Real Time Counterpoint Synthesizer

Christopher Chin, Elizabeth Hass

6.111 Final Project Report

10 December 2010

Abstract
The goal of this project is to let a person otherwise stuck in a karaoke bar create their own music. Our device takes the input signal of a person singing, shifts it to a different note, and adds effects to the sound. The result is a person singing harmoniously with themselves in real time.

Table of Contents

List of Figures

Overview

People love to make music. Even people who have not learned an instrument express themselves musically through activities like karaoke and games like Rock Band. These two mediums, however, are very limited. Played perfectly, Rock Band produces a recorded soundtrack that the view has heard before. With karaoke the singer's voice can change, but he is stuck with the same recorded background music, and must bend his voice to a recording. The learning curve between the layperson and a composer is almost insurmountable. One must spend years learning to play an instrument and then more years learning music theory.

The goal of this project is to allow a person to sing with themselves with no previous musical background.The project accomplishes this by producing a pitch shifted version of the singer's voice at intervals that change according to classical music theory's 18th century counterpoint.

Half of the project entailed building audio filters and the pitch shifter. The other half of the project involved implementing the a fast Fourier transform, determining the pitch of the incoming signal, and determining the correct counterpoint for the sequence of notes played.

The incoming audio signal goes down two distinct paths: the counterpoint path and the filtering path. All of the filters are completed in the time domain. The paths converge at the pitch shifter, which completes a time domain shift based on an interval fed to it from the frequency path counterpoint module.

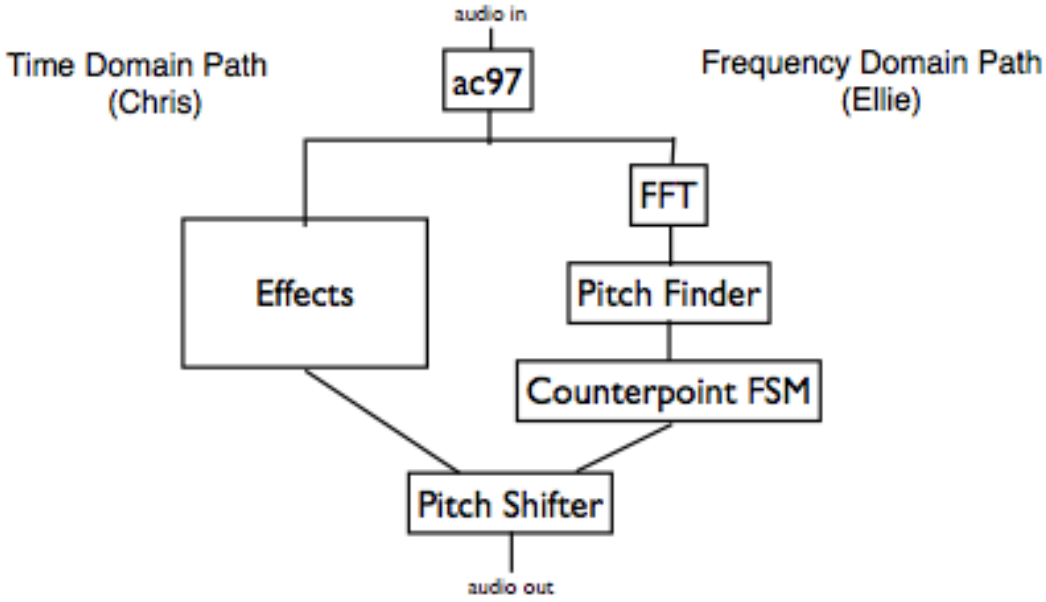
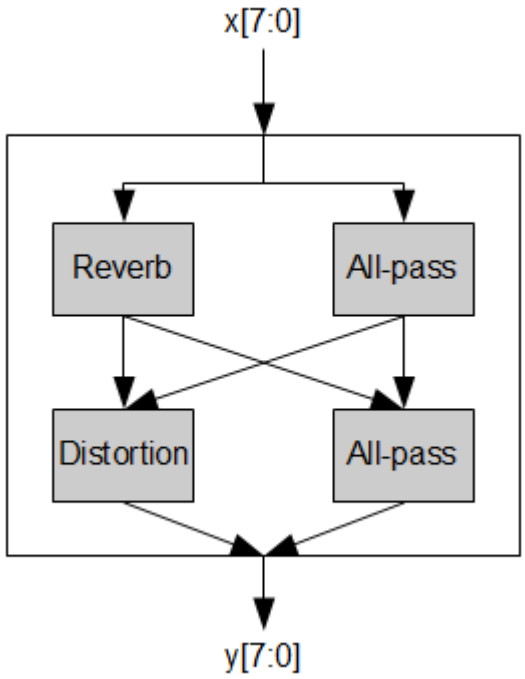


Figure 1. System Block Diagram

Effects Module:
This module applies up to two different effects to an incoming audio signal. These effects serve to shape the produced sound in aesthetically pleasing ways, and can make the counterpoint more interesting to listen to. It takes as input an 8-bit audio sample (x) and a ready signal (ready) from the AC97 audio module, a reverb-enabling switch (rev_enable), and a distortion-enabling switch (dist_enable). It outputs a processed audio signal (y).

The block itself is laid out as shown in Figure 2. The input audio signal is routed to either the reverb module or an all-pass module in the first stage of effects processing, depending on if rev_enable is high. The output of this first stage is similarly routed to either another all-pass module or the distortion module, depending on whether



dist_enable is high. This allows the user to turn on individual effects as desired.

Figure 2. Block Diagram of Effects Module

All-Pass:
This module applies no change to the incoming audio signal, which allows the effects block to use this module as a bypass for the reverb and distortion modules. It takes as input an audio sample, and outputs the same audio sample.

Reverb:
This module is responsible for adding a standard reverberation effect to the audio signal

by adding a delayed, attenuated copy of the input signal to itself. This effect lends a more natural tone to the audio signal by simulating the effects of echo in a room. It takes as input an eight bit audio sample, and outputs an eighteen-bit audio sample, of which only the eight most significant are actually used.

