf

<http://www.cs.columbia.edu/~sedwards/classes/2008/4840/Wolfson-WM8731-audio-CODEC.pdf>

<https://www.intel.com/content/www/us/en/docs/programmable/683364/18-1/streaming-interfaces.html>

https://cdrdv2-public.intel.com/667068/mnl_avalon_spec-683091-667068.pdf

<https://github.com/BelaPlatform/bela-online-course/blob/master/lectures/lecture-20/code-examples/fft-pitchshift.zip>

https://www.youtube.com/watch?v=2p_-jbl6Dyc

https://www.vlsiuniverse.com/verilog-code-for-sine-cos-and-tan-cordic/#3_Sine_and_Cosine_Implementation_in_Verilog_using_CORDIC_algorithm