6.111 Final Project Report

iSing Voice Harmonizer

Cyril Lan, Jessie Li, Darren Yin

December 10, 2009

Abstract

Our project is a voice harmonizer which detects the frequency of a sung note and pitch shifts the note to match the keys played on a keyboard. A 2048 point FFT was implemented for pitch detection, a pitchshifter module was written for pitch shifting, and a central CPU was written to control the flow of data between modules. Due to hardware constraints and lack of time, the system was not functional as a whole. However, the FFT module was able to correctly detect pitch, the keyboard inputs were properly converted to midi frequencies, and the pitchshifter was able to shift a 750 Hz tone up an octave with some added noise.

Contents

List of Figures

1 Overview

The basic system consists of three main components: pitch detection of the sung note using a FFT, pitch shifting the note several times to match each of the keyboard inputs, and adding the pitch shifted signals into one signal that is sent to the speakers. A block diagram of our system is in Figure 1.

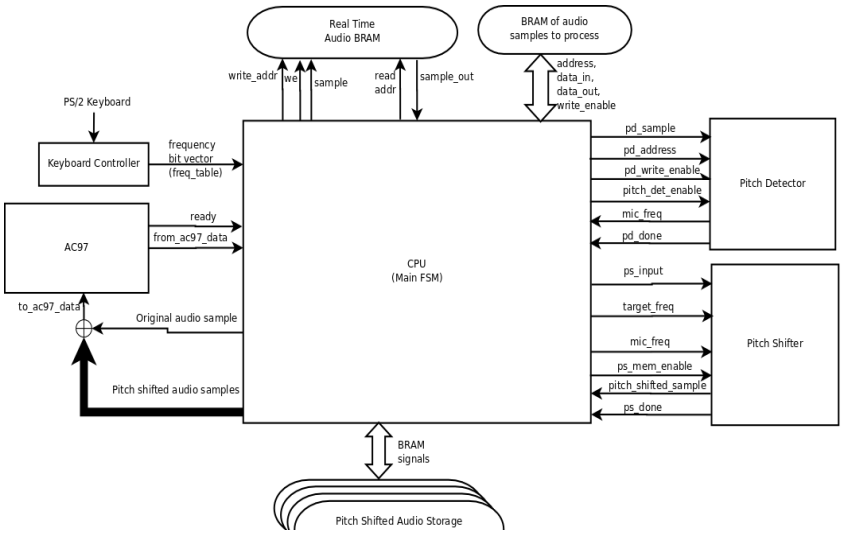

Figure 1: Block diagram of entire system

When the user presses x keys simultaneously on the keyboard, the midi controller detects which keys are pressed and maps the keys to appropriate midi frequencies. Furthermore, incoming microphone samples are stored in a microphone buffer which the CPU reads from. When enough samples are written into the buffer, the CPU reads off 4096 samples and sends them to the FFT for pitch detection. After the FFT has detected the most prominent note frequency, the CPU sends this information along with the pressed midi frequencies and the 4096 samples to the pitchshifter. The pitch shifter module uses the midi frequencies as the target frequencies to shift to and outputs the pitchshifted samples to the CPU which sends them to the AC97.

2 Description of Each Module

2.1 Fast Fourier Transform (Cyril)

2.1.1 Input and Output

Input and output to the FFT module is similar to that of a memory module. Data is written directly to the FFT BRAM by inputting the address, data, and asserting a write-enable bit. To start the FFT transform, the FFT-enable bit is asserted for one clock cycle. The module will perform calculations and assert a done bit for one cycle upon completion. After that, output data values can be read out by providing the address. During the time that the FFT module is busy, attempts at data writes or data reads will do nothing.

2.1.2 Data Width

The AC97 records 8 signed bits of audio data. Data is stored as 32-bit complex numbers inside the FFT memory (16 real, 16 imaginary), so the imaginary component is initially assigned to zero. These 32-bit complex coefficients are returned as output. One potential issue that arises is whether 16 bits is enough to

store the potentially large coefficients that result from a Fourier transform. Parseval's theorem will allow us to analyze this issue.

The theorem essentially states that the total energy in the audio signal equals the total energy in the transform. The equation is

$$\sum_{N-1}^{0} |x[n]|^2 = \frac{1}{N} \sum_{N-1}^{0} |X[k]|^2$$

where $x(t)$ is the time-domain signal and $X(f)$ is the frequency domain signal. In the worst case scenario where x(t) is as large as possible, the left hand sum becomes $2048 * 128^2 = 2^{11} * 2^{(7*2)} = 2^{25}$. The worst case scenario for the right hand side is if there are only two frequency coefficients present in the transform (since the FFT is symmetric), giving us that $|X[i]|^2 = 2^{35}$. So up to 19 bits could be used in storing the FFT coefficients. Generally, this does not present an issue as most natural audio signals have a wide frequency spectrum.

2.1.3 The Cooley-Tukey Algorithm

The algorithm used is this FFT module is the Cooley-Tukey algorithm. The algorithm requires that the sample size N to be a power of 2. The process is divided up into stages, with the total number of stages equal to the log base 2 of the number of samples. In our case, there are 2048 samples and 11 stages.

At each stage, $N/2$ butterfly operations are performed on pairs of samples. A butterfly takes two complex coefficients from the memory, A and B as well as a twiddle factor, w. The butterfly calculates $A + B * w$ and $A - B * w$ and stores those two results back into the memory locations of A and B, respectively. The twiddle factor w is a complex exponential $e^{(\frac{2\pi * n}{N})}$, where N is the number of samples and n is the twiddle factor coefficient. Details on how to obtian the memory addresses for A and B, as well as the twiddle factor coefficient, will be discussed in the section on the FFT addresser.

There are on the order of N butterfly operations per stage and on the order of $log_2(N)$ stages, giving us a runtime of $N * log(N)$.

2.1.4 FFT BRAM memory

The FFT module used a dual-port BRAM module generated by COREGEN. The memory was 32 bits in width (16 real bits, 16 imaginary bits) and 2048 registers in length. A dual-port memory module was chosen because the butterfly calculation reads and writes two entries of data in one clock cycle.

2.1.5 FFT Addresser

The Addresser module performs bit-reversing and bit-circulation to generate the memory addresses in the order required by the Cooley-Tukey algorithm. The Addresser module also generated twiddle factor coefficients to match with each memory address. The stage number and the group number (there are 1024 butterfly groups) are stored in registers in the Addresser.

The addresser takes in an enable bit, and if the bit is asserted at the rising edge of a clock, the addresser will increment the group number (or the stage number if the group number reaches its maximum) to move on to the next pair of addresses. Addresses are calculated using combinatorial logic as to avoid having an extra clock in the butterfly calculation pipeline. Let S be the 4-bit stage number (the first stage is stage 0), and let G be the 10-bit group number. First, the 11-bit values of G*2 and G*2+1 are bit-reversed. Next, the bit-reversed values are bit-circulated toward the right by S bits. These two values now are the two memory addresses. To find the twiddle factor coefficient, we take G and set the right-most 10-S bits to 0.