This module was implemented through a finite state machine. In the S_WAIT_FOR_READY state, the module waits for a ready signal, and increments a counter every time ready goes high. Every eighth time ready goes high, the current sample is saved, an offset is incremented and the module transitions into the S_CALCULATE_DECAY state. In this state, a sample is read off a circular buffer at the offset plus one (which simulates reading off of the opposite end of the buffer) and multiplied by a gain of 0.8, which is done by taking the eight high-order bits of the 18-bit result of multiplying the sample by 0.8*1024. The state then transitions into the S_CALCULATE_MIX stage, in which the saved sample from the S_WAIT_FOR_READY stage is added to the attenuated, delayed sample from the S_CALCULATE_DECAY stage and saved into the circular buffer at the offset. The sample at the opposite end of the buffer is multiplied by 0.9 and run through a 31-tap FIR low-pass filter, which is signaled to begin calcuations in the following S_OUTPUT_ONE state, after which the system waits fot the low-pass filter to finish computing within the S_OUTPUT_TWO state. After the low-pass filter finishes computing, the module returns to the S_WAIT_FOR_READY state, in which the module will set as its output the current sample added to the output from the low-pass filter.

Distortion:
This module attempts to simulate the overdrive effect commonly seen in guitar pedals by applying a gain to the incoming audio signal, then clipping it. The resulting audio waveform more resembles a square wave than a sinusoid, which gives the perceived sound a distinctly harsher tone. The module, like the others, takes as input an eight-bit audio sample, and outputs an eight-bit audio sample.

This module calculates its output with a two-state finite machine. In the first state, S_WAIT_FOR_READY, the module waits for a ready signal, in which case it will multiply the current input by two and transition into the S_CALCULATE_CLIP stage, where the result is compared against the 11111111 threshold for 8-bit signals; if the result is greater, the module clips the result at 11111111 and sets it as the output, and if not, it leaves the result as-is and sets it as the output.

Pitch Shifter Module:
This module performs a time-domain pitch shift on an audio signal; it lowers the perceived frequency of the sounds in the audio while keeping their durations constant. Since musical intervals work on relative frequency ratios, it is possible to shift any signal a specified interval and still have it sound in tune with the original. This can be achieved by separating the input signal into segments and sampling each segment at indices that increment by values smaller than or greater than 1, thereby simulating playing back the waveform at a lower or higher speed for that segment only. When multiple segments are

joined together, the overall playback speed remains the same while the frequencies of the notes are changed.

This module takes as input the 8-bit audio samples from the AC97 module, a 4-bit interval from the counterpoint FSM module, and outputs 8-bit pitchshifted audio samples. It performs the pitch shift by processing blocks of 1024 samples at a time with two dual-port 1024x8 BRAMs, reading from one BRAM while writing to the other and vice versa, alternating roles when the BRAM being written to is filled. When the module reads from a BRAM, it increments a counter by the appropriate alpha value (here defined as the ratio of the target frequency to the original frequency) and reads the sample from the counter rounded down to the nearest integer value and the sample right after it. These two samples are sent to an interpolator that takes a weighted average of the two and returns a sample that is set to the output of the module.

Fast Fourier Transform (FFT) Module

The Fast Fourier Transform module is pipelined, 512 point transform that takes 8 bit audio signals as input. In its final implementation, the transform will be 2048 points.

The human voice range extends from 82 Hz to 1046 Hz, equivalent to the notes E2 to C6 on a piano. At the lower end of the range, the difference in frequencies between notes is very small. The interval between E2 and F2, for example, is a mere 4.9Hz. To implement a counterpoint, the system must know when a person changes what pitch they are singing, and therefore the Fourier transform must have a high enough **caliber** to assign different index values to frequencies separated by 5hz. At a sample rate of 48khz, this requires a 10240 point transform. The goal of this project is to produce a counterpoint in real time, and this transform is too large to make that a possibility. Instead, the transform looks at only 1 in 5 of the incoming audio signals, with a sample rate of 9.6khz.

Each frequency band of 4.68hz is associated with an index bin. The real and imaginary components of the FFT and the associated bin number are passed on to the FFT computations module.

FFT Computations

The module sums the squares of the real and imaginary values associated with each bin. It compares the sum of each bin against the other points in the fft, finding the one with the largest magnitude. It passes the index of the bin with the largest magnitude on to the pitch finder.

Theoretically, harmonics could cause this largest magnitude bin to be a an octave higher than the incoming singer’s voice instead of the actual note, in which case the bin value would be off by an octave. However, in the world of music octaves are essentially equivalent, they sound the same. The counterpoint module handles this false octave signal and a real octave shift in exactly the same way.

Pitch Finder
The relation between index bins and the pitches they correspond to is not one to one, as each octave is double the frequency of the last. The pitch finder is essentially a look-up table that associates a pitch value with a set of indices.

The lookup table numbers the pitches from 0 for E2, the lowest note, to 45 for C6, the highest note.

