FPGA DJ

CSEE W4840: Embedded Systems
Spring 2024 Report

Written By:
Joy He, Oliver MacGregor, Harrison Riley, Joshua Zhou

1 Overview 3
System Architecture 3
2 Hardware 4
2.1 FFT/IFFT 4
The 'Butterfly' Unit 4
Pipelined FFT & IFFT 5
Frequency Domain Filtering 6
Audio Interface 7
Fifo Storage 7
Codec 8
3 Hardware-Software Interface 10
4 Software 11
4.1 Parsing WAV Files 11
4.2 Keyboard Input 11
4.3 Thread Communication 12
5 Validation & Testing 12
FFT/IFFT Block 12
References 14
Code 14
Hardware 14

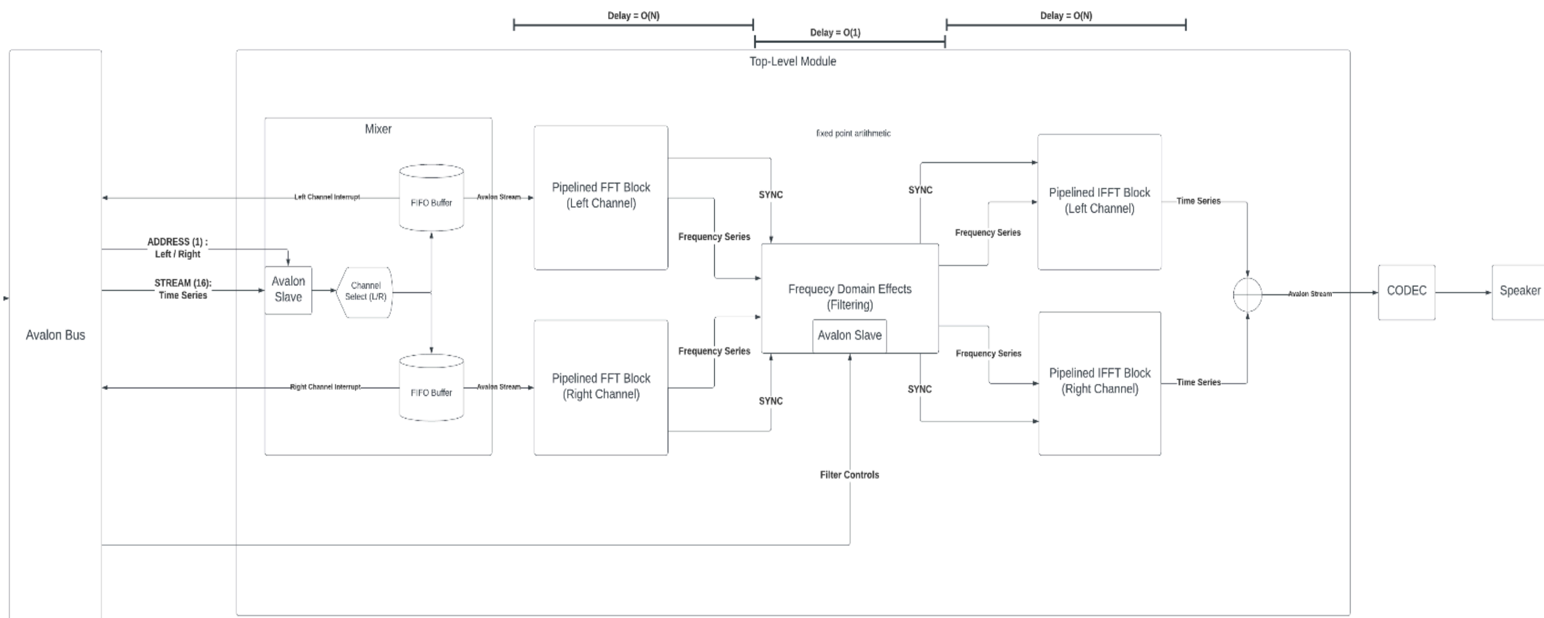
1 Overview

System Architecture

The main design challenge was to reconcile the irregular data transfer that is expected when streaming with a general-purpose computer running an operating system, and the regular synchronous data processing that is required for continuous real-time audio processing.

As samples are streamed by the software, through the Avalon bus, to the mixer module, samples are placed in each buffer according to the address they were written to. This continues until either of the buffers reaches a predetermined ‘almost full’ threshold, at which point an interrupt is asserted by the module, signaling to the software that it should stop streaming data for the associated channel until the interrupt is un-asserted. There are no timing requirements for when samples must be sent from software, other than the average rate at which samples are sent over a long period is 48 kHz (the sampling frequency of the audio channels), such that data is always available to enter the pipeline.

The pipelined FFT and IFFT blocks will only progress one sample of data through the pipeline when a ‘clock enable’ signal is asserted. This signal is driven by the ‘ready’ status of the CODEC’s Avalon streaming sink, allowing samples to progress only when there is space in its internal buffer for another sample. So long as there is always data in the input FIFO buffer, the pipeline will always be able to progress as required by the CODEC.

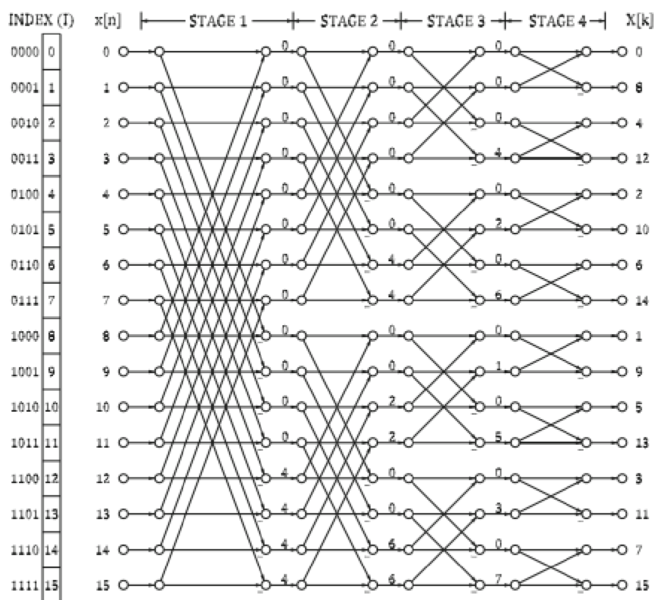


2 Hardware

The hardware in this project is primarily responsible for applying frequency domain effects to our tracks. To achieve this, windows of audio samples are streamed from the hardware, the frequency series is analyzed, filter windows are applied in the frequency domain, and the resulting frequency series is used to synthesize the filtered output time-series.

2.1 FFT/IFFT

The Fast Fourier Transform (FFT) and Inverse Fast Fourier Transform (IFFT) are algorithms that compute the Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT) more efficiently by using a divide and conquer algorithm. This approach uses memoization instead of repeated calculation of intermediate results, ultimately reducing the time complexity of computing the DFT from $O(n^2)$ to $O(n * \log(n))$.

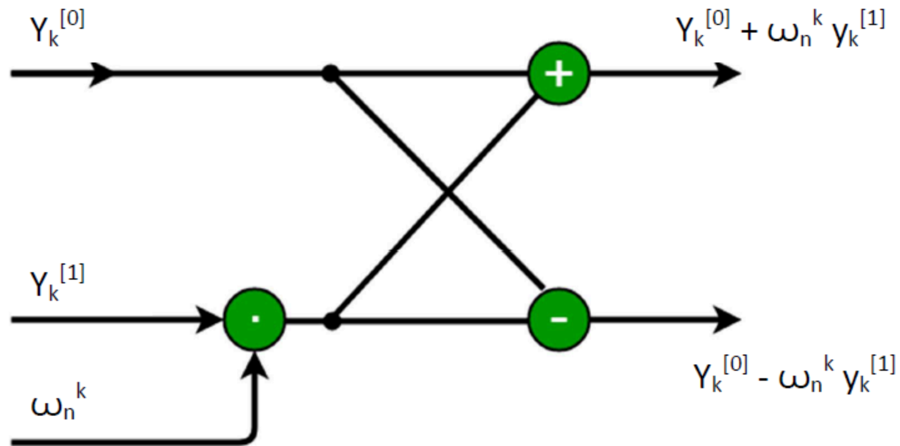


For many applications, it is common to apply the DFT to an entire time series, thus extracting data about the frequency components across the entire series. However, for analysis of a song, where the frequency components change throughout the song, a Short Time Fourier Transform (STFT) is more appropriate, which analyzes the frequency components over short intervals of the song.

The ‘Butterfly’ Unit

The most commonly used FFT implementation is the Cooley-Tukey algorithm. This is a divide-and-conquer algorithm that recursively breaks down a DFT into two smaller DFTs, and

ultimately into many DFTs with N = 2. This computation is often referred to as a 'butterfly' multiplication, due to its shape, as seen in figure XX . These are easily computed, as a simple cross multiplication with one of the roots of unity.

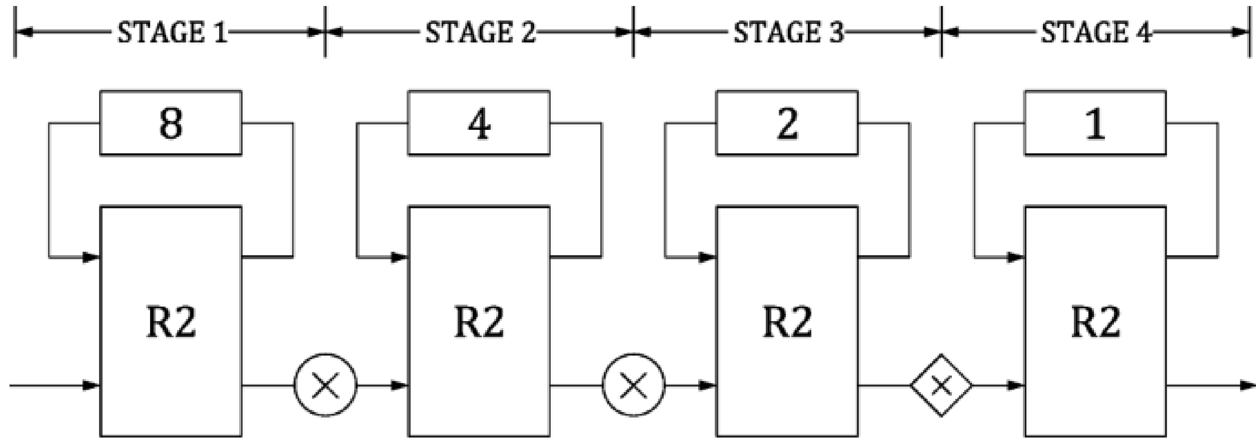


The butterfly unit is a simple, repeatable block that can be built easily and repeatedly with DSP units on an FPGA.

Pipelined FFT & IFFT

As previously established, the FFT is achieved by computing a 'butterfly' multiplication N * log(N) times, using intermediate results as the pairs of inputs to the computation. This is achieved simply when computed serially, where the order in which intermediate values are computed is unimportant, as long as all intermediate values are computed for each stage in their entirety before moving on to the next stage.

However, if some care is taken with the ordering of butterfly computations, multiple butterfly units can work in sequence, each working on a subsequent stage of the computation. This pipelined approach uses log(N) butterfly units. As such, the N *log(N) butterfly operations that need to be computed becomes O(N).



Each stage of the pipelined FFT is structurally the same but has a different length delay buffer and a different data set of roots of unity. These are passed as parameters into the blocks of each.

The FFT and IFFT blocks in the project are generated using an open-sourced pipelined FFT/IFFT block generator, which allows for the generation of FFT/IFFT blocks with customizable window size, sample bit width, DSP use, etc. The generated source code is integrated directly into the project and is included in the tar file.

Frequency Domain Filtering

In the frequency domain, applying a filter is achieved simply by multiplying the envelope element-wise with each bin in the frequency series. For our application, a Hanning window was applied.

Filters in our system are implemented based on ROM of size M that is initialized at the instantiation of the block. The contents and size of the ROM is generated ahead of time using a simple Python script (generate_filter.py), which encodes the values as a hex file. These are passed into the filter block as a parameter.

Each filter takes as an input an integer, C, between 0 and N/2, which is the cutoff bin (frequency) of the filter. The filter is synchronized with the output of the previous stages ‘pulse’ signal.

Out[i] = ROM[0] * In[i], i = [0, C]

After this pulse, the filter scales the first C frequencies with the first value in the filter ROM. After C samples, the next M samples are multiplied by the M gains stored in the ROM.