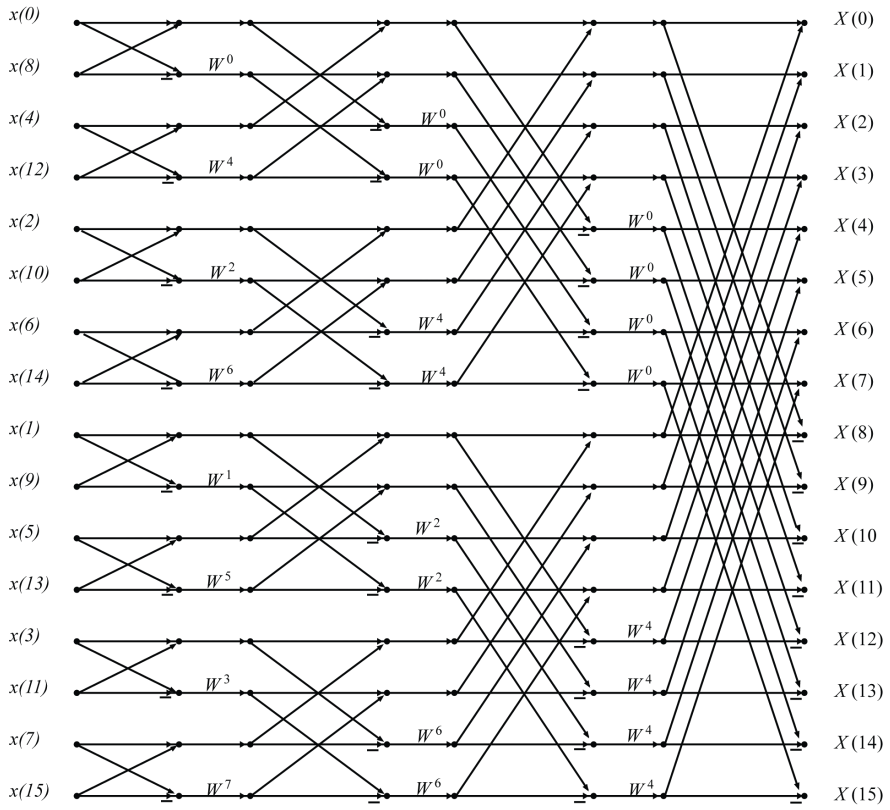

Figure 2: Diagram of butterfly calculations for a 16-pt FFT

2.1.6 Sine/Cosine Lookup Table

The twiddle factor coefficient is passed into the trigonometric lookup table to compute the twiddle factor. We used the COREGEN to create a 2048-row sine-cosine lookup table that produced sine and cosine values with 16 bits of accuracy.

2.1.7 Butterfly Module

The Butterfly Module lies at the heart of the FFT. The module takes in two 32-bit complex data values A and B as well as a 32-bit complex twiddle factor w, and produces two new complex data values Y and Z. We pipelined our Butterfly module into two stages. In the first stage, the complex multiplication $B * w$ is foiled out using four parallel 18x18 multipliers. The results are stored in registers used in the next stage, when the addition $A + B * w$ is performed.

We had originally planned to use this pipelining to achieve a throughput of one butterfly calculation every two clock cyles (the clock cycles would alternate between reading from the BRAM and writing the results to the BRAM). However, such an architecture would result in different addresses for the read and write operations, since we would be reading new data points followed by writing old data points. Due to time constraints, we did not get a chance to test out this pipelined architecture, and went with the simpler design of waiting for the results to be written before reading new data points.

2.1.8 FFT Controller

The FFT controller is a state machine that regulates the modules in the FFT and controls the flow of data. The controller begins by sending an enable signal to the Butterfly module to start the butterfly calculation. Next, it waits for two clock cycles and asserts the write-enable signal to the memory module. Finally, the controller sends an enable signal to the Addresser to tell it to increment its pair of addresses. The controller runs for 1024 groups * 11 stages = 11263 cycles.

2.2 Pitch Detector (Cyril)

The Pitch Detector module is basically an extension of the FFT module, but the output of the FFT is piped through a magnitude module and a peak finder module. We used the magnitude module from the PerfectPitch project by Grace Cheung and Karl Rieb (2007), and piped the output of the FFT into the magnitude module.

The peak finder module used a linear counting method to find the index of the maximum magnitude. The index then translated into the frequency associated with the corresponding FFT bin. In an FFT with N bins and a sampling rate of Fs, the maximum frequency is Fs/2, and the minimum frequency is 0. Additionally, the FFT also produces negative frequencies from 0 to -Fs/2. Hence, the frequency resolution of the FFT is Fs/N. The first bin (index 0) will correspond to frequency -Fs/2, the two middle bins 0, and the last bin Fs/2.

2.3 Keyboard Controller (Darren)

The keyboard controller ensures that keys pressed on the attached PS/2 keyboard are translated into memory, and it consists of two submodules: a keyboard input deserializer and a target pitch frequency memory manager.

The keyboard controller module takes as input the keyboard inputs and writes to a memory with the number of locations equal to the number of keys enabled on the keyboard. If the key corresponding to a particular midi frequency is pressed, there is a 1 in the memory slot. Otherwise, there is a 0.

The PS/2 keyboard protocol consists of key-up and key-down signals, among other things. Every byte of each command is transmitted via an 11 bit frame and synchronized by a clock from the keyboard. To read in each byte, the keyboard controller uses a keyboard serializer submodule which ensures In addition, there is some very important debouncing logic which ensures that bytes are read correctly.

2.4 Pitch Shifter (Jessie)

Pitch shifting is the process of changing the frequency of a signal while keeping its duration constant. There are two main approaches to pitchshifting: either performing it in the frequency domain or performing it in the time domain. The frequency domain approach requires using mathematical functions such as arctangent, sine, cosine, and square root which are hard to synthesize in hardware. Furthermore, most frequency domain approaches require using floating point numbers to keep many bits of precision, which is infeasible in hardware. Therefore, I chose to take the time domain approach with the added assumption that the signal is periodic in the 4096 samples that are being processed at a time.

The idea of the pitchshifting algorithm is to time expand the signal by making copies of the samples and then resample the signal by reading off every alpha-th sample. Here alpha is the pitchshift factor, defined as the ratio of the target frequency to the actual note frequency. This approach worked fairly well in the software model because the signal was mostly periodic in the 4096 samples that are processed each time so putting copies of the signal next to each other would time expand it without changing its frequency.

Increasing the number of samples that are processed each time improves the sound quality significantly, but due to limitations in bram size on the FPGA, we chose to stick to 4096 samples.

During the process of resampling, we sample indices that are alpha apart from each other, but because alpha may not be an integer,we need to perform linear interpolation to get a weighted average of two adjacent samples. For example, if I'm pitchshifting up a major third, alpha = 1.26, I would sample indices 0, 1.26, 2.52, 3.78, ... When I'm sampling index 1.26, I take 0.26 of the sample at index 1 and add it to 0.74 of the sample at index 2. When I'm sampling index 2.52, I take 0.5 of sample 2 and add it to 0.48 of sample 3. In this way, I am mimicking taking every alpha-th sample even when alpha is not an integer.

