Sound Localization Project

Matheu Campbell (mgc2171), Dawn Yoo (dy2486),
Peiran Wang (pw2593), Elvis Wang (yw4082)

Spring Semester, 2024

1 Introduction

Sensor arrays are often used in applications where spatial and temporal information about signals provides useful information about its source. By transforming signals measured at multiple points in time, these arrays can determine when, from where, and with what properties an incoming signal has arrived. The spatial signal processing techniques involved in making these inferences find applications in wireless communication, radar, seismology, and audio analysis. In this project, we will use two linear microphone arrays, mounted perpendicularly, to determine the angle of arrival of a sound (of known frequency) originating from a point source.

2 System Overview

2.1 Algorithm

The core algorithm the system implements is the Bartlett method of beamforming. Generically, beamforming is the technique of applying certain transformations to make a microphone array sensitive to signals coming from a specific direction. With the right manipulations, signals from other directions destructively cancel each other out, while signals in the target direction constructively reinforce each other. To perform Bartlett beamforming, the frequency-domain representation of each signal in the array at the target frequency is multiplied by an appropriate complex exponential. This corresponds to a phase shift in the time domain. The geometry of the array determines which set of complex exponentials will be applied to realign signals from a target direction. Once all multiplications are performed, the array output power can be calculated as the sum of the magnitudes of each of the products. Figure 1 shows the geometry involved in calculating the Bartlett coefficients for a linear array (Source).

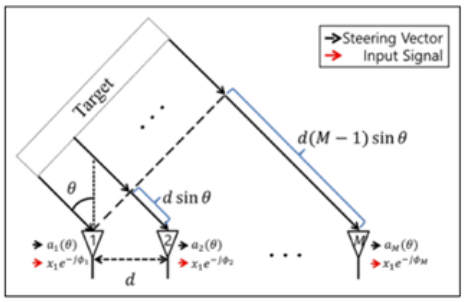


Figure 1: Calculation of Time Delays

To localize an incoming sound, the array output power can be calculated at a finite set of target angles to generate a spatial power spectral density. The maximum among these angles is assumed to be approximately the origin of the source of the sound. Note that for a single linear array, the angles are symmetric about the axis of the array, as the realignment coefficients are identical regardless of which side the sound source is on. In the final version of this project, we use two linear arrays arranged

perpendicularly to one another to localize the sound to one position. Below, a diagram of a linear array and incident angles as well as a sample PSD for an angle of arrival of 75 degrees are shown.

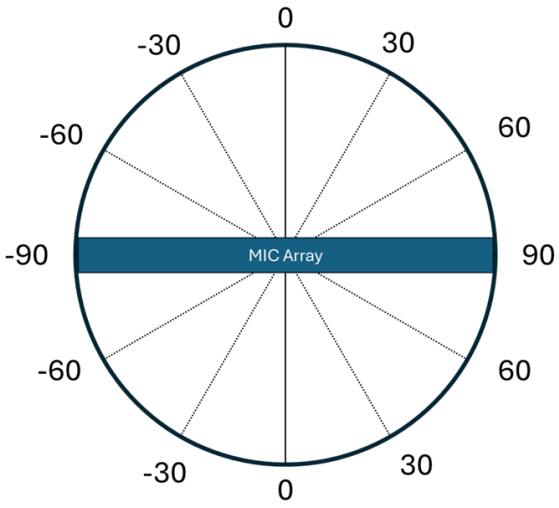


Figure 2: Visual Diagram of Microphone Array

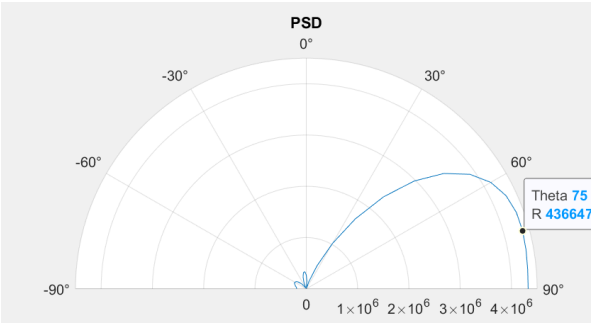


Figure 3: PSD for an Angle of Arrival of 75

Another crucial algorithm for this system is the Fast Fourier Transform (FFT). This algorithm transforms time-domain data into frequency-domain data. Only in the frequency domain can we extract the signal of interest and perform the appropriate multiplications. A dedicated IP block implements the Cooley-Tukey algorithm for FFT, decimating the signal in frequency using a divide-and-conquer strategy to generate our transforms in real-time, allowing real-time updates to the output angle that change as the position of the signal changes.

2.2 Implementation

In our design, the FPGA, interfaced with a microphone array, shows the direction of the audio source on two 7-segment displays. The microphone array lies stationary in the center of the room, and the audio source is a wireless speaker that we manually move around.

Computation takes place in four phases. In the first phase, the FPGA records data from the microphones and stores it in memory. To simplify calculations, the source is a single narrow-band high-frequency audio signal, which is chosen to be 2 kHz. In the second phase, the FPGA performs the FFT algorithm and selects the frequency of interest based on the maximum power point in a spectrum band covering 2 kHz to avoid out-of-band noise. In the third phase, the FPGA calculates the angular offset of the audio source using pre-calculated weights. Finally, the angle information will be displayed on 7-segment displays in real-time.

Our design has six main modules (`Microphone Array, Time Series FIFO, 1024-point Streaming FFT, Frequency Detect, DOA Controller`), Figure 4 showing the block diagram of signal bit-width of one dimension microphone array. Let's break it down.

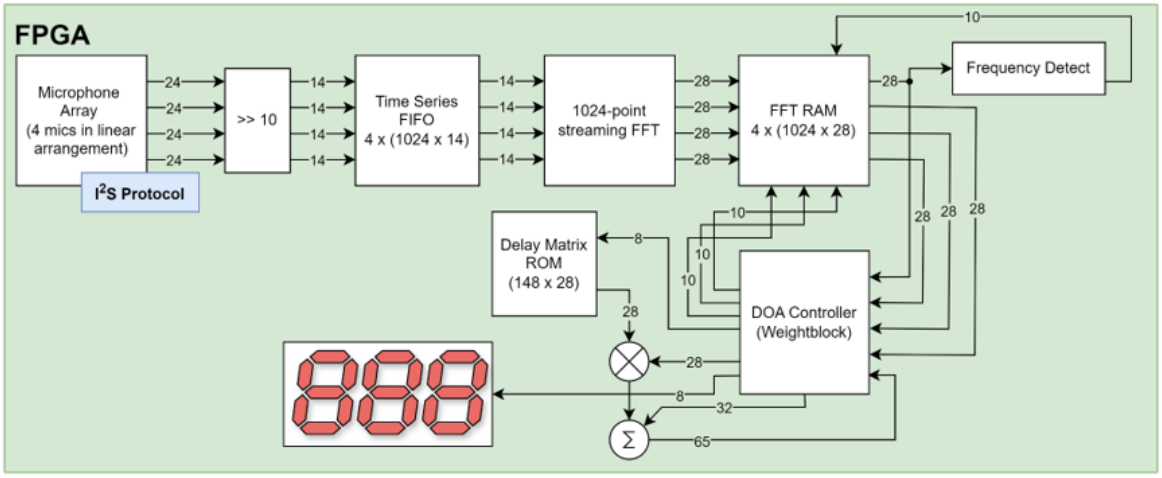


Figure 4: Block Diagram of the System

First, four microphones collect 24-bit word-length audio samples. Then, the higher 14 bits are kept, which is saved to the time series FIFO for crossing time domains. Third, the 1024-point streaming FFT module calculates the spectrum of the inputs, generating 28-bit spectrum data, each consisting of a 14-bit real part and a 14-bit imaginary part, which is subsequently stored in FFT RAM. Fourth, the frequency detector module starts iterating through the FFT RAM to obtain the address of the frequency of interest (i.e., 2 kHz) in the RAM. Next, the direction of arrival (DOA) controller reads the corresponding spectrum samples using the previous frequency detection calculation address and multiplies them by the weights that are pre-loaded in the delay matrix ROM. Finally, products are summed to get the weighted summation value for each direction, with the largest summation being the estimated direction of arrival.

3 Hardware Design

Hardware design has four main modules, including **audio.sv, fft_wrapper.sv, freqdetect.sv, weightblock.sv**.

3.1 Digital Microphone Array and I2S Interface

A linear microphone array with four microphones collects the audio data in one dimension. As shown in Figure 6, two microphone arrays are implemented, with a constant distance of 7.9 cm between each mircophone. Note that there are two more microphones in every dimension to ensure the other four are synchronized correctly because four microphones cannot work properly. In total, only eight microphones are used in our design.

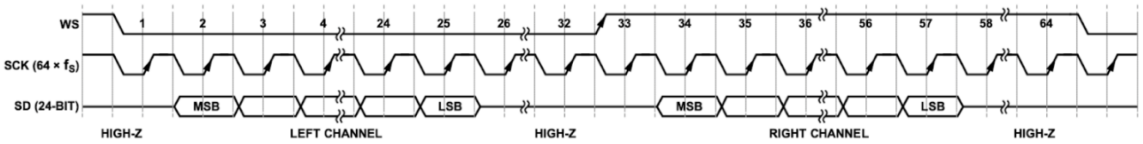


Figure 5: I2S Data Stream Format (Source)

INMP441 is chosen in our design, which has digital inputs/outputs with I2S protocol and stereo configuration mode. Figure 5 shows the I2S data stream of stereo mode. I2S consists of three important lines: word select (WS), serial clock (SCK), and serial data (SD). WS controls the sampling rate and denotes the right or left channels on the SD line, which is set to 48kHz in our design. SCK clock triggers the SD output stream and is 64 times faster than the WS clock (3.072MHz). SD outputs 24-bit word length, MSB first serial data.

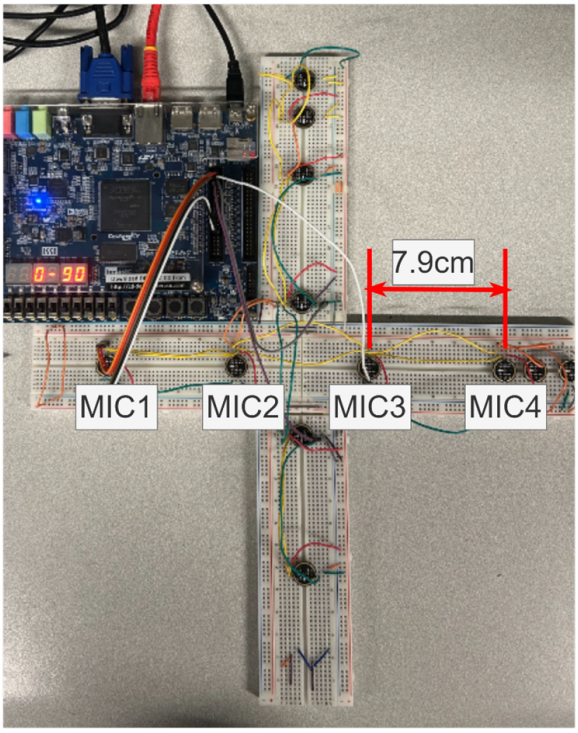


Figure 6: Microphone Array

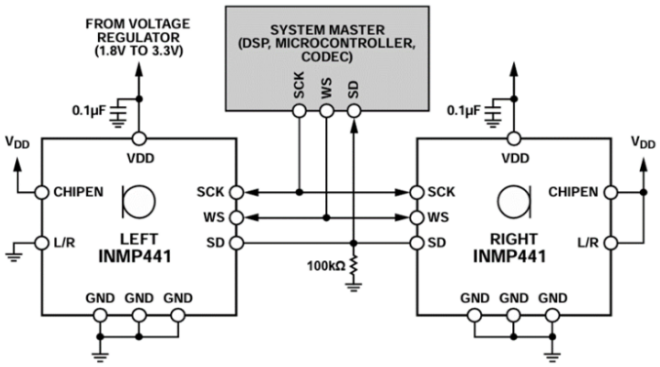


Figure 7: I2S Stereo Configuration (Source)

INMP441 supports stereo configuration, which means that two microphones can share the same SD line, as shown in Figure 7. This method saves some GPIO resources, ensuring that eight microphones use four SD lines in two dimensions. The configuration of the single-axis system is shown in Figure 8. The wiring instruction with GPIO is shown in Figure 9.