Out[i] = ROM[i - C] * In[i], i = [C, C + M)

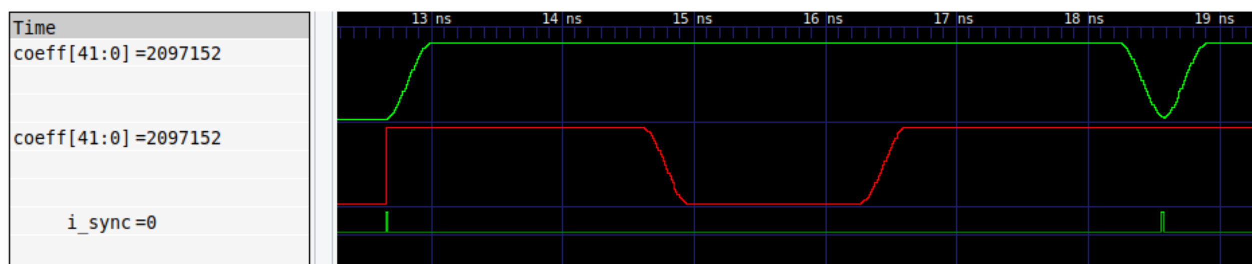
At the end of the transition window, all remaining values are multiplied by the final value in the ROM.

Out[i] = ROM[M - 1] * In[i], i = [C + M, N/2]

Because the FFT is symmetric for purely real samples, the output will be symmetrical. As such, this process is applied in reverse for all samples after N/2 samples have passed after the synchronization pulse.

Out[i] = Out[N - i - 1], i = [0, N]

The high-pass filters and low-pass filters are implemented using these blocks by applying roll-on and roll-off windows



Since the samples are represented as integers, and the roll-off period samples are scaled by a factor between 0 and 1, fixed point multiplication was used for the integer samples. For a sample of n bits, an intermediate value is created with the sample in the upper n bits, and zeroes in the lower n bits, with the fixed point being in the middle. This number is multiplied by the n-bit scaling factor that is read from ROM. The scaled value, rounding down, is then the upper n bits of this result.

Audio Interface

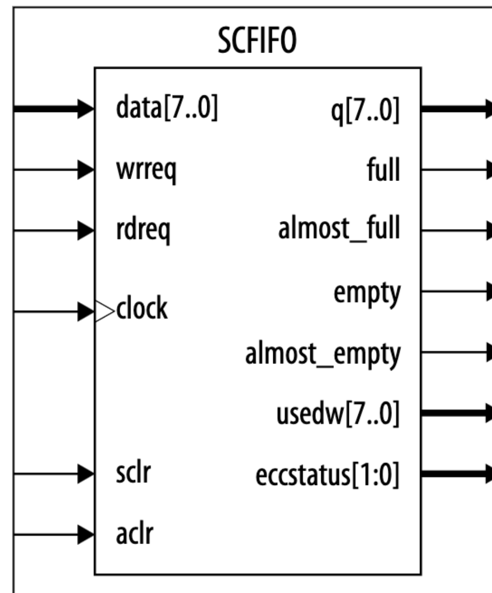
Fifo Storage

To ensure the song data gets passed to the Codec at the correct frequency while also allowing for the FFT module to process data as quickly as required, we pass audio samples for each song through three Altera SCFIFO's before passing them to the Codec to be played. This IP block was generated with Quartus and includes almost empty, almost full, empty, and full wires which allow us to deal with flow control. The mixer module receives audio samples from the Avalon bus and stores it in a fifo, sending interrupts to software whenever the fifo is almost full or almost empty so the software knows when to send or stop sending samples.

We connect the mixer module and the DJ module through an Avalon streaming interface, which allows for the transfer of the raw audio samples to DJ module's first fifo through ready, valid, and

data wires. If the first fifo in the DJ module has space for more samples, the ready wire of the Avalon streaming sink (on the DJ module side) will go high, allowing for the mixer module to set the valid wire high and send audio samples through the data wire.

If filtering is enabled the samples from this fifo will be sent through the FFT/Filtering blocks and the results of the final IFFT are placed in a second results fifo. If filtering is not enabled, the samples from the first fifo will go straight into the results fifo. The DJ module then coordinates the transfer of samples in the results fifo to the Codec.

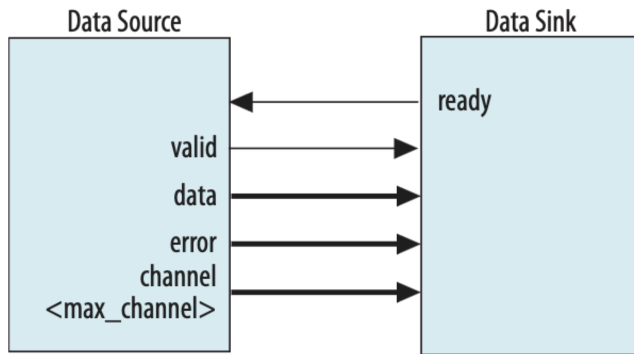


Single Clock Fifo

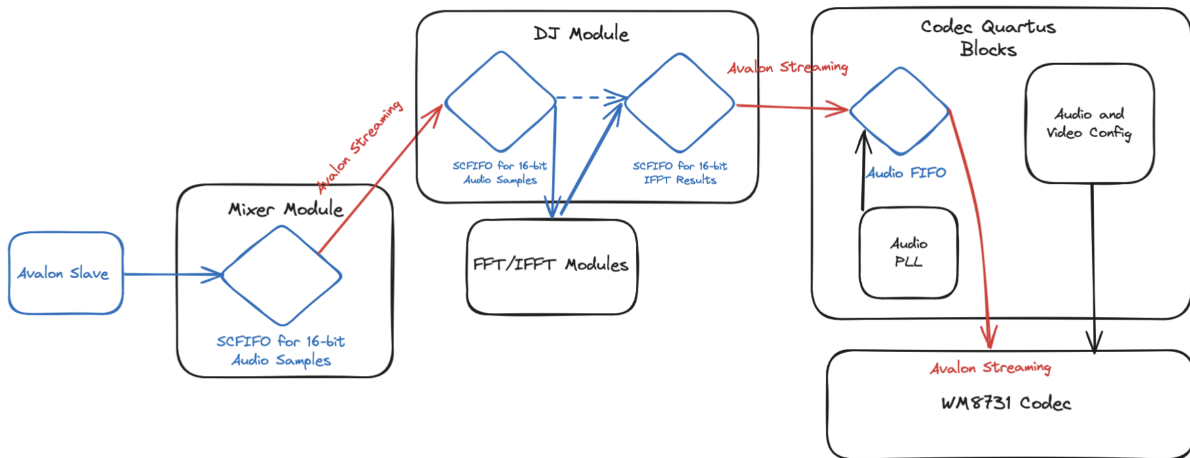
Codec

We used the Altera Audio, Audio and Video Config, and Audio PLL IP blocks to interface with the WM8731 Codec. The Audio block provides an Avalon Streaming Sink through which we transfer audio data to the Codec. The Audio and Video Config block helps us configure the Codec to have a 48 kHz sampling rate with left-justified, 16-bit samples. The Audio PLL block creates a 12.288 MHz clock from the reference clock of 50 MHz to act as a clock divider, such that the Codec can operate at the correct clock frequency.

The Avalon Streaming Source is in the DJ module. When the ready wire of the streaming sink in the audio block goes high, we count up to a threshold of 1000 clock cycles (to slow the rate of data transfer to the 48kHz frequency of the Codec fifo). Once this threshold has been reached, we set the valid flag high and send a read request to the results fifo to acquire either filtered audio samples or the raw audio samples from software. The data read from this fifo is then sent through the data wire of the Avalon streaming sink/source, which is connected to the DAC output wire of the Codec, allowing music to be played.



Avalon Streaming Interface



Audio Architecture

3 Hardware-Software Interface

System Contents | Address Map | Interconnect Requirements

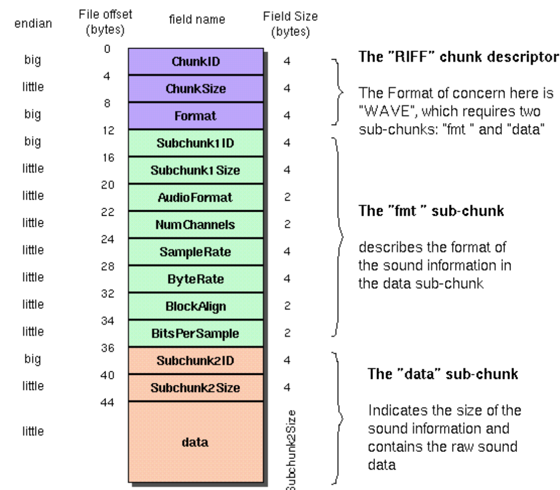
System: soc_system **Path:** clk_0

Name	Description	Export	Clock	Base	End	IRQ
clk_0	Clock Source					
clk_in	Clock Input	clk	exported			
clk_in_reset	Reset Input	reset				
clk	Clock Output	Double-click to	clk_0			
clk_reset	Reset Output	Double-click to				
hps_0	Arria V/Cyclone V Hard Proce...					
h2f_user1_clock	Clock Output	Double-click to	hps_0_h2...			
memory	Conduit	hps_ddr3				
hps_io	Conduit	hps				
h2f_reset	Reset Output	Double-click to				
h2f_axi_clock	Clock Input	Double-click to	clk_0			
h2f_axi_master	AXI Master	Double-click to	[h2f_axi_...			
f2h_axi_clock	Clock Input	Double-click to	clk_0			
f2h_axi_slave	AXI Slave	Double-click to	[f2h_axi_...			
h2f_lw_axi_clock	Clock Input	Double-click to	clk_0			
h2f_lw_axi_master	AXI Master	Double-click to	[h2f_lw_a...			
f2h_irq0	Interrupt Receiver	Double-click to		IRQ 0	IRQ 31	
f2h_irq1	Interrupt Receiver	Double-click to		IRQ 0	IRQ 31	
mixer_0	Mixer					
clock	Clock Input	Double-click to	clk_0			
reset	Reset Input	Double-click to	[clock]			
mixer	Conduit	mixer	[clock]			
avalon_slave_0	Avalon Memory Mapped Slave	Double-click to	[clock]	0x0000_0000	0x0000_000f	
avalon_streaming_source_right	Avalon Streaming Source	Double-click to	[clock]			
avalon_streaming_source_left	Avalon Streaming Source	Double-click to	[clock]			
interrupt_sender	Interrupt Sender	Double-click to	[clock]			0
DJ_0	DJ					
left_fifo_source	Avalon Streaming Sink	Double-click to	[clk]			
right_fifo_source	Avalon Streaming Sink	Double-click to	[clk]			
reset	Reset Input	Double-click to	[clk]			
clk	Clock Input	Double-click to	clk_0			
codec_left_fifo_sink	Avalon Streaming Source	Double-click to	[clk]			
codec_right_fifo_sink	Avalon Streaming Source	Double-click to	[clk]			
effects	Avalon Memory Mapped Slave	Double-click to	[clk]	0x0000_0080	0x0000_00bf	
audio_and_video_config_0	Audio and Video Config					
clk	Clock Input	Double-click to	clk_0			
reset	Reset Input	Double-click to	[clk]			
avalon_av_config_slave	Avalon Memory Mapped Slave	Double-click to	[clk]	0x0000_0020	0x0000_002f	
external_interface	Conduit	audio_and_video_c...				
audio_0	Audio					
clk	Clock Input	Double-click to	clk_0			
reset	Reset Input	Double-click to	[clk]			
avalon_left_channel_source	Avalon Streaming Source	Double-click to	[clk]			
avalon_right_channel_source	Avalon Streaming Source	Double-click to	[clk]			
avalon_left_channel_sink	Avalon Streaming Sink	Double-click to	[clk]			
avalon_right_channel_sink	Avalon Streaming Sink	Double-click to	[clk]			
external_interface	Conduit	audio_0_external_i...				
audio_pll_0	Audio Clock for DE-series Boa...					
ref_clk	Clock Input	Double-click to	clk_0			
ref_reset	Reset Input	Double-click to				
audio_clk	Clock Output	audio_pll_0_audio_...	audio_pll...			
reset_source	Reset Output	Double-click to				

We implemented two device drivers. The control driver allows software to tell the FFT module whether to apply filtering and, if so, what values to use for the low and high pass cutoffs. The stream driver lets software write song data to the mixer module. Because the write speed of software is unpredictable, we utilize a combination of a FIFO and interrupts to ensure that there is always data available to be passed to the FFT module (note that this FIFO is separate from the one in the FFT module described above). Every time the driver writes data, it checks a ready flag (which is initially set to true). When the FIFO has available room, the driver simply allows data to be written from software. When the number of used words in the FIFO exceeds an “almost full” threshold, an interrupt is asserted and the value 0xf (for full) is written to readdata. The driver detects the interrupt and executes an interrupt handler, which reads the ‘f’ and unsets the ready flag. Now, if the software attempts to write data, it is put to sleep on a wait queue. When the FIFO’s used word count falls below an “almost empty” threshold, an interrupt is asserted again and the value 0xe (for empty) is written to readdata. The driver’s interrupt handler reads the ‘e’, sets the ready flag, and wakes up any processes on the wait queue. The cycle repeats.