Figure 3 shows a block diagram of the pitchshifter module.

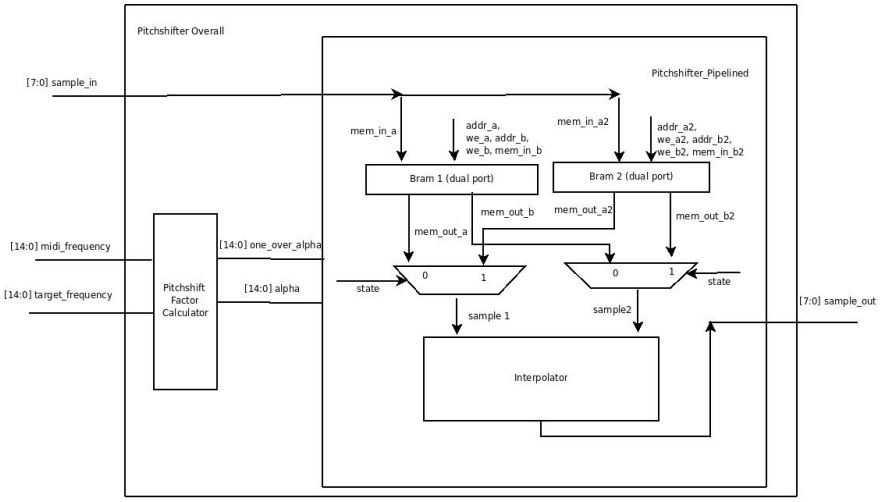

Figure 3: Block diagram of Pitchshifter Module

The pitchshifter module takes as inputs the target pitch, the detected note frequency, and microphone samples, and returns as output pitchshifted samples. The pitchshifter module contains two internal dual port brams, each of which can store up to 4096 8 bit audio samples. When samples are being written to one bram, samples are read from the other bram and processed. Using a dual port bram allows two samples to be read and sent to the interpolator on the same clock cycle. In this way, processing could be done in the same number of clock cycles as it took to write the samples into bram. After the processing is finished, the roles of the brams swap. In this way, there is a constant stream of pitchshifted samples coming out of the pitchshifter module and no samples are dropped.

Time expansion and resampling of the signals is accomplished by looping several times through the bram that is in read mode and incrementing by alpha a counter called index that keeps track of my index. When index reaches last = alpha*(n-1), where n is the number of samples, I know that I have finished resampling. At this point, I swap the roles of the brams and perform the same process on the other bram.

Figure 4 shows a state machine of the pitchshifter module.

In state write_one_read_two, I am writing to bram 1 and reading from bram 2. The roles of the brams are swapped in the other state. As long as I have not incremented index alpha times, I stay in the same state. When index is greater than or equal to alpha*(n-1), I transition into the other state. During the state transition, I reset my addresses to zero and set the write enable signals on the two brams appropriately.

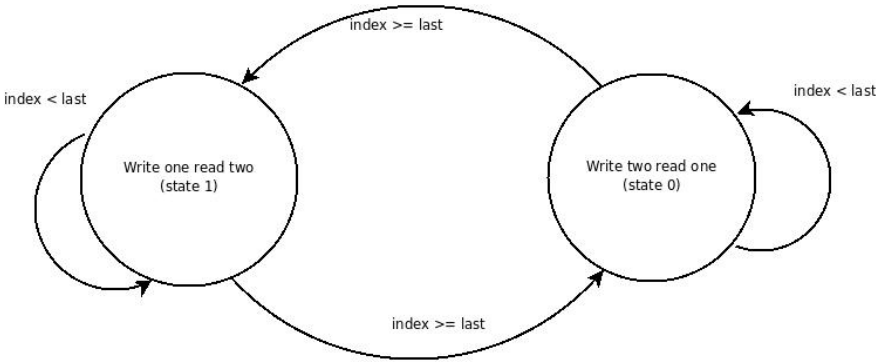

Figure 4: State Machine of Pitchshifter Module

We were able to get a working software model of the algorithm, and we were able to verify in Modelsim that the outputs are what we expect. However, when the results were put on the FPGA, we heard a lot of high frequency noise that impaired the quality of our signal. Despite the high frequency noise and aliasing effects, we were able to verify using Adam's spectrum analyzer that our module can pitchshift a 750 Hz tone up by an octave.

2.5 CPU (Darren)

The CPU is the main FSM that controls all of the slave modules such as the FFT and the pitch shifter.

The CPU consists of five main states and within each state, there are substates which specify more detailed functionality. In the reset state, the local audio buffer is empty and the mic frequency and target frequency are initialized to zero. When reset is deasserted, the CPU transitions into the Initialization state. The Initialization state contains two substates: one where we are copying samples from the external bram which stores all incoming audio samples to the local audio buffer which stores samples that are currently being processed. After the local audio buffer is filled, the frequencies corresponding to keys pressed on the keyboard are copied into another bram called target_freqs.

Subsequently, the state machine transitions into the Pitch Detection state which also consists of two substates. In the first substate, we are filling the FFT buffer with audio samples and in the second substate, we enable pitch_detection_enable for one clock cycle and wait for the FFT to finish pitch detection. When the FFT asserts the pd_done signal, the CPU transitions into the Pitch_Shifter state.

The Pitch_Shifter state contains four substates. In the first substate, the CPU passes a target frequency to the pitch shifter. In the next substate, 4096 samples are passed to the pitchshifter in 4096 clock cycles. After 4096 samples have been passed to the pitch shifter, the fsm enters the third substate where the pitchshifter is doing processing. When the pitchshifter is done with its processing, the CPU reads out 4096 samples and goes back to the first substate.

When there are no more target frequencies to pass to the pitchshifter, the CPU transitions into Audio Playback state, which iterates over the pitchshifted samples and the original samples and pipes the sum of the samples to the ac97 at 48 KHz.