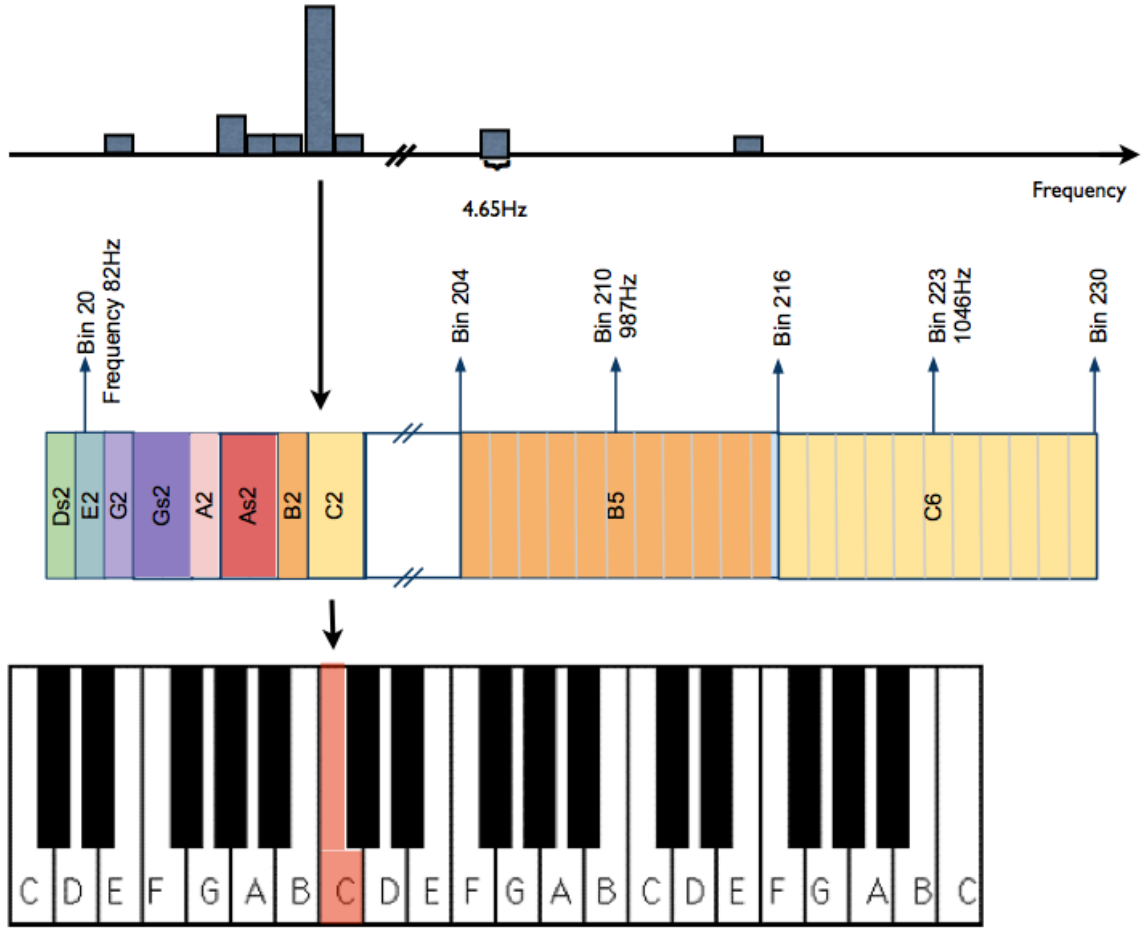


Figure 3. The pitch finder module associates an FFT index bin with a note.

Counterpoint Finite State Machine
The counterpoint module is the logical heart of the system. In music theory, counterpoint intervals are determined by previous intervals and changes in the melody. The music is entirely relative to itself, it is not based on absolute note values. This module uses this fact to implement a state machine to determine what counterpoint intervals to play.

The first step the module takes is to reduce its inputs to intervals from the absolute pitches it received from the pitch finder module. It calculates the melody interval to be the difference between the incoming note and the previous note played. The counter-

point interval is the state of the state machine. The state machine defaults to state 3, a major third. Given a state, a counterpoint interval, the module uses the melody interval to determine the next state. The full state table is shown in Figure 5.

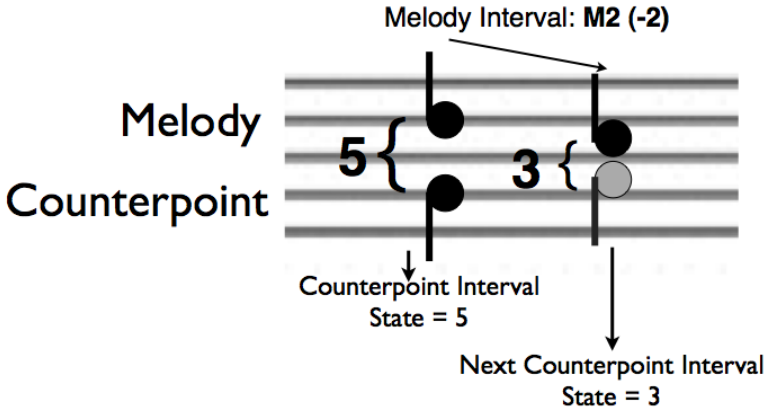


Figure 4. Counterpoint module example.

Counterpoint Interval / Melody Interval	M3	P4	P5	M6	P8	M10
m2 (1)						
M2 (2)	P5	P5	P8	P8	M10	P8
m3 (3)						
M3 (4)	M3	P8	M6	P5	M6	P8
P4 (5)	P5	P8	M6	P8	M10	M10
A4 (6)						
P5 (7)	P5	P5	M10	P8	M10	M10
-m2 (-1)						
-M2 (-2)	M3	M3	M3	M3	M6	P8
-m3 (-3)						
-M3 (-4)	M3	M3	M3	M3	M6	M6
-P4 (-5)	M3	M3	M3	M3	P4	P5
-A4 (-6)						
-P5 (-7)	M3	M3	M3	M3	M3	P5

Figure 5. Table of Counterpoint State Changes. Given a counterpoint interval, the next counterpoint interval is determined by the melody interval. The orange square corresponds to the state change in Figure 4.

Given more time, I would have added in the minor intervals as states and reworked the major interval states to include them. The music produced would be significantly more interesting than just major chords. I also would have included more melody intervals in the state machine. Any intervals that I could not implement, either because they would have required a minor chord to follow or for the sake of simplicity, remain in the same state. The result should be a sound that switches musical keys as a person sings and keeps within the states allowed.

Western music divides the frequency domain into a repeating set of twelve notes. These notes correspond to the keys on keyboard. At the end of the set of twelve, you reach an octave, a doubling of the frequency of the first note played. If an input signal jumps an octave or more, the counterpoint module reduces that interval to the equivalent interval within an octave, one of the twelve notes. Essentially, if you do not change note names, you do not change what counterpoint rules apply. At the point where a singer makes that jump they are breaking the rules anyways and the module simply follows along.

The module state machine attempts to follow the simplest rules of 18th century counter-point:

1. No interval larger than a tenth will be played
2. Allowed intervals are limited to the major third, perfect fourth, perfect fifth, major sixth, perfect eight, and major tenth.
3. Contrary motion between the melody and counterpoint lines played when possible
4. Parallel fifths and octaves are forbidden
5. Avoid dissonant intervals - seconds, sevenths, and the augmented fourth

Testing, Integration, and Debugging:

Effects Module:

Each of the effects in this module was implemented and tested in ModelSim before testing on an FPGA. However, when placed on the FPGA, the reverb filter did not seem to produce an audible effect in the output sound at first. This was partially because of the fact that the circular buffer used as the delay was rather small during testing, but the issue cropped up even when the buffer was very large. This was because the buffer was being looped through too fast for the delayed signals to be audible and separate from the original input; part of the reason for only advancing the offset in the delay every eight samples (and running the output of the buffer through the low-pass filter) was to extend the amount of time before the offset traversed the whole of the circular buffer. At a 48.1KHz sampling rate, the effective sampling rate for the buffer was 6KHz; combined with a buffer of length 1024, this afforded about 0.17 seconds of delay before the offset looped back and started playing the saved samples.

Pitch Shifter:

While this module functioned as specified, it fell short in actual implementation due to the fact that it introduced a significant amount of high-frequency noise into the output. Increasing the length of the BRAMs (so that there are less jumps between segments of

the signal) also increased the quality of the sound immensely, but also increased the compilation time of the project to on the order of hours; even a pair of modest 1024-sample BRAMs took an hour to compile.