4 Software

4.1 Parsing WAV Files



The software is responsible for reading the user-specified WAV files and processing keyboard input. WAV files are composed of multiple subchunks; we are primarily concerned with parsing the fmt subchunk (which contains metadata like the number of channels and the byte rate) and the data subchunk (which contains the actual sound data). We check that the metadata matches what we expect. If so, we begin passing song data to the stream device driver using ioctl calls. This is accomplished simply by calling ioctl in a loop; the device driver takes care of timing concerns.

4.2 Keyboard Input

In a separate thread, we listen for keyboard input using the usbkeyboard library. We support the following commands:

1. Toggle between play and pause for the first song (Fn1), second song (Fn2), or both songs (Fn3)
2. Rewind or fast forward a specified number of seconds
    a. A and D to rewind or fast forward the first song
    b. Left arrow and Right arrow to rewind or fast forward the second song
3. Increase or decrease the gain
    a. W and S to increase or decrease the gain for the first song
    b. Up arrow and Down arrow to increase or decrease the gain for the second song
4. Toggle between echo and no echo for the first song (E) or the second song (R)
5. Toggle between looping (i.e., replaying the song after it finishes) and no looping for the first song (O) or the second song (P)

6. Toggle between filtering and no filtering for the first song (F) or the second song (G)
7. Set the values for the LPF and HPF cutoffs
    a. H and J to set the LPF and HPF cutoffs for the first song
    b. K and L to set the LPF and HPF cutoffs for the second song
8. Airhorn! Space bar.

Some commands require additional input, such as rewinding/fast forwarding (need the number of seconds) and LPF/HPF (need the cutoff value). For these commands, after pressing the associated key, the user inputs the desired number. They then press Enter to finish the command.

4.3 Thread Communication

The keyboard thread and main thread (which makes the ioctl calls) communicate using global variables. The keyboard thread writes to these variables, and the main thread reads from them. The values that the main thread writes to the stream device depend on the state of the global variables. For instance, if the keyboard thread indicates that the user paused a song, the main thread writes zeros for that song's data. Moreover, if the keyboard thread indicates that the user requested filtering to occur, the main thread executes ioctl calls to the control driver to signal it to begin filtering and pass it the user-specified cutoff values.

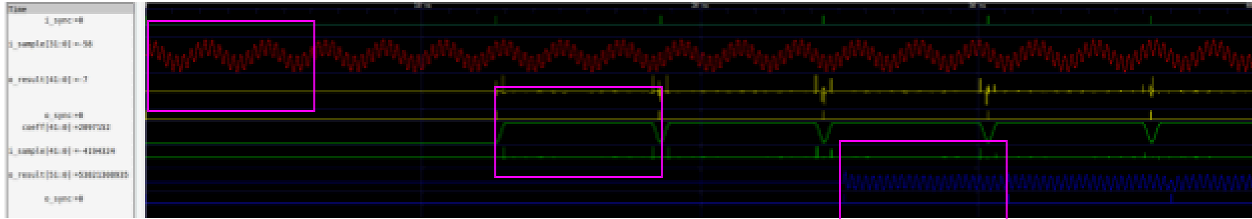
5 Validation & Testing

To validate the correctness of our system before final validation on hardware, several simulations and test benches were developed.

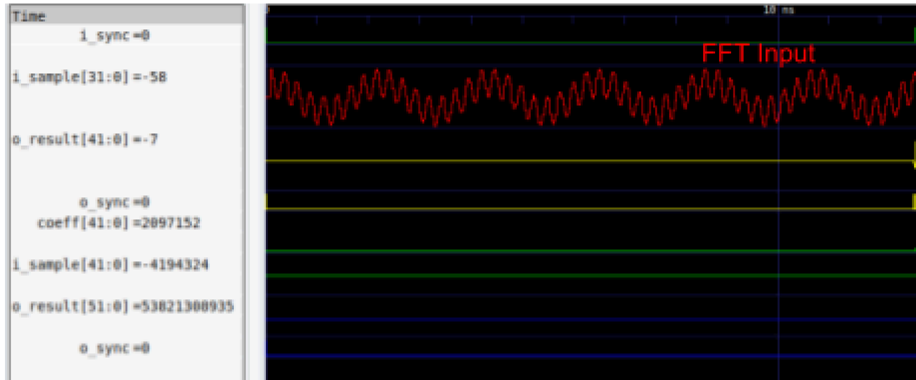
FFT/IFFT Block

To initially validate the functionality of the FFT and IFFT block, a simple analysis/filter/synthesis procedure was performed. For this test, the input was two superimposed sine waves, one with a frequency about an order of magnitude larger than the other. The goal of the test is to analyze the frequency components, apply a high-pass filter, and to resynthesize the higher-frequency sine wave, thus filtering out the lower-frequency component.
The figure below shows the important signals from this test. In subfigure (a), the first window is shown at several stages as it passes through the pipeline. The pink boxes indicate the relative delay for the first window before the FFT stage, after the FFT stage, and after the IFFT stage.

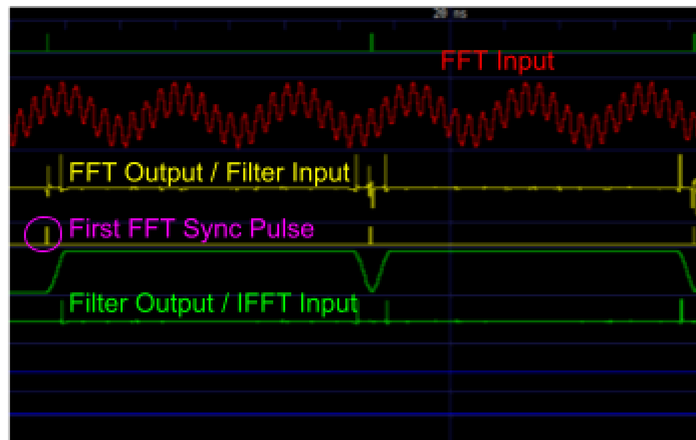
Subfigure (b) shows the initial window input to the system. Some fixed delay later, subfigure (c) shows the results of the FFT analysis, with two peaks seen in the spectrum near the start of the window, indicated by the rising SYNC pulse. Following this, the peak of the lower frequency component is filtered out, leaving only the higher frequency component. Finally, (d) shows the synthesis of the output waveform using the IFFT block, which contains only the high frequency component.



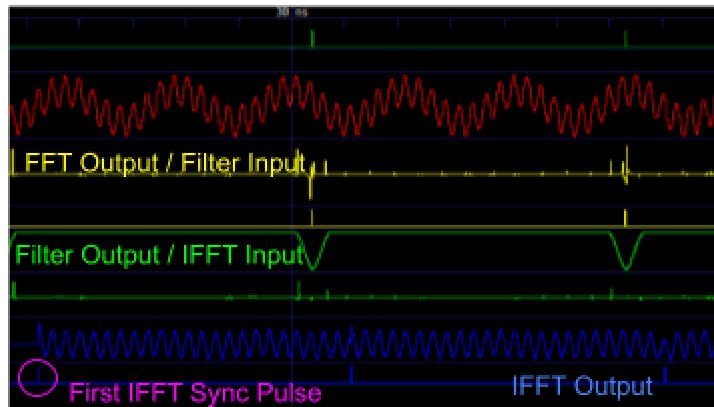
a. Experiment Overview



b. Input Waveform



c. analysis of input wave + frequency domain filter



d. Synthesis of filtered wave

References

- https://cdn.sparkfun.com/datasheets/Dev/Arduino/Shields/WolfsonWM8731.pdf
- https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/18.1/University_Program_IP_Cores/Audio_Video/Audio_and_Video_Config.pdf
- https://www-ug.eecg.toronto.edu/msl/nios_devices/datasheets/Audio.pdf
- http://ridl.cfd.rit.edu/products/manuals/altera/User%20Guides%20and%20AppNotes/FIFO/ug_fifo.pdf
- https://www.intel.com/content/www/us/en/docs/programmable/683364/18-1/streaming-interfaces.html
- https://www.cl.cam.ac.uk/teaching/1920/ECAD+Arch/optional-tonegen.html
- https://zipcpu.com/dsp/2018/10/02/fft.html
-

Code

Hardware

Top-level:
- mixer.sv

Frequency Domain Effects:
- DJ.sv

FFT/IFFT Common
- fftstage.sv
- laststage.sv
- qtrstage.sv
- hwbfly.sv
- bitreverse.sv
- convround.sv

FFT
- fftmain.sv
- cmem_8.hex
- cmem_16.hex
- cmem_.32hex
- cmem_.64hex
- cmem_.128hex
- cmem_.256hex
- cmem_.512hex

IFFT

- ifftmain.sv
- icmem_8.hex
- icmem_16.hex
- icmem_.32hex
- icmem_.64hex
- icmem_.128hex
- icmem_.256hex
- icmem_.512hex

Filters:

- filter.sv
- hpf.hex
- lpf.hex

Other:

- Gen_filter.py

Software

Kernel Modules:

- mixer.c

Userspace Program:

- bitsandbeats.c

Other:

- usbkeyboard.c

Github: https://github.com/20joshuaz/Embedded-Systems/tree/integration-cleanup/final-project

Roles

Oliver:

- FFT / IFFT Block Generation + Testing
- Frequency Domain FIltering Blocks
- Pipeline Construction + Synchronization

Joy:

- Audio IP block integration
- Fifo implementations/synchronizations

Joshua
- Device driver implementation
- Software elements: WAV file parsing and interrupt handling

Harrison
- Top-level system design
- Audio IP block integration

Code

Hardware

mixer.sv

```
Unset
/*
 * Avalon memory-mapped peripheral that generates VGA
 *
 * Stephen A. Edwards
 * Columbia University
 */

module mixer(
    input logic clk,
    input logic reset,
    input logic [31:0]  writedata,
    input logic     write,

    input logic read,
    input logic chipselect,
    input logic [1:0] address,

    output logic [31:0] readdata,
```

```
output logic [7:0] VGA_R, VGA_G, VGA_B,
output logic    VGA_CLK, VGA_HS, VGA_VS,
          VGA_BLANK_n,
output logic VGA_SYNC_n,

input left_ready, right_ready,
output [15:0] left_data, right_data,
output logic left_valid, right_valid,

output logic irq);

logic [31:0] fifo_stream;
logic async_clr = 0;
logic read_req = 0;
logic empty;
logic full;
logic almost_empty;
logic almost_full;
logic [9:0] used_words;
reg [11:0] counter;

//store song samples
fifo_interrupt song_data1(.aclr(async_clr), .clock(clk), .data(writedata),
    .rdreq(read_req), .wrreq(chipselect && write),
    .almost_empty(almost_empty), .almost_full(almost_full),
    .empty(empty), .full(full), .q(fifo_stream), .usedw(used_words));

enum logic [1:0] {EMPTY, FULL, FILLING} state;
    always_ff @(posedge clk) begin
    if (reset) begin
          counter <= 0;
          async_clr <= 1'b1;
          left_valid <= 0;
          right_valid <= 0;
          read_req <= 0;
          irq <= 1;
          state <= EMPTY;
    end else begin
          //handle writes from software
          case (state)
          EMPTY: begin
                 readdata <= 16'he;
                 if (chipselect && read) begin
```

```
                    irq <= 0;
                end
                if (!almost_empty) begin
                    state <= FILLING;
                end
            end
            FILLING: begin
                irq <= 0;
                if (almost_empty) begin
                    irq <= 1;
                    state <= EMPTY;
                end else if (almost_full) begin
                    irq <= 1;
                    state <= FULL;
                end
            end
            FULL: begin
                readdata <= 16'hf;
                if (chipselect && read) begin
                    irq <= 0;
                end
                if (!almost_full) begin
                    state <= FILLING;
                end
            end
            default: ;
            endcase
            //send data to fft block
            if (!empty && left_ready == 1 && right_ready == 1) begin
                left_valid <= 1;
                right_valid <= 1;
                read_req <= 1;
                left_data <= (fifo_stream[31:16]);
                right_data <= (fifo_stream[15:0]);
                async_clr <= 1'b0;
            end else begin
                left_valid <= 0;
                right_valid <= 0;
                read_req <= 0;
                async_clr <= 1'b0;
            end
        end
    end
endmodule
```