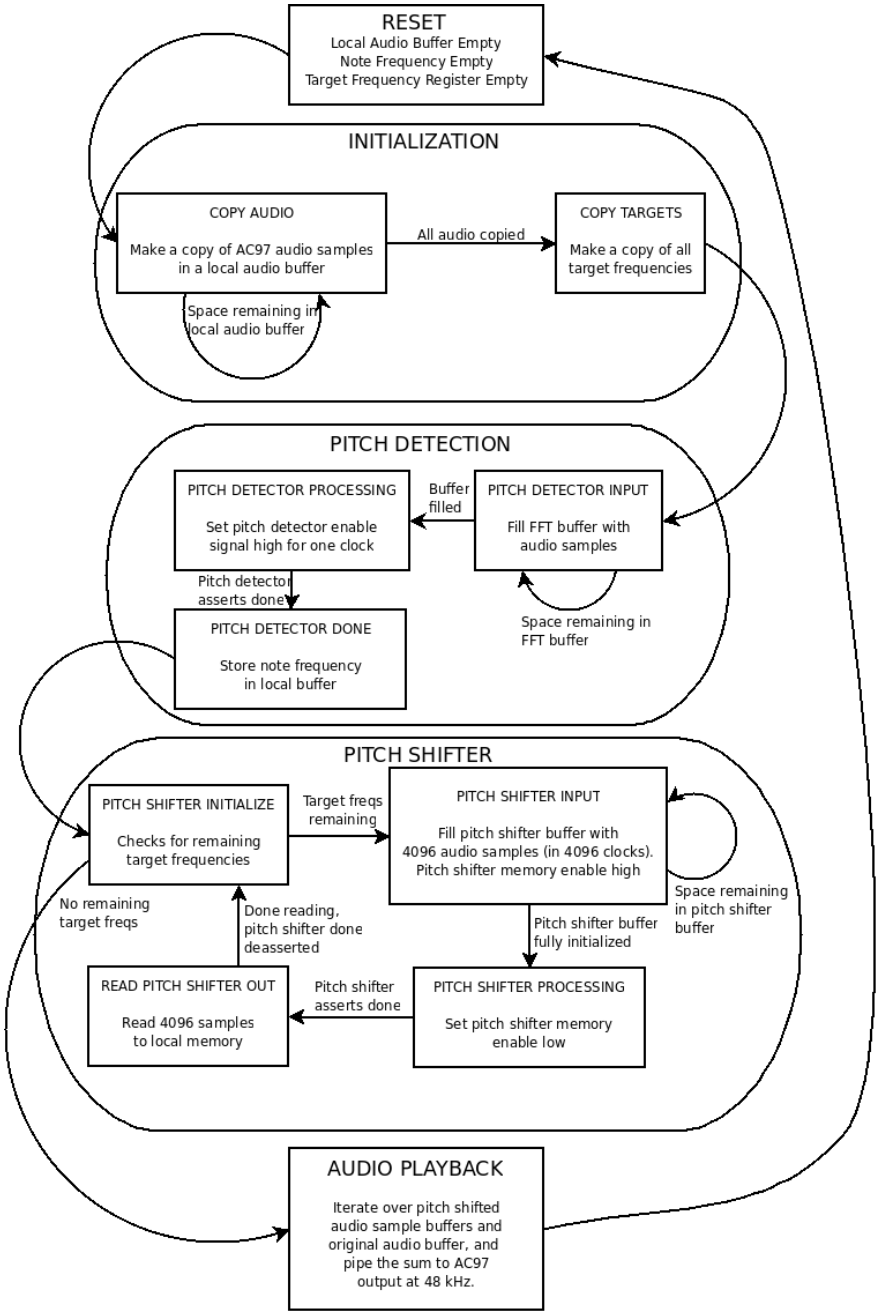

Figure 5: Main finite state machine diagram

3 Testing and Debugging

Overall, our system took longer to implemented and test than we expected, and we were unable to successfully integrate our modules together. The techniques that we used to debug our individual modules were to perform simulation in Modelsim and to feed them mock inputs and comparing actual output with expected output. We also individually tested each module on the FPGA.

3.1 FFT Testing

In the design stages, the FFT algorithm was first implemented in Python using only integer arithmetic to prove that the algorithm worked. Next, a scaled down version of the FFT using only 16 points was implemented. The model was tested in Modelsim, and then on the FPGA to verify that the transform worked. Data was fed in using a dummy runner module, and output coefficients were displayed on the 16-character LED display. We compared these coefficients to answers found using MATLAB. After they matched, the FFT was scaled up to a 2048 point transform.
The 2048 point FFT was more difficult to test on the FPGA as it performed a huge number of calculations and required on the scale of 1 millisecond to complete the transform. To test it, we wrote a peak detector module to return the most prominent frequency in the input. We used another dummy runner module to test inputs sawtooth waves of differing frequencies. Initial tests showed that the 2048 point FFT module worked properly, although we did not have enough time to run enough tests to conclusively show that the module functioned.

3.2 Pitchshifting Testing

Pitchshifting proved to be an extremely computationally intensive algorithm when implemented correctly, and due to the limitations of the hardware, we settled for an approximation that did not work as well as we expected. The surprising discovery was that our software model using the approximation worked fine, but maintaining the quality of the sound after implementing the approximation in hardware proved to be difficult. Some of the pitchshifted sounds contained tones of the right frequency, but there was a lot of high frequency noise.

Given more time and resources, a different, more accurate method of pitchshifting could have been tried. A possible algorithm is the Shift Overlapp Add algorithm which breaks up the time domain signal into overlapping segments and then shifts them apart and overlap adds them to create a time expanded signal. We did not implement this algorithm because the algorithm requires computing cross correlation coefficients between different segments, and the complexity of the computation seemed too great for the hardware to synthesize.

The accurary of pitchshifting was tested by first performing simulation in Modelsim using the 750 Hz tone and then loading the module onto the FPGA. When testing on the FPGA, there were two modes: one where we record voice and another where we use the 750 Hz tone. The pitchshift factor was determined by adjusting switches.

3.3 CPU and Keyboard Testing

When testing the CPU, mock FFT and pitchshifter modules that produced a constant stream of outputs were used. Based on these outputs, we could determine the expected state transitions. We then observed the actual state transitions on the logic analyzer. The logic analyzer was clocked to the 27 mhz clock which allowed state transitions to be observed. It was verified that state transitions were happening properly and that midi frequencies were detected properly.

Originally, we planned to use a midi keyboard, but due to compatibility issues, a QWERTY keyboard was used instead. The midi keyboard communication protocol involves detecting zeros and ones by changing the direction of current through a diode. However, this change of current was very hard to detect because the FPGA uses voltage controlled power sources.

The keyboard controller module was debugged using the LCD display and the logic analyzer.

The FFT module was first tested in Modelsim by feeding in a sequence of time domain samples and observing the real and imaginary coefficients that are generated. Later, the FFT module was put onto the FPGA and tested using the LCD display.

4 Conclusion

Although we did not have time to fully integrate our system, some parts worked individually. The 2048 point FFT and the keyboard key conversion to MIDI frequencies both worked. The pitchshifter succesfully shifted a test signal by an octave and by a major third, despite the fact that it also produced large amounts of high frequency noise. Given more time, we would have been able to implement more efficient algorithms and fully integrate our system into a working voice harmonizer.