3.1.1 AUDIO Block (Top Module)

1. **Inputs:**
   - `clk`: Clock signal line (50MHz).
   - `reset`: Resets the state of the block to an initial state.
   - `chipselect, read, write`: Avalon Bus control signal.
   - `[31:0]writedata`: Avalon data bus.
   - `[2:0]address`: Avalon address bus.
   - `SD1, SD2, SD3, SD4`: Connected to I2S conduit. Serial data line.

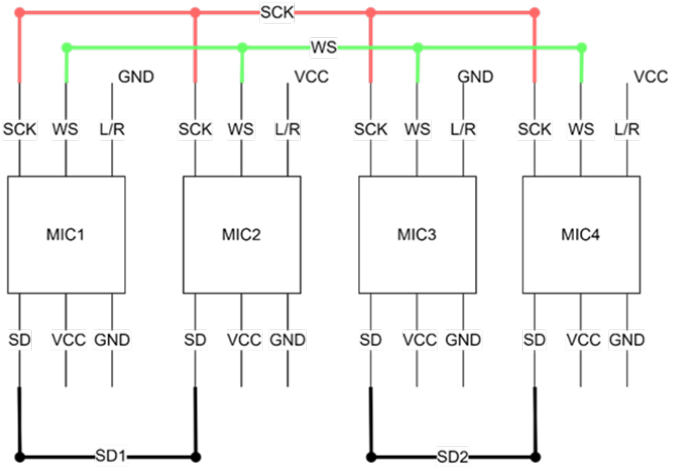


Figure 8: Wiring Configuration of Single-axis System

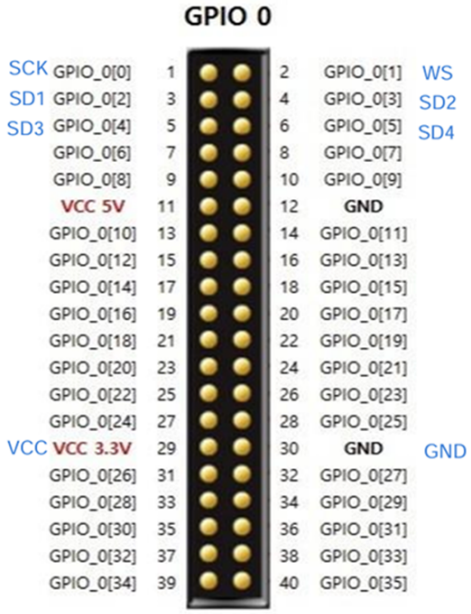


Figure 9: GPIO Usage

- `SCK`: Connected to a phase-locked loop (PLL). Serial clock line.

2. **Outputs:**
   - `WS`: Word select line.
   - `[31:0]readdata`: Avalon data bus.
   - `[6:0]disp2, disp1, disp0`: Connected to 7-segment displays for x-axis.
   - `[6:0]disp5, disp4, disp3`: Connected to 7-segment displays for y-axis.
3. **Functionality:**

   This module acted as the top module of the hardware system. The corresponding source code is in `audio.sv`. Two conduits are instantiated here to connect GPIOs and 7-segment displays, respectively, as shown in Figure 10. In addition, `time series FIFO, fft_wrapper, freqdetect,` and `weightblock` are instantiated here and connected together.

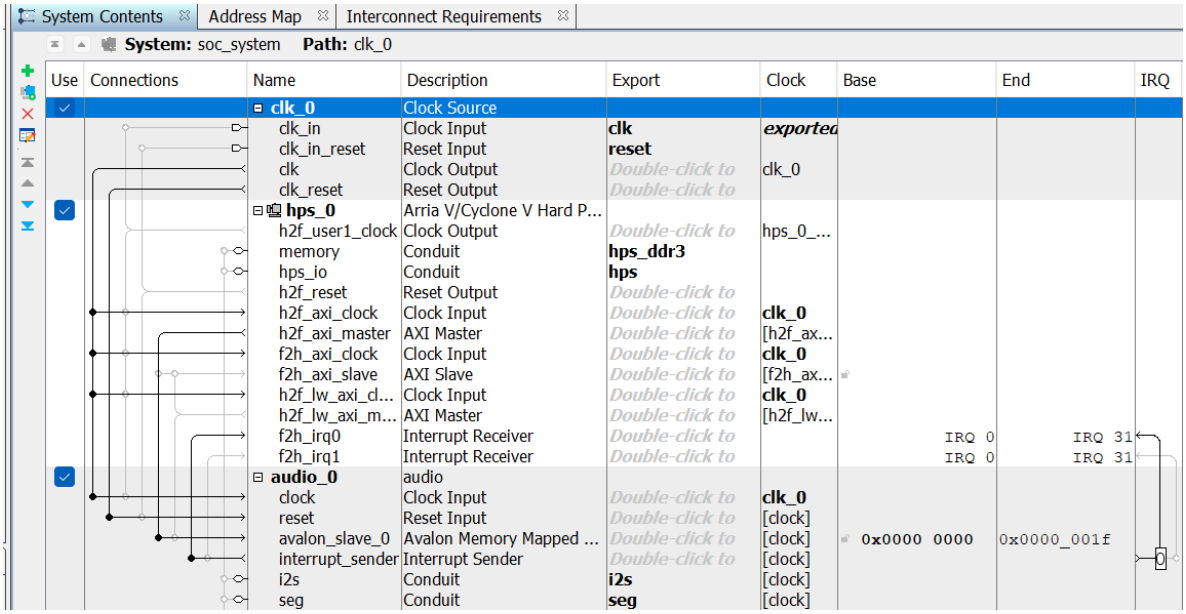


Figure 10: Qsys Connctions

3.2 1024-point Streaming FFT

1024-point Streaming FFT block gets raw audio data from the previous FIFO, performs FFT, and saves the frequency-domain results in the RAM.

3.2.1 FFT WRAPPER Block

1. **Inputs:**
   - `clk`: Clock signal line.
   - `rst_n`: Resets the state of the block to an initial state. Active low.
   - `go`: Starts a new cycle of calculation.
   - `ready`: Indicates the completion of collecting raw audio data.
   - `[13:0]data_in`: Raw audio data input bus line.
   - `[9:0]rd_addr_fft`: Read address bus line of FFT RAM.
2. **Outputs:**
   - `fftdone`: Indicates the completion of FFT calculation.
   - `rdreq`: Read request. It is asserted when read the raw audio data from FIFO
   - `[27:0]ram_q`: FFT RAM output bus.
3. **Functionality:**

   1024-point Streaming FFT is implemented after the time series FIFO. The corresponding source code is in `fft_wrapper.sv`. From the data we collected through the microphones, we need to perform FFT in order to detect the frequency of interest, i.e., to find the maximum power frequency bin. We instantiated an FFT IP core and a two-port RAM to obtain the frequency-domain data and store the FFT results.

   `FFT WRAPPER` has three main parts: FFT IP instance, two-port RAM IP instance, and FFT wrapper control logic.

   In the first place, an FFT IP is instantiated. It runs on variable streaming FFT mode, which is set to fixed-point computation and the natural order of inputs/outputs in the wizard. Note that the input data has to run in a data stream. As shown in Figure 11, `sink_sop` and `sink_eop` signals

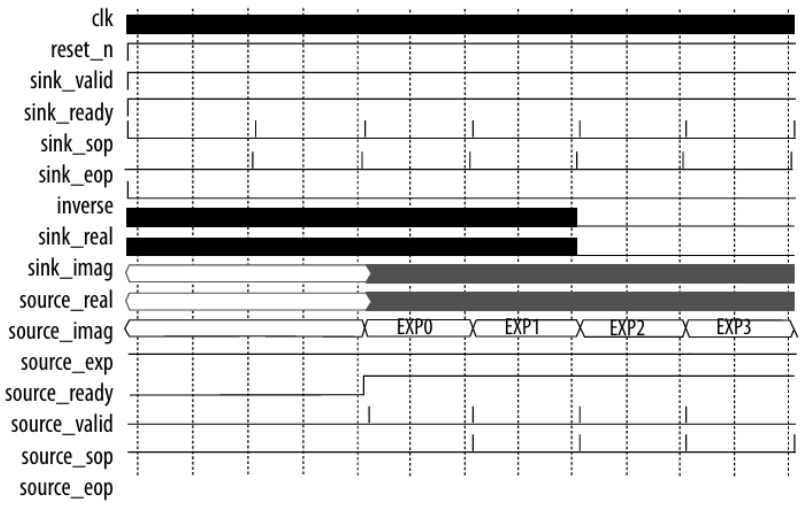


Figure 11: Variable Stream Mode Waveform (Source)

indicate the start of packet and end of packet of the input time-domain data, while `source_sop` and `source_eop` signals denote the output frequency-domain data, similarly.

In the second place, a two-port RAM is instantiated, which has separate read and write operations. The write address signal is controlled by the a FSM to ensure the FFT result can be saved to FFT RAM according to the `source_eop` signal.

Finally, another FSM controls the FFT and two-port RAM instances, ensuring that reading raw audio data and writing FFT results are correctly timed. The FSM has four states, as shown in Figure 12. In `READ` state, it reads raw audio data from the time series FIFO. In `write` state, it writes the FFT results to FFT RAM.

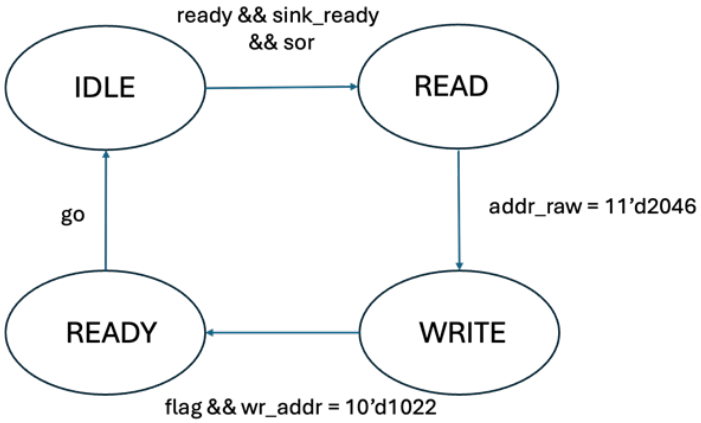


Figure 12: FSM of FFT Wrapper

3.2.2 FFT WRAPPER Simulation

Figure 13 shows the simulated waveform of the entire FFT wrapper. We can see that the control signals are in a continuous stream and the FFT data result is shown at the right timing. `fftdone` is asserted when FFT RAM stores 1024 samples and then sent to the next block, `freqdetect`.

