6.111 Final Project
Digital Effects Box for Guitar

Matthew Moskwa and Schuyler Senft-Grupp

12/13/2005

I. Introduction

Processing analog signals is often complex, innacurate, and lossy. By transferring these signals into the digital domain prior to processing, we can retain more of the original features of the signal, as well as perform more advanced processing of the signal. The design that follows is one that allows for modification of individual tones of a signal from a musical instrument. This would be next to impossible with analog compoments. While not intirely impossible, the amount of precision hardware involved (i.e. very, very finely tuned capacitors to perform very tight band-pass filtering) would make costs quite prohibitive. However, once the signal is fully digitized, the processing can be done with simple FIR filters and the more complex, but fully realizable FFT.

The goal of this design is for the user to be able to play an electrified instrument into the device, have the tones and all associated harmonics detected, and then, having chosen the particular modifications to be made to the tones, hear the output immediately. Since the AC97 codec used in the design has a 48KHz clock rate, there is a limited time in which to complete all the processing. The FFT is a demanding process, so other steps in the processing pipeline must be designed with this in mind.

In order to detect a wide range of tones, several FFTs with different sampling rates will have to be used. The reason for this is the logarithmic nature of musical tones. The distance between notes in the Western musical system increases by powers of 2. Therefore, the resolution needed to detect low tones is much higher than that needed for higher tones. All of these FFTs can be run synchronously and in parallel through the use of frame buffers on their inputs. In order to avoid aliasing of the frequencies output by the lower-order FFTs, low-pass filters must be used before the buffers.

The output of FFTs will then be fed into a ROM containing specific frequency information about the specific tones. An FFT module does not actually output the exact frequency content of a signal, but instead divides the output into a ranges of frequencies determined by the size of the fft (in sample points) and the sampling rate. We will refer to these ranges as “bins”. Since we know what bins the possible musical tones will fall in, we can wait for these bins, and pass their contents onto the ROM. The ROM will output the exact frequencies of the notes (in floating-point form) and magnitudes determined by the FFT onto a counter and FX module.

Since possible effects depend on the duration of the note, the counter will keep track of this information. The FX module will then do the actually modification of the frequencies, based on the time information from the counter and parameters entered by the user. Some possible effects that will be described in this design are a pitch bender, a brief vibrato, and an appeggio effect. From the FX module, the newly modified frequencies are sent to a synthesizer that creates sine waves of particular magnitudes and frequencies as PCM data that is sent back to the AC97 codec for D/A conversion and output.

The design also features a GUI that displays important information about the currently chosen effect, the input spectrum, and the output spectrum. The user will also make the choice of effect through the GUI, using buttons on the labkit.