DJ.sv

```
Unset
module    DJ #(
    parameter FFT_IWIDTH = 16,
    parameter FFT_OWIDTH = 21,
    parameter IFFT_IWIDTH = 21,
    parameter IFFT_OWIDTH = 26,
    parameter LGWIDTH = 9
)(
    input clk,  //  Master Clock
    input reset,

    // Effects Controller Avalon Slave
    input     chipselect,
    input [7:0]    effects_writedata,
    input     effects_write,
    output [7:0]    effects_readdata,
    input     effects_read,
    input [5:0]    effects_address,


    // Left Channel Input Buffer
    input signed [FFT_IWIDTH - 1:0]    left_fifo_source_stream, // to fft block
    input                  left_fifo_source_valid,
    output                 left_fifo_source_ready,


    // Right Channel Input Buffer
    input signed [FFT_IWIDTH - 1:0]    right_fifo_source_stream, // to fft
block
    input                  right_fifo_source_valid,
    output                 right_fifo_source_ready,

    //

    // Left Channel Output Codec Buffer
    output signed [FFT_IWIDTH - 1:0]    codec_left_fifo_sink_stream, // from
fft bus
```

```
    output                        codec_left_fifo_sink_valid,
    input                         codec_left_fifo_sink_ready,


    // Right Channel Output Codec Buffer
    output signed [FFT_IWIDTH - 1:0]    codec_right_fifo_sink_stream, // from
fft bus
    output                        codec_right_fifo_sink_valid,
    input                         codec_right_fifo_sink_ready
);
    /******************
       LEFT DECLARATIONS
    ********************/

    // FFT Block
    wire                    left_fft_ce;
    wire [(2 * FFT_OWIDTH - 1):0]    left_fft_o_result;
    wire                    left_fft_o_sync;

    // IFFT Block
    wire                    left_ifft_i_ce;
    wire [(2 * IFFT_IWIDTH - 1):0]    left_ifft_i_sample;
    reg                     left_ifft_i_reset;
    wire [2 * IFFT_OWIDTH - 1:0]    left_ifft_o_result;
    wire                    left_ifft_o_sync;

    // LPF Block
    reg [(LGWIDTH - 1):0]         left_lpf_cutoff;
    wire                    left_lpf_o_ce, left_lpf_o_sync;
    wire [(2 * IFFT_IWIDTH - 1):0]    left_lpf_o_sample;

    // HPF Block
    reg [(LGWIDTH - 1):0]         left_hpf_cutoff;
    wire                    left_hpf_o_ce, left_hpf_o_sync;
    wire [(2 * IFFT_IWIDTH - 1):0]    left_hpf_o_sample;


    /******************
       RIGHT DECLARATIONS
    ********************/
    /*
    // FFT Block
    wire                    right_fft_ce;
    wire [(2 * FFT_OWIDTH - 1):0]    right_fft_o_result;
```

```
wire                    right_fft_o_sync;

// IFFT Block
wire                    right_ifft_i_ce;
wire [(2 * IFFT_IWIDTH - 1):0]    right_ifft_i_sample;
reg                     right_ifft_i_reset;
wire [2 * IFFT_OWIDTH - 1:0]    right_ifft_o_result;
wire                    right_ifft_o_sync;

// LPF Block
reg [(LGWIDTH):0]       right_lpf_cutoff;
wire                    right_lpf_o_ce, right_lpf_o_sync;
wire [(2 * IFFT_IWIDTH - 1):0]    right_lpf_o_sample;
*/

logic do_left_filtering;
// logic do_right_filtering;

/******************
   MIXER LOGIC
********************/

logic async_clr;

//fifo wires for data from mixer module (unfiltered)
logic left_song_read_req;
logic right_song_read_req;

logic left_song_almost_empty;
logic right_song_almost_empty;

logic left_song_almost_full;
logic right_song_almost_full;

logic left_song_empty;
logic right_song_empty;

logic left_song_full;
logic right_song_full;

logic [10:0] left_song_usedw;
logic [10:0] right_song_usedw;

logic [15:0] left_song_fifo_stream;
```

```
    logic [15:0] right_song_fifo_stream;


    fifo_interrupt left_song_data(.aclr(async_clr), .clock(clk),
.data(left_fifo_source_stream),
        .rdreq(left_song_read_req), .wrreq(left_fifo_source_valid &&
right_fifo_source_valid),
        .almost_empty(left_song_almost_empty),
.almost_full(left_song_almost_full),
        .empty(left_song_empty), .full(left_song_full),
.q(left_song_fifo_stream), .usedw(left_song_usedw));

    fifo_interrupt right_song_data(.aclr(async_clr), .clock(clk),
.data(right_fifo_source_stream),
        .rdreq(right_song_read_req), .wrreq(left_fifo_source_valid &&
right_fifo_source_valid),
        .almost_empty(right_song_almost_empty),
.almost_full(right_song_almost_full),
        .empty(right_song_empty), .full(right_song_full),
.q(right_song_fifo_stream), .usedw(right_song_usedw));

    reg [10:0] counter;

    //write data from fifos to codec
    always @(posedge clk) begin
        if (reset) begin
               async_clr <= 1;
        end
        if (codec_left_fifo_sink_ready && codec_right_fifo_sink_ready && counter
< 1020) begin
               codec_left_fifo_sink_valid <= 0;
               codec_right_fifo_sink_valid <= 0;
               left_result_read_req <= 0;
               right_result_read_req <= 0;
               counter <= counter + 1;
               async_clr <= 0;
        end else if (!left_result_empty && !right_result_empty &&
codec_left_fifo_sink_ready && codec_right_fifo_sink_ready && counter == 1020)
begin
               codec_left_fifo_sink_valid <= 1;
               codec_right_fifo_sink_valid <= 1;
               left_result_read_req <= 1;
               right_result_read_req <= 1;
               counter <= 0;
```

```
            async_clr <= 0;
        end else begin
            codec_left_fifo_sink_valid <= 0;
            codec_right_fifo_sink_valid <= 0;
            left_result_read_req <= 0;
            right_result_read_req <= 0;
            async_clr <= 0;
        end
    end


    /******************
        LEFT LOGIC
    ********************/


    // only allow pipeline to proceed if values are available at the input
    // AND the output buffer is not at risk of overflowing
    assign left_fft_ce = do_left_filtering && !left_song_empty &&
!left_result_almost_full;
    assign left_song_read_req = !left_song_empty && !left_result_almost_full;
    //allow mixer module to write data to song fifo
    assign left_fifo_source_ready = !left_song_almost_full;

    //codec left DAC output stream
    assign codec_left_fifo_sink_stream = left_result_stream +
right_result_stream;

    wire    left_w_syncd;
    reg    left_r_syncd;

    // Synchronize and latch FFT windows with IFFT
    always @(posedge clk) begin
        if (reset)
            left_r_syncd <= 1'b0;
        else
            left_r_syncd <= left_r_syncd || left_fft_o_sync;
    end

    assign    left_w_syncd = left_r_syncd || left_fft_o_sync;

    // Generate FFT Block for left song
    fftmain left_fft(
        .i_clk(clk),
```

```
        .i_reset(reset),
        .i_ce(left_fft_ce),
        .i_sample({left_song_fifo_stream, 16'b0}),
        .o_result(left_fft_o_result),
        .o_sync(left_fft_o_sync));

    // Generate LPF Block
    filter #(
        .LGWIDTH(LGWIDTH),
        .LGFILTWIDTH(5),
        .COEFFILE("lpf.hex")
    ) left_lpf (
        .i_cutoff(left_lpf_cutoff),
        .i_clk(clk),
        .i_ce(((left_fft_ce)&&(left_w_syncd))),
        .i_sample(left_fft_o_result),
        .i_sync(left_fft_o_sync),
        .o_ce(left_lpf_o_ce),
        .o_sample(left_lpf_o_sample),
        .o_sync(left_lpf_o_sync));

    filter #(
        .LGWIDTH(LGWIDTH),
        .LGFILTWIDTH(5),
        .COEFFILE("hpf.hex")
    ) left_hpf (
        .i_cutoff(left_hpf_cutoff),
        .i_clk(clk),
        .i_ce(left_lpf_o_ce),
        .i_sample(left_lpf_o_sample),
        .i_sync(left_lpf_o_sync),
        .o_ce(left_hpf_o_ce),
        .o_sample(left_hpf_o_sample),
        .o_sync(left_hpf_o_sync));

    assign left_ifft_i_ce = left_hpf_o_ce;
    assign left_ifft_i_sample = left_hpf_o_sample;

    // Generate IFFT Block
    ifftmain left_ifft(
        .i_clk(clk),
        .i_reset(reset),
        .i_ce(left_ifft_i_ce),
        .i_sample(left_ifft_i_sample),
```

```
        .o_result(left_ifft_o_result),
        .o_sync(left_ifft_o_sync));

    logic left_samples_ready;

    //check when first sample is ready from the ifft
    always @(posedge clk) begin
        if (reset) begin
               left_samples_ready <= 0;
        end else if(left_ifft_o_sync) begin
               left_samples_ready <= 1;
        end
    end

    //wires for fifos that store left ifft results
    logic left_result_read_req;
    logic left_result_almost_empty;
    logic left_result_almost_full;
    logic left_result_empty;
    logic left_result_full;
    logic [15:0] left_result_stream;
    logic [10:0] left_usedw;

    logic left_result_ready;
    assign left_result_ready = left_samples_ready && left_ifft_i_ce;

    fifo_interrupt left_result(.aclr(async_clr), .clock(clk),
        .data(do_left_filtering ? left_ifft_o_result[FFT_IWIDTH + IFFT_OWIDTH -
1: IFFT_OWIDTH] : left_song_fifo_stream),
        .rdreq(left_result_read_req), .wrreq(do_left_filtering ?
left_result_ready : left_song_read_req),
        .almost_empty(left_result_almost_empty),
.almost_full(left_result_almost_full),
        .empty(left_result_empty), .full(left_result_full),
.q(left_result_stream), .usedw(left_usedw));



    /******************
        RIGHT LOGIC
    *******************/


    // only allow pipeline to proceed if values are available at the input
```

```
    // AND the output buffer is not at risk of overflowing
    // assign right_fft_ce = do_right_filtering && !right_song_empty &&
!right_result_almost_full;
    assign right_song_read_req = !right_song_empty &&
!right_result_almost_full;

    assign right_fifo_source_ready = !right_song_almost_full;

    assign codec_right_fifo_sink_stream =  left_result_stream  +
right_result_stream;

    /*
    wire    right_w_syncd;
    reg    right_r_syncd;

    // Synchronize and latch FFT windows with IFFT
    always @(posedge clk) begin
        if (reset)
            right_r_syncd <= 1'b0;
        else
            right_r_syncd <= right_r_syncd || right_fft_o_sync;
    end

    assign    right_w_syncd = right_r_syncd || right_fft_o_sync;


    // Generate FFT Block for right song
    fftmain right_fft(
        .i_clk(clk),
        .i_reset(reset),
        .i_ce(right_fft_ce),
        .i_sample({right_song_fifo_stream, 16'b0}),
        .o_result(right_fft_o_result),
        .o_sync(right_fft_o_sync));

    assign right_lpf_cutoff = 10'd40;

    // Generate LPF Block
    filter #(
        .LGWIDTH(LGWIDTH),
        .LGFILTWIDTH(5),
        .COEFFILE("lpf.hex")
    ) left_lpf (
        .i_cutoff(right_lpf_cutoff),
```

```
    .i_clk(clk),
    .i_ce(((right_fft_ce)&&(right_w_syncd))),
    .i_sample(right_fft_o_result),
    .i_sync(right_fft_o_sync),
    .o_ce(right_lpf_o_ce),
    .o_sample(right_lpf_o_sample),
    .o_sync(right_lpf_o_sync));

assign right_hpf_cutoff = 10'd10;

filter #(
    .LGWIDTH(LGWIDTH),
    .LGFILTWIDTH(5),
    .COEFFILE("hpf.hex")
) right_hpf (
    .i_cutoff(right_hpf_cutoff),
    .i_clk(clk),
    .i_ce(right_lpf_o_ce),
    .i_sample(right_lpf_o_sample),
    .i_sync(right_lpf_o_sync),
    .o_ce(right_hpf_o_ce),
    .o_sample(right_hpf_o_sample),
    .o_sync(right_hpf_o_sync));

assign right_ifft_i_ce = right_hpf_o_ce;
assign right_ifft_i_sample = right_hpf_o_sample;

// Generate IFFT Block
ifftmain right_ifft(
    .i_clk(clk),
    .i_reset(reset),
    .i_ce(right_ifft_i_ce),
    .i_sample(right_ifft_i_sample),
    .o_result(right_ifft_o_result),
    .o_sync(right_ifft_o_sync));

logic right_samples_ready;

always @(posedge clk) begin
    if (reset) begin
           right_samples_ready <= 0;
    end else if(right_ifft_o_sync) begin
           right_samples_ready <= 1;
    end
```

```
    end
    */

    logic right_result_read_req;
    logic right_result_almost_empty;
    logic right_result_almost_full;
    logic right_result_empty;
    logic right_result_full;
    logic [15:0] right_result_stream;
    logic [10:0] right_usedw;

    logic right_result_ready;
    // assign right_result_ready = right_samples_ready && right_ifft_i_ce;

    fifo_interrupt right_result(.aclr(async_clr), .clock(clk),
.data(right_song_fifo_stream),
        .rdreq(right_result_read_req), .wrreq(right_song_read_req),
        .almost_empty(right_result_almost_empty),
.almost_full(right_result_almost_full),
        .empty(right_result_empty), .full(right_result_full),
.q(right_result_stream), .usedw(right_usedw));

    /*reg [2:0] ctr;

    wire clk_div;
    assign clk_div = ctr[2];

    always @(posedge clk) ctr <= ctr + 1;*/


    // assign codec_left_fifo_sink_ready = (ctr == 0) ? 1'b1 : 1'b0;

    // assign codec_right_fifo_source_ready = codec_left_fifo_source_ready;

    /*********************
        EFFECTS
    *********************/

    always @(posedge clk) begin
        if (reset) begin
            left_lpf_cutoff <= 9'd40;
            left_hpf_cutoff <= 9'd10;
            do_left_filtering <= 0;
        end else if (chipselect && effects_write) begin
```

```
            case (effects_address)
            6'd0: do_left_filtering <= effects_writedata;
            6'd1: left_lpf_cutoff <= effects_writedata;
            6'd2: left_hpf_cutoff <= effects_writedata;
            default: ;
            endcase
        end
    end
endmodule
```