After confirming the results with the ModelSim simulation, we conducted a test using a 1 kHz sound source and read the FFT RAMs by Avalon Bus. The results peaked at approximately 1kHz, as shown in Figure.

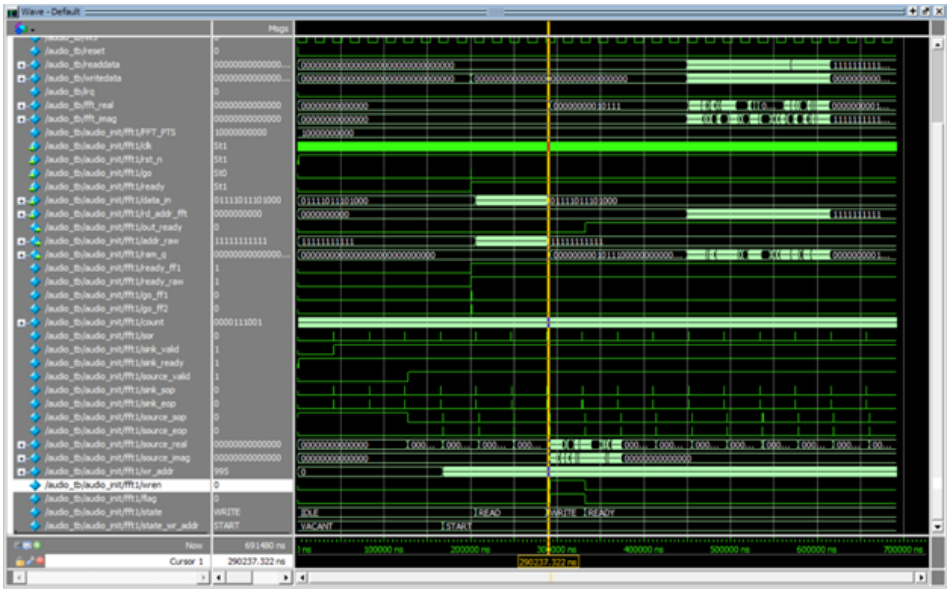


Figure 13: FFT Simulation Waveform

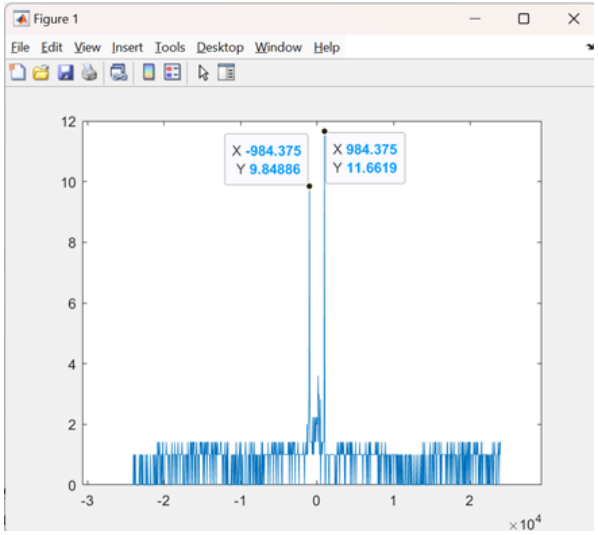


Figure 14: FFT Results in 1 kHz Test

3.3 DOA Computation

DOA computation block consists of two main parts: `frequent detect`, `DOA controller`. They obtain the FFT result from FFT RAM, find the frequency of interest, and calculate the angle of DOA.

3.3.1 FREQ. DETECT Block

1. **Inputs:**
   - `clk`: Drives the timing of the operations within the block. (50MHz)
   - `reset`: Used to reset the state of the block to an initial state.
   - `fftdone`: Denotes the completion of the FFT operation.
   - `[27:0]ramq`: Connected to FFT RAM output bus from the previous module.
2. **Outputs:**
   - `detectdone`: Indicates the completion of the detect operation.

- `[9:0]ramaddr`: Iterates the address of FFT RAM in previous module.
- `[9:0]maxbin`: Indicates the index of max bin, i.e., address of frequency of interest in the FFT RAM.

3. **Functionality:**

   The frequency detection module is triggered to start once the FFT has been fully computed. Then, it iterates through each bin in a predefined range to find the address of the bin containing the frequency of maximum magnitude. Though we know where the maximum signal should appear in our transform because we guarantee that it is 2 kHz, we detect which bin it is in to make the system more noise-tolerant and robust. This way, any problems that may lead to the FFT being different from ideal should not affect the calculations significantly. As long as the phase information is correct at the peak, the algorithm should proceed successfully.

   The iteration is performed by incrementing a wire that controls the read address of the first of the four FFT RAMs. After each increment, multipliers and adders connected combinationally compute the squared magnitude of each of the FFT values, stored as two 14-bit numbers, one real and one imaginary. Every time a new maximum is encountered, its index is updated to be the new maximum bin. Once all addresses are checked, the block signals the next block to start, as the value for the maximum bin is guaranteed to be correct.

3.3.2 DOA Controller (WEIGHT BLOCK)

1. **Inputs:**

   - `clk`: Drives the timing of the operations within the block. (50MHz)
   - `reset`: Used to reset the state of the block to an initial state.
   - `detectedtone`: Indicates the completion of frequency detect.
   - `[9:0]maxbin`: Connected to the frequency detect module.
   - `[27:0]ramq1, ramq2, ramq3, ramq4`: Connected to the output buses FFT RAMs.

2. **Outputs:**

   - `[9:0]rdaddr2, rdaddr3, rdaddr4`: Will be set to `maxbin` for all FFT RAMs.
   - `weightdone`: Indicates the completion of weight block.
   - `[5:0]bnum`: Indicates the ordinal number of the arrival directions. (0 to 35)
   - `[7:0]doa`: Indicates the angle of direction of arrival. (-90 to 90)
   - `[6:0]disp0, disp1, disp2`: Connected to the 7-segment displays.

3. **Functionality:**

   This block is responsible for performing the Bartlett multiplications. A ROM is preloaded with the Bartlett weights, and as the block iterates through each possible arrival direction, it reads the new delay coefficients from the ROM and multiplies them by the FFT signals extracted at the bin of the maximum FFT magnitude. Similarly to how the frequency detection block operates, the estimated angle of arrival is updated every time a new maximum output power is found. Once all possible angles have been checked, the module sets a "done" signal high for one clock cycle to update the numbers displayed on 6 7-segment displays. Below, an FSM describes the behavior of the block, and a block diagram shows the connections between the weight block and the frequency detection block.

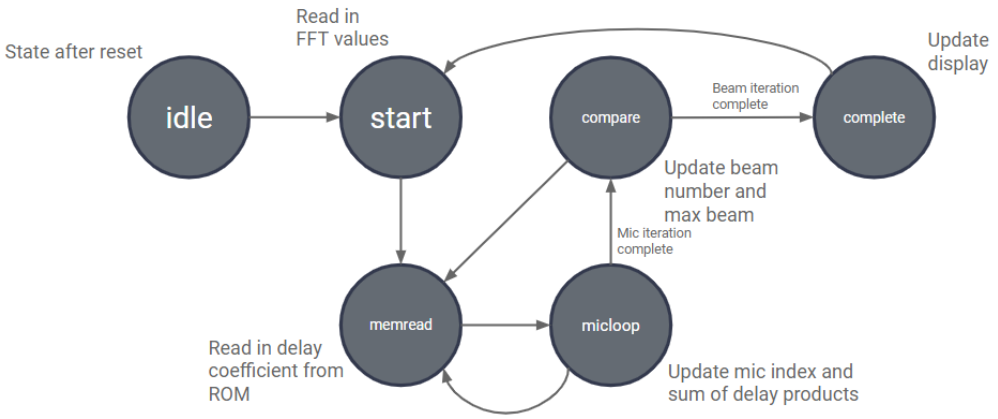


Figure 15: Weight Block FSM

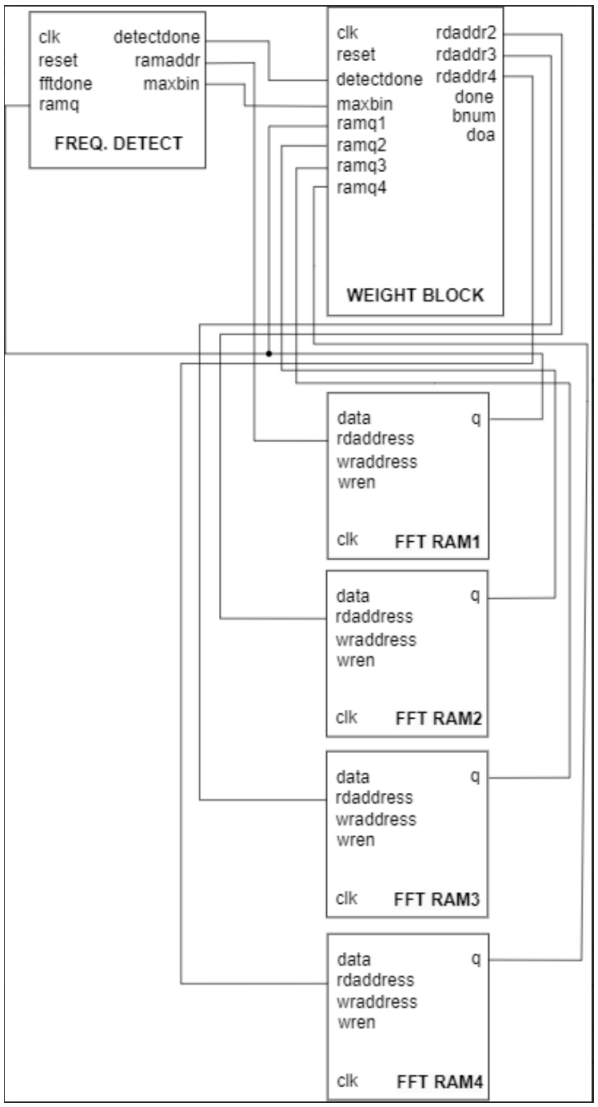


Figure 16: Weightblock/Freqdetect Connections

3.4 IP Core Generation Parameters

1. **myfifo**: FIFO - stores the raw audio data.
    - FIFO width: 16 bits.
    - FIFO deep: 1024 words.
    - Synchronized to `rdclk, wrclk`.
    - Read-side full signal.
2. **fft_block**: FFT
    - 1024 points.
    - Direction: Bi-directional.
    - Data Flow: Variable Streaming.
    - Input / Output Order: Natural.
    - Data Input Width: 14 bits.
    - Data Output Width: 14 bits.
3. **ram_fft_output**: RAM - stores the results of FFT computation.
    - 2-port 28-bit width and depth of 1024
4. **compmult**: ALTMULT_COMPLEX - computes product of frequency-domain signal and delay coefficient.
    - 28-bit input (14 signed real — 14 signed imag).
    - 56-bit output.
    - No pipelining.
5. **realmult**: LPM_MULT - computes the square of imaginary and real components to be summed for $|z|$.
    - Input is squared.
    - 32-bit signed input.
    - 64-bit unsigned output.
6. **delay_ROM**: ROM - stores the delay coefficients for each of directions to achieve 5 deg resolution.
    - 1-port 28-bit width and depth of 148.