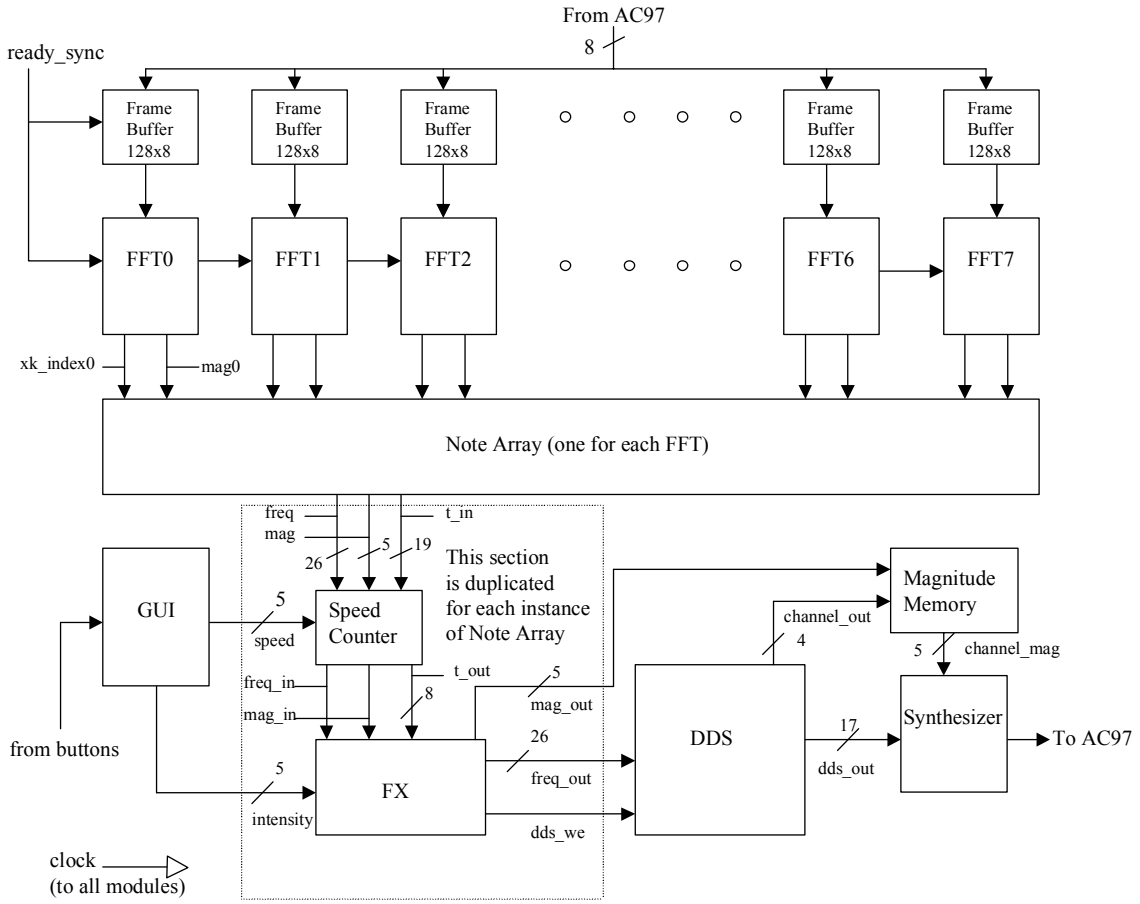

Figure 1: Full Block Diagram

II. Modules

1. Buffers

Because each FFT (there will be 8 in total) requires a different sample rate, we will need 8 separate frame buffers to feed the appropriate data into the associated FFT. A buffer-less solution is possible, but requires that the FFTs sample the incoming data themselves, which is not desirable in continous processing. We would rather have a complete frame presented to the FFT at the beginning of each processing stage, so we have the buffers sample the incoming data instead.

The audio data will not be coming directly from the AC97 codec, which outputs 20-bit two's-complement PCM data. The data the buffers receive comes from the audio.v module, which was written for 6.111 lab 3. It has not been modified for use in this design. Therefore, the buffers will be storing 8-bit PCM data. Since the FFTs are 128-point, the buffers will have to be 128×8 bit in size. Clock dividers running off of the system clock, which is set at 50MHz, will control them. Because the input and output

will be controlled by separate signals running at different clock rates, we will use a two-port memory. The input port will be write-only, and the aforementioned clock divider will set the *write-enable* signal. The output port will accordingly be read-only, and will be controlled by the *fft_start* signal. The addresses for reads and writes are set after each write, when the write pointer is incremented by one, and the read pointer is set to be one greater than the write pointer, so the FFT always reads the oldest data first.

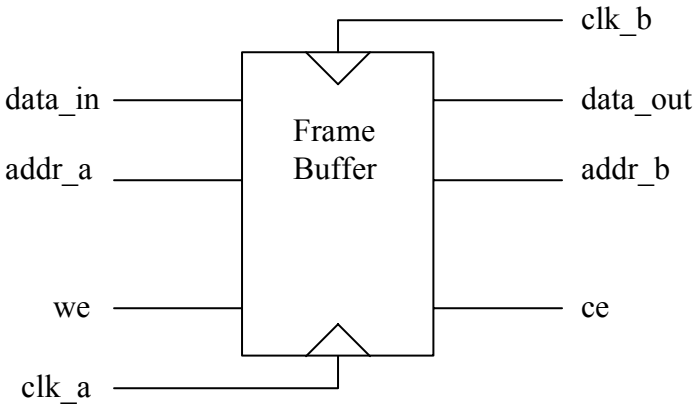

Figure 2: Frame Buffer. The frame buffer has two ports so the address lines can be modified independently at different clock rates.

2. FFT

To achieve the detection of all the notes possible on an instrument like an electric guitar, either an FFT of very large sample size is needed, or several FFTs analyzing different parts of the incoming signal. This design chooses the latter, because the computation time of the FFT scales as $\Theta(n) = N\log(N)$ where N is the sample size. Also, the resolution of the lower order FFTs will need to be much higher than the upper FFTs, because of the logarithmic nature of the tonal sytem (see Fig. X). The resolution of the frequency bins is based on the sample size and the sampling frequency according to the following equation:

$$W_{bin} = \frac{f_{sample}}{N}.$$

With the lowest sampling frequency of 280 Hz, this gives a resolution of approximately 2.19 Hz, which is more than enough for our lowest tone, which is at 73 Hz. The FFTs are not actually sampling the incoming data at this sample rate; instead, the frame buffers sample the data, and the FFT computes the frequencies at the system clock rate. The algorithm does not require that the FFT run at the sample rate, so to improve performace immensely, and to allow for completely continous processing, the FFT simply loads in a frame from the buffer every time the *fft_start* signal, which is tied to the *ready_sync* signal from the AC97 codec, goes high. At this time, the FFT takes in each stored sample

Table 1: Selected notes and their frequencies. Note that tones an octave higher have double the

Note	Frequency (Hz)
E2	82.41
F2	87.31
.	.
.	.
E3	164.81
F3	174.61
.	.
.	.
E4	329.63
F4	349.23
.	.
.	.
E5	659.26
F5	698.46

from the buffer, beginning with the least-recently-written address. After loading all N samples, the FFT enters the computation stage, which takes some time (about half of the available time for the 128-point FFT used here). After this, the data is unloaded, in bit-reversed order due to the particulars of the algorithm, as an index and associated magnitude. The index will be converted to a frequency by next stage, the Note Array ROM, and the magnitude will be passed along to the final stage, the synthesizer. At the completion of the computation stage, the FFT sends out a *done* signal, which is received by the Note Array so it can begin its own loading.

Because of difficulties with the final design, only the lowest-order FFT was actually implemented, but it worked as described, albeit with significant aliasing problems that might have been alleviated with a low-pass filter on the input to the buffer.