5 Appendices

5.1 FFT Main Module

```
module fftx82048(input wire clock, reset,
  input wire [7:0] input_data_in,
  input wire [10:0] input_addr,
  input wire input_we,
  input wire input_enable,
  output reg done,
  output wire [31:0] data_out);
//To start off the FFT, assert input_enable for 1 cycle

wire mem_we;
wire [10:0] mem_addr0;
wire [10:0] mem_addr1;
wire [31:0] mem_data_in0;
wire [31:0] mem_data_in1;
wire [31:0] mem_data_out0;
wire [31:0] mem_data_out1;

wire addr_enable;
wire addr_writemode;
wire butterfly_enable;
wire ctrl_done;

wire [10:0] tf_addr;
wire [31:0] tf_data;

//Working memory
bram_2048_32 mem(.clka(clock),
.clkb(clock),
.addra(~done ? mem_addr0 : input_addr),
.addrb(mem_addr1),
.wea(~done ? mem_we : input_we),
.web(mem_we),
.douta(mem_data_out0),
```

```
.doutb(mem_data_out1),
.dina(~done ? mem_data_in0 : {{8{input_data_in[7]}}, input_data_in , 16'b0}),
.dinb(mem_data_in1)
);

assign data_out = mem_data_out0;

//FFT controller
fftctrl2048 cpu(.clock(clock), .reset(reset),
.enable(input_enable),
.addr_enable(addr_enable),
.addr_writemode(addr_writemode),
.butterfly_enable(butterfly_enable),
.done(ctrl_done));

//Addresser
fftaddr2048 addresser(.clock(clock), .reset(reset),
.enable(addr_enable),
.writemode(addr_writemode),
.mem_we(mem_we), //write enable for memory
.mem_addr0(mem_addr0), //memory address
.mem_addr1(mem_addr1),
.tf_addr(tf_addr));

//Twiddle factor generator
//exp2048 tfgen(.clock(clock), .reset(reset), .addr(tf_addr), .data_out(tf_data));
exp2048cg tfgen(.CLK(clock), .THETA(tf_addr), .COSINE(tf_data[31:16]), .SINE(tf_data[15:0]));

//Butterfly
butterflyx8 b(.clock(clock), .reset(reset),
  .enable(butterfly_enable),
  .a(mem_data_out0),
  .b(mem_data_out1),
  .tf(tf_data), //twiddle factor
  .y(mem_data_in0),
  .z(mem_data_in1));

always @(posedge clock) begin
if (reset) begin
done <= 1;
end
if (input_enable) begin
done <= 0;
end
if (ctrl_done) begin
done <= 1;
end
end

endmodule
```

5.2 FFT Controller Module

```
module fftctrl2048(input wire clock, input wire reset,
input wire enable,

output reg addr_enable,
output reg addr_writemode,
output reg butterfly_enable,
output reg done);

reg [3:0] state;
reg [15:0] iterations;
reg [3:0] waitcounter;
/*
States
0: Stopped
1: Started, addresser is outputing correct address
2: Butterfly enable asserted
3: Butterfly enable deasserted, waiting for butterfly
4: Butterfly done, write is asserted
5: Write deasserted, addresser enabled to advance to next address
6: Waiting for addresser to finish
7: Done, Check for correct # of iterations
*/


always @(posedge clock) begin
if (reset) begin
state <= 0;
iterations <= 0;
waitcounter <= 0;
done <= 0;
addr_enable <= 0;
addr_writemode <= 0;
butterfly_enable <= 0;
end

if (state == 0 && enable) begin
state <= 1;
done <= 0;
end
if(state == 1) begin
state <= 2;
butterfly_enable <= 1;
end
/*
if(state == 1 && waitcounter < 1) begin
waitcounter <= waitcounter + 1;
end
if(state == 1 && waitcounter == 1) begin
waitcounter <= 0;
state <= 2;
```

```
butterfly_enable <= 1;
end
*/
if (state == 2) begin
state <= 3;
butterfly_enable <= 0;
end
if (state == 3 && waitcounter < 1) begin
waitcounter <= waitcounter + 1;
end
if (state == 3 && waitcounter == 1) begin
waitcounter <= 0;
state <= 4;
addr_writemode <= 1;
end
if (state == 4) begin
state <= 5;
addr_writemode <= 0;
end
if (state == 5) begin
state <= 6;
addr_enable <= 1;
end
if (state == 6) begin
addr_enable <= 0;
if (iterations == 11263) begin
iterations <= 0;
state <= 7;
done <= 1;
end
else begin
iterations <= iterations + 1;
state <= 1;
end
end
if (state == 7) begin
done <= 0;
state <= 0;
end
end
endmodule
```

5.3 FFT Addresser Module with bit operations

```
module fftaddr2048(input wire clock, reset,
input wire enable, //makes the addresser count up
input wire writemode, //read or write mode for addresser (asserted TRUE for writing to MEMORY)
output wire mem_we, //write enable for memory
output wire [10:0] mem_addr0, //memory addresses
output wire [10:0] mem_addr1,
output wire [10:0] tf_addr); //twiddle factor address
```

```
/*
Description:

After receiving an enable signal (with writemode = 0), the addresser module does the following:
1. Assert mem_we = 0
2. Output the 2 memory addresses
3. Keep i the same

After receiving an enable signal (with we = 1), the addressor module does the following:
1. Assert mem_we = 1
2. Output the 2 memory addresses
3. Increase i by 2
*/

reg [14:0] index;
// Bits 14-11 are the stage number, bits 10-1 are the group number, bit 0 is the item number

always @(posedge clock) begin
if (reset) begin
index <= 0;
//Initially: stage = 0, group = 0, item = 0
end
if (enable) begin
index <= index + 2;
end
end

//Perform bit reversing and bit circulation
wire [10:0] data0;
wire [10:0] data1;
bitrev11 br0(index[10:0], data0);
bitrev11 br1(index[10:0]+1, data1);
bitcirc_r11 bc0(index[14:11], data0, mem_addr0);
bitcirc_r11 bc1(index[14:11], data1, mem_addr1);

assign mem_we = writemode;

//Get twiddle factor theta
//Address input to CORGEN module: theta = 2pi * (INPUT / 2^INPUT_WIDTH) radians
assign tf_addr = (index[10:0] >> (11-index[14:11])) << (10-index[14:11]);

endmodule

module bitrev11(input wire [10:0] data_in, output wire [10:0] data_out);
assign data_out[0] = data_in[10];
assign data_out[1] = data_in[9];
assign data_out[2] = data_in[8];
assign data_out[3] = data_in[7];
assign data_out[4] = data_in[6];
assign data_out[5] = data_in[5];
assign data_out[6] = data_in[4];
```

```
assign data_out[7] = data_in[3];
assign data_out[8] = data_in[2];
assign data_out[9] = data_in[1];
assign data_out[10] = data_in[0];
endmodule

module bitcirc_r11(input wire [3:0] n, input wire [10:0] data_in, output wire [10:0] data_out);
   //Bit circulates data_in to the right by n bits
wire [10:0] data_out3;
wire [10:0] data_out2;
wire [10:0] data_out1;

assign data_out3 = (n[3] == 1) ? {data_in[7:0], data_in[10:8]} : data_in;
assign data_out2 = (n[2] == 1) ? {data_out3[3:0], data_out3[10:4]} : data_out3;
assign data_out1 = (n[1] == 1) ? {data_out2[1:0], data_out2[10:2]} : data_out2;
assign data_out = (n[0] == 1) ? {data_out1[0], data_out1[10:1]} : data_out1;

endmodule // bitcirc_r11
```