mixer.sv

```
Unset
module filter #(

             parameter WIDTH=21,
             parameter LGWIDTH=9,
             parameter LGP1=LGWIDTH+1,
             parameter  LGFILTWIDTH=3,
             parameter COEFFILE="hpf.hex"

       ) (
             // {{{

             input wire   [(LGWIDTH - 1):0] i_cutoff,

             input  wire   i_clk, i_ce,
        input wire   [(2*WIDTH-1):0]     i_sample,
             input  wire   i_sync,

             output  reg   o_ce,
             output reg       [(2*WIDTH-1):0] o_sample,
          output    reg                        o_sync
             // }}}
       );

       reg[(LGWIDTH):0] bin_idx;

       wire signed [WIDTH - 1: 0] sample_re, sample_im;
       reg signed [WIDTH - 1: 0] out_re, out_im;
```

```
    wire [LGFILTWIDTH: 0] filter_width;
    assign filter_width = 'b1 << LGFILTWIDTH;

    wire signed [2 * WIDTH - 1: 0] fixed_point_sample_re;
    wire signed [2 * WIDTH - 1: 0] fixed_point_sample_im;

    wire [LGWIDTH - 1: 0]      filter_idx;


    reg    signed [(2*WIDTH-1):0]     coeff;

    wire [LGWIDTH - 1:0] symm_bin_idx;


    // Read in coefficients
  reg      [(2*WIDTH-1):0]     cmem [0:((1<<LGWIDTH)-1)];
    initial      $readmemh(COEFFILE,cmem);


    assign sample_re = i_sample[2 * WIDTH - 1 : WIDTH];
    assign sample_im = i_sample[WIDTH - 1: 0];


    // Symmetrical index for real series DFT
    assign symm_bin_idx = (bin_idx) < ('b1 << (LGWIDTH - 1)) ? (filter_width
+ bin_idx ): ('b1 << (LGWIDTH)) - (bin_idx - filter_width);


    // keep filter index within bounds of cmem
    assign filter_idx = symm_bin_idx > i_cutoff ? (symm_bin_idx < i_cutoff +
filter_width ? symm_bin_idx - i_cutoff : filter_width - 1) : 0;

    // assign outputs
    assign fixed_point_sample_re = sample_re * coeff;
    assign fixed_point_sample_im = sample_im * coeff ;

    // assign o_sample[2 * WIDTH - 1 : WIDTH] = fixed_point_sample_re[2*WIDTH
- 1: WIDTH];
    // assign o_sample[WIDTH - 1: 0] = fixed_point_sample_im[2*WIDTH - 1:
WIDTH];


    always @(posedge i_clk) begin
```

```
            o_ce <= i_ce;
            o_sync <= i_sync;


            if (i_ce) begin
                  if (i_sync) begin
                        bin_idx <= 1 ;
                        coeff <= cmem[filter_width > i_cutoff ? (0 < i_cutoff
? filter_width - i_cutoff : filter_width - 1) : 0];

                  end

                  else begin
                        bin_idx <= bin_idx + 1;
                        coeff <= cmem[filter_idx];

                  end

                  o_sample[2 * WIDTH - 1 : WIDTH] <=
fixed_point_sample_re[2*WIDTH - 1: WIDTH];
                  o_sample[WIDTH - 1: 0] <= fixed_point_sample_im[2*WIDTH -
1: WIDTH];



            end

      end


endmodule
```

fftmain.sv

```
Unset

module fftmain #(
      parameter IWIDTH=16,
      parameter OWIDTH=21
      // LGWIDTH=9;
```

```
	//
)
(i_clk, i_reset, i_ce,
		i_sample, o_result, o_sync);
	// The bit-width of the input, IWIDTH, output, OWIDTH, and the log
	// of the FFT size.  These are localparams, rather than parameters,
	// because once the core has been generated, they can no longer be
	// changed.  (These values can be adjusted by running the core
	// generator again.)  The reason is simply that these values have
	// been hardwired into the core at several places.

	input	wire			i_clk, i_reset, i_ce;
	//
	input	wire	[(2*IWIDTH-1):0]	i_sample;
	output	reg	[(2*OWIDTH-1):0]	o_result;
	output	reg			o_sync;


	// Outputs of the FFT, ready for bit reversal.
	wire			br_sync;
	wire	[(2*OWIDTH-1):0]	br_result;


	// A hardware optimized FFT stage
	wire		w_s512;
	wire	[33:0]	w_d512;
	fftstage	#(
		// {{{
		.IWIDTH(IWIDTH),
		.CWIDTH(IWIDTH+4),
		.OWIDTH(17),
		.LGSPAN(8),
		.BFLYSHIFT(0),
		.OPT_HWMPY(1),
		.CKPCE(1),
		.COEFFILE("cmem_512.hex")
		// }}}
	) stage_512(
		// {{{
		.i_clk(i_clk),
		.i_reset(i_reset),
		.i_ce(i_ce),
		.i_sync(!i_reset),
		.i_data(i_sample),
```

```
		.o_data(w_d512),
		.o_sync(w_s512)
		// }}}
	);


	// A hardware optimized FFT stage
	wire		w_s256;
	wire	[35:0]	w_d256;
	fftstage	#(
		// {{{
		.IWIDTH(17),
		.CWIDTH(21),
		.OWIDTH(18),
		.LGSPAN(7),
		.BFLYSHIFT(0),
		.OPT_HWMPY(1),
		.CKPCE(1),
		.COEFFILE("cmem_256.hex")
		// }}}
	) stage_256(
		// {{{
		.i_clk(i_clk),
		.i_reset(i_reset),
		.i_ce(i_ce),
		.i_sync(w_s512),
		.i_data(w_d512),
		.o_data(w_d256),
		.o_sync(w_s256)
		// }}}
	);

	// A hardware optimized FFT stage
	wire		w_s128;
	wire	[35:0]	w_d128;
	fftstage	#(
		// {{{
		.IWIDTH(18),
		.CWIDTH(22),
		.OWIDTH(18),
		.LGSPAN(6),
		.BFLYSHIFT(0),
		.OPT_HWMPY(1),
		.CKPCE(1),
```

```
		.COEFFILE("cmem_128.hex")
		// }}}
	) stage_128(
		// {{{
		.i_clk(i_clk),
		.i_reset(i_reset),
		.i_ce(i_ce),
		.i_sync(w_s256),
		.i_data(w_d256),
		.o_data(w_d128),
		.o_sync(w_s128)
		// }}}
	);

	// A hardware optimized FFT stage
	wire		w_s64;
	wire	[37:0]	w_d64;
	fftstage	#(
		// {{{
		.IWIDTH(18),
		.CWIDTH(22),
		.OWIDTH(19),
		.LGSPAN(5),
		.BFLYSHIFT(0),
		.OPT_HWMPY(1),
		.CKPCE(1),
		.COEFFILE("cmem_64.hex")
		// }}}
	) stage_64(
		// {{{
		.i_clk(i_clk),
		.i_reset(i_reset),
		.i_ce(i_ce),
		.i_sync(w_s128),
		.i_data(w_d128),
		.o_data(w_d64),
		.o_sync(w_s64)
		// }}}
	);

	// A hardware optimized FFT stage
	wire		w_s32;
	wire	[37:0]	w_d32;
	fftstage	#(
```

```
		// {{{
		.IWIDTH(19),
		.CWIDTH(23),
		.OWIDTH(19),
		.LGSPAN(4),
		.BFLYSHIFT(0),
		.OPT_HWMPY(1),
		.CKPCE(1),
		.COEFFILE("cmem_32.hex")
		// }}}
	) stage_32(
		// {{{
		.i_clk(i_clk),
		.i_reset(i_reset),
		.i_ce(i_ce),
		.i_sync(w_s64),
		.i_data(w_d64),
		.o_data(w_d32),
		.o_sync(w_s32)
		// }}}
	);

	// A hardware optimized FFT stage
	wire		w_s16;
	wire	[39:0]	w_d16;
	fftstage	#(
		// {{{
		.IWIDTH(19),
		.CWIDTH(23),
		.OWIDTH(20),
		.LGSPAN(3),
		.BFLYSHIFT(0),
		.OPT_HWMPY(1),
		.CKPCE(1),
		.COEFFILE("cmem_16.hex")
		// }}}
	) stage_16(
		// {{{
		.i_clk(i_clk),
		.i_reset(i_reset),
		.i_ce(i_ce),
		.i_sync(w_s32),
		.i_data(w_d32),
		.o_data(w_d16),
```

```
		.o_sync(w_s16)
		// }}}
	);

	// A hardware optimized FFT stage
	wire		w_s8;
	wire	[39:0]	w_d8;
	fftstage	#(
		// {{{
		.IWIDTH(20),
		.CWIDTH(24),
		.OWIDTH(20),
		.LGSPAN(2),
		.BFLYSHIFT(0),
		.OPT_HWMPY(1),
		.CKPCE(1),
		.COEFFILE("cmem_8.hex")
		// }}}
	) stage_8(
		// {{{
		.i_clk(i_clk),
		.i_reset(i_reset),
		.i_ce(i_ce),
		.i_sync(w_s16),
		.i_data(w_d16),
		.o_data(w_d8),
		.o_sync(w_s8)
		// }}}
	);

	wire		w_s4;
	wire	[41:0]	w_d4;
	qtrstage	#(
		// {{{
		.IWIDTH(20),
		.OWIDTH(21),
		.LGWIDTH(9),
		.INVERSE(0),
		.SHIFT(0)
		// }}}
	) stage_4(
		// {{{
		.i_clk(i_clk),
		.i_reset(i_reset),
```

```
		.i_ce(i_ce),
		.i_sync(w_s8),
		.i_data(w_d8),
		.o_data(w_d4),
		.o_sync(w_s4)
		// }}}
	);
	// verilator lint_off UNUSED
	wire		w_s2;
	// verilator lint_on  UNUSED
	wire	[41:0]	w_d2;
	laststage	#(
		// {{{
		.IWIDTH(21),
		.OWIDTH(21),
		.SHIFT(0)
		// }}}
	) stage_2(
		// {{{
		.i_clk(i_clk),
		.i_reset(i_reset),
		.i_ce(i_ce),
		.i_sync(w_s4),
		.i_val(w_d4),
		.o_val(w_d2),
		.o_sync(w_s2)
		// }}}
	);


	wire	br_start;
	reg	r_br_started;
	initial	r_br_started = 1'b0;
	always @(posedge i_clk)
	if (i_reset)
		r_br_started <= 1'b0;
	else if (i_ce)
		r_br_started <= r_br_started || w_s2;
	assign	br_start = r_br_started || w_s2;

	// Now for the bit-reversal stage.
	bitreverse	#(
		// {{{
		.LGSIZE(9), .WIDTH(21)
```

```
		// }}}
	) revstage (
		// {{{
		.i_clk(i_clk),
		.i_reset(i_reset),
		.i_ce(i_ce & br_start),
		.i_in(w_d2),
		.o_out(br_result),
		.o_sync(br_sync)
		// }}}
	);


	// Last clock: Register our outputs, we're done.
	initial		o_sync  = 1'b0;
	always @(posedge i_clk)
	if (i_reset)
		o_sync  <= 1'b0;
	else if (i_ce)
		o_sync  <= br_sync;

	always @(posedge i_clk)
	if (i_ce)
		o_result  <= br_result;


endmodule
```