3. Note Array

The main function of the note array is to correct for the fact that the FFTs output a magnitude value that covers a range of frequency values. The note array runs once every 48KHz cycle and takes in the FFT output index (*fft_index*) and checks to see if that index corresponds with a frequency range containing a note frequency. (The FFTs are sized to only ever have one note per frequency bin.) If the input index corresponds to a note, the note array checks to see if the corresponding input magnitude (*mag_in*) is greater than the previous cycle's magnitude plus some error called *MAGNITUDE_TOLERANCE*. If the new magnitude is greater it starts a timer, called *t[number]*, to record how long the note has been played.

The important outputs from note array are the magnitude of the current index (*mag*), the frequency (*freq)*, which is one of 12 constant note frequencies bitshifted by some input value (*bitshift)* to correspond with the correct octave, and the number of 48KHz cycles the note has been played (*t_out*). The note array also passes out a new index value (*index_out)* which is a number between 0 and 11. This index is used for

controlling which channel of the DDS should be written too. Also when a valid note frequency gets output, note array brings a write enable (*we*) signal high for one clock cycle. This is used to trigger other modules, especially the DDS.

Table 2: Sampling Rates and Frequency Ranges for each FFT/ Buffer combination.

Sampling Rate (Hz)	Note Range	Frequency Range
280	D2 - C#3	73.42 - 138.59
560	D3 - C#4	146.83 - 277.18
1120	D4 - C#5	293.66 - 554.37
2240	D5 - C#6	587.33 - 1108.73
4480	D6 - C#7	1174.66 - 2217.46
8960	D7 - C#8	2349.32 - 4434.92
17920	D8 -	4698.64 -

4. Speed Counter

The Speed Counter module keeps track of how long each note has been played. Since the design calls for 8 FFTs and 8 Note Arrays, there must be 8 counters as well. The counter consists of three separate RAMs that store information about each note. The first acts as the low-order counter, whose maximum is set by the user through the *speed* input. This input has a possibility of 32 settings, but since we are limited in the math that we can do because of the time taken by the FFT stage. To determine the size of the counter, the following simple math is performed:

$$Count_{\max} = (32 - speed) >> 8 .$$

This can be done very quickly, which is ideal, because there is still quite a lot of processing to do, and a very limited time in which to do it (we've used up more than half the available clock cycles with the first two stages alone).

When the low-order counter reaches its maximum, the higher-order counter stored in another RAM increments by one, and this number is output as *t_out* to the FX module. Also stored in RAM is the *t_in* for each note from the Note Array. The Speed Counter additionally passes through the frequency and magnitude values from the Note Array with no change.

5. FX

The FX module currently has three options for processing of the frequencies. These are Pitch Bender, Vibrato, and Arpeggiator.

Pitch Bender: The pitch bender uses the following formula to determine the output frequency of the module:

$$f_{out} = f_{in} + (f_{in} << 7) \cdot t_{out},$$

where t_{out} is the *t_out* value from the Speed Counter. In this way, the pitch of the note will increase over time at a speed set by the user. If a new *f_in* is detected, however, the *t_out* value is reset, so the bending will start again.

Vibrato: The vibrato implemented in this design is not quite like the vibrato produced by real musician, which is a rapid, unmeasured bend between a note and tones slightly above and below it. What this design does is discretize those bends to create a more "warbling" sound. The user has control over not only the speed, in the same way as described above, but also over the distance between the fundamental note and the upper and lower vibrato pitches. This distance is set with the *intensity* input.

Arpeggio: The arpeggio effect is similar to the vibrato, except the output frequency moves to very specific pitches that spell out a major chord. The effect caues the fundamental note to cycle up through the third above the octave and back done, over 8 cycles. In ths way, the arpeggio effect is a simple FSM. To achieve this, the FX module looks at the *t_out* value from the speed counter, performs division mod 8 on it, and assigns the following values to the output, based on the state:

State 0: f_{in}
State 1: $f_{in} + f_{in}>>2$ (third)
State 2: $f_{in} + f_{in}>>1$ (fifth)
State 3: $f_{in}<<1$ (octave)
State 4: $f_{in}<<1 + f_{in}>>1$ (tenth)
State 5: $f_{in}<<1$ (octave)
State 6: $f_{in} + f_{in}>1$ (fifth)
State 7: $f_{in} + f_{in}>>2$ (third)

The speed of the changes is calculated in the same way as the other two effects.

Since there are 8 different Note Arrays, there will also have to be 8 FX modules, so the entire range of frequencies present in each note from the instrument can be modified in the same way. However, since this implementation only used one FFT, only one FX module was used, but that was not the goal of the design.

6. Direct Digital Synthesis (DDS)

The DDS module generates sine waves of specified frequency. The DDS is generated by the IP Core Generator, which is part of the Xilinx toolkit, using several parameters selected by the user. We chose a 12 channel DDS which means that we can supply it 12 different frequencies and the DDS will output the corresponding sine values by time-sharing the output. To get the resolution (.1 Hz) and range (up to 20KHz) needed for our project, the input frequency must be a 26 bit value. This input value is related to the output frequency by

$$f_{out} = \frac{f_{clk} \cdot f_{in}}{2^B},$$

where B is the number of bits in f_{in}. In our case, f_{clk} is the 49.85 MHz clock divided by 12, the number of channels. To obtain good quality sound output we selected the sine output values to be 17 bits. (The AC97 output is 20 bits.) The synthesis module samples the DDS module once every 48KHz cycle.