In addition, the algorithm used was inherently flawed since some of the original audio never gets played, as our intervals only specify a downwards shift in pitch; a better idea might have been to employ a shift-overlap-add configuration to ensure continuous sound, but its implementation would have rivaled a frequency-domain based pitch shifter in terms of difficulty.

Conclusion

Our system implements several filters and a very good pitch shifter. Currently, the FFT is not integrated into the system, but a fake fft input runs through the rest of system and successfully generates counterpoint intervals. This last component of the system will be added during IAP. We were pleasantly surprised at how much signal integrity the pitch shifter kept when dealing with a human voice. We succeeded in making a system that operates in real time, capable of doing almost everything we intended.

References

1. Hom, G. FFT Modules. Fall 2007. Massachusetts Institute of Technology. <http://web.mit.edu/6.111/www/f2007/handouts/fft.v>.
2. Stojanovic, Vladimir. 6.111. Fall. 2010. Massachusetts Institute of Technology. <http://web.mit.edu/6.111/www/f2010/index.html>.

Appendix A: FFT Computations Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////
// Company:
// Engineer:
//
// Create Date:    16:08:06 12/07/2010
// Design Name:
// Module Name:    fft_computations
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////
module fft_computations (     clk,
                                                        clock_27mhz,
                                                        reset,
                                                        ready,
                                                        sel,
                                                        from_ac97_data,
                                                        debug,xk_re,
                                                        xk_im,
                                                        xk_index,
                                                        xk_done,
                                                        pitch,
                                                        interval,
                                                        done,
                                                        fm_done,
                                                        pitch_done,
                                                        hmag,
                                                        hindex,
                                                        mag,
                                                        mag_index
                                                );

  input clk, reset, ready, debug, clock_27mhz;
  input [3:0] sel;
      input [7:0] from_ac97_data;
      output [5:0] pitch;
  output [3:0] interval;
  output reg done;
      output signed [22:0] xk_re,xk_im;
      output [13:0] xk_index;
      output xk_done;
  output fm_done;
  output pitch_done;
```

```
  output reg [19:0] hmag;
  output reg [13:0] hindex;
  output [19:0] mag;
  output [13:0] mag_index;
  reg fm_ready;
  reg [5:0] final_pitch;
  wire [3:0] state;
      wire cpt_done;
      assign interval = state;

      // variables to downsample the fft from 48khz to 9.6khz
      reg fft_ready;
      reg [2:0] fft_count;

  // find magnitude function settings: debug OFF
  findmagnitude magnitude_tone      (     .clk(clk),
                                          .reset(reset),
                                          //.ready(fm_ready),
                                          .ready(ready),
                                          .sel(sel),
                                          .debug(1'b0),
                                          .xk_re(xk_re),
                                          .xk_im(xk_im),
                                          .xk_index(xk_index),
                                          .mag(mag),
                                          .mag_index(mag_index),
                                          .done(fm_done)
                                          ); // add in other variables

      pitchfinder pitch_tone  (     .clk(clk),
                                                          .reset(reset),
                                                          .ready(fm_done),
                                                          .debug(debug),
                                                          .index(hindex),
                                                          .pitch(pitch),
                                                          .done(pitch_done)
                                                    );

  counterpoint_fsm counterpoint_tone(     .clk(clk),
                                    .reset(reset),
                                    .ready(done),
                                    .debug(debug),
                                    .pitch(final_pitch),
                                    .state(state),
                                    .done(cpt_done)
                                    );
//    myfft fft(  .clk(clock_27mhz),
//                              .ce(reset | fft_ready),
//                              .xn_re(from_ac97_data),
//                              .xn_im(8'b0),
//                              .start(1'b1),
//                              .fwd_inv(1'b1),
//                              .fwd_inv_we(reset),
//                              .xk_re(xk_re),
//                              .xk_im(xk_im),
```

```
//                              .xk_index(xk_index),
//                              .done(xk_done)
//                         );
      fakefft fft(
        .clk(clock_27mhz),
            .ready(ready),
            .debug(1'b0),
            .xk_re(xk_re),
            .xk_im(xk_im),
            .xk_index(xk_index),
            .fft_done(xk_done)
    );

  initial begin
    hmag = 0;
    hindex = 0;
    fm_ready = 0;
  end

      always @(posedge clock_27mhz) begin
//          if(ready) fft_count <= (fft_count == 4) ? (0) : (fft_count + 1);
//          if(ready) fft_ready <= (fft_count == 4) ? (1) : (0);
//          else fft_ready <= 0;

             if (xk_done)
                begin
                hmag<=0;
                    hindex<=0;
                    end
             else
                begin
                if ((xk_index) > 16 && (xk_index) < 231)
                   begin
            // fm_ready<=1;  // do the find magnitude calculation
                   if (fm_done)
                      begin // when it finishes...
                      if((mag)>(hmag))
                         begin
                                          hmag<=mag;
                                          hindex<=mag_index;
                         end
                      else hmag<=(hmag);
                      end    // fm_done
                  end

                 else if ((xk_index)>=231)
                        begin
                        final_pitch<=pitch;
                        done<=1;
                        end
                 else done<=0; // index<17, which should never happen