5.4 FFT Butterfly Module

```
module butterflyx8(input wire clock, reset,
  input wire enable,
  input wire [31:0] a,
  input wire [31:0] b,
  input wire [31:0] tf, //twiddle factor
  output reg [31:0] y,
  output reg [31:0] z);
//Inputs: Two 32-bit complex numbers (16 signed bits real - more signfct. bits,
//   16 signed bits complex - less signfct. bits)

reg state;
reg signed [31:0] r_1;
reg signed [31:0] r_2;
reg signed [31:0] j_1;
reg signed [31:0] j_2;

wire signed [15:0] b_r = b[31:16];
wire signed [15:0] b_j = b[15:0];
wire signed [15:0] tf_r = tf[31:16];
wire signed [15:0] tf_j = tf[15:0];

always @(posedge clock) begin
if(reset) begin
state <= 0;
y <= 0;
z <= 0;
r_1 <= 0;
r_2 <= 0;
j_1 <= 0;
j_2 <= 0;
end
```

```
if(enable && state == 0) begin
r_1 <= b_r * tf_r;
r_2 <= b_j * tf_j;
j_1 <= b_r * tf_j;
j_2 <= b_j * tf_r;
state <= 1;
end
//Note: For twiddle factors, 2^14 = +1, -2^14 = -1
//My own lookup table: 2^15 and -2^15
/*if(state == 1) begin
y[31:16] <= a[31:16] + r_1[29:14] - r_2[29:14];
y[15:0] <= a[15:0] + j_1[29:14] + j_2[29:14];
z[31:16] <= a[31:16] - r_1[29:14] + r_2[29:14];
z[15:0] <= a[15:0] - j_1[29:14] - j_2[29:14];
state <= 0;
end*/
if(state == 1) begin
y[31:16] <= a[31:16] + r_1[30:15] - r_2[30:15];
y[15:0] <= a[15:0] + j_1[30:15] + j_2[30:15];
z[31:16] <= a[31:16] - r_1[30:15] + r_2[30:15];
z[15:0] <= a[15:0] - j_1[30:15] - j_2[30:15];
state <= 0;
end
end
endmodule
```

5.5 Pitchshifter Software Model

```
###############################################################################
## Python Software Model for Pitchshifter. Uses waveforms.py file written
## by Chris Terman.
###############################################################################

import math
import wave, struct, numpy, os
from array import array
from waveforms import *

#Pitchshifts the samples in a file by a factor of alpha.
def pitchshift4(alpha, file_name):
        # 12 fractional bits
        f = pow(2, 12)
        integer_alpha = int(alpha*f)

        one_over_alpha = (f*f/integer_alpha)


        b = pow(2, 7)
        raw_inputs = read_wavfile(file_name, 20000)[0].samples
        all_input = []
```

```
        input_file = open('pitchshifting.samples', 'w')
        for mic_input in raw_inputs:
                # Convert into 8 bit number
                fixed_mic_input = int(mic_input/b)
                all_input.append(fixed_mic_input)
                input_file.write(str(fixed_mic_input) + '\n')

        input_file.close()
        output_file = open('pitchshifting.output', 'w')
        output_file.write('')
        output_file.close()
        #Splice up all_input into 4096 bit chuncks and do processing on those
        all_output = array('h')
        for x in range(0,len(all_input), 4096):
            segment = all_input[x:min(len(all_input),x+4096)]
            pitchshifted = pitchshift(segment, integer_alpha, one_over_alpha)
            all_output.extend(pitchshifted)

        writewavefile(all_output,file_name, file_name + '_' + str(alpha) + '.wav')

# Performs pitchshifting on a batch of 4096 samples.
def pitchshift(all_input, integer_alpha, one_over_alpha):
        f = pow(2, 12)
        output = array('h')
        temp = integer_alpha
        # Expand the time domain signal to alpha*N by making copies of the
        # signal and putting them next to each other or truncating the signal.
        # This is where I'm assuming the signal is periodic and resembles
        # a sinusoid.
        while (temp > f):
                output.extend(all_input[0:len(all_input)])
                temp = temp - f
        output.extend(all_input[0:len(all_input)*temp/f])

        final_output = array('h')
        index = 0

        # Resample the signal at 1/alpha times the sampling rate.
        end = (len(output)-1)*f
        output_file = open('pitchshifting.output', 'a+')
        while (index < end):
                left = output[index/f]
                right = output[index/f +1]
                if (integer_alpha < f):
                        interpolated = interpolate(left,right,integer_alpha,f)/f
                else:
                        interpolated = interpolate(left,right,one_over_alpha,f)/f

                output_file.write('   ' + str(interpolated) + '\n')
                final_output.append(interpolated)
                index = index + integer_alpha
```

```
        output_file.close()
        return final_output

# Perform linear interpolation on two samples
def interpolate(sample1, sample2, fraction, f):
        return fraction*sample1 + (f - fraction)*sample2

# Writes to output wave file
def writewavefile(data,original_file_name, new_file_name):
    read = wave.open(original_file_name, 'r')
    w = wave.open(new_file_name, 'wb')
    w.setparams(read.getparams())
    w.writeframes(data.tostring())
    read.close()
    w.close()
    read2 = wave.open(new_file_name, 'r')
    print "read.getnchannels() =", read2.getnchannels()
    print "read.getsampwidth() =", read2.getsampwidth()
    print "read.getframerate() =", read2.getframerate()
    print "read.getnframes() =", read2.getnframes()
    read2.close()

if __name__ == "__main__":
    pitchshift4(2.0, 'flute2.wav')
```