In out initial plan we intended to have multiple DDS modules, each one corresponding to an octave of inputs. However we never reached the stage where we needed to generate more than 12 tones, so we only ever created one instance of the DDS module.

7. Synthesizer Module

The synthesizer module takes the sine outputs from the different DDS modules and adds them together to get one value to output to the AC97. (Since in our project we only reached the point of implementing one DDS, the synthesizer module only took one sine input, *sine0*.) The synthesizer module is triggered by the start signal *start_synth* which we connected to the synchronous AC97 signal *ready_sync*. On receiving the start signal, the synthesizer module reads in the next 12 values presented at the output of the DDS. It also receives magnitude values corresponding to each sine value, and scales the sine value by the magnitude. (Currently in our code, for testing purposes, this is a binary scaling – either the note is being played or not.) The output signal, *sound_out*, goes to AC97.

8. Magnitude Memory Module

Because the note magnitudes are output from the note array module before they need to be used by the synthesizer module, we wrote a small module to store the magnitudes. The magnitudes are read in synchronously from the note array and stored in a twelve-row array of registers called *mag*. The module also constantly reads in which channel is being read out from the DDS and asynchronously outputs the corresponding magnitude value, *mag_out*.

9. Graphical User Interface (GUI) Module

The GUI controls the output on the screen, gives the user an easy interface to control effects, and outputs the user's preferences to the associated modules. The screen consists of 6 'slider bars' to control the three effects (bender, vibrato, and arpeggio) and delay. Each slider bar and slider is controlled by a sprite module, blob, which generates a rectangle in a given location with a designated height, width, and color. The other graphics this module is responsible for generating are all the text on the screen (done with a character module and font rom), the outside border, and inner lines dividing the screen into the control section and the spectrum analyzer section.

The user interfaces with our project through the buttons on the labkit. The left/right buttons select the slider bar the user wants to control and the up/down buttons increase or decrease the level of the selected slider bar. To change which effect is currently running the user moves to the corresponding slider bar and presses a select button.

The GUI has several important outputs. The first is *pixel_out*, a 3 bit color value that gets sent to the module. The signal *fx* goes to the FX module and selects which effect is currently being selected. Depending on the value of *fx*, the corresponding *speed* and *intensity* value are also output to the FX module. To control the rate and intensity of delay, a *delay_speed* and *delay_int* are constantly output. Unfortunately we did not have time to implement the delay feature.

Graph Module

This module takes in index and magnitude data and plots a spectrum bar graph. The module uses an array of registers to store the data. The output graph is plotted using a logic statement that checks to see if the current pixel location falls within the data of magnitudes. This is made particularly easy by the use of the array.

10. Wet Graph Data Storage Module

To plot the data coming out of the effect module we need some way of determining which index of the bar graph the magnitude should be plotted. This module looks at the actual frequency value coming in from the FX module and checks to see which index, from 0 to 11, this corresponds to. The output *index_out* is determined through a logic statement which determines each bits value. This data is then fed to the graph module. In hindsight, one limitation of this module is that the FX module can increase a frequency to above the octave it initially begins in. This means that if a note gets changed to above a Db it will always just get recorded of having an index of 11, or the highest possible value.

III. Discussion

Unfortunately we were unable to fully complete our project as planned. The major obstacle we ran into was with the FFTs. As earlier discussed, we needed multiple FFTs each sampling at increasing frequencies from 260Hz to 48KHz. However we failed to recognize the need to filter the signals going to the FFTs to make sure that there were no frequencies above the sampling frequency present in the signal. This caused severe aliasing problems which for the most part rendered the output of the FFT useless. There was just too much noise to make the output enjoyable to listen to. We attempted to

implement some quick filters but were unable to design anything that worked well enough in the time we had.

Our project was trying to do something very difficult: recreate a guitar input with its full harmonic spectrum at the output. Unfortunately because we only ever got one FFT/DDS working at a time and because we had to only use the top few bits of the magnitude to remove the aliasing noise, this was impossible. Instead we ended up making, for all intents and purposes, a guitar input to midi output. Had this been our initial goal, to just pick up the few loudest notes being played and ignore harmonics, we would have had a very different and simpler design.