4 Software Design

The software part is mainly used to reset the computation in the final design version. It sends a reset signal that asserts 'go' to the FPGA every 0.5 seconds to perform the whole end-to-end calculation.

```
while (1) {
    address.go = 1;
    write_addr(&address);
    address.go = 0;
    write_addr(&address);
    usleep(500000);
}
```

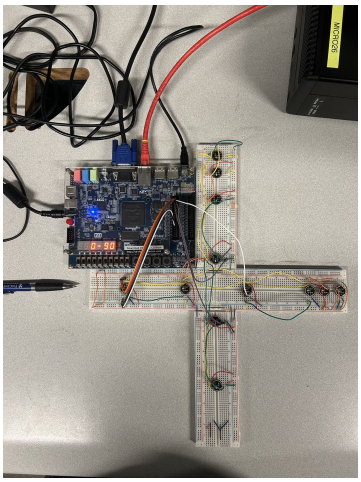
The most important function of the software part is testing. For example, the software can read the audio samples that the I2S interface decodes or the FFT results that the FFT instance outputs by simply iterating either RAM in the design. It is a helpful tool when we want to debug the hardware modules.

Here is one piece of testing code that we used during the FFT debugging process. Software iterates the address from 0 to 1023 and reads the data out of the FFT RAM. The outputs of 28-bit FFT RAMs have 14-bit real parts and 14-bit imaginary parts. So, the software obtains the data and separates the real and imaginary parts using `/, %` calculation. Bit shifting is used to extend the sign bit.

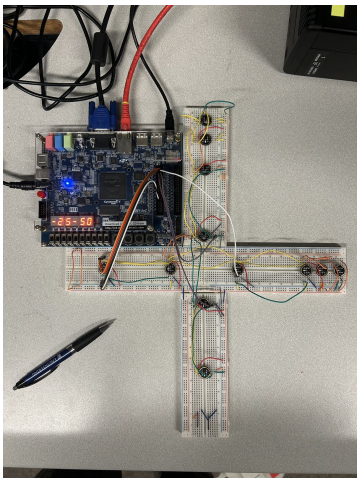
```
void read_audio() {
  audio_arg_t vla;
  if (ioctl(audio_fd, AUDIO_READ, &vla)) {
      perror("ioctl(AUDIO_READ) failed");
      return;
  }
  data1[buf_index] = vla.audio.left1;
  data2[buf_index] = vla.audio.right1;
  data3[buf_index] = vla.audio.left2;
  data4[buf_index] = vla.audio.right2;
  buf_index++;
}
...
while (buf_index < BUF_SIZE) {
  address.addr = buf_index;
  write_addr(&address);
  read_audio();
}
printf("done\n");
for (int i = 0; i < BUF_SIZE; i++) {    // Received data is 28 bits wide
  fprintf(fd1, "%d\n", ((data1[i] / 16384) << 18) >> 18); // Extend the sign
  fprintf(fd2, "%d\n", ((data1[i] % 16384) << 18) >> 18); // Extend the sign
...
}
```

5 Results

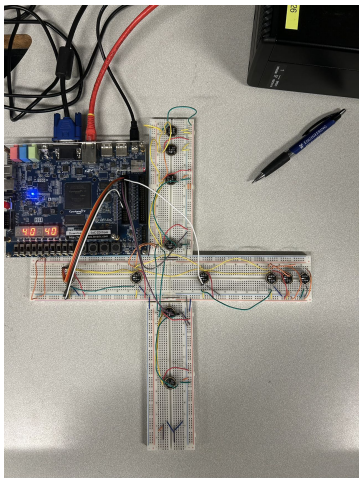
We can get accurate direction of arrival results, as shown in the figures below. The pen showed the direction of the audio source. The right three 7-segment displays illustrate the angle of DOA of the x-axis microphone array, and the left displays give the results of the y-axis.



(a) $(0, -90)$ expected



(b) $(-50, -25)$ expected



(c) $45, 45$ expected

Figure 17: System Results

6 Team Roles

Peiran Wang (pw2593)

My role is developing the I2S interface, wiring, and debugging the HW/SW system, including:

- Wrote SystemVerilog code for the I2S interface to collect audio data.
- Wrote SystemVerilog code for hardware and software communication by the Avalon Bus.
- Wrote testbenches to ensure the I2S interface worked properly.
- Implemented the layout of the microphone array.
- Extensive debugging in collaboration with teammates to combine all modules into a final solution.

As a summary of this project, I would emphasize the importance of performing small-scale tests before implementing the FPGA in terms of hardware design. Debugging took me a lot of time until I realized that doing small-scale experiments and ModelSim simulations would help me locate the bugs faster. Speaking of future projects, it is advisable to make use of simulation tools such as ModelSim to debug, even if the project deals with some real data, such as our project that collects real-world audio signals. Testing the modules one by one is a good idea because we usually make mistakes while connecting the modules. Python and MATLAB are great tools that can help along the way.

Dawn Yoo (dy2486)

I developed the FFT implementation on the HW interface side.

- Generated a FFT block using Quartus ip core and implemented control signals to correctly trigger the FFT block as we desired.
- Wrote testbench to verify the FFT module and checked its functionality in Questasim.
- Collaborated in debugging this module when incorporating the entire system together.

Elvis Wang (yw4082)

- Conducted testing to ensure proper reception of audio signals by the system.
- Reviewed and provided feedback on the audio.sv file to improve its functionality and performance.
- Collaborated in debugging efforts to ensure the system's reliability and accuracy.

Matheu Campbell (mgc2171)

- Researched beamforming methods and decided on final algorithm.
- Simulated the system in Matlab with synthetic and real data.
- Wrote Matlab scripts to calculate Bartlett coefficients and generate ROM initialization files.
- Wrote testbenches to test `freqdetect` and `weightblock` modules to test individually and when connected to each other, then confirmed correct operation in ModelSIM
- Wrote SystemVerilog to implement frequency detection, Bartlett multiplication, and angle display and updating on 7-segment displays.

The project was a valuable exercise in system design. By separating the system into discrete sequential parts early, we could each develop robust modules under the assumption that previous modules worked as expected. Then, the challenge becomes integrating them with one another. A more distributed system would be an interesting next challenge, where the processes aren't as linear. Alternatively, pipelining this system to make it even faster would add another layer of complexity.

References

Jin, S., Kim, D., Kim, H. S., Lee, C. H., Choi, J. S., & Jeon, J. W. (2008, July). Real-time sound source localization system based on FPGA. *2008 6th IEEE International Conference on Industrial Informatics* (pp. 673-677). IEEE.

Abusultan, M., Harkness, S., LaMeres, B. J., & Huang, Y. (2010, March). FPGA implementation of a Bartlett direction of arrival algorithm for a 5.8 ghz circular antenna array. *2010 IEEE Aerospace Conference* (pp. 1-10). IEEE.

A MATLAB Code

doa.m

```
% Implements DOA algorithm on simulated linear array
% Author: Matheu
close all

% Array Parameters
m = 4; % number of microphones
d = 0.104;  % distance between microphones (in meters)
Fs = 48000;  % sampling rate of the microphones

% Signal Parameters
f = 2000;  % frequency of signal of interest
aoa = 75;  % intended angle of arrival (in degrees)
snr = 5;  % signal power to noise power ratio in dBW

% Trial Parameters
n = 1024;  % samples in data block
snum = 36; % number of sectors to split half-circle into

bnum = snum+1;  % number of beams to form
arrsig = generate_array_signals(m, d, aoa, f, n, Fs, snr);

% Plot signals to verify delay
figure(1);
title("Input Signals")
xlabel("Sample")
ylabel("Value")
hold on;
for i = 1:m
    plot(arrsig(i, 1:50))
end
hold off;

% Generate FFT for each microphone
L = 1024;
spatial_spectrum = zeros(m, L);

% figure(2);
for i=1:m
    spatial_spectrum(i, :) = fft(arrsig(i, :), L);
    subplot(4, 2, i)
%     plot(Fs/L*(-L/2:L/2-1), angle(spatial_spectrum(i, :)))
    plot(Fs/L*(-L/2:L/2-1), abs(fftshift(spatial_spectrum(i, :))))
end

% Extract bin with FFT peak
[~, tbin] = max(spatial_spectrum(1, 1:L/2));

% Compute delay matrix
dmat = zeros(m, bnum);  % Rows are sensors, columns are directions
angs = (-pi/2:pi/snum:pi/2);
wlength = 343/f;

for n = 1:m  % Iterate over sensors
```

```
    shift_constant = (2*pi*d*(n-1))/wlength;
    shvec = shift_constant*sin(angs);
    eshifts = exp(-1i*shvec);
    dmat(n, :) = eshifts./abs(eshifts);
end

% Scale delay matrix to fit in 14-bit integer.
dmat = dmat * 10^3;

% Apply delays and calculate power
sigvec = spatial_spectrum(:, tbin);
outvec = dmat.'*sigvec;
pwrvec = abs(outvec).^2;

[pmax, ind] = max(pwrvec);
doa_res = angs(ind)*180/pi;

% Plot PSD
figure(3);
polarplot(angs, pwrvec);
title("PSD");
thetalim([-90, 90]);
ax = gca;
ax.ThetaZeroLocation = 'top';
ax.ThetaDir = 'clockwise';
```

generate_array_signals.m

```
function outmat = generate_array_signals(m, d, aoa, f, n, Fs, snr)
%GENERATE_SIGNALS Synthesizes n samples of data for an m-element
    linear array
%   m: number of sensors
%   d: distance between sensors
%   aoa: angle of arrival
%   f: signal frequency
%   n: number of samples
%   Fs: sampling rate
%   snr: signal to noise power ratio

% Create base signal: n samples of f Hz sine wave sampled at Fs
svec = (1:n);
tvec = svec/Fs;

% Create time delay without (i-1) term
wlength = 343/f;
tdelay = 2*pi*d*sin(pi*aoa/180)/wlength;

% Create (m x n) output vector where each row is the original signal
% delayed the appropriate amount
outmat = zeros(m, n);

for i = 1:m
    outmat(i, :) = awgn(sin(2*pi*f*tvec + (i-1)*tdelay), snr, '
        measured');
%     outmat(i, :) = cos(2*pi*f*tvec + (i-1)*tdelay);
end
```

```
end
```

createMIF.m

```
function createMIF(infilepath, width, fwidth, scale, fname)
    %% Parameters
    % infilepath: filepath to .mat
    % width: words in RAM
    % fdepth: width of fractional portion of binary val
    % scale: scale factor for each value
    % fname: output file name

    % Converts matrix into .mif format for loading into ROM
    datastruct = load(infilepath);
    datamat = datastruct.spatial_spectrum4 .* scale;
    dims = size(datamat);
    rowcount = dims(1);
    colcount = dims(2);

    % Define data format for components of delay coefficients
    % Ex: depth = 28, fdepth = 4 --> [10b.4b | 10b.4b]
    q = quantizer([width/2, fwidth]);

    header = sprintf('WIDTH=%d;\nDEPTH=%d;\n', width, rowcount*
        colcount);
    header = header + "ADDRESS_RADIX=HEX;" + newline + "DATA_RADIX=BIN
        ;" + newline;
    writelines(header, fname);
    writelines("CONTENT", fname, WriteMode="append");
    writelines("BEGIN", fname, WriteMode="append");

    % Delay ROM Structure
    % [  B1  ][  B2  ][  B3  ][  B4  ]
    % [M1..M4][M1..M4][M1..M4][M1..M4]
    % Matrix Structure = 4 x 13 (mics x directions)
    for bnum = 1:colcount
        for mnum = 1:rowcount
            addr = dec2hex((bnum-1)*rowcount + (mnum-1), 2);
            val = datamat(mnum, bnum);
            binval_a = num2bin(q, real(val));
            binval_b = num2bin(q, imag(val));
            binval = strcat(binval_a, binval_b);
            nextline = sprintf(' %s : %s;', addr, binval);
            writelines(nextline, fname, Writemode="append");
        end
    end

    writelines("END;", fname, WriteMode="append");
end
```