            end
      end
endmodule // fft_computations
```

Appendix B: Pitch Finder Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
/////
// Company: 6.111 Laboratories
// Engineer: ehass
//
// Create Date:    15:47:31 12/07/2010
// Design Name:
// Module Name:    pitchfinder
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
/////
// Pitchfinder Module
// Tested, works correctly and in 1 clock cycle
// Takes an index and returns an integer value for the associated pitch
//////////////////////////////////////////////////////////////////////////////////
/////


module pitchfinder(clk, reset, ready, debug, index, pitch, done);
   input clk, reset,ready, debug;
   input [13:0] index;
   output reg [5:0] pitch;
   output reg done;

   always @(posedge clk) begin  // index to pitch mapping
      if(reset) begin
         done=0;
         pitch=6'bXXXXXX;
     end

   else if (ready) begin
      if(index<17) pitch =6'bXXXXXX;
      else if(index==17) pitch=0; //e2
      else if(index==18) pitch=1; //f2
      else if(index==19) pitch=2; //fs2
      else if(index==20) pitch=3; // g2
      else if(index>=21 && index<=22) pitch=4; // gs2
      else if(index==23) pitch=5; // a2
      else if(index==24) pitch=6; // as2
      else if(index>=25 && index<=26) pitch=7; // b2
      else if(index==27) pitch=8; // c3
      else if(index>=28 && index<=29) pitch=9; // cs3
```

```
      else if(index>=30 && index<=31) pitch=10; // d3
      else if(index>=32 && index<=33) pitch=11; // ds3
      else if(index>=34 && index<=35) pitch=12; // e3
      else if(index>=36 && index<=37) pitch=13; // f3
      else if(index>=38 && index<=39) pitch=14; // fs3
      else if(index>=40 && index<=42) pitch=15; // g3
      else if(index>=43 && index<=45) pitch=16; // gs3
      else if(index>=46 && index<=48) pitch=17; // a3
      else if(index>=49 && index<=50) pitch=18; // as3
      else if(index>=51 && index<=53) pitch=19; // b3
      else if(index>=54 && index<=57) pitch=20; // c4
      else if(index>=58 && index<=60) pitch=21;// cs4
      else if(index>=61 && index<=63) pitch=22; // d4
      else if(index>=64 && index<=67) pitch=23; // ds4
      else if(index>=68 && index<=71) pitch=24; // e4
      else if(index>=72 && index<=76) pitch=25; // f4
      else if(index>=77 && index<=80) pitch=26; // fs4
      else if(index>=81 && index<=85) pitch=27; // g4
      else if(index>=86 && index<=90) pitch=28; // gs4
      else if(index>=91 && index<=95) pitch=29; // a4
      else if(index>=96 && index<=102) pitch=30; // as4
      else if(index>=103 && index<=107) pitch=31; // b4
      else if(index>=108 && index<=114) pitch=32; // c5
      else if(index>=115 && index<=121) pitch=33; // cs5
      else if(index>=122 && index<=128) pitch=34; // d5
      else if(index>=129 && index<=135) pitch=35; // ds5
      else if(index>=136 && index<=143) pitch=36; // e5
      else if(index>=144 && index<=152) pitch=37; // f5
      else if(index>=153 && index<=162) pitch=38; // fs5
      else if(index>=163 && index<=173) pitch=39; // g5
      else if(index>=174 && index<=183) pitch=40; // gs5
      else if(index>=184 && index<=194) pitch=41; // a5
      else if(index>=195 && index<=203) pitch=42; // as5
      else if(index>=204 && index<=215) pitch=43; // b5
      else if(index>=216 && index<=230) pitch=44; // c6
      else pitch=6'bXXXXXX;

      done=1;
      end
  else done = 0;
   end // always @ posedge clk
endmodule // pitchfinder
```

Appendix C: Magnitude Finder Verilog

```
module findmagnitude(clk, reset, ready, sel, debug, xk_re, xk_im, xk_index,
mag,mag_index,done);
  input clk, reset,ready,debug;
  input [3:0] sel;
  input signed [22:0] xk_re,xk_im;
  input [13:0] xk_index;
  output reg [13:0] mag_index;
  output reg [19:0] mag;
  output reg done;

  //wire signed [9:0] xk_re_scaled = xk_re >> sel;
  //wire signed [9:0] xk_im_scaled = xk_im >> sel;
  wire signed [9:0] xk_re_scaled;
  wire signed [9:0] xk_im_scaled;

  assign xk_re_scaled = xk_re;
  assign xk_im_scaled = xk_im;


    // process fft data
  reg [2:0] state;
  reg [9:0] haddr;
  reg [19:0] rere,imim;
  reg [19:0] mag2;

  always @ (posedge clk) begin
    if (reset) begin
      state <= 0;
      done <=0;
      if(debug) $display("reset, ready=%d",ready);
    end
    else case (state)
     3'h0: begin
        if (ready) state <= 1;
        else state<=0;

        if(debug) $display("state 0, ready=%d",ready);
        done <=0;
       end
     3'h1: begin
              // only process data with index < 1024
              state <= (xk_index[13:10] == 0) ? 2 : 0;
           haddr <= xk_index[9:0];
           rere <= xk_re_scaled * xk_re_scaled;
           imim <= xk_im_scaled * xk_im_scaled;
           done <=0;
           if(debug) $display("state 1, xk_re_scaled=%d, xk_im_scaled=%d",
xk_re_scaled, xk_im_scaled);
           end
     3'h2: begin
           state <= 0;
             mag <= rere + imim;
             mag_index <= haddr-1;
             done <=1;
```

```
            if(debug) $display("state 2:rere=%d,imim=%d, mag=%d, mag_index=
%d",rere, imim,mag,mag_index);
          end
       default: state<=0;
    endcase
    //if (debug) $display("done=%d, sqrt_done=%d",done,sqrt_done);
  end

endmodule
```