IV. Appendix

```
//-----------------------------------------------------------------------
------
//                6.111 Final Project
//                        by
//    Schuyler Senft-Grupp and Matthew Moskwa
//
//                Completed 12/12/05
//
//  Our project is a digital effects box that can alter individual
//  frequencies of an incoming audio source

//-----------------------------------------------------------------------
-------
// top level module

module main_file(
               beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
ac97_synch,
             ac97_bit_clock,

             vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
             vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
             vga_out_vsync,

             tv_out_ycrcb, tv_out_reset_b, tv_out_clock,
tv_out_i2c_clock,
             tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
             tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

             tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
             tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
             tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
             tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

             ram0_data, ram0_address, ram0_adv_ld, ram0_clk,
ram0_cen_b,
             ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

             ram1_data, ram1_address, ram1_adv_ld, ram1_clk,
ram1_cen_b,
             ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

             clock_feedback_out, clock_feedback_in,

             flash_data, flash_address, flash_ce_b, flash_oe_b,
flash_we_b,
             flash_reset_b, flash_sts, flash_byte_b,

             rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

             mouse_clock, mouse_data, keyboard_clock, keyboard_data,
```

```
              clock_27mhz, clock1, clock2,

              disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
              disp_reset_b, disp_data_in,

              button0, button1, button2, button3, button_enter,
button_right,
              button_left, button_down, button_up,

              switch,

              led,

              user1, user2, user3, user4,

              daughtercard,

              systemace_data, systemace_address, systemace_ce_b,
              systemace_we_b, systemace_oe_b, systemace_irq,
systemace_mpbrdy,

              analyzer1_data, analyzer1_clock,
              analyzer2_data, analyzer2_clock,
              analyzer3_data, analyzer3_clock,
              analyzer4_data, analyzer4_clock
                   );

   output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
   input  ac97_bit_clock, ac97_sdata_in;

   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
        vga_out_hsync, vga_out_vsync;

   output [9:0] tv_out_ycrcb;
   output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
tv_out_i2c_data,
        tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b,
tv_out_blank_b,
        tv_out_subcar_reset;

   input  [19:0] tv_in_ycrcb;
   input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
tv_in_aef,
        tv_in_hff, tv_in_aff;
   output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock,
tv_in_iso,
        tv_in_reset_b, tv_in_clock;
   inout  tv_in_i2c_data;

   inout  [35:0] ram0_data;
   output [18:0] ram0_address;
   output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b,
ram0_we_b;
   output [3:0] ram0_bwe_b;

   inout  [35:0] ram1_data;
```

```
   output [18:0] ram1_address;
   output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b,
ram1_we_b;
   output [3:0] ram1_bwe_b;

   input  clock_feedback_in;
   output clock_feedback_out;

   inout  [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b,
flash_byte_b;
   input  flash_sts;

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   inout  mouse_clock, mouse_data;
   //input  mouse_clock, mouse_data;
   input  keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

   output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
   input  disp_data_in;
   output  disp_data_out;

   input  button0, button1, button2, button3, button_enter,
button_right,
        button_left, button_down, button_up;
   input  [7:0] switch;
   output [7:0] led;

   inout [31:0] user1, user2, user3, user4;

   inout [43:0] daughtercard;

   inout  [15:0] systemace_data;
   output [6:0]  systemace_address;
   output systemace_ce_b, systemace_we_b, systemace_oe_b;
   input  systemace_irq, systemace_mpbrdy;

   output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
             analyzer4_data;
   output analyzer1_clock, analyzer2_clock, analyzer3_clock,
analyzer4_clock;


//////////////////////////////////////////////////////////////////////
/////
   //
   // I/O Assignments
   //

//////////////////////////////////////////////////////////////////////
/////
```

```
   // Audio Input and Output
   assign beep= 1'b0;
//   assign audio_reset_b = 1'b0;
//   assign ac97_synch = 1'b0;
//   assign ac97_sdata_out = 1'b0;

   // ac97_sdata_in is an input

   // VGA Output
   //assign vga_out_red = 10'h0;
   //assign vga_out_green = 10'h0;
   //assign vga_out_blue = 10'h0;