B Hardware Code

audio.sv

```
/* audio.sv
 * Top Module
 * Author: Peiran
 * Notes:
 * 1. There is a startup time for mic.
 * 2. Contention across clock domains matters.
 */
module audio(
        input logic clk,        // 50M, 20ns
        input logic reset,
        input logic chipselect,
        input logic read,
        input logic write,
        input logic [31:0] writedata,
        input logic [2:0] address,
        input logic SD1,        // Serial data input: microphone set 1
        input logic SD2,        // Set 2
        input logic SD3,        //
        input logic SD4,        //
        input logic SCK,        // Sampling rate * 32 bits * 2 channels: 320

        output logic WS,        // Sampling rate
        output logic irq,       // Reserved
        output logic [31:0] readdata,
        // X
        output logic [6:0] disp2,
        output logic [6:0] disp1,
        output logic [6:0] disp0,
        // Y
        output logic [6:0] disp5,
        output logic [6:0] disp4,
        output logic [6:0] disp3,

        // VGA
        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
);

logic rst_n = 0;
logic sck_rst = 1;
logic [3:0] count1 = 4'd0;
logic [3:0] count2 = 4'd0;
logic [5:0] clk_cnt;                            // 64 counter to generate WS signal
logic [4:0] stretch_cnt1, stretch_cnt2; // Strech signal for synchro
logic go, go_SCK;// go command to start sampling and calculation
logic [23:0] right1, left1, right2, left2, right3, left3, right4, left4;  // Temp memory
// RAM for raw data
logic wrreq;                    // write enable for raw data RAM
logic [10:0] wr_addr;   // RAM write address
logic [10:0] rd_addr;   // RAM read address
// RAM inputs
logic [15:0] ram1_in, ram2_in, ram3_in, ram4_in, ram5_in, ram6_in, ram7_in, ram8_in;
logic [15:0] ram1q, ram2q, ram3q, ram4q, ram5q, ram6q, ram7q, ram8q;    // RAM outputs
// Asserted when raw data RAM is full
logic ready1, ready2, ready3, ready4, ready5, ready6, ready7, ready8;
logic rdreq1, rdreq2, rdreq3, rdreq4, rdreq5, rdreq6, rdreq7, rdreq8;
// FFT wrapper
// RAM outputs for fft RAM
logic [27:0] ram1_fft, ram2_fft, ram3_fft, ram4_fft, ram5_fft, ram6_fft, ram7_fft, ram8_fft;