5.6 Pitchshifter Verilog

```
`timescale 1ns / 1ps

/* Top Level Module for Pitchshifter that takes in samples to process, a note frequency,
a midi frequency, an enable signal, and outputs pitchshifted samples and a done signal.
*/
module pitchshifter_overall
#(parameter N = 12, W= 15, P = 14, F = 12, B = 8)(input clock, reset,ps_enable,
input wire signed [B-1:0] mic_input,
input wire [P-1:0] note_freq,
input wire [P-1:0] midi_freq,
output wire signed [B-1:0] pitchshiftedsignal,
output wire psdone
);

wire [W-1:0] alpha;
wire [W-1:0] one_over_alpha;
        // Assume that pitchshifter module constantly outputs samples so there are always pitchshifted
        // samples to send to the CPU.
assign psdone = (alpha >0) ? 1: 0;
calculate_alphas #(.N(N), .F(F), .B(B), .W(W), .P(P)) calculate1 (
      .clock(clock),
.reset(reset),
.note_freq(note_freq),
.midi_freq(midi_freq),
.alpha(alpha),
```

```
.one_over_alpha(one_over_alpha)
   );

       wire [N+W:0] index;
wire signed [B-1:0] sample1;
wire signed [B-1:0] sample2;
wire [2:0] addr_a;
wire [N-1:0] addr_a2;
wire [N-1:0] addr_b;
wire [N-1:0] addr_b2;
wire we_a;
wire we_b;
wire we_a2;
wire we_b2;
wire signed  [7:0] mem_in_a;
wire signed  [7:0] mem_in_a2;
wire signed  [7:0] mem_in_b;
wire signed  [7:0] mem_in_b2;
wire signed  [7:0] mem_out_a;
wire signed  [7:0] mem_out_b;
wire signed  [7:0] mem_out_a2;
wire signed  [7:0] mem_out_b2;
wire [N+W:0] last;
wire [1:0] pitchshifter_state;
wire [1:0] pitchshifter_next_state;
wire ready;
wire mem_enable;

pitchshifter_pipelined #(.N(N), .F(F), .B(B), .W(W), .P(P)) uut (
.clock(clock),
.reset(reset),
.ready(ready),
.mic_input(mic_input),
.alpha(alpha),
.one_over_alpha(one_over_alpha),
.pitchshiftedsignal(pitchshiftedsignal),
.index(index),
.last(last),
.sample1(sample1),
.sample2(sample2),
.state(pitchshifter_state),
.next_state(pitchshifter_next_state),

.addr_a(addr_a),
.we_a(we_a),
.mem_in_a(mem_in_a),
.mem_out_a(mem_out_a),

.addr_a2(addr_a2),
.we_a2(we_a2),
.mem_in_a2(mem_in_a2),
```

```
.mem_out_a2(mem_out_a2),

.addr_b(addr_b),
.we_b(we_b),
.mem_in_b(mem_in_b),
.mem_out_b(mem_out_b),

.mem_in_b2(mem_in_b2),
.addr_b2(addr_b2),
.we_b2(we_b2),

.mem_out_b2(mem_out_b2),
.ps_enable(psenable),
.mem_enable(mem_enable)

);

endmodule


/* Pitchshifts incoming samples by a factor of alpha.
*/
module pitchshifter_pipelined #(parameter N = 3, W= 15, P = 24, F = 12, B = 8)
        (input clock, reset,ready, ps_enable, mem_enable,
input wire signed [B-1:0] mic_input,
input wire [W-1:0] alpha,
input wire [W-1:0] one_over_alpha,
output wire signed [B-1:0] pitchshiftedsignal,
output reg [N+W:0] index,
output wire signed [B-1:0] sample1,
output wire signed [B-1:0] sample2,
output reg [1:0] state,
output reg [1:0] next_state,
output reg [N-1:0] addr_a,addr_a2,
output reg [N-1:0] addr_b,addr_b2,
output reg we_a, we_b,we_a2, we_b2,
output wire [B-1:0] mem_in_a,mem_in_a2,
output wire signed [B-1:0]  mem_in_b,mem_in_b2,
output wire signed [B-1:0] mem_out_a, mem_out_b,
output wire signed [B-1:0] mem_out_a2, mem_out_b2,
output wire [W+N:0] last

);
parameter write_one_read_two= 0;
parameter write_two_read_one = 1;
parameter number_samples = 1<<N;

// coregen dual port 4096 by 8 bram.
dual_bram_4096_8 bram(.clka(clock), .clkb(clock),.wea(we_a), .addra(addr_a),
  .dina(mem_in_a), .douta(mem_out_a), .web(we_b),
  .addrb(addr_b), .dinb(mem_in_b), .doutb(mem_out_b));
```

```
assign mem_in_a = mic_input;
assign mem_in_a2 = mic_input;

// coregen dual port 4096 by 8 bram.
dual_bram_4096_8 bram2(.clka(clock), .clkb(clock),.wea(we_a2), .addra(addr_a2),
  .dina(mem_in_a2), .douta(mem_out_a2), .web(we_b2),
  .addrb(addr_b2), .dinb(mem_in_b2), .doutb(mem_out_b2));

assign last = alpha * number_samples - (1<<F);
wire [W-1:0] fraction;
wire signed [B-1:0] interpolated_sample;

assign sample1 = (state == write_two_read_one) ? mem_out_a : mem_out_a2;
assign sample2 = (state == write_two_read_one) ? mem_out_b : mem_out_b2;
assign fraction[11:0] = index[11:0];
assign fraction[W-1:12] = 0;

interpolate #(.F(F)) interpolator(.sample_one(sample1), .sample_2(sample2),.fraction(fraction),
  .ready(1'b1), .interpolated_sample(interpolated_sample));

assign pitchshiftedsignal = interpolated_sample;

always @ * begin
   if (reset == 1) begin
    next_state = write_one_read_two;

end
    case(state)
        write_one_read_two: begin
      if (index < last) begin
   next_state = write_one_read_two;
end
else begin
   next_state = write_two_read_one;

end
  end

        write_two_read_one: begin
   if (index < last) begin
next_state = write_two_read_one;

end
else begin
   next_state = write_one_read_two;

end
  end

        default: next_state = write_one_read_two;
```

```
    endcase
end//always @ *


reg first_sample;
assign diff = (state != next_state);
always @(posedge clock) begin

if (reset == 1) begin
 addr_a <=0;
 addr_b <=0;
 addr_a2 <=0;
 addr_b2 <=0;
 we_a <=0;
 we_b <=0;
 we_a2 <=0;
 we_b2 <=0;
 index <= 0;


end
        else if (diff) begin
                 //reset index and addresses on state transition.
 index <= 0;
 addr_a <= 0;
 addr_b <= 0;
 addr_a2 <= 0;
 addr_b2 <=0;
 if (we_a == 0 && we_b == 0) begin
 we_a <= 1;
 we_a2 <=0;
 we_b2 <=0;
 end
 else if (we_a2 == 0 && we_b2==0) begin
 we_a2 <= 1;
 we_a <=0;
 we_b <=0;
 end

end
else begin
 if (state == write_one_read_two ) begin
        //increment write address   addr_a <= addr_a + 1;
                            // increment read addresses
  addr_a2 <= (index >> F);
  addr_b2 <= (index >> F) + 1;
  index <= index + alpha;

 end
 else if (state == write_two_read_one) begin
```

```
  addr_a2 <= addr_a2 + 1;
          addr_a <= (index >> F);
          addr_b <= (index >> F) + 1;
          index <= index + alpha;

 end

end//else
state <= next_state;
end//always @ posedge clock
endmodule
/*
Computes pitchshift factor and reciprocal of pitchshift factor from target frequency
and note frequency.
*/
module calculate_alphas #(parameter N = 12, W= 15, P = 14, F = 12, B = 8)(input clock, reset,
input wire [P-1:0] note_freq,
input wire [P-1:0] midi_freq,
output wire [W-1:0] alpha,
output wire [W-1:0] one_over_alpha);

wire [13:0] remainder1, remainder2;
wire [25:0] quotient1, quotient2;
wire rfd1, rfd2;
wire [25:0] dividend2 = note_freq*(1<<12);
wire [25:0] dividend1 = midi_freq*(1<<12);

//coregen 26 bit dividend, 14 bit divisor
divider_14_by_14 div1(.dividend(dividend1), .divisor(note_freq),
    .clk(clock), .quotient(quotient1), .remainder(remainder1), .rfd(rfd1));

assign alpha = quotient1[W-1:0];
//coregen 26 bit dividend, 14 bit divisor
divider_14_by_14 div2(.dividend(dividend2), .divisor(midi_freq), .clk(clock), .quotient(quotient2),
    .remainder(remainder2), .rfd(rfd2));

assign one_over_alpha = quotient2[W-1:0];

endmodule
```