fftstage.sv

```
Unset

`default_nettype	none
//
module fftstage #(
		// {{{
		parameter	IWIDTH=16,CWIDTH=20,OWIDTH=17,
		// Parameters specific to the core that should be changed when
		// this core is built ... Note that the minimum LGSPAN (the base
		// two log of the span, or the base two log of the current FFT
		// size) is 3.  Smaller spans (i.e. the span of 2) must use the
		// dbl laststage module.
		// Verilator lint_off UNUSED
```

```
		parameter	LGSPAN=8, BFLYSHIFT=0, // LGWIDTH=9
		parameter [0:0]	OPT_HWMPY = 1,
		// Clocks per CE.  If your incoming data rate is less than 50%
		// of your clock speed, you can set CKPCE to 2'b10, make sure
		// there's at least one clock between cycles when i_ce is high,
		// and then use two multiplies instead of three.  Setting CKPCE
		// to 2'b11, and insisting on at least two clocks with i_ce low
		// between cycles with i_ce high, then the hardware optimized
		// butterfly code will used one multiply instead of two.
		parameter	CKPCE = 1,
		// The COEFFILE parameter contains the name of the file
		// containing the FFT twiddle factors
		parameter	COEFFILE="cmem_512.hex"
		// Verilator lint_on  UNUSED

	) (
		// {{{
		input	wire				i_clk, i_reset,
							i_ce, i_sync,
		input	wire	[(2*IWIDTH-1):0]	i_data,
		output	reg	[(2*OWIDTH-1):0]	o_data,
		output	reg				o_sync

		// }}}
	);

	// Local signal definitions
	// {{{
	// I am using the prefixes
	//	ib_*	to reference the inputs to the butterfly, and
	//	ob_*	to reference the outputs from the butterfly
	reg	wait_for_sync;
	reg	[(2*IWIDTH-1):0]	ib_a, ib_b;
	reg	[(2*CWIDTH-1):0]	ib_c;
	reg	ib_sync;

	reg	b_started;
	wire	ob_sync;
	wire	[(2*OWIDTH-1):0]	ob_a, ob_b;

	// cmem is defined as an array of real and complex values,
	// where the top CWIDTH bits are the real value and the bottom
	// CWIDTH bits are the imaginary value.
	//
```

```
// cmem[i] = { (2^(CWIDTH-2)) * cos(2*pi*i/(2^LGWIDTH)),
//           (2^(CWIDTH-2)) * sin(2*pi*i/(2^LGWIDTH)) };
//
reg   [(2*CWIDTH-1):0]    cmem [0:((1<<LGSPAN)-1)];

initial      $readmemh(COEFFILE,cmem);

reg   [(LGSPAN):0]        iaddr;
reg   [(2*IWIDTH-1):0]    imem   [0:((1<<LGSPAN)-1)];

reg   [LGSPAN:0]          oaddr;
reg   [(2*OWIDTH-1):0]    omem   [0:((1<<LGSPAN)-1)];

wire                      idle;
reg   [(LGSPAN-1):0]             nxt_oaddr;
reg   [(2*OWIDTH-1):0]    pre_ovalue;
// }}}

// wait_for_sync, iaddr
// {{{
initial wait_for_sync = 1'b1;
initial iaddr = 0;
always @(posedge i_clk)
if (i_reset)
begin
       wait_for_sync <= 1'b1;
       iaddr <= 0;
end else if ((i_ce)&&((!wait_for_sync)||(i_sync)))
begin
       //
       // First step: Record what we're not ready to use yet
       //
       iaddr <= iaddr + { {(LGSPAN){1'b0}}, 1'b1 };
       wait_for_sync <= 1'b0;
end
// }}}

// Write to imem
// {{{
always @(posedge i_clk) // Need to make certain here that we don't read
if ((i_ce)&&(!iaddr[LGSPAN])) // and write the same address on
       imem[iaddr[(LGSPAN-1):0]] <= i_data; // the same clk
// }}}
```

```
// ib_sync
// {{{
// Now, we have all the inputs, so let's feed the butterfly
//
// ib_sync is the synchronization bit to the butterfly.  It will
// be tracked within the butterfly, and used to create the o_sync
// value when the results from this output are produced
initial ib_sync = 1'b0;
always @(posedge i_clk)
if (i_reset)
	ib_sync <= 1'b0;
else if (i_ce)
begin
	// Set the sync to true on the very first
	// valid input in, and hence on the very
	// first valid data out per FFT.
	ib_sync <= (iaddr==(1<<(LGSPAN)));
end
// }}}

// ib_a, ib_b, ib_c
// {{{
// Read the values from our input memory, and use them to feed
// first of two butterfly inputs
always @(posedge i_clk)
if (i_ce)
begin
	// One input from memory, ...
	ib_a <= imem[iaddr[(LGSPAN-1):0]];
	// One input clocked in from the top
	ib_b <= i_data;
	// and the coefficient or twiddle factor
	ib_c <= cmem[iaddr[(LGSPAN-1):0]];
end
// }}}

// idle
// {{{
// The idle register is designed to keep track of when an input
// to the butterfly is important and going to be used.  It's used
// in a flag following, so that when useful values are placed
// into the butterfly they'll be non-zero (idle=0), otherwise when
// the inputs to the butterfly are irrelevant and will be ignored,
// then (idle=1) those inputs will be set to zero.  This
```

```
// functionality is not designed to be used in operation, but only
// within a Verilator simulation context when chasing a bug.
// In this limited environment, the non-zero answers will stand
// in a trace making it easier to highlight a bug.


	assign idle = 0;



//////////////////////////////////////////////////////////////////////////
//
// Instantiate the butterfly
// {{{
//////////////////////////////////////////////////////////////////////////
//
//

//
generate if (OPT_HWMPY)
begin : HWBFLY

	hwbfly #(
		// {{{
		.IWIDTH(IWIDTH),
		.CWIDTH(CWIDTH),
		.OWIDTH(OWIDTH),
		.CKPCE(CKPCE),
		.SHIFT(BFLYSHIFT)
		// }}}
	) bfly(
		// {{{
		.i_clk(i_clk), .i_reset(i_reset), .i_ce(i_ce),
		.i_coef( (idle && !i_ce) ? {(2*CWIDTH){1'b0}}:ib_c),
		.i_left( (idle && !i_ce) ? {(2*IWIDTH){1'b0}}:ib_a),
		.i_right((idle && !i_ce) ? {(2*IWIDTH){1'b0}}:ib_b),
		.i_aux(ib_sync && i_ce),
		.o_left(ob_a), .o_right(ob_b), .o_aux(ob_sync)
		// }}}
	);

end else begin : FWBFLY

	butterfly #(
```

```
			// {{{
			.IWIDTH(IWIDTH),
			.CWIDTH(CWIDTH),
			.OWIDTH(OWIDTH),
			.CKPCE(CKPCE),
			.SHIFT(BFLYSHIFT)
			// }}}
		) bfly(
			// {{{
			.i_clk(i_clk), .i_reset(i_reset), .i_ce(i_ce),
			.i_coef( (idle && !i_ce)? {(2*CWIDTH){1'b0}} :ib_c),
			.i_left( (idle && !i_ce)? {(2*IWIDTH){1'b0}} :ib_a),
			.i_right((idle && !i_ce)? {(2*IWIDTH){1'b0}} :ib_b),
			.i_aux(ib_sync && i_ce),
			.o_left(ob_a), .o_right(ob_b), .o_aux(ob_sync)
			// }}}
		);

	end endgenerate


	// }}}

	// oaddr, o_sync, b_started
	// {{{
	// Next step: recover the outputs from the butterfly
	//
	// The first output can go immediately to the output of this routine
	// The second output must wait until this time in the idle cycle
	// oaddr is the output memory address, keeping track of where we are
	// in this output cycle.
	initial oaddr     = 0;
	initial o_sync    = 0;
	initial b_started = 0;
	always @(posedge i_clk)
	if (i_reset)
	begin
		oaddr     <= 0;
		o_sync    <= 0;
		// b_started will be true once we've seen the first ob_sync
		b_started <= 0;
	end else if (i_ce)
	begin
		o_sync <= (!oaddr[LGSPAN])?ob_sync : 1'b0;
```

```
		if (ob_sync||b_started)
			oaddr <= oaddr + 1'b1;
		if ((ob_sync)&&(!oaddr[LGSPAN]))
			// If b_started is true, then a butterfly output
			// is available
			b_started <= 1'b1;
	end
	// }}}

	// nxt_oaddr
	// {{{
	always @(posedge i_clk)
	if (i_ce)
		nxt_oaddr[0] <= oaddr[0];
	generate if (LGSPAN>1)
	begin : WIDE_LGSPAN

		always @(posedge i_clk)
		if (i_ce)
			nxt_oaddr[LGSPAN-1:1] <= oaddr[LGSPAN-1:1] + 1'b1;

	end endgenerate
	// }}}

	// omem
	// {{{
	// Only write to the memory on the first half of the outputs
	// We'll use the memory value on the second half of the outputs
	always @(posedge i_clk)
	if ((i_ce)&&(!oaddr[LGSPAN]))
		omem[oaddr[(LGSPAN-1):0]] <= ob_b;
	// }}}

	// pre_ovalue
	// {{{
	always @(posedge i_clk)
	if (i_ce)
		pre_ovalue <= omem[nxt_oaddr[(LGSPAN-1):0]];
	// }}}

	// o_data
	// {{{
	always @(posedge i_clk)
	if (i_ce)
```

```
            o_data <= (!oaddr[LGSPAN]) ? ob_a : pre_ovalue;
      // }}}

endmodule
```