```

```
63
64  // Frequency detector
65  logic fftdone, detectdone;
66  logic [9:0] rd_addr_fd_x, maxbin_x;
67  logic [9:0] rd_addr_fd_y, maxbin_y;
68
69  // Weight block
70  logic [9:0] rdaddr2_wb_x, rdaddr3_wb_x, rdaddr4_wb_x;
71  logic [9:0] rdaddr2_wb_y, rdaddr3_wb_y, rdaddr4_wb_y;
72  logic wbdone, wbdone_SCK;
73  logic [5:0] bnum_x, bnum_y;
74  logic [7:0] doa_x, doa_y;
75
76  // VGA logic
77  logic [10:0] xcoor;
78  logic [9:0] ycoor;
79
80  // Testing logic
81  logic [9:0] rd_addr_fft1, rd_addr_fft2, rd_addr_fft3, rd_addr_fft4;
82  logic [9:0] rd_addr_fft5, rd_addr_fft6, rd_addr_fft7, rd_addr_fft8;
83
84  enum {IDLE, WRITE, READ} state;
85
86  /* Generate reset signal
87   */
88  always_ff @(posedge clk) begin
89          count1 <= count1 + 4'd1;
90          if (count1 == 4'b1111)
91                  rst_n <= 1'd1;
92  end
93
94  always_ff @(negedge SCK) begin
95          count2 <= count2 + 4'd1;
96          if (count2 == 4'b1111)
97                  sck_rst = 1'd0;
98  end
99
100 /* Go signal synchronizer
101  * go -> go_SCK
102  * wbdone -> wbdone_SCK
103  * Faster clk -> Slower SCK
104  * 320/20 = 16
105  * Stretch the go_clk signal so that SCK can get
106  */
107 always_ff @(posedge clk) begin
108         if (~rst_n) begin
109                 stretch_cnt1 <= 5'd0;
110                 stretch_cnt2 <= 5'd0;
111         end else begin
112                 if (go) begin
113                         stretch_cnt1 <= 5'd16;
114                 end else if (stretch_cnt1 > 5'd0) begin
115                         stretch_cnt1 <= stretch_cnt1 - 5'd1;
116                 end
117                 if (wbdone) begin
118                         stretch_cnt2 <= 5'd16;
119                 end else if (stretch_cnt2 > 5'd0) begin
120                         stretch_cnt2 <= stretch_cnt2 - 5'd1;
121                 end
122         end
123 end
124 assign go_SCK = (stretch_cnt1 > 0) ? 1'd1 : 1'd0;
```

```
125 assign wbdone_SCK = (stretch_cnt2 > 0) ? 1'd1 : 1'd0;
126
127 /* WS clock generator
128  * 64 division
129  */
130 always_ff @(negedge SCK) begin // Negedge of SCK
131         if (sck_rst) begin
132                 clk_cnt <=  6'd0;
133         end else begin
134                 clk_cnt <= clk_cnt + 6'd1;
135         end
136 end
137
138 assign WS = clk_cnt[5];  // Flip at 31st cycles
139
140 /* I2S decoder
141  * Get left and right channels based on the clk_cnt counter
142  * 0-25 left channel
143  * 32-57 right channel
144  */
145 always_ff @(negedge SCK) begin
146         if (sck_rst) begin              // Initialize
147                 left1 <= 24'd0;
148                 right1 <= 24'd0;
149                 left2 <= 24'd0;
150                 right2 <= 24'd0;
151                 left3 <= 24'd0;
152                 right3 <= 24'd0;
153                 left4 <= 24'd0;
154                 right4 <= 24'd0;
155                 ram1_in <= 16'd0;
156                 ram2_in <= 16'd0;
157                 ram3_in <= 16'd0;
158                 ram4_in <= 16'd0;
159                 ram5_in <= 16'd0;
160                 ram6_in <= 16'd0;
161                 ram7_in <= 16'd0;
162                 ram8_in <= 16'd0;
163
164                 wr_addr <= 11'd2047;// 0 address is avaible
165                 wrreq <= 1'd0;  // Initialize with 0 to reset RAMs
166                 state <= IDLE;
167         end else begin
168                 // Read from the bus
169                 if (clk_cnt > 0 && clk_cnt < 25) begin // Left channel, 24-bit, MSB first
170                         left1 <= {left1[22:0], SD1};
171                         left2 <= {left2[22:0], SD2};
172                         left3 <= {left3[22:0], SD3};
173                         left4 <= {left4[22:0], SD4};
174                 end else if (clk_cnt > 32 && clk_cnt < 57) begin        // Right channel
175                         right1 <= {right1[22:0], SD1};
176                         right2 <= {right2[22:0], SD2};
177                         right3 <= {right3[22:0], SD3};
178                         right4 <= {right4[22:0], SD4};
179                 end
180                 // FSM:
181                 // IDLE: Transit to WRITE state when go_SCK is high
182                 // WRITE: Write raw data to RAMs
183                 // READ: Ready to be read to the FFT wrapper
184                 case (state)
185                         IDLE: begin
186                                 if (go_SCK)
```

```
                                        state <= WRITE;
                                else
                                        state <= IDLE;
                        end
                        WRITE:begin
                                if (clk_cnt == 57) begin
                                        ram1_in <= left1[23:8]; // Discard the least 8 bits
                                        ram2_in <= right1[23:8];
                                        ram3_in <= left2[23:8];
                                        ram4_in <= right2[23:8];
                                        ram5_in <= left3[23:8];
                                        ram6_in <= right3[23:8];
                                        ram7_in <= left4[23:8];
                                        ram8_in <= right4[23:8];

                                        wrreq <= 1'd1;
                                        wr_addr <= wr_addr + 11'd1; // Start with address 0
                                end else if (clk_cnt == 58) begin
                                        wrreq <= 1'd0;
                                end

                                if (wr_addr == 10'd1023)
                                        state <= READ;
                                else
                                        state <= WRITE;
                        end
                        READ:   begin
                                wr_addr <= 11'd2047;
                                if (go_SCK) begin
                                        state <= WRITE;
                                end else begin
                                        state <= READ;
                                end
                        end
                        default: begin
                                        state <= IDLE;
                        end
                endcase
        end
end

/* Two Port RAM Instantiation
 * Raw data from i2s bus
 */
myfifo fifo1(
        .data   (ram1_in),
        .rdclk  (clk),
        .rdreq  (rdreq1),
        .wrclk  (SCK),
        .wrreq  (wrreq),
        .q              (ram1q),
        .rdfull (ready1),
        .wrempty()
);

myfifo fifo2(
        .data   (ram2_in),
        .rdclk  (clk),
        .rdreq  (rdreq2),
        .wrclk  (SCK),
        .wrreq  (wrreq),
        .q              (ram2q),
```

```
249         .rdfull (ready2),
250         .wrempty()
251 );
252
253 myfifo fifo3(
254         .data   (ram3_in),
255         .rdclk  (clk),
256         .rdreq  (rdreq3),
257         .wrclk  (SCK),
258         .wrreq  (wrreq),
259         .q              (ram3q),
260         .rdfull (ready3),
261         .wrempty()
262 );
263
264 myfifo fifo4(
265         .data   (ram4_in),
266         .rdclk  (clk),
267         .rdreq  (rdreq4),
268         .wrclk  (SCK),
269         .wrreq  (wrreq),
270         .q              (ram4q),
271         .rdfull (ready4),
272         .wrempty()
273 );
274
275 myfifo fifo5(
276         .data   (ram5_in),
277         .rdclk  (clk),
278         .rdreq  (rdreq5),
279         .wrclk  (SCK),
280         .wrreq  (wrreq),
281         .q              (ram5q),
282         .rdfull (ready5),
283         .wrempty()
284 );
285 myfifo fifo6(
286         .data   (ram6_in),
287         .rdclk  (clk),
288         .rdreq  (rdreq6),
289         .wrclk  (SCK),
290         .wrreq  (wrreq),
291         .q              (ram6q),
292         .rdfull (ready6),
293         .wrempty()
294 );
295 myfifo fifo7(
296         .data   (ram7_in),
297         .rdclk  (clk),
298         .rdreq  (rdreq7),
299         .wrclk  (SCK),
300         .wrreq  (wrreq),
301         .q              (ram7q),
302         .rdfull (ready7),
303         .wrempty()
304 );
305 myfifo fifo8(
306         .data   (ram8_in),
307         .rdclk  (clk),
308         .rdreq  (rdreq8),
309         .wrclk  (SCK),
310         .wrreq  (wrreq),
```

```
311         .q              (ram8q),
312         .rdfull (ready8),
313         .wrempty()
314 );
315
316 /* FFT wrapper module instantiation
317  */
318 // X axis
319 fft_wrapper fft1(
320         .clk(clk),
321         .rst_n(rst_n),
322         .go(detectdone),
323         .ready(ready1),             // Raw data ready
324         .data_in(ram1q[15:2]),  // Raw data in
325         .rd_addr_fft(rd_addr_fft1),// Read address of fft RAMs
326
327         .fftdone(fftdone),
328         .rdreq(rdreq1),
329         .ram_q(ram1_fft)                        // fft results from fft RAMs
330 );
331
332 fft_wrapper fft2(
333         .clk(clk),
334         .rst_n(rst_n),
335         .go(detectdone),
336         .ready(ready2),
337         .data_in(ram2q[15:2]),
338         .rd_addr_fft(rd_addr_fft2),
339
340         .fftdone(),
341         .rdreq(rdreq2),
342         .ram_q(ram2_fft)
343 );
344
345 fft_wrapper fft3(
346         .clk(clk),
347         .rst_n(rst_n),
348         .go(detectdone),
349         .ready(ready3),
350         .data_in(ram3q[15:2]),
351         .rd_addr_fft(rd_addr_fft3),
352
353         .fftdone(),
354         .rdreq(rdreq3),
355         .ram_q(ram3_fft)
356 );
357
358 fft_wrapper fft4(
359         .clk(clk),
360         .rst_n(rst_n),
361         .go(detectdone),
362         .ready(ready4),
363         .data_in(ram4q[15:2]),
364         .rd_addr_fft(rd_addr_fft4),
365
366         .fftdone(),
367         .rdreq(rdreq4),
368         .ram_q(ram4_fft)
369 );
370 // Y
371 fft_wrapper fft5(
372         .clk(clk),
```

```
373         .rst_n(rst_n),
374         .go(detectdone),
375         .ready(ready5),
376         .data_in(ram5q[15:2]),
377         .rd_addr_fft(rd_addr_fft5),
378 
379         .fftdone(),
380         .rdreq(rdreq5),
381         .ram_q(ram5_fft)
382 );
383 fft_wrapper fft6(
384         .clk(clk),
385         .rst_n(rst_n),
386         .go(detectdone),
387         .ready(ready6),
388         .data_in(ram6q[15:2]),
389         .rd_addr_fft(rd_addr_fft6),
390 
391         .fftdone(),
392         .rdreq(rdreq6),
393         .ram_q(ram6_fft)
394 );
395 fft_wrapper fft7(
396         .clk(clk),
397         .rst_n(rst_n),
398         .go(detectdone),
399         .ready(ready7),
400         .data_in(ram7q[15:2]),
401         .rd_addr_fft(rd_addr_fft7),
402 
403         .fftdone(),
404         .rdreq(rdreq7),
405         .ram_q(ram7_fft)
406 );
407 fft_wrapper fft8(
408         .clk(clk),
409         .rst_n(rst_n),
410         .go(detectdone),
411         .ready(ready8),
412         .data_in(ram8q[15:2]),
413         .rd_addr_fft(rd_addr_fft8),
414 
415         .fftdone(),
416         .rdreq(rdreq8),
417         .ram_q(ram8_fft)
418 );
419 
420 // Test Interface
421 assign rd_addr_fft1 = rd_addr_fd_x;
422 assign rd_addr_fft2 = rdaddr2_wb_x;
423 assign rd_addr_fft3 = rdaddr3_wb_x;
424 assign rd_addr_fft4 = rdaddr4_wb_x;
425 assign rd_addr_fft5 = rd_addr_fd_y;
426 assign rd_addr_fft6 = rdaddr2_wb_y;
427 assign rd_addr_fft7 = rdaddr3_wb_y;
428 assign rd_addr_fft8 = rdaddr4_wb_y;
429 
430 /* Frequency detector instantiation
431 */
432 freqdetect fd_inst1(
433         .clk            (clk),          // 50 MHz, 20 ns
434         .reset          (~rst_n),
```

```
435         .fftdone        (fftdone),      // Set high upon FFT block finishing
436         .ramq           (ram1_fft),     // Output port of channel 1 FFT RAM
437
438         .detectdone     (detectdone),           // Set high when iteration is complete
439         .ramaddr        (rd_addr_fd_x),         // Address to read from RAM
440         .maxbin         (maxbin_x)// Index of max bin
441 );
442
443 freqdetect fd_inst2(
444         .clk            (clk),          // 50 MHz, 20 ns
445         .reset          (~rst_n),
446         .fftdone        (fftdone),      // Set high upon FFT block finishing
447         .ramq           (ram5_fft),     // Output port of channel 1 FFT RAM
448
449         .detectdone     (),             // Set high when iteration is complete
450         .ramaddr        (rd_addr_fd_y),         // Address to read from RAM
451         .maxbin         (maxbin_y)// Index of max bin
452 );
453
454 /* Weight block instantiation
455 */
456 weightblock wb_inst1(
457         .clk            (clk),
458         .reset          (~rst_n),
459         .detectdone     (detectdone),
460         .maxbin         (maxbin_x),
461         .ramq1          (ram1_fft),     // FFT RAMs
462         .ramq2          (ram2_fft),
463         .ramq3          (ram3_fft),
464         .ramq4          (ram4_fft),
465
466         .rdaddr2        (rdaddr2_wb_x), // Will be set to maxbin for all FFT RAMs
467         .rdaddr3        (rdaddr3_wb_x),
468         .rdaddr4        (rdaddr4_wb_x),
469         .weightdone     (wbdone),
470         .bnum           (bnum_x),               // (0 to 36)
471         .doa            (doa_x),                // (-90 to 90)
472
473         .disp2          (disp2),
474         .disp1          (disp1),
475         .disp0          (disp0)         // 7seg displays
476 );
477
478 weightblock wb_inst2(
479         .clk            (clk),
480         .reset          (~rst_n),
481         .detectdone     (detectdone),
482         .maxbin         (maxbin_y),
483         .ramq1          (ram5_fft),     // FFT RAMs
484         .ramq2          (ram6_fft),
485         .ramq3          (ram7_fft),
486         .ramq4          (ram8_fft),
487
488         .rdaddr2        (rdaddr2_wb_y), // Will be set to maxbin for all FFT RAMs
489         .rdaddr3        (rdaddr3_wb_y),
490         .rdaddr4        (rdaddr4_wb_y),
491         .weightdone     (),
492         .bnum           (bnum_y),               // (0 to 36)
493         .doa            (doa_y),                // (-90 to 90)
494
495         .disp2          (disp5),
496         .disp1          (disp4),
```

```
        .disp0          (disp3)         // 7seg displays
);

vga_ball vga_inst(
    clk, ~rst_n, xcoor, ycoor,

        VGA_R, VGA_G, VGA_B,
        VGA_CLK, VGA_HS, VGA_VS,
        VGA_BLANK_n,
        VGA_SYNC_n
);

/* Avalon bus configuration
 * readdata: FPGA -> HPS
 * writedata: HPS -> FPGA
 */
always_ff @(posedge clk) begin
        if (reset) begin
                irq <= 1'd0;
                readdata <= 32'd0;
                go <= 1'd0;
                xcoor <= 11'd630;
                ycoor <= 10'd240;
        end else if (chipselect && read) begin
                case (address)
                3'h0: readdata <= {{24{doa_x[7]}}, doa_x};
                3'h1: readdata <= {{24{doa_y[7]}}, doa_y};
                endcase
        end else if (chipselect && write) begin
                case (address)
                3'h0 : go <= writedata[0];
                3'h1 : xcoor <= writedata[10:0];        // Lower 8 digits of xcoor
                3'h2 : ycoor <= writedata[9:0]; // Lower 8 digits of ycoor
                endcase
        end
end

endmodule
```

fft_wrapper.sv

```
/* FFT Wrapper for One Channel
 * Author: Peiran, Dawn
 * Includes:
 * 1. FFT ip core, RAM ip core
 * 2. FFT control logic
 * FFT Mode: Variable Streaming
 * Version Notes:
 * I made the variable names consistent with fft_block's for clarity.
 */
module fft_wrapper(
        input logic clk,
        input logic rst_n,                      // Active low
        input logic go,                         // Reset FSM
        input logic ready,                      // Raw data RAMs ready to be read
        input logic [13:0]data_in,      // Raw data input
        input logic [9:0]rd_addr_fft,   // Read address of fft RAMs

        output logic fftdone,           // fft wrapper output ready
        output logic rdreq,                     // Read request
        output logic [27:0]ram_q                // fft results from fft RAM
```

```
);

logic [9:0]count;               // 1024 counter
logic sor;                              // start of read (raw data)
// fft ip singals
logic sink_valid, sink_ready, source_valid;
logic sink_sop, sink_eop, source_sop, source_eop;
logic [13:0] source_real, source_imag;
// RAM signals
logic [9:0] wr_addr;
logic wren;
logic [10:0] addr_raw;
logic flag; // Toggle to differ control wren

enum {IDLE, READ, WRITE, READY} state;
enum {VACANT, START} state_wr_addr;
parameter FFT_PTS = 11'd1024;

/* Generate sink_eop & sink_sop stream signals for fft_block
 * sor (start of raw) ensures addr_raw to output correctly,
 * which can align with the sink_sop & sink_eop stream
 */
always @(posedge clk) begin
        if (~rst_n) begin
                count = 10'd1;
                sink_valid <= 1'd0;
                sink_eop <= 1'd0;
                sink_sop <= 1'd0;
                sor <= 1'd0;
        end else begin
                count <= count + 1'b1;
                if (count == 10'd0) begin
                        sink_eop <= 0;
                        sink_sop <= 1;
                        sink_valid <= 1;
                end else if(count == 10'd1) begin
                        sink_sop <= 0;
                end else if (count == 10'd1021) begin
                        sor <= 1;
                end else if (count == 10'd1022) begin
                        sor <= 0;
                end else if (count == 10'd1023) begin
                        sink_eop <= 1;
                end
        end
end

/* Control Calculation
 * FSM
 * IDLE: Wait for the ready signals from raw data RAM and FFT engine
 * READ: Read the raw data out of the RAM
 * WRITE: Write the fft results to RAM when source_eop
 * (I intentionally used eop instead of sop since eop is one cycle earlier than sop)
 * (So that it won't miss the first output in terms of wren)
 * READY: fft RAM is ready to be read to the next stage
 */
always_ff @(posedge clk) begin
        if (~rst_n) begin
                state <= IDLE;
                addr_raw <= 11'd2047;
                wren <= 1'd0;
                fftdone <= 1'd0;
```

```
                rdreq <= 1'd0;
                flag <= 1'd0;
        end else begin
                case (state)
                        IDLE: begin
                                fftdone <= 1'd0;
                                if (ready && sink_ready && sor)
                                        state <= READ;
                                else
                                        state <= IDLE;
                        end
                        READ: begin
                                addr_raw <= addr_raw + 11'd1;
                                if (addr_raw == 11'd1023) begin
                                        rdreq <= 1'd0;
                                end else if (addr_raw == 11'd2046) begin
                            // because the calculation delay of fft is about 1024
                                        state <= WRITE;
                                end else if (addr_raw == 11'd2047) begin
                                        rdreq <= 1'd1;
                                end else begin
                                        state <= READ;
                                end
                        end
                        WRITE: begin
                                if (source_eop) begin
                                        wren <= 1'd1;
                                        flag <= 1'd1;
                                end
                                if (flag && wr_addr == 10'd1022) begin
                                        state <= READY;
                                end else begin
                                        state <= WRITE;
                                end
                        end
                        READY: begin
                                flag <= 1'd0;
                                wren <= 1'd0;
                                fftdone <= 1'd1;
                                if (go)
                                        state <= IDLE;
                                else
                                        state <= READY;
                        end
                        default: begin
                                state <= IDLE;
                        end
                endcase
        end
end

/* Generate Control Singals for fft RAM
 * FSM
 * VACANT: wait until source_eop to start wr_addr streaming
 * START: generate wr_addr singal stream for FFT RAM
 */
always @(posedge clk) begin
        if (~rst_n) begin
                wr_addr <= 10'd0;
                state_wr_addr <= VACANT;
        end else begin
                case (state_wr_addr)
```

```
                        VACANT: begin
                                wr_addr <= 10'd0;
                                if (source_eop)
                                        state_wr_addr <= START;
                                else
                                        state_wr_addr <= VACANT;
                        end
                        START: begin
                                wr_addr <= wr_addr + 10'd1;
                        end
                        default: state_wr_addr <= VACANT;
                endcase
        end
end

fft_block fft_inst(
        .clk(clk),
        .reset_n(rst_n),
        .sink_valid(sink_valid), // Asserted when data is valid
        .sink_ready(sink_ready), // Output. Asserted when fft engine can accept data.
        .sink_error(2'b00),     // Error
        .sink_sop(sink_sop),            // Start of input
        .sink_eop(sink_eop),            // End of input
        .sink_real(data_in),// Real input data (signed)
        .sink_imag(14'd0),
        .fftpts_in(FFT_PTS),                    // The number of points
        .inverse(1'b0),
        .source_valid(source_valid),            // Output valid
        .source_ready(1'b1),// Asserted when downstream module is able to accept data
        .source_error(),                                        // Output error
        .source_sop(source_sop), // Start of output. Only valid when source valid
        .source_eop(source_eop),
        .source_real(source_real),
        .source_imag(source_imag),
        .fftpts_out()
);

ram_fft_output fft_ram1(
        .clock          (clk),
        .data           ({source_real, source_imag}),   // 28 bits width
        .rdaddress      (rd_addr_fft),  // 10 bits width address
        .wraddress      (wr_addr),
        .wren           (wren),
        .q              (ram_q)
);

endmodule
```