Appendix D: Counterpoint FSM Verilog

```
module counterpoint_fsm(clk, reset, ready, debug, pitch, state, done);
   input clk, reset, ready,debug;
   input [5:0] pitch;
   output reg [3:0] state;
   output done;

   reg [3:0] last_state;
   reg [5:0] last_pitch;
   wire signed [5:0] real_interval;
   reg signed [5:0] melody_interval;
   wire cheats;

//    parameter S0=4'b0000;   // Unison. Should never be selected.
//   parameter S3=4'b1100;  // Major 3rd
//   parameter S4=4'b0001;  // Perect 4th
//   parameter S5=4'b0011;  // Perfect 5th
//   parameter S6=4'b0111;  // Major 6th
//   parameter S8=4'b1111;  // Perfect 8th
//   parameter S10=4'b1110; // Major 10th

    parameter S0=4'd0;
    parameter S3=4'd3;
    parameter S4=4'd4;
    parameter S5=4'd5;
    parameter S6=4'd6;
    parameter S8=4'd8;
    parameter S10=4'd10;

   cheat melodycheat(.clk(clk),
.melody_interval(melody_interval),.cheats(cheats));

   //assign melody_interval=(last_pitch-pitch)%12;
   assign real_interval=(pitch-last_pitch);

   always @(posedge clk) begin  // should be done in 1 clock cycle (how do i
prove this is true?)

      // MODULUS for my purposes
      // take the real interval between two notes and turn it into an
interval less than an octave
      // interval is limited to 45, which allows this nice, fast cheat
        if(real_interval>12) melody_interval<=real_interval-12;
        else if(real_interval>24) melody_interval<=real_interval-24;
        else if(real_interval>36) melody_interval<=real_interval-36;
        else melody_interval<=real_interval;

        if(real_interval<-36) melody_interval<=real_interval+36;
        else if(real_interval<-24) melody_interval<=real_interval+24;
        else if(real_interval<-12) melody_interval<=real_interval+12;
        else melody_interval<=real_interval;



      if(debug) $display("pitch=%d, last_pitch=%d, real_interval=%d,
```

```
melody_interval=%d, state=%d, last_state=%d",pitch, last_pitch,
real_interval, melody_interval, state, last_state);

      if (reset) state <= S3;
      else if (melody_interval==0)
        begin  // if the note doesn't change, hold the state as it is
        state<=state;
        end

      else begin
        //if(debug) $display("state cases");
        case (last_state)
                          S0: state<=S3;
             S3: begin
                      if(melody_interval<0) begin
                        if(debug) $display("state 3, interval=n2");
                        state<=S3;
                      end
                      else case (melody_interval)
                        2: begin
                         if(debug) $display("state 3, interval=p2");
                          state<=S5; //M2
                          end
                        4: state<=S5; //M3
                        5: state<=S5; //P4
                        7: state<=S5; //P5
                        default: state<=last_state;
                      endcase // case (melody_interval)
            end
            S4: begin
             if(melody_interval<0) state<=S3;
             else case (melody_interval)
                    2: state<=S5; //M2
                    4: state<=S8; //M3
                    5: state<=S8; //P4
                    7: state<=S5; //P5
                    default: state<=last_state;
                  endcase // case (melody_interval)
            end
            S5: begin
             if(melody_interval<0) state<=S3;
             else case (melody_interval)
                    2: state<=S8; //M2
                    4: state<=S6; //M3
                    5: state<=S6; //P4
                    7: state<=S10; //P5
                    default: state<=last_state;
                  endcase // case (melody_interval)
            end
            S6: begin
             if(melody_interval<0) state<=S3;
             else case (melody_interval)
                    2: state<=S8; //M2
                    4: state<=S5; //M3
                    5: state<=S8; //P4
                    7: state<=S8; //P5
```

```
                  default: state<=last_state;
                endcase // case (melody_interval)
            end
            S8: begin
             case (melody_interval)
               -2: state<=S6; //-M2
               -4: state<=S6; //-M3
               -5: state<=S4; //-P4
               -7: state<=S3; //-P5
               2: state<=S10; //M2
               4: state<=S6; //M3
               5: state<=S10; //P4
               7: state<=S10; //P5
               default: state<=last_state;
             endcase // case (melody_interval)
            end
            S10: begin
             case (melody_interval)
               -2: state<=S8; //-M2
               -4: state<=S6; //-M3
               -5: state<=S5; //-P4
               -7: state<=S5; //-P5
               2: state<=S8; //M2
               4: state<=S8; //M3
               5: state<=S10; //P4
               7: state<=S10; //P5
               default: state<=last_state;
             endcase // case (melody_interval)
            end // case: S10
            default: begin
              if(debug) $display("default");
              state<=S3; // might be better as state=last_state?
             end
          endcase // case (last_state)
        end // else
        last_pitch<=pitch;
        last_state<=state;
   end // always @ (posedge clk)
endmodule // counterpoint_fsm



module cheat(clk,melody_interval, cheats);
   input clk;
   input signed [5:0] melody_interval;
   output reg           cheats;

   always @(posedge clk) begin
       if(melody_interval==0 ||  //0
   melody_interval==1 ||  //m2
       melody_interval==3 ||  //m3
       melody_interval==6 ||  //A4
       melody_interval==8 ||  //m6
       melody_interval==9 ||  //M6
       melody_interval==10 ||  //m7
```

```
       melody_interval==11 || //M7
       melody_interval==12 ) begin  //P8

          cheats=1;
      end
      else cheats=0;
   end // always @ (posedge clk)
endmodule // cheat
```

Appendix E: Effects Module Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////
// Company:
// Engineer:
//
// Create Date:    21:12:10 11/23/2010
// Design Name:
// Module Name:    effects
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////
module effects_block(clock, reset, ready, rev_enable, dist_enable, x, y);
      input wire clock;
      input wire reset;
      input wire ready;
      input wire rev_enable;
      input wire dist_enable;
      input wire [7:0] x;
      output reg [7:0] y;

      wire [17:0] rev_output;
      wire [7:0] dist_output;
      wire [7:0] rev_pass;
      wire [7:0] dist_pass;
      wire reverb_done, pass_done;

      reverb #(.DELAY(10)) rev_blk
                                    (     .clock(clock),
                                          .reset(reset),
                                          .ready(ready),
                                          .x(x),
                                          .y(rev_output[17:0]),
                                          .reverb_done(reverb_done)     );
      all_pass rev_all_pass(  .clock(clock),
                              .ready(ready),
                              .x(x),
                              .y(rev_pass),
                              .done(pass_done)  );

      distortion_simple dist_blk(   .clock(clock),
                                    .reset(reset),
                                    .ready((rev_enable) ? reverb_done :
```

```
pass_done),                                          .x((rev_enable) ?
rev_output[7:0] : rev_pass),
                                    .y(dist_output) );
      all_pass dist_all_pass( .clock(clock),
                              .ready((rev_enable) ? reverb_done : pass_done),
                              .x((rev_enable) ? rev_output[7:0] : rev_pass),
                              .y(dist_pass));


      always @(posedge clock) begin
            y <= (dist_enable) ? dist_output : dist_pass;
      end