   //assign vga_out_sync_b = 1'b1;
   //assign vga_out_blank_b = 1'b1;
   //assign vga_out_pixel_clock = 1'b0;
   //assign vga_out_hsync = 1'b0;
   //assign vga_out_vsync = 1'b0;

   // Video Output
   assign tv_out_ycrcb = 10'h0;
   assign tv_out_reset_b = 1'b0;
   assign tv_out_clock = 1'b0;
   assign tv_out_i2c_clock = 1'b0;
   assign tv_out_i2c_data = 1'b0;
   assign tv_out_pal_ntsc = 1'b0;
   assign tv_out_hsync_b = 1'b1;
   assign tv_out_vsync_b = 1'b1;
   assign tv_out_blank_b = 1'b1;
   assign tv_out_subcar_reset = 1'b0;

   // Video Input
   assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b0;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b0;
   assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = 1'b0;
   assign tv_in_i2c_data = 1'bZ;
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs
   //assign ram0_data = 36'hZ;
   //assign ram0_address = 19'h0;
   assign ram0_adv_ld = 1'b0;
   //assign ram0_clk = 1'b0;
   //assign ram0_cen_b = 1'b1;
   assign ram0_ce_b = 1'b1;
   assign ram0_oe_b = 1'b1;
   //assign ram0_we_b = 1'b1;
   assign ram0_bwe_b = 4'hF;
   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;
   assign ram1_cen_b = 1'b1;
```

```
   assign ram1_ce_b = 1'b1;
   assign ram1_oe_b = 1'b1;
   assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'hF;
   assign clock_feedback_out = 1'b0;

   // clock_feedback_in is an input

   // Flash ROM
   assign flash_data = 16'hZ;
   assign flash_address = 24'h0;
   assign flash_ce_b = 1'b1;
   assign flash_oe_b = 1'b1;
   assign flash_we_b = 1'b1;
   assign flash_reset_b = 1'b0;
   assign flash_byte_b = 1'b1;
   // flash_sts is an input

   // RS-232 Interface
   assign rs232_txd = 1'b1;
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are
inputs

   // LED Displays
/*
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;
*/
   // disp_data_in is an input

   // Buttons, Switches, and Individual LEDs
   //lab3 assign led = 8'hFF;
   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

   // User I/Os
   assign user1 = 32'hZ;
   assign user2 = 32'hZ;
   assign user3 = 32'hZ;
   assign user4 = 32'hZ;

   // Logic Analyzer
   assign analyzer1_data = 16'h0;
   assign analyzer1_clock = clock_50MHz;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = clock_50MHz;
   assign analyzer3_data = 16'h0;
   assign analyzer3_clock = clock_50MHz;
   assign analyzer4_data = 16'h0;
```

```
   assign analyzer4_clock = clock_50MHz;

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;
   // systemace_irq and systemace_mpbrdy are inputs

   ////////////////////////////////////////
   // master clock for this system is the clock_27mhz

   // use FPGA's digital clock manager to produce a 50 Mhz clock from
27 Mhz
   // actual frequency: 49.85 MHz
   wire clock_50mhz_unbuf,clock_50MHz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_50mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 13
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_50MHz),.I(clock_50mhz_unbuf));

   wire power_on_reset;
   SRL16 reset_sr (.D(1'b0), .CLK(clock_50MHz), .Q(power_on_reset),
               .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

   wire user_reset;
   debounce dbreset(1'b0,clock_50MHz,~button_enter,user_reset);

   wire reset = power_on_reset | user_reset;


   ////////////////////////////////////////
   // Final Project - Guitar Effects

   //wire declarations

   wire audio_ready;  //un_synced ready signal when ac97 receives/send
data

   wire [19:0] from_ac97_data, to_ac97_data;  //data to and from AC97

   //note frequencies - higher octaves are multiples of 2
   parameter D = 12'd1186;
   parameter Eb =  12'd1257;
   parameter E = 12'd1331;
   parameter F = 12'd1410;
   parameter Gb = 12'd1494;
   parameter G = 12'd1583;
   parameter Ab = 12'd1677;
   parameter A = 12'd1777;
```

```
   parameter Bb = 12'd1883;
   parameter B = 12'd1995;
   parameter C = 12'd2113;
   parameter Db = 12'd2239;

   //fft wires that go to all fft modules

   wire [7:0]  xn_im = 0;
   wire        fwd_inv = 1;
   wire        fwd_inv_we;
   wire [7:0]  scale_sch = 8'hAA;
   wire        scale_sch_we;


   wire signed [19:0] from_synth1;  //wire that connects the
synthesizer to the
                                    //AC97 output

   wire ready_sync; //synchronous audio_ready

   wire up, down, left, right, dbutton3;

   wire [1:0] fx;
   wire [4:0] intensity, speed, delay_speed, delay_int;

   //sets read from buffer
   reg [6:0] ready_count;
   reg ready_cycle;
   wire new_aframe;

   always @ (posedge clock_50MHz) begin
      ready_cycle <= (ready_count < 127);
      ready_count <= (new_aframe) ? 0 : ((ready_count == 127) ?
ready_count :           ready_count + 1);
   end

   //sync audio_ready

   reg [2:0] audio_ready_sync;
   always @ (posedge clock_50MHz) audio_ready_sync <=
      {audio_ready_sync[1:0],audio_ready};
   assign ready_sync = audio_ready_sync[2] & audio_ready_sync[1];

   //generate a test tone for debugging
   wire [19:0] tone;
   tone750hz xxx(clock_50MHz, ready_sync, tone);

   reg [1:0]   initclk;

   //set the scaling for the FFTs

   reg [7:0]   old_scale_sch;
   always @(posedge clock_50MHz) old_scale_sch <= scale_sch;