freqdetect.sv

```
// Author: Matheu
module freqdetect(
                input logic clk,                        // 50 MHz, 20 ns
                input logic reset,                      // Reset key is 0
                input logic fftdone,            // Set high upon FFT block finishing
                input logic [27:0] ramq,        // Output port of channel 1 FFT RAM

                output logic detectdone,        // Set high when iteration is complete
                output logic [9:0] ramaddr,     // Address to read from RAM
                output logic [9:0] maxbin       // Index of max bin
);
```

```
12
13 enum {idle, readone, readtwo, multiply, compare, compareone, comparetwo, complete} state;
14 // logic [9:0] ramaddr_rv;  // Bit-reversal of ramaddr (restores linear index in FFT)
15 logic signed [13:0] real_c;
16 logic signed [13:0] imag_c;
17 logic [28:0] cursqmag;
18 logic [28:0] maxsqmag;
19
20 assign cursqmag = real_c*real_c + imag_c*imag_c;
21 // assign ramaddr_rv = {ramaddr[0], ramaddr[1], ramaddr[2], ramaddr[3], ramaddr[4], ramaddr[5]
22
23 always_ff @(posedge clk) begin
24         if (reset) begin
25                 detectdone <= 0;
26
27                 ramaddr <= 10'b0;
28                 maxbin <= 10'b0;
29                 real_c <= 14'b0;
30                 imag_c <= 14'b0;
31                 maxsqmag <= 27'b0;
32
33                 state <= idle;
34         end else begin
35                 case (state)
36                         idle:
37                                 if (fftdone && !detectdone)     state <= readone;
38                         readone:                        // First read cycle; ramaddr set duri
39                                 state <= readtwo;
40                         readtwo:                        // Second read cycle
41                                 state <= multiply;
42                         multiply: begin         // Read components into multiplier input regis
43                                 real_c <= ramq[27:14];
44                                 imag_c <= ramq[13:0];
45                                 state <= compare;
46                         end
47                         compare: begin          // Update bin corresponding to squared mag ma
48                                 if ((cursqmag > maxsqmag) && (ramaddr > 10'd30)) begin
49                                         maxbin <= ramaddr;
50                                         maxsqmag <= cursqmag;
51                                 end
52
53                                 if (ramaddr == 10'd100) begin
54                                         detectdone <= 1;
55                                         state <= compareone;
56                                 end else begin
57                                         state <= readone;
58                                         ramaddr <= ramaddr + 10'd1;
59                                 end
60                         end
61                         compareone: begin
62                                 ramaddr <= maxbin;
63                                 detectdone <= 0;
64                                 state <= comparetwo;
65                 // Wait for detectdone to reset fft_wrapper
66                         end
67                         comparetwo: begin
68                                 state <= complete;
69                         end
70                         complete: begin // Hold ramaddr at maxbin until fftdone signal
71                                 if (fftdone) begin
72                                         ramaddr <= 10'b0;
73                                         maxbin <= 10'b0;
```

```
                                        real_c <= 14'b0;
                                        imag_c <= 14'b0;
                                        maxsqmag <= 27'b0;
                                        state <= readone;
                                    end else begin
                                        state <= complete;
                                    end
                                end

                                default: state <= idle;
                        endcase;
                end
        end
end

endmodule
```

weightblock.sv

```
// Author: Matheu
module weightblock(
        input logic clk,
        input logic reset,
        input logic detectdone,

        input logic [9:0] maxbin,
        input logic [27:0] ramq1,       // FFT RAMs
        input logic [27:0] ramq2,
        input logic [27:0] ramq3,
        input logic [27:0] ramq4,

        output logic [9:0] rdaddr2,     // Will be set to maxbin for all FFT RAMs
        output logic [9:0] rdaddr3,
        output logic [9:0] rdaddr4,
        output logic weightdone,
        output logic [5:0] bnum,                // (0 to 36)
        output logic signed [7:0] doa,  // (-90 to 90)

        output logic [6:0] disp2, disp1, disp0          // 7seg displays
);

enum logic [4:0] {idle, start, memread, micloop, compare, complete} state;
logic rcount;                           // Read cycle counter
logic [2:0] mnum;                       // Microphone number
logic [5:0] maxbnum;            // Beam corresponding to max power
logic [7:0] dladdr;                     // ROM address to get delay coefficient
logic [27:0] dcoeff;            // Delay coefficient from ROM
logic [55:0] delayprod;         // Product of FFT and delay coefficient
logic signed [31:0] ssreal;// real(sigsum); sigsum = sum of all delayprods for a given bnum
logic signed [31:0] ssimag; // imag(sigsum)
logic [63:0] ssrealsq;          // real(sigsum)^2
logic [63:0] ssimagsq;          // imag(sigsum)^2
logic [64:0] sigpwr;            // |sigsum|^2, i.e. array output power
logic [64:0] maxpwr;            // Max array output power
logic signed [27:0] sspec [3:0];        // FFTs by channel at maxbin

// Note: rdaddr1 is set to maxbin by freqdetect block while it is in its complete state
assign rdaddr2 = maxbin;
assign rdaddr3 = maxbin;
assign rdaddr4 = maxbin;
assign dladdr = 4*bnum + mnum;
assign doa = -90 + 5*maxbnum;
```

```
44  assign sigpwr = ssrealsq + ssimagsq;
45  assign sspec[3] = ramq4;
46  assign sspec[2] = ramq3;
47  assign sspec[1] = ramq2;
48  assign sspec[0] = ramq1;
49
50  // Delay matrix preloaded into ROM
51  delay_ROM drom (
52          .clock          (clk),
53          .address        (dladdr),
54          .q                      (dcoeff)
55  );
56
57  // Complex multiplier for delay * FFT
58  compmult cmult (
59          .dataa_real             (dcoeff[27:14]),
60          .dataa_imag             (dcoeff[13:0]),
61          .datab_real             (sspec[mnum][27:14]),
62          .datab_imag             (sspec[mnum][13:0]),
63          .result_real    (delayprod[55:28]),
64          .result_imag    (delayprod[27:0])
65  );
66
67  // Multiplier IP computes square for real(sigsum)^2
68  realmult m1 (
69          .dataa  (ssreal),
70          .result (ssrealsq)
71  );
72
73  // Multiplier IP computes square for imag(sigsum)^2
74  realmult m2 (
75          .dataa  (ssimag),
76          .result (ssimagsq)
77  );
78
79  // 7 Segment Displays
80  angdisplay disp (
81          .clk            (clk),
82          .wbdone         (weightdone),
83      .reset      (reset),
84          .angle          (doa),
85          .signdisp       (disp2),
86          .disp1          (disp1),
87          .disp0          (disp0)
88  );
89
90  always_ff @(posedge clk) begin
91          if (reset) begin
92                  maxpwr <= 65'b0;        //
93                  maxbnum <= 6'b0;
94                  ssreal <= 32'b0;
95                  ssimag <= 32'b0;
96                  {mnum, bnum} <= 9'b0;
97                  rcount <= 0;
98                  weightdone <= 0;
99
100                 state <= idle;
101         end else if (dladdr < 248) begin
102                 case (state)
103                         idle: begin
104                                 if (detectdone && !weightdone) state <= start;
105                         end
```

```
                        start: begin// Two cycle delay to allow ramq1 to update with maxbin
                                if (rcount != 1)
                                        rcount <= 1'd1;
                                else begin
                                        rcount <= 1'd0;
                                        state <= memread;
                                end
                        end
                        memread: begin
                                if (rcount != 1) begin
                                        rcount <= 1'd1;
                                end else begin
                                        rcount <= 1'd0;
                                        state <= micloop;
                                end
                        end
                        micloop: begin  // Loop through channels and multiply by dcoeffs
                                // Multipliers work combinationally
                                ssreal <= ssreal + {{4{delayprod[55]}}, delayprod[55:28]};
                                ssimag <= ssimag + {{4{delayprod[27]}}, delayprod[27:0]};
                                if (mnum == 3'd3)       begin
                                        state <= compare;
                                end else begin
                                        mnum <= mnum + 3'd1;
                                        state <= memread;
                                end
                        end
                        compare: begin
                                // Update maxpwr and maxbnum
                                if (sigpwr > maxpwr) begin
                                        maxpwr <= sigpwr;
                                        maxbnum <= bnum;
                                end
                                if (bnum == 6'd36) begin
                                        weightdone <= 1'd1;
                                        state <= complete;
                                end else begin
                                        bnum <= bnum + 6'd1;
                                        mnum <= 3'd0;
                                        ssreal <= 32'b0;
                                        ssimag <= 32'b0;
                                        state <= memread;
                                end
                        end
                        complete: begin
                                weightdone <= 0;
                                if (detectdone) begin
                                        maxpwr <= 65'b0;
                                        maxbnum <= 6'b0;
                                        ssreal <= 32'b0;
                                        ssimag <= 32'b0;
                                        {mnum, bnum} <= 9'b0;
                                        rcount <= 1'd0;
                                        state <= start; //
                                end else begin
                                        state <= complete;
                                end
                        end
                endcase;
        end
end
```

```
endmodule
```