endmodule
```

Appendix F: Reverb Module Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////
// Company:
// Engineer:
//
// Create Date:    21:10:05 11/23/2010
// Design Name:
// Module Name:    reverb
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////
module reverb #(parameter DELAY = 4)
                                (clock, reset, ready, x, y, reverb_done, state,
lpf_ready, lpf_done, filtered, mix);
      input wire clock;
      input wire reset;
      input wire ready;
      input wire [7:0] x;
      output reg [17:0] y;
      output reg reverb_done;
      output reg [2:0] state;

      parameter S_WAIT_FOR_READY = 0;
      parameter S_CALCULATE_DECAY = 1;
      parameter S_CALCULATE_MIX = 2;
      parameter S_OUTPUT_ONE = 3;
      parameter S_OUTPUT_TWO = 4;

      reg [2:0] next_state;
      reg [DELAY-1:0] offset;
      reg [DELAY-1:0] delay_pointer;
      reg [7:0] delay_buffer [(1<<DELAY)-1:0];

      reg [17:0] decay_result;
      output reg [17:0] mix;
      reg [7:0] sample;
      reg [3:0] counter;

      integer i;

      initial begin
            offset = 0;
```

```
            $display("begin zero initialization for array length ",
(1<<DELAY));
            for (i = 0; i < (1<<DELAY); i = i + 1) begin
                  delay_buffer[i] = 8'b0;
            end
            $display("end initial");
      end

      output reg lpf_ready;
      output [17:0] filtered;
      output lpf_done;

      fir31_lpf lpf(.clock(clock), .ready(lpf_ready), .reset(reset),
.x(mix[17:10]), .y(filtered), .done(lpf_done));

      always @(posedge clock) begin
            if (reset)
                  begin
                  offset = 0;
                  delay_pointer = 0;

                  reverb_done = 0;
                  y = 0;
                  counter = 0;

                  next_state = S_WAIT_FOR_READY;
                  end

            else begin
                  case (state)
                        S_WAIT_FOR_READY:
                              begin
                              if (ready)
                                    begin
                                    counter = (counter + 1);
                                    if (counter == 8)
                                          begin
                                          counter = 0;
                                          $display("offset = ",((offset ==
                                                (1<<DELAY) - 1) ? 0 : offset
                                                + 1));
                                          offset = ((offset == (1<<DELAY) -
                                                1) ?  0 : offset + 1);

                                          $display("delay_pointer = ",
                                                ((offset+1 == (1<<DELAY)) ? 0
                                                : offset + 1));
                                          delay_pointer = (offset+1 ==
                                                (1<<DELAY)) ? 0 :
                                                offset + 1;
                                          next_state = S_CALCULATE_DECAY;
                                          sample = x;
                                          end
                                    else
                                      begin
                                      y = x + filtered[17:10];
```

```
                                    reverb_done = 1;
                                    end
                                end
                        end

                    S_CALCULATE_DECAY:
                        begin
                        $display("delayed sample:
                                        ",delay_buffer[delay_pointer]
);

                        decay_result = 819*delay_buffer[delay_pointer];
                        next_state = S_CALCULATE_MIX;
                        end

                    S_CALCULATE_MIX:
                        begin
                        $display("calculating mix");

                        delay_buffer[offset] = sample +
decay_result[17:10];
                        mix = 922*delay_buffer[delay_pointer];
                        next_state = S_OUTPUT_ONE;
                        end

                    S_OUTPUT_ONE:
                        begin
                        $display("output: sample: ", sample, " delay:
",                                  mix[17:10]);
                        lpf_ready = 1;
                        next_state = S_OUTPUT_TWO;
                        end

                    S_OUTPUT_TWO:
                        begin
                        lpf_ready = 0;
                        if (lpf_done)
                            begin
                            next_state = S_WAIT_FOR_READY;
                            end
                        end

                    default:
                        begin
                        next_state = S_WAIT_FOR_READY;
                        end
                endcase
            end
            state = next_state;
      end
endmodule
```

Appendix G: Distortion Module Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////
// Company:
// Engineer:
//
// Create Date:    20:46:37 11/30/2010
// Design Name:
// Module Name:    distortion_simple
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////
module distortion_simple(clock, reset, ready, x, y, dist_done, state);
      input wire clock;
      input wire reset;
      input wire ready;
      input wire [7:0] x;
      output reg [7:0] y;
      output reg dist_done;

      output reg state;
      reg next_state;

      reg [11:0] x_mult;
      parameter S_WAIT_FOR_READY = 0;
      parameter S_CALCULATE_CLIP = 1;

      always @(posedge clock) begin
            if (reset)
                  begin
                        y = 0;
                        dist_done = 0;
                        next_state = S_WAIT_FOR_READY;
                  end
            else begin
                  case (state)

                        S_WAIT_FOR_READY:
                              begin
                              if (ready)
                                    begin
                                          $display("triggered with sample:
",x);
                                          x_mult = 2 * x;
```

```
                                    dist_done = 0;
                                    next_state = S_CALCULATE_CLIP;

                              end
                        end

                  S_CALCULATE_CLIP:
                        begin
                        y = (x_mult[11:0] > 11'b0000_11111111) ?
                                          8'b11111111 : x_mult[7:0];
                        dist_done = 1;
                        next_state = S_WAIT_FOR_READY;
                        end

                  default:
                        begin
                        next_state = S_WAIT_FOR_READY;
                        end

            endcase
      end
      state = next_state;
end

endmodule
```

Appendix H: All-Pass Module Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////
// Company:
// Engineer:
//
// Create Date:    21:11:48 11/23/2010
// Design Name:
// Module Name:    all_pass
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////
module all_pass(clock, ready, x, y, done);
      input wire clock;
      input wire ready;
      input wire [7:0] x;
      output reg [7:0] y;
      output reg done;

      always @(posedge clock) begin
            if (ready)
                  begin
                  y <= x;
                  done <= 1;
                  end
      end
endmodule
```

Appendix I: Pitch Shifter Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////
// Company:
// Engineer:
//
// Create Date:    00:59:03 12/03/2010
// Design Name:
// Module Name:    pitchshifter
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////
module pitchshifter     #(parameter LOG_SL = 8)
                                        (     clock,
                                              reset,
                                              ready,
                                              audio_in,
                                              interval,
                                              audio_out,
                                              // debug outputs
                                              state,
                                              index,
                                              sample_out_a,
                                              sample_out_b,
                                              int_alpha,
                                              alpha_count,
                                              alpha_index,
                                              alpha_index_plus_one
                                        );

      input clock;
      input reset;
      input ready;
      input [7:0] audio_in;
      input [3:0] interval;
      output [7:0] audio_out;

      parameter S_READ_A_WRITE_B = 0;
      parameter S_READ_B_WRITE_A = 1;

      output reg state;
      reg next_state;
      output reg [(LOG_SL):0] index;
```

```
	wire [7:0] mem_out_a;
	wire [7:0] mem_out_a2;
	wire [7:0] mem_out_b;
	wire [7:0] mem_out_b2;

	wire we_a, we_b;

	wire [(LOG_SL-1):0] addr;
	assign addr = index;
	wire [(LOG_SL-1):0] addr_plus_one;
	assign addr_plus_one = addr + 1;

	output wire [7:0] sample_out_a;
	output wire [7:0] sample_out_b;

	assign sample_out_a = (state == S_READ_A_WRITE_B) ? mem_out_a :
mem_out_b;
	assign sample_out_b = (state == S_READ_A_WRITE_B) ? mem_out_a2 :
mem_out_b2;
	assign we_a = (state == S_READ_A_WRITE_B) ? 1'b0 : 1'b1;
	assign we_b = (state == S_READ_A_WRITE_B) ? 1'b1 : 1'b0;

	output reg [12:0] int_alpha;
	output reg [((LOG_SL-1) + 12):0] alpha_count;
	output wire [(LOG_SL-1):0] alpha_index;
	output wire [(LOG_SL-1):0] alpha_index_plus_one;
	assign alpha_index = alpha_count[((LOG_SL-1) + 12):12];
	assign alpha_index_plus_one = (alpha_index == ((1<<LOG_SL)-1)) ? (0) :
(alpha_index + 1);

	wire [11:0] fraction;
	assign fraction = alpha_count[11:0];