   assign scale_sch_we = initclk[0] | ~(scale_sch==old_scale_sch);

   always @(posedge clock_50MHz)
     initclk <= reset ? 0 : (((initclk<3))  ? initclk + 1 : initclk);
```

```
   assign fwd_inv_we = (initclk==2'd1);

   reg old_ready;
   always @ (posedge clock_50MHz) old_ready <= ready_sync;
   assign new_aframe = ready_sync & ~old_ready;


   //module to interface with the AC97
   //used signals are the to/from ac 97 data, and the audio_ready
   audio myaudio(clock_27mhz, power_on_reset, from_ac97_data,
to_ac97_data,
            audio_ready, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
            ac97_synch, ac97_bit_clock);
   defparam myaudio.VOLUME = 4'd8;


   //set 0
   wire [3:0] dds_channel_out0;  //relays which channel data is coming
from on                                  //DDS

   wire signed [16:0] sine0;  //data output from DDS
   wire dds_we0; //enable a write of frequency data into selected DDS
channel
   wire dds_rdy0; //signal that DDS output data is valid


   //see call to FFT module for explanations of these wires
   wire [7:0]  xk_re0,xk_im0;
   wire [6:0]  xn_index0, xk_index0;
   wire        fft_rfd0;
   wire        fft_busy0;
   wire        fft_dv0;
   wire        fft_edone0;
   wire        fft_done0;

   wire [3:0] tolerance0 = 0; //used to adjust when a new magnitude
occurred
                              //we never got around to using it


   // buffer clocks
   // clock 0: 280 hz
   reg [17:0] dcount0;
   wire clk_dsp0 = (dcount0==178035);

   always @(posedge clock_50MHz) dcount0 <= (clk_dsp0) ? 0 : dcount0+1;


   //////////////////  Buffers ////////////////////
   // portA is input (clk_dsp), portB is output(clk triggered from
ready_sync)

   reg [6:0] addra0;
   wire we0 = clk_dsp0;
   always @ (posedge clock_50MHz) begin
      if (clk_dsp0)
            addra0 <= (addra0 == 127) ? 0: addra0 + 1;
```

```
      end

   reg [6:0] addrb0;

   always @ (posedge clock_50MHz) begin
      if (ready_cycle)
            addrb0 <= (addrb0 == 127) ? 0 : addrb0 + 1;
     else addrb0 <= addra0 + 1;
   end

   reg [7:0]   xn_re0;
   wire [7:0] pdat0;
   always @ (posedge clock_50MHz)
      xn_re0 <= pdat0;

   wire [7:0] to_buffer0 = from_ac97_data[19:12];

   //-----------------------------------------------------------------
   //module calls
   //-----------------------------------------------------------------

   buffer2 b0(addra0, addrb0, clock_50MHz, clock_50MHz, to_buffer0,
pdat0, we0);      //make sure bits are right length

   fft_128pt_8bit fft0(xn_re0,      // real data, input
             xn_im,     // imaginary data, input
             new_aframe,      // start data loading & conversion, in
             fwd_inv,         // forward or inverse, input
             fwd_inv_we,      // write enable for fwd_inv, input
             scale_sch, // scaling schedule, input
             scale_sch_we,    // write enable for scale_sch, input
             clock_50MHz,           // system synchronous clock
             xk_re0,          // output real data
             xk_im0,          // output imaginary data
             xn_index0, // index of input data (output)
             xk_index0, // index of output data (output)
             fft_rfd0,        // ready for data, out
             fft_busy0, // high while core is computing fft
             fft_dv0,         // data valid, output
             fft_edone0,      // early done strobe, output
             fft_done0);      // fft complete strobe, output;

   wire [8:0] sum_mag0;
   reg [7:0] pos_re0;
   reg [7:0] pos_im0;


   //convert the imaginary and real values from the FFT to positive
numbers
   //and add them
   //we use this approximation of the magnitude for debugging and
because we
   //drop the lower order bits
   always @(xk_re0)
   begin
      if (xk_re0[7])
            pos_re0 = ~(xk_re0-1);
```

```
      else
            pos_re0 = xk_re0;
   end

   always@(xk_im0)
   begin
      if (xk_im0[7])
            pos_im0 = ~(xk_im0-1);
      else
            pos_im0 = xk_im0;
   end

   assign sum_mag0 = pos_re0 + pos_im0;

   //wire assignments between modules

   wire [4:0]na_mag_out0, na_mag_out1, na_mag_out2;
   wire [19:0]na_t_out0, na_t_out1, na_t_out2;
   wire [25:0] na_freq_out0, na_freq_out1, na_freq_out2;
   wire [3:0] na_index_out0, na_index_out1, na_index_out2;
   wire na_done0, na_done1, na_done2;
   wire na_we0, na_we1, na_we2;
   wire [25:0] fx_freq_out0;
   wire [4:0] fx_mag_out0;
   wire [3:0] fx_index_out0;
   wire [4:0] mem_mag_out0;


   wire synth_index_out0;

   wire [25:0] fx_freq0;
   wire [4:0] fx_mag0;
   wire [3:0] fx_index0;

   wire [7:0] ts_out0;

   note_array na0(clock_50MHz, reset, xk_index0, sum_mag0[8:4], 4'b1,
      tolerance0, fft_dv0,
      D, Eb, E, F, Gb, G, Ab, A, Bb, B, C, Db, na_mag_out0,
      na_t_out0, na_freq_out0, na_index_out0, na_done0, na_we0,
      dbutton3, switch);

   speed_counter speed_c0(clock_50MHz, na_t_out0, na_index_out0,
na_mag_out0,            na_freq_out0,
            speed, ts_out0, fx_index0, fx_mag0, fx_freq0);

   fx fx0(clock_50MHz, fx_freq0, fx_mag0, fx_index0, ts_out0,
fx_freq_out0,
            fx_mag_out0, fx_index_out0, na_we0, dds_we0, fx, speed,