angdisplay.sv

```
// Author: Matheu
module angdisplay(input logic clk,
                  input logic wbdone,        // done signal from weightblock
                  input logic reset,
                              input logic signed [7:0] angle,

                      output logic [6:0] signdisp, disp1, disp0);

    logic [7:0] absang;
    logic [7:0] tens;

    assign absang = angle[7] ? ~angle + 8'd1 : angle;

    always_comb begin
        if (0 <= absang && absang < 10)
            tens = 8'd0;
        else if (10 <= absang && absang < 20)
            tens = 8'd10;
        else if (20 <= absang && absang < 30)
            tens = 8'd20;
        else if (30 <= absang && absang < 40)
            tens = 8'd30;
        else if (40 <= absang && absang < 50)
            tens = 8'd40;
        else if (50 <= absang && absang < 60)
            tens = 8'd50;
        else if (60 <= absang && absang < 70)
            tens = 8'd60;
        else if (70 <= absang && absang < 80)
            tens = 8'd70;
        else if (80 <= absang && absang < 90)
            tens = 8'd80;
        else if (90 <= absang && absang < 100)
            tens = 8'd90;
        else
            tens = 0;
    end

        always_ff @(posedge clk) begin
        if (reset) begin
            signdisp <= 7'b011_1111;
            disp0 <= 7'b011_1111;
            disp1 <= 7'b011_1111;
        end else if (wbdone) begin
            signdisp <= angle[7] ? 7'b011_1111 : 7'b111_1111;
            disp0 <= absang[0] ? 7'b001_0010 : 7'b100_0000;  // Ones can only be 0 or 5

            // Convert |angle| to two 7segs (-90 to 90)
            // Tens place
            case (tens)
                8'd0:    disp1 <= 7'b111_1111;       // blank
                8'd10:   disp1 <= 7'b111_1001;       // 1
                8'd20:   disp1 <= 7'b010_0100;       // 2
                8'd30:   disp1 <= 7'b011_0000;       // 3
                8'd40:   disp1 <= 7'b001_1001;       // 4
                8'd50:   disp1 <= 7'b001_0010;       // 5
                8'd60:   disp1 <= 7'b000_0010;       // 6
```

```
58                  8'd70:   disp1 <= 7'b111_1000;       // 7
59                  8'd80:   disp1 <= 7'b000_0000;       // 8
60                  8'd90:   disp1 <= 7'b001_0000;       // 9
61                  default: disp1 <= 7'b011_1111;       // -
62              endcase
63          end
64      end
65
66  endmodule
```

C Software Code

hello.c

```
/*
 * Userspace program that communicates with the vga_ball device driver
 * through ioctls
 *
 * Origin: Stephen A. Edwards (Columbia University)
 * Author: Sound localizer
 */

#include <stdio.h>
#include "audio.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include "write_wav.h"
#include <math.h>

#define PI 3.14159265358979

int audio_fd;
int data1, data2;

void calcCoor(double radius, double degrees, double *x, double *y) {
    double radians = degrees * (PI / 180.0);
    *x = radius * cos(radians);
    *y = radius * sin(radians);
}

// Calculate the degree
int calcDeg(int x, int y) {
        int dir;
  if (x >= -45 && x <= 45) {
    if (y >= 45 && y <= 90)
      dir = 90 - x;
    else if (y >= -90 && y <= -45)
      dir =  270 + x;
                else
                        dir = 0;
  } else {
    if (x > 45 && x < 90)
```

```
      dir = 360 + y;
    else if (x > -90 && x < -45)
      dir =  180 - y;
                else
                        dir = 0;
  }
        return dir;
}

// Read audio data
void read_audio() {
  audio_arg_t vla;
  if (ioctl(audio_fd, AUDIO_READ, &vla)) {
      perror("ioctl(AUDID_READ) failed");
      return;
  }
  data1 = vla.audio.left1;
  data2 = vla.audio.right1;
}

// Write address
void write_addr(addr_t *address) {
  addr_arg_t vla;
  vla.addr = *address;
  if (ioctl(audio_fd, ADDR_WRITE, &vla)) {
      perror("ioctl(ADDR_WRITE) failed");
      return;
  }
}

int main()
{
  double radius, degrees, dou_x, dou_y;
  int dir;
  double x_center;
        int y_center;
  addr_t address;
  static const char filename[] = "/dev/audio";  // Open the driver
  // static const char file1[] = "./test1.wav"; // Microphone 1 .wav directory
  // static const char file2[] = "./test2.wav";

  printf("Audio record program started\n");
  if ( (audio_fd = open(filename, O_RDWR)) == -1) {
    fprintf(stderr, "could not open %s\n", filename);
    return -1;
  }
  // Init
  radius = 100;
  x_center = 630;
  y_center = 240;
  address.xcoor = 100;
  address.ycoor = 100;

  // Start the program
        while (1) {
    address.go = 1;
```

```
    write_addr(&address);
    address.go = 0;
    write_addr(&address);
    usleep(500000);

}
  printf("done\n");
  printf("Audio record program terminating\n");
  return 0;
}
```

audio.h

```
#ifndef _AUDIO_H
#define _AUDIO_H

#include <linux/ioctl.h>

// Package 1
typedef struct {
         int left1;
         int right1;
} audio_t;

typedef struct {
        int audio_ready;
} audio_ready_t;

typedef struct {
        audio_t audio;
        // audio_ready_t ready;
} audio_arg_t;

// Package 2
typedef struct {
        int go;
        int xcoor;
        int ycoor;
} addr_t;

typedef struct
{
        addr_t addr;
} addr_arg_t;


#define AUDIO_MAGIC 'q'

/* ioctls and their arguments */
#define AUDIO_READ  _IOR(AUDIO_MAGIC, 1, audio_arg_t)
#define ADDR_WRITE      _IOW(AUDIO_MAGIC, 2, addr_arg_t)
#define AUDIO_IRQ_READ _IOR(AUDIO_MAGIC, 3, audio_arg_t)

#endif
```

audio.c

```
/* Device driver for the I2S
 *
 * 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/
 * NoCallerID
 *
 * "make" to build
 * insmod audio.ko
 *
 * Team: Sound-localizer
 */

#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 <linux/sched.h>
#include <linux/interrupt.h>
#include <linux/of_irq.h>
#include "audio.h"

#define DRIVER_NAME "audio"

// Initialize a wait queue to sleep user level process until irq raised
DECLARE_WAIT_QUEUE_HEAD(wq);

/* Device registers */
#define DATA1_L(x) (x)
#define DATA1_R(x) ((x)+4)
// #define RESET_IRQ(x) ((x)+8)

#define GO(x) (x)
#define XCOOR(x) ((x)+4)
#define YCOOR(x) ((x)+8)

/*
 * Information about our device
 */
```

```
struct audio_dev { // audio_dev
        struct resource res; /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
        audio_t audio; // audio_color_t background;
                addr_t  addr;
                                audio_ready_t ready;
                                int irq_num;
} dev;

/* Read audio data from device
 */
static void read_audio(audio_t *audio)
{
        audio->left1 = ioread32(DATA1_L(dev.virtbase));
        audio->right1 = ioread32(DATA1_R(dev.virtbase));
        // ioread32(RESET_IRQ(dev.virtbase));
        dev.audio = *audio;
}

/* Write address to device
*/
static void write_address(addr_t *addr)
{
        iowrite32(addr->go, GO(dev.virtbase));
        iowrite32(addr->xcoor, XCOOR(dev.virtbase));
        iowrite32(addr->ycoor, YCOOR(dev.virtbase));
        dev.addr = *addr;
}

/* Handle interrupts raised by our device. Read samples,
 * clear the interrupt, and wake the user level program.
 */
static irqreturn_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
        audio_t audio;
        audio_ready_t ready;
        read_audio(&audio);

        ready.audio_ready = 1;
        dev.ready = ready;
        wake_up_interruptible(&wq);

        // IRQ_RETVAL(val): If val is non-zero, this macro returns IRQ_HANDLED
        return IRQ_RETVAL(1);
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long audio_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
        audio_arg_t vla;
        audio_ready_t ready;
        addr_arg_t vla_addr;
```

```
        audio_t audio;
        switch (cmd) {
        case ADDR_WRITE:
                if (copy_from_user(&vla_addr, (addr_arg_t *) arg, sizeof(addr_arg_t)))
                        return -EACCES;
                write_address(&vla_addr.addr);
                break;

        case AUDIO_READ:
                read_audio(&audio);
                vla.audio = dev.audio;
                if (copy_to_user((audio_arg_t *) arg, &vla, sizeof(audio_arg_t)))
                        return -EACCES;
                break;

        case AUDIO_IRQ_READ:
                // Sleep the process until woken by the interrupt handler, and the data
                        wait_event_interruptible_exclusive(wq, dev.ready.audio_ready);
                        // Data is ready
                        vla.audio = dev.audio;
                        ready.audio_ready = 0;
                        dev.ready = ready;
                if (copy_to_user((audio_arg_t *) arg, &vla, sizeof(audio_arg_t)))
                        return -EACCES;
                break;

        default:
                return -EINVAL;
        }

        return 0;
}

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

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

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init audio_probe(struct platform_device *pdev)
{
  // vga_ball_color_t beige = { 0xf9, 0xe4, 0xb7 };
        int ret;
        int irq;
```

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

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);

        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

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

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

        /* Determine the interrupt number associated with our device */
        irq = irq_of_parse_and_map(pdev->dev.of_node, 0);
        dev.irq_num = irq;

        /* Request our interrupt line and register our handler */
        ret = request_irq(irq, (irq_handler_t) irq_handler, 0, "csee4840_audio", NULL);

        if (ret) {
                printk("request_irq error: %d", ret);
                ret = -ENOENT;
                goto out_deregister;
        }

        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        free_irq(dev.irq_num, NULL);
        misc_deregister(&audio_misc_device);
        return ret;
}


/* Clean-up code: release resources */
static int audio_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        free_irq(dev.irq_num, NULL);
        misc_deregister(&audio_misc_device);
```

```
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id audio_of_match[] = {
        { .compatible = "csee4840,audio-1.0" },
        {},
};
MODULE_DEVICE_TABLE(of, audio_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver audio_driver = {
        .driver = {
                .name   = DRIVER_NAME,
                .owner  = THIS_MODULE,
                .of_match_table = of_match_ptr(audio_of_match),
        },
        .remove = __exit_p(audio_remove),
};

/* Called when the module is loaded: set things up */
static int __init audio_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&audio_driver, audio_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit audio_exit(void)
{
        platform_driver_unregister(&audio_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(audio_init);
module_exit(audio_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Columbia University");
MODULE_DESCRIPTION("I2S Audio driver");
```