	// output reg [11:0] alpha_index;
	// output wire [11:0] alpha_index_plus_one;
	// assign alpha_index_plus_one = (alpha_index == SAMPLE_LENGTH) ? (0) :
(alpha_index + 1);

	dual_bram  #(.LOGSIZE(LOG_SL), .WIDTH(8))
		memory_a (
		.addra((state == S_READ_A_WRITE_B) ? alpha_index : addr),
		.addrb((state == S_READ_A_WRITE_B) ? alpha_index_plus_one :
addr_plus_one),
		.clk(clock),
		.dina(audio_in),
		.dinb(audio_in),
		.douta(mem_out_a),
		.doutb(mem_out_a2),
		.wea(we_a),
		.web(we_a)
		);

	dual_bram #(.LOGSIZE(LOG_SL), .WIDTH(8))
		memory_b (
		.addra((state == S_READ_A_WRITE_B) ? addr : alpha_index),
```

```
            .addrb((state == S_READ_A_WRITE_B) ? addr_plus_one :
alpha_index_plus_one),
            .clk(clock),
            .dina(audio_in),
            .dinb(audio_in),
            .douta(mem_out_b),
            .doutb(mem_out_b2),
            .wea(we_b),
            .web(we_b)
            );

      interpolator interpolator(
                    .clock(clock),
                    .fraction(fraction),
                    .sample_a(sample_out_a),
                    .sample_b(sample_out_b),
                    .interpolated(audio_out)
                   );


      always @* begin
            case (interval)

      //    parameter S0=4'b0000;  // Unison,      alpha = 1;
      // parameter S3=4'b1100;  // Major 3rd,   alpha = 0.794;
      // parameter S4=4'b0001;  // Perfect 4th, alpha = 0.749;
      // parameter S5=4'b0011;  // Perfect 5th, alpha = 0.667;
      // parameter S6=4'b0111;  // Major 6th,   alpha = 0.595;
      // parameter S8=4'b1111;  // Perfect 8th, alpha = 0.500;
      // parameter S10=4'b1110; // Major 10th,  alpha = 0.442;


                  // This outputs alpha multiplied by 2**12!

//                4'b0000: int_alpha = 4096;
//                4'b1100: int_alpha = 3252;
//                4'b0001: int_alpha = 3068;
//                4'b0011: int_alpha = 2732;
//                4'b0111: int_alpha = 2437;
//                4'b1111: int_alpha = 2048;
//                4'b1110: int_alpha = 1810;
//                default: int_alpha = 4096;

                  4'b0000: int_alpha = 4096;
                  4'b0011: int_alpha = 3252;
                  4'b0100: int_alpha = 3068;
                  4'b0101: int_alpha = 2732;
                  4'b0110: int_alpha = 2437;
                  4'b1000: int_alpha = 2048;
                  4'b1010: int_alpha = 1810;
                  default: int_alpha = 4096;

            endcase
      end

      always @(posedge clock) begin
```

```
            if (reset) begin
                        index = 0;
                        alpha_count = 0;
                        next_state = S_READ_A_WRITE_B;
                  end
            else begin

                  case (state)
                        S_READ_A_WRITE_B:
                              begin
                              if (ready)
                                    begin

                                    alpha_count = (alpha_count + int_alpha);
                                    if (alpha_count[((LOG_SL-1) + 12):12] >=
(1<<LOG_SL))
                                      begin
                                      // alpha_count =
{(alpha_count[((LOG_SL-1) + 12):12] - (1<<LOG_SL)),(alpha_count[11:0])};
                                      alpha_count = {(alpha_count[((LOG_SL-1)
+ 12):12] - (1<<LOG_SL)),(alpha_count[11:0])};
                                      end

                                    index = (index + 1);
                                    $display(index);

                                    end

                              if ((index) == (1<<LOG_SL))
                                    begin
                                    index = 0;
                                    alpha_count = (alpha_count[11:0]);
                                    next_state = S_READ_B_WRITE_A;
                                    end
                              end

                        S_READ_B_WRITE_A:
                              begin
                              if (ready)
                                    begin


                                    alpha_count = (alpha_count + int_alpha);
                                    if (alpha_count[((LOG_SL-1) + 12):12] >=
(1<<LOG_SL))
                                      begin
                                      alpha_count = {((alpha_count[((LOG_SL-
1) + 12):12]) - (1<<LOG_SL)),(alpha_count[11:0])};
                                      end

                                    index = (index + 1);
                                    end

                              if ((index) == (1<<LOG_SL))
                                    begin
                                    index = 0;
```

```
                                    alpha_count = (alpha_count[11:0]);
                                    next_state = S_READ_A_WRITE_B;
                                    end
                              end

                        default:
                              begin
                                    index = 0;
                                    alpha_count = 0;
                                    next_state = S_READ_A_WRITE_B;
                              end
                  endcase
            end
            state = next_state;
      end

endmodule
```

Appendix J: Interpolator Verilog

```
`timescale 1ns / 1ps

module interpolator(clock, fraction, sample_a, sample_b, interpolated);
  input clock;
  input [11:0] fraction; //multiplied by 2**12...
  input [7:0] sample_a;
  input [7:0] sample_b;
  output [7:0] interpolated;

  reg [19:0] interpolated_reg;

  assign interpolated = interpolated_reg[19:12];

  wire [11:0] one_minus_fraction;
  assign one_minus_fraction = (4096 - fraction);

  always @(posedge clock) begin
    interpolated_reg = ((sample_a*one_minus_fraction) + (sample_b*fraction));
  end

endmodule
```

Appendix K: Dual BRAM Verilog

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////
// Company:
// Engineer:
//
// Create Date:    19:50:16 12/04/2010
// Design Name:
// Module Name:    dual_bram
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
//////
module dual_bram  #(parameter LOGSIZE=12, WIDTH=8)
                              (     input wire [LOGSIZE-1:0] addra,
                                    input wire [LOGSIZE-1:0] addrb,
                                    input wire clk,
                                    input wire [WIDTH-1:0] dina,
                                    input wire [WIDTH-1:0] dinb,
                                    output reg [WIDTH-1:0] douta,
                                    output reg [WIDTH-1:0] doutb,
                                    input wire wea,
                                    input wire web    );
      (* ram_style = "block" *)
      reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];

      always @(posedge clk) begin
            if (wea) mem[addra] <= dina;
            if (web) mem[addrb] <= dinb;
            douta <= mem[addra];
            doutb <= mem[addrb];
      end

endmodule
```