Software

mixer.c

```
C/C++
/* * Device driver for the VGA video generator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *           drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod mixer.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_ball.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
```

```
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "mixer.h"
#include <linux/wait.h>
#include <linux/interrupt.h>
#include <linux/of_irq.h>

#define CTRL_DRIVER_NAME "CTRL"
#define STREAM_DRIVER_NAME "STREAM"

/* Device registers */

//control slave
#define CTRL_L_GAIN(x) (x)
#define CTRL_R_GAIN(x) ((x)+1)

//stream slave
#define STREAM(x) (x)

//gain macros
#define L_GAIN 1
#define R_GAIN 1

/*
 * Information about our device
 */
struct ctrl_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */
       ctrl_gain_t gain;
} ctrl_dev;

struct stream_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */
    stream_t stream;
    int irq_num;
    int ready;
} stream_dev;

DECLARE_WAIT_QUEUE_HEAD(wq);
```

```
irqreturn_t irq_handler(int irq, void *dev_id)
{
    if (irq != stream_dev.irq_num) {
        return IRQ_NONE;
    }

    stream_dev.ready = (ioread32(stream_dev.virtbase) == 0xe);
    wake_up_interruptible(&wq);
    return IRQ_HANDLED;
}

static void write_gain(ctrl_gain_t *gain)
{
    iowrite8(gain->l_gain, CTRL_L_GAIN(ctrl_dev.virtbase) );
    iowrite8(gain->r_gain, CTRL_R_GAIN(ctrl_dev.virtbase) );
    ctrl_dev.gain = *gain;
}

static void write_stream(stream_t *stream)
{
    wait_event_interruptible(wq, stream_dev.ready);
    // wait_event_interruptible_timeout(wq, stream_dev.ready, 10);
    iowrite32(stream->stream1 << 16 | stream->stream2, stream_dev.virtbase);
    // iowrite16(stream->stream2, stream_dev.virtbase + 2);
    stream_dev.stream = *stream;
}

static void read_stream(stream_t *stream)
{
    stream->stream1 = ioread16(stream_dev.virtbase);
    // stream->stream2 = ioread16(stream_dev.virtbase + 2);
}

/*
 * Handle ioctl() calls from userspace
 */
static long ctrl_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    ctrl_arg_t ctrl;

    switch (cmd) {
    case CTRL_WRITE_GAIN:
        if (copy_from_user(&ctrl, (ctrl_arg_t *) arg,
                           sizeof(ctrl_arg_t)))
```

```c
                return -EACCES;
            write_gain(&ctrl.gain);
            break;
    default:
            return -EINVAL;
    }

    return 0;
}

static long stream_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    stream_arg_t stream;

    switch (cmd) {
    case STREAM_WRITE_STREAM:
        if (copy_from_user(&stream, (stream_arg_t *) arg,
                           sizeof(stream_arg_t)))
                return -EACCES;
        write_stream(&stream.stream);
        break;
    case STREAM_READ_STREAM:
        read_stream(&stream.stream);
        if (copy_to_user((stream_arg_t *)arg, &stream,
                sizeof(stream_arg_t))) {
                return -EACCES;
        }
        break;
    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations ctrl_fops = {
    .owner     = THIS_MODULE,
    .unlocked_ioctl = ctrl_ioctl,
};

static const struct file_operations stream_fops = {
    .owner     = THIS_MODULE,
    .unlocked_ioctl = stream_ioctl,
```

```
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice ctrl_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = CTRL_DRIVER_NAME,
    .fops       = &ctrl_fops,
};

static struct miscdevice stream_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = STREAM_DRIVER_NAME,
    .fops       = &stream_fops,
};

/*
 * Initialization code: get resources (registers)
 */
static int __init ctrl_probe(struct platform_device *pdev)
{
    int ret;

    /* Register ourselves as a misc device: creates /dev/vga_ball */
    ret = misc_register(&ctrl_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &ctrl_dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(ctrl_dev.res.start, resource_size(&ctrl_dev.res),
        CTRL_DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    /* Arrange access to our registers */
    ctrl_dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (ctrl_dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
```

```
    }

    return 0;

out_release_mem_region:
    release_mem_region(ctrl_dev.res.start, resource_size(&ctrl_dev.res));
out_deregister:
    misc_deregister(&ctrl_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int ctrl_remove(struct platform_device *pdev)
{
    iounmap(ctrl_dev.virtbase);
    release_mem_region(ctrl_dev.res.start, resource_size(&ctrl_dev.res));
    misc_deregister(&ctrl_misc_device);
    return 0;
}

static int __init stream_probe(struct platform_device *pdev)
{
    int ret, irq;

    /* Register ourselves as a misc device: creates /dev/vga_ball */
    ret = misc_register(&stream_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &stream_dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(stream_dev.res.start,
resource_size(&stream_dev.res),
        STREAM_DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    /* Arrange access to our registers */
    stream_dev.virtbase = of_iomap(pdev->dev.of_node, 0);
```

```c
    if (stream_dev.virtbase == NULL) {
        ret = -ENOMEM;
        goto out_release_mem_region;
    }

    irq = irq_of_parse_and_map(pdev->dev.of_node, 0);
    stream_dev.irq_num = irq;

    pr_alert("irq_of_parse_and_map worked\n");

    ret = request_irq(irq, irq_handler, 0, "csee4840_vga_ball", NULL);
    pr_alert("request_irq finished\n");
    if (ret) {
        pr_err("request_irq failed: %d\n", ret);
        ret = -ENOENT;
        goto out_release_mem_region;
    }

    stream_dev.ready = 1;

    return 0;

out_release_mem_region:
    release_mem_region(stream_dev.res.start, resource_size(&stream_dev.res));
out_deregister:
    misc_deregister(&stream_misc_device);
    return ret;
}

/* Clean-up code: release resources */
static int stream_remove(struct platform_device *pdev)
{
    free_irq(stream_dev.irq_num, NULL);
    iounmap(stream_dev.virtbase);
    release_mem_region(stream_dev.res.start, resource_size(&stream_dev.res));
    misc_deregister(&stream_misc_device);
    return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id ctrl_of_match[] = {
    { .compatible = "csee4840,ctrl-1.0" },
    {},
```

```
};
MODULE_DEVICE_TABLE(of, ctrl_of_match);
static const struct of_device_id stream_of_match[] = {
    { .compatible = "csee4840,vga_ball-1.0" },
    {},
};
MODULE_DEVICE_TABLE(of, stream_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver ctrl_driver = {
    .driver    = {
        .name     = CTRL_DRIVER_NAME,
        .owner     = THIS_MODULE,
        .of_match_table = of_match_ptr(ctrl_of_match),
    },
    .remove    = __exit_p(ctrl_remove),
};

static struct platform_driver stream_driver = {
    .driver    = {
        .name     = STREAM_DRIVER_NAME,
        .owner     = THIS_MODULE,
        .of_match_table = of_match_ptr(stream_of_match),
    },
    .remove    = __exit_p(stream_remove),
};

/* Called when the module is loaded: set things up */
static int __init mixer_init(void)
{
    int ret;

    pr_info(CTRL_DRIVER_NAME ": init\n");
    ret = platform_driver_probe(&ctrl_driver, ctrl_probe);
    if (ret < 0)
        return ret;
    pr_info(STREAM_DRIVER_NAME ": init\n");
    return platform_driver_probe(&stream_driver, stream_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit mixer_exit(void)
{
```

```
    platform_driver_unregister(&ctrl_driver);
    platform_driver_unregister(&stream_driver);
    pr_info(CTRL_DRIVER_NAME ": exit\n");
    pr_info(STREAM_DRIVER_NAME ": exit\n");
}

module_init(mixer_init);
module_exit(mixer_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Joy, Oli, Harrison, Joshua. Columbia University");
MODULE_DESCRIPTION("FPGA DJ");
```

bitsandbeats.c

```C/C++
/*
 * Userspace program that communicates with the vga_ball device driver
 * through ioctls
 *
 * Stephen A. Edwards
 * Columbia University
 */

#include <assert.h>
#include <endian.h>
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include "mixer.h"
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <unistd.h>
#include "usbkeyboard.h"
#include <pthread.h>
#include <stdlib.h>
#include "usbhidkeys.h"
#include <sys/stat.h>
```

```c
#define GAIN_UP 1.1
#define GAIN_DOWN 0.9
#define LE16TOH(x) (le16toh(*(uint16_t *)(x)))
#define LE32TOH(x) (le32toh(*(uint32_t *)(x)))
#define FMT_OF(wi) (wi.file + wi.fmt_offset)
#define DATA_OF(wi) (wi.file + wi.data_offset)
#define SAMPLE_AT(wi, i) \
    (*(int16_t *)(DATA_OF(wi) + 8 + wi.bytes_per_whole_sample*(i)))
#define SAMPLE_AT_WO(wi, i) SAMPLE_AT(wi, (i + wi.song_offset)%wi.num_samples)
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#define MAX(a, b) ((a) > (b) ? (a) : (b))

#define SAMPLES_PER_SECOND 48000

struct subchunk_header {
    char name[4];
    uint32_t size;
};

struct wav_info {
    char *filename;
    size_t filesize;

    char *file;
    uint32_t fmt_offset;
    uint32_t data_offset;
    uint16_t audio_format;
    uint16_t num_channels;
    uint32_t sample_rate;
    uint32_t byte_rate;
    uint16_t block_align;
    uint16_t bits_per_sample;

    int bytes_per_whole_sample;
    int num_samples;
    volatile int do_silent;
    volatile int song_offset;
    volatile float gain;
    volatile int do_echo;
    volatile int echo_offset;
};

int ctrl_fd;
```

```c
int stream_fd;
struct libusb_device_handle *keyboard;
uint8_t endpoint_address;
struct wav_info song1, song2;
volatile int do_airhorn;

int16_t clamp(int32_t a)
{
    int32_t b = MIN(a, INT16_MAX);
    return MAX(b, INT16_MIN);
}

int32_t get_sample(struct wav_info wi, int ndx)
{
    float sample;
    static const float echo_weights[] = { 0.5, 0.25, 0.125 };

    if (wi.do_silent) {
        return 0;
    }

    sample = SAMPLE_AT_WO(wi, ndx);
    if (wi.do_echo) {
        for (int i = 0; i < sizeof(echo_weights)/sizeof(float); i++) {
              sample += echo_weights[i]*SAMPLE_AT_WO(wi, ndx + (i +
1)*wi.echo_offset);
        }
    }
    return (int32_t)(wi.gain*sample);
}

struct wav_info new_wav_info(const char *filename)
{
    uint32_t curr_offset = 12;  // skip first chunk
    struct wav_info wi = { 0 };
    uint32_t data_size;
    struct stat st;
    int fd;
    char *file;

    assert(lstat(filename, &st) >= 0);
    assert((fd = open(filename, O_RDONLY)) >= 0);
    assert((file = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0)) !=
MAP_FAILED);
```

```c
    assert(strncmp(file, "RIFF", 4) == 0 && strncmp(file + 8, "WAVE", 4) == 0);

    wi.filename = (char *)filename;
    wi.filesize = st.st_size;
    wi.file = file;

    while (!wi.fmt_offset || !wi.data_offset) {
        char *curr_subchunk = file + curr_offset;
        struct subchunk_header curr_header =
              *(struct subchunk_header *)curr_subchunk;

        curr_header.size = le32toh(curr_header.size);
        printf("%.*s: %d\n", 4, curr_header.name, curr_header.size);
        if (strncmp(curr_header.name, "fmt\040", 4) == 0) {
              wi.fmt_offset = curr_offset;
        } else if (strncmp(curr_header.name, "data", 4) == 0) {
              wi.data_offset = curr_offset;
        }

        curr_offset += 8 + curr_header.size;
    }

    assert(wi.fmt_offset && wi.data_offset);

    wi.audio_format = LE16TOH(FMT_OF(wi) + 8);
    wi.num_channels = LE16TOH(FMT_OF(wi) + 10);
    wi.sample_rate = LE32TOH(FMT_OF(wi) + 12);
    wi.byte_rate = LE32TOH(FMT_OF(wi) + 16);
    wi.block_align = LE16TOH(FMT_OF(wi) + 20);
    wi.bits_per_sample = LE16TOH(FMT_OF(wi) + 22);

    wi.bytes_per_whole_sample = wi.num_channels * (wi.bits_per_sample / 8);
    data_size = LE32TOH(DATA_OF(wi) + 4);
    wi.num_samples = data_size / wi.bytes_per_whole_sample;

    wi.do_silent = 0;
    wi.song_offset = 0;
    wi.gain = 1.0;
    wi.do_echo = 0;
    wi.echo_offset = 0;

    return wi;
}
```

```
void print_wav_info(struct wav_info wi)
{
    printf("audio_format=%d\n", wi.audio_format);
    printf("num_channels=%d\n", wi.num_channels);
    printf("sample_rate=%d\n", wi.sample_rate);
    printf("byte_rate=%d\n", wi.byte_rate);
    printf("block_align=%d\n", wi.block_align);
    printf("bits_per_sample=%d\n", wi.bits_per_sample);
    printf("bytes_per_whole_sample=%d\n", wi.bytes_per_whole_sample);
    printf("num_samples=%d\n", wi.num_samples);
}

void *keyboard_fn(void *arg)
{
    struct usb_keyboard_packet packet;
    int transferred;
    // char keystate[12];
    uint8_t prev_key0 = 0;
    uint8_t listen_for_song_offset = 0;
    int song_offset = 0;

    for (;;) {
        libusb_interrupt_transfer(keyboard, endpoint_address,
              (unsigned char *)&packet, sizeof(packet),
              &transferred, 0);

        // pprintf(keystate, "%02x %02x %02x", packet.modifiers,
packet.keycode[0], packet.keycode[1]);
        // printf("%s\n", keystate);

        if (transferred != sizeof(packet)) {
              continue;
        }

        uint8_t key0 = packet.keycode[0];

        /*
         * fn1: Play/pause song 1
         * fn2: Play/pause song 2
         * A/D: Scroll through song 1
         * Left/right arrows: Scroll through song 2
         */

        if (prev_key0 == key0) {
```

```
            continue;
        }
        prev_key0 = key0;

        switch (key0) {
        case KEY_F1:
            song1.do_silent = !song1.do_silent;
            printf("%s %s\n",
                song1.do_silent ? "Paused" : "Playing", song1.filename);
            break;
        case KEY_F2:
            song2.do_silent = !song2.do_silent;
            printf("%s %s\n",
                song2.do_silent ? "Paused" : "Playing", song2.filename);
            break;
        case KEY_F3:
            song1.do_silent = !song1.do_silent;
            song2.do_silent = !song2.do_silent;
            printf("%s %s. %s %s.\n",
                song1.do_silent ? "Paused" : "Playing", song1.filename,
                song2.do_silent ? "Paused" : "Playing", song2.filename);
            break;
        case KEY_A:
        case KEY_D:
        case KEY_LEFT:
        case KEY_RIGHT:
            listen_for_song_offset = key0;
            song_offset= 0;
            printf("Listening for song offset\n");
            break;
        case KEY_ENTER:
            switch (listen_for_song_offset) {
            case KEY_A:
                song1.song_offset -= song_offset * SAMPLES_PER_SECOND;
                printf("Rewinded %s by %ds\n", song1.filename,
song_offset);
                break;
            case KEY_D:
                song1.song_offset += song_offset * SAMPLES_PER_SECOND;
                printf("Forwarded %s by %ds\n", song1.filename,
song_offset);
                break;
            case KEY_LEFT:
                song2.song_offset -= song_offset * SAMPLES_PER_SECOND;
```

```
                    printf("Rewinded %s by %ds\n", song2.filename,
song_offset);
                    break;
              case KEY_RIGHT:
                    song2.song_offset += song_offset * SAMPLES_PER_SECOND;
                    printf("Forwarded %s by %ds\n", song2.filename,
song_offset);
                    break;
              default:
                    break;
              }
              listen_for_song_offset = song_offset = 0;
              break;
        case KEY_W:
              song1.gain *= GAIN_UP;
              printf("Set gain for %s to %.2f\n", song1.filename, song1.gain);
              break;
        case KEY_S:
              song1.gain *= GAIN_DOWN;
              printf("Set gain for %s to %.2f\n", song1.filename, song1.gain);
              break;
        case KEY_UP:
              song2.gain *= GAIN_UP;
              printf("Set gain for %s to %.2f\n", song2.filename, song2.gain);
              break;
        case KEY_DOWN:
              song2.gain *= GAIN_DOWN;
              printf("Set gain for %s to %.2f\n", song2.filename, song2.gain);
              break;
        case KEY_E:
              song1.do_echo = !song1.do_echo;
              song1.echo_offset = -(SAMPLES_PER_SECOND/8);
              printf("%s echo for %s\n",
                    song1.do_echo ? "Starting" : "Stopped", song1.filename);
              break;
        case KEY_R:
              song2.do_echo = !song2.do_echo;
              song2.echo_offset = -(SAMPLES_PER_SECOND/8);
              printf("%s echo for %s\n",
                    song2.do_echo ? "Starting" : "Stopped", song2.filename);
              break;
        case KEY_SPACE:
              do_airhorn = 1;
              printf("Playing airhorn\n");
```

```
                break;
            default:
                if (listen_for_song_offset && key0 >= KEY_1
                        && key0 <= KEY_0) {
                        song_offset = song_offset*10 + (key0 == KEY_0 ? 0 : key0 -
KEY_1 + 1);
                }
                break;
            }
    }
}

int main(int argc, char **argv)
{
    static const char ctrl_filename[] = "/dev/CTRL";
    static const char stream_filename[] = "/dev/STREAM";
    static const char airhorn_filename[] = "AIRHORN.wav";
    pthread_t keyboard_thread;
    struct wav_info airhorn;

    assert(argc == 3);
    char *song1_filename = argv[1];
    char *song2_filename = argv[2];

    printf("Mixer program started\n");

    if ((ctrl_fd = open(ctrl_filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", ctrl_filename);
        return -1;
    }

    if ((stream_fd = open(stream_filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", stream_filename);
        return -1;
    }

    if ((keyboard = openkeyboard(&endpoint_address)) == NULL) {
        fprintf(stderr, "Did not find keyboard\n");
        exit(1);
    }

    song1 = new_wav_info(song1_filename);
    song2 = new_wav_info(song2_filename);
    airhorn = new_wav_info(airhorn_filename);
```

```
    printf("\n");
    printf("SONG 1: %s\n", song1.filename);
    print_wav_info(song1);

    printf("\n");
    printf("SONG 2: %s\n", song2.filename);
    print_wav_info(song2);
    assert(song1.num_channels == 1 && song1.bits_per_sample == 16);
    assert(song2.num_channels == 1 && song2.bits_per_sample == 16);

    printf("\n");

    pthread_create(&keyboard_thread, NULL, keyboard_fn, NULL);

    int song1_ndx, song2_ndx, airhorn_ndx;
    airhorn_ndx = 0;
    for (song1_ndx = 0, song2_ndx = 0; ; song1_ndx += !song1.do_silent,
song2_ndx += !song2.do_silent) {
        // printf("SAMPLE %d\n", i);
        int32_t sample1 = get_sample(song1, song1_ndx);
        int32_t sample2 = get_sample(song2, song2_ndx);

        // uint32_t sample = sample1 << 16 | sample2;
        // printf("sample=%x\t", sample);

        int32_t airhorn_sample = 0;
        if (do_airhorn) {
              airhorn_sample = SAMPLE_AT(airhorn, airhorn_ndx);
              airhorn_ndx++;
              if (airhorn_ndx >= airhorn.num_samples) {
                    do_airhorn = airhorn_ndx = 0;
              }
        }

        stream_t stream = {
              .stream1 = clamp(sample1 + airhorn_sample*4),
              .stream2 = clamp(sample2)
        };
        stream_arg_t stream_arg = { .stream = stream };
        if (ioctl(stream_fd, STREAM_WRITE_STREAM, &stream_arg)) {
              perror("ioctl(STREAM_WRITE) failed");
              return -1;
        }
```

```
        // usleep(1);
        /*
        if (ioctl(stream_fd, STREAM_READ_STREAM, &stream_arg)) {
              perror("ioctl(STREAM_READ) failed");
              return -1;
        }*/
        // printf("stream1=%x\n", stream_arg.stream.stream1);
    }

    munmap(song1.file, song1.filesize);
    munmap(song2.file, song2.filesize);
    munmap(airhorn.file, airhorn.filesize);
    printf("Mixer program terminating\n");
    return 0;
}
```