5.7 Main FSM Verilog

```
////////////////////////////////////////////////////////////////////////////////
//
// Main FSM
//
// This controls what happens when, and is central to the entire
// system working as planned.
//
////////////////////////////////////////////////////////////////////////////////
```

```
// Signal snapshots
reg [4*FREQ_BITS-1:0] target_freqs_to_process;
// We shift target_freq off of target_freqs_to_process

// State data
reg [2:0]             fsm_state;
localparam [2:0]
  S_RESET = 3'b000,
  S_INIT = 3'b001,
  S_PITCH_DETECTOR = 3'b010,
  S_PITCH_SHIFTER = 3'b011;

reg                   fsm_state_init;
localparam
  S_INIT_TARGET_FREQS = 1'b0,
  S_INIT_MEM = 1'b1;

reg                   fsm_state_init_mem;
localparam
  S_INIT_MEM_WRITE = 1'b0,
  S_INIT_MEM_WRITE_WAIT = 1'b1;

reg                   fsm_state_pd;
localparam
  S_PD_MEM = 1'b0,
  S_PD_WAIT = 1'b1;

reg                   fsm_state_pd_mem;
localparam
  S_PD_MEM_WR = 1'b0, // pitch detector, memory writing
  S_PD_MEM_WR_WAIT = 1'b1; // pitch detector, waiting for load from frozen

reg                   fsm_state_ps;
localparam
  S_PS_MEM = 1'b0,
  S_PS_WAIT = 1'b1;

reg                   fsm_state_ps_mem;
localparam
  S_PS_MEM_WR = 1'b0, // pitch detector, memory writing
  S_PS_MEM_WR_WAIT = 1'b1; // pitch detector, waiting for load from frozen

/*-----------------------
 In general, this fsm resets each top-level state's internal
 state every time it goes to another top-level state.
 -----------------------*/
always @(posedge clock)
  if (reset)
    fsm_state <= S_RESET;
  else
    case (fsm_state)
```

```
S_RESET:
  begin
            // Initialization registers
     target_freqs_to_process <= 0;
            mic_freq <= 0;
            frozen_bram_address <= 0;
            frozen_bram_we <= 0;

            // Pitch shifter registers
            fft_we <= 0;
            fft_enable_processing <= 0;
            fft_address <= 0;

            // State registers
     fsm_state <= S_INIT;
            fsm_state_init <= S_INIT_TARGET_FREQS;
            fsm_state_init_mem <= S_INIT_MEM_WRITE;
            fsm_state_pd <= S_PD_MEM;
            fsm_state_pd_mem <= S_PD_MEM_WR;
            fsm_state_ps <= S_PS_MEM;
            fsm_state_ps_mem <= S_PS_MEM_WR;
  end // case: S_RESET

        // Initialize the target frequencies, and then initialize the
        // audio memory samples
        S_INIT:
          case (fsm_state_init)
    S_INIT_TARGET_FREQS: // get frequencies
      if (target_freqs_ready)
        begin
                    // copies target_freqs
   target_freqs_to_process <= target_freqs;

                    // initialization for audio memory snapshot
                    address_audiobram_b <= bram_address + 1;
                    frozen_bram_address <= 0;
   fsm_state_init <= S_INIT_MEM;
        end
            S_INIT_MEM: // Initialize the frozen bram with audio
              case (fsm_state_init_mem)
                S_INIT_MEM_WRITE:
                  begin
                     frozen_bram_we <= 1;
                     fsm_state_init_mem <= S_INIT_MEM_WRITE_WAIT;
                  end
                S_INIT_MEM_WRITE_WAIT:
                  begin
                     frozen_bram_we <= 0;
                     if (&frozen_bram_address) // frozen bram full
                       begin
                          // initialize signals for pitch detector
```

```
                        fft_we <= 0;
                        fft_address <= 0;
                        frozen_bram_address <= 0;
                        fsm_state <= S_PITCH_DETECTOR;

                        // cleanup the mess
                        fsm_state_init <= S_INIT_TARGET_FREQS;
                        fsm_state_init_mem <= S_INIT_MEM_WRITE;
                      end
                    else
                      begin
                        address_audiobram_b <= address_audiobram_b + 1;
                        frozen_bram_address <= frozen_bram_address + 1;
                        fsm_state_init_mem <= S_INIT_MEM_WRITE;
                      end
                end // case: S_INIT_MEM_WRITE_WAIT
            endcase // case (fsm_state_init_mem)
          endcase // case (fsm_state_init)

        // The pitch detector has two stages. We write to its memory
        // (it practically inherits from the bram module definition,
        // so to speak), and then we activate the processing
        // component.
S_PITCH_DETECTOR:
          case (fsm_state_pd)
            // First, write into the memory of the pitch detector.
            S_PD_MEM:
              case (fsm_state_pd_mem)
                S_PD_MEM_WR:
                  begin
                    fft_we <= 1;
                    fsm_state_pd_mem <= S_PD_MEM_WR_WAIT;
                  end
                S_PD_MEM_WR_WAIT: // opportunity for a pipeline here.
                  begin
                    fft_we <= 0;
                    fsm_state_pd_mem <= S_PD_MEM_WR;
                    if (&fft_address)
                      begin
                        pd_enable_processing <= 1;
                        fsm_state_pd <= S_PD_WAIT;
                      end
                    else
                      begin
                        frozen_bram_address <= frozen_bram_address + 1;
                        fft_address <= fft_address + 1;
                      end
                end // case: S_PD_MEM_WR_WAIT
              endcase // case (fsm_state_pd_mem)
            // Next, wait for the pitch detector to come up with the
            // pitch
```

```
            S_PD_WAIT:
              if (pd_done)
                begin
                   // cleanup
                   fsm_state_pd <= S_PD_MEM;
                   pd_enable_processing <= 0;

                   mic_freq <= pd_output_freq;
                   if (|target_freqs_to_process[FREQ_BITS-1:0])
                     begin
                        target_freq <= target_freqs_to_process[FREQ_BITS-1:0];
                        target_freqs_to_process <= {FREQ_BITS*{1'b0}, target_freqs_to_process[4*FREQ_BI
                        fsm_state <= S_PITCH_SHIFTER;
                     end
                   else
                     fsm_state <= S_PLAY;
                end
          endcase // case (fsm_state_pd)

        // We shift off the current frequency out of the
        // target_freqs_to_process and run processing on it.
S_PITCH_SHIFTER:
          case (fsm_state_ps)
            S_PS_MEM:
              case (fsm_state_ps_mem)
                PS_MEM_WR:
                  begin
                     fsm_state_ps_mem <= S_PS_MEM_WR_WAIT;
                  end
                PS_MEM_WR_WAIT:
                  begin
                     // debug
                     fsm_state_ps <= S_PS_WAIT;
                     // cleanup
                     fsm_state_ps_mem <= S_PS_MEM_WR;
                  end
              endcase
            S_PS_WAIT:
              begin
              end
          endcase // case (fsm_state_ps)
        S_PLAY:

      endcase // case (fsm_state)
```