Testing / Generation

tb.cpp

```
C/C++
#include <stdio.h>
#include <math.h>
#include <assert.h>

#include "verilated.h"
#include "Vsoc_system.h"
#include "verilated_vcd_c.h"


#include "AudioFile.h"

#define VCD



class  TB {
public:
      Vsoc_system   *m_tb;
```

```
// double           *m_tb_buf;
// int              m_ntest;
bool         m_syncd;
unsigned long m_tickcount;
VerilatedVcdC*      m_trace;

AudioFile<int16_t> curr_wav;
int wav_idx;
int sample_idx;

TB(void) {
       m_tb = new Vsoc_system;
       #ifdef VCD
       Verilated::traceEverOn(true);
       #endif

       curr_wav.setNumChannels(2);
       curr_wav.setAudioBufferSize(2, 300000);
       curr_wav.setSampleRate(44100);
       curr_wav.setBitDepth(16);



       wav_idx = 0;
       sample_idx = 0;
}

virtual void opentrace(const char *vcdname) {
       #ifdef VCD
       if (!m_trace) {
              m_trace = new VerilatedVcdC;
              m_tb->trace(m_trace, 99);
              m_trace->open(vcdname);
       }
       #endif
}

// void      tick(void) {
//     m_tb->clk_clk = 0;
//     m_tb->eval();
//     m_tb->clk_clk = 1;
//     m_tb->eval();
```

```
	// }

	void	tick(void) {
		m_tickcount++;

		m_tb->clk_clk = 0;
		m_tb->eval();
		#ifdef VCD
		if (m_trace)
			m_trace->dump((vluint64_t)(10*m_tickcount-2));
		#endif

		m_tb->clk_clk = 1;
		m_tb->eval();
		#ifdef VCD
		if (m_trace)
			m_trace->dump((vluint64_t)(10*m_tickcount));
		#endif
		m_tb->clk_clk = 0;
		m_tb->eval();
		#ifdef VCD
		if (m_trace) {
			m_trace->dump((vluint64_t)(10*m_tickcount+5));
			m_trace->flush();
		}
		#endif
	}

	void	reset(void) {

		m_tb->reset_reset_n = 0;
		tick();
		m_tb->reset_reset_n = 1;
		tick();


	}


};

#define LEFT_LPF_CUTOFF 0b10
#define LEFT_HPF_CUTOFF 0b100
```

```cpp
AudioFile<int32_t> audioFile;
AudioFile<int32_t> audioFile2;

int     main(int argc, char **argv, char **envp) {
        Verilated::commandArgs(argc, argv);
        TB *tb = new TB;
        FILE    *fpout;

        printf("IFFT TB\n");
        fpout = fopen("tb.dbl", "w");
        if (NULL == fpout) {
                fprintf(stderr, "Cannot write output file, fft_tb.dbl\n");
                exit(-1);
        }

        tb->opentrace("tb.vcd");

        tb->reset();

        // audioFile.load("../../SFD.wav");
        // audioFile.load("../../RhymeDust.wav");


        Vsoc_system * dj = tb->m_tb;

        // SET FILTER CUTOFFS

        dj->dj_0_effects_address  = LEFT_LPF_CUTOFF;
        dj->dj_0_effects_writedata = 200;
        dj->dj_0_effects_write = 1;
        tb->tick();

        dj->dj_0_effects_write = 0;
        tb->tick();

        dj->dj_0_effects_address  = LEFT_HPF_CUTOFF;
        dj->dj_0_effects_writedata = 30U;
        dj->dj_0_effects_write = 1;
        tb->tick();

        dj->dj_0_effects_write = 0;
        tb->tick();
```

```
    int i;
    for(i = 0; i < 30000; i++) {
        double cl, cr, cp, W;
        uint32_t val;
        int j;


        if (i % 2 == 0) {

            dj->mixer_0_avalon_slave_0_chipselect =1;
            dj->mixer_0_avalon_slave_0_address = 0;

            static int k = 0;
            dj->mixer_0_avalon_slave_0_write = 1;

            W = 2.0 * M_PI / 512;
            cl = cos(W * (6 * k )) * 500.0;

            cr = cos(W * (8 * k)) * 0000.00;

            cp = cos(W * (50*k)) * 500.00;

            // val = audioFile.samples[0][k];

            k++;

            val = cr + cl + cp;

            dj->mixer_0_avalon_slave_0_writedata = (val  << 16) |
((int32_t)(0x0000FFFF & val));
            tb->tick();



            // for(j = 0; j < val % 4 + 1; j++)
{dj->dj_0_avalon_slave_0_write = 0;tb->tick();}
            // dj->dj_0_avalon_slave_0_write = 1;

        }
        else {

            dj->mixer_0_avalon_slave_0_chipselect =0;
            tb->tick();
```

```
            }

            if ((i % 4 == 0)) {
                   dj->dj_0_codec_left_fifo_sink_ready = 1;
                   dj->dj_0_codec_right_fifo_sink_ready = 1;
                   tb->tick();
                   tb->curr_wav.samples[0][tb->sample_idx++] =
(int16_t)(dj->dj_0_codec_left_fifo_sink_data);
                   // dj->dj_0_avalon_slave_0_read = 0;


            }

            // if ((uint16_t)dj->dj_0_right_stream_interrupt_irq == 0) {
            //     static int k = 0;

            //     if (k < audioFile2.getNumSamplesPerChannel()) {

            //            dj->dj_0_avalon_slave_0_address = 1;


            //            dj->dj_0_avalon_slave_0_write = 1;



            //            val = audioFile2.samples[0][k];

            //            k++;


            //            dj->dj_0_avalon_slave_0_writedata = val;
            //            tb->tick();

            //     }

            // } else {

            //     dj->dj_0_avalon_slave_0_write = 0;
            //     tb->tick();

            // }
```

```
        }

        tb->curr_wav.save("OUT.wav");

        fclose(fpout);
}
```

Gen_filter.py

```python
## HPF

filter_width = 32
c = 4
sample_width = 21

f = open("hw/fft-sv/hpf.hex", "w")

f.write("// HPF, LEN = " + str(filter_width) + "\n")

f.write("000000\n")
for x in range(1, filter_width -1):
    val = int(2 ** sample_width * (1 - c/(x + c)+ c/(c+32)))
    f.write(("%0.6X" % val) + "\n")
f.write(("%0.6X" % 2 ** sample_width) + "\n")
f.close()

## LPF

filter_width = 32
c = 3

f = open("hw/fft-sv/lpf.hex", "w")

f.write("// LPF, LEN = " + str(filter_width) + "\n")

f.write(("%0.6X" % 2 ** sample_width) + "\n")
for x in range(1, filter_width - 1):
    val = int(2 ** 21 * (1 - c/(32 - x + c) + c/(c+32)))
    f.write(("%0.6X" % val) + "\n")
```

```
f.write("000000\n")

f.close()
```