                  intensity);

   mag_memory magnitudes0(clock_50MHz, fx_mag_out0, fx_index_out0,
mem_mag_out0,           dds_channel_out0);

   dds_point1 dds0(fx_freq_out0, dds_we0, {1'b0, fx_index_out0},
clock_50MHz,            1'b1, reset, dds_channel_out0, dds_rdy0,
sine0);
```

```
   wire [4:0] wet_data_out0;
   wire [6:0] wet_index_out0;

   wet_graph_data wet0(clock_50MHz, fx_freq_out0, 4'b0, fx_mag_out0,
            D, Eb, E, F, Gb, G, Ab, A, Bb, B, C, Db,  wet_data_out0,
                  wet_index_out0);

   synthesizer synth(clock_50MHz, mem_mag_out0, sine0, ready_sync,
from_synth1,            reset, from_zbt, delay_int);

   //for debugging allow for test tone out
   wire dbutton0;
   assign to_ac97_data = (dbutton0) ? tone : from_synth1;


   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;


  // feed XVGA signals to modules
   wire [2:0] pixel1, pixel2, pixel3, bpixel;
   wire phsync,pvsync,pblank;

   //display code

   // generate basic XVGA video signals
   xvga xvga1(clock_50MHz, hcount, vcount, hsync, vsync, blank);

   graph graph_dry(clock_50MHz,reset, na_mag_out0,
                  na_index_out0,
                  hcount,vcount,hsync,vsync,blank, pixel1);

   defparam wet.LEFT = 10'd408;

   graph wet(clock_50MHz,reset,
                  wet_data_out0,
                  wet_index_out0,
                  hcount,vcount,hsync,vsync,blank, pixel2);

   debounce dup(1'b0,clock_50MHz,~button_up,up);
   debounce ddown(1'b0,clock_50MHz,~button_down,down);
   debounce dleft(1'b0,clock_50MHz,~button_left,left);
   debounce dright(1'b0,clock_50MHz,~button_right,right);
   debounce dbutton3_debounce(1'b0,clock_50MHz,~button3,dbutton3);
   debounce dbutton0_debounce(1'b0,clock_50MHz,~button0,dbutton0);


   gui main_gui(clock_50MHz,reset, up, down, left, right, dbutton3,

      hcount,vcount,hsync,vsync,blank,phsync,pvsync,pblank,pixel3,
            fx, intensity, speed, delay_speed, delay_int);

   reg [2:0] rgb;
   reg b,hs,vs;
```

```
   always @(posedge clock_50MHz) begin
       hs <= phsync;
       vs <= pvsync;
       b <= pblank;
       rgb <= (pixel1 | pixel2 | pixel3 | bpixel);
   end

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_50MHz.
   assign vga_out_red = {8{rgb[2]}};
   assign vga_out_green = {8{rgb[1]}};
   assign vga_out_blue = {8{rgb[0]}};
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~clock_50MHz;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

endmodule

//-----------------------------------------------------------------------
//Note Array Module
//For every FFT module there is a corresponding Note Array Module.
//The Note Array Module pitch corrects the frequency values coming
//from the FFT and outputs the standard frequency for a given note.
//It also records how long a note has been played and outputs that
information.

//Currently, for debugging because we could not get the guitar input
working
//correctly the magnitudes are set to switches

module note_array(clk, reset, fft_index, mag_in, bitshift,
MAG_TOLERANCE, fft_dv,
            D, Eb, E, F, Gb, G, Ab, A, Bb, B, C, Db, mag_out,
            t_out, freq_out, index_out, na_done, we, dbutton3, switch);

    input dbutton3;

   input clk, reset;
   input [6:0] fft_index; //incoming index of output from FFT
   input [4:0] mag_in;  //incoming magnitude from FFT for given index
   input [3:0] bitshift; //amount to shift note frequencies to get
correct octave
   input [3:0] MAG_TOLERANCE; //tolerance in magnitude values before a
time reset
   input fft_dv; // data valid from fft
   input [7:0] switch;
   input [11:0] D, Eb, E, F, Gb, G, Ab, A, Bb, B, C, Db;  //note
frequencies

   output [4:0] mag_out;  //output note magnitude
   output [19:0] t_out; //output length of time that note has been
played
   output [25:0] freq_out; //output nbte frequency
   output [3:0] index_out;  //index for DDS channel
   output na_done;//goes high when cycle of new notes is done
```

```
output we; //goes high when data valid

reg na_done;
reg mag_out;
reg t_out;
reg freq_out;
reg freq_index;
reg index_out;

reg [19:0] t0;
reg [19:0] t1;
reg [19:0] t2;
reg [19:0] t3;
reg [19:0] t4;
reg [19:0] t5;
reg [19:0] t6;
reg [19:0] t7;
reg [19:0] t8;
reg [19:0] t9;
reg [19:0] t10;
reg [19:0] t11;

reg [4:0] old_mag0;
reg [4:0] old_mag1;
reg [4:0] old_mag2;
reg [4:0] old_mag3;
reg [4:0] old_mag4;
reg [4:0] old_mag5;
reg [4:0] old_mag6;
reg [4:0] old_mag7;
reg [4:0] old_mag8;
reg [4:0] old_mag9;
reg [4:0] old_mag10;
reg [4:0] old_mag11;

reg [4:0] mag0;
reg [4:0] mag1;
reg [4:0] mag2;
reg [4:0] mag3;
reg [4:0] mag4;
reg [4:0] mag5;
reg [4:0] mag6;
reg [4:0] mag7;
reg [4:0] mag8;
reg [4:0] mag9;

reg [4:0] mag10;
reg [4:0] mag11;


reg old_ready;
reg old_channel_in;
reg we;


   always @(posedge clk)
   begin
```

```
		if (reset)
		begin
			t1 <= 0;
			t2 <= 0;
			t3 <= 0;
			t4 <= 0;
			t5 <= 0;
			t6 <= 0;
			t7 <= 0;
			t8 <= 0;
			t9 <= 0;
			t10 <= 0;
			t11 <= 0;
			t0 <= 0;
			old_mag0 <= 0;
			old_mag1 <= 0;
			old_mag2 <= 0;
			old_mag3 <= 0;
			old_mag4 <= 0;
			old_mag5 <= 0;
			old_mag6 <= 0;
			old_mag7 <= 0;
			old_mag8 <= 0;
			old_mag9 <= 0;

			old_mag10 <= 0;
			old_mag11 <= 0;

			mag0 <= 0;
			mag1 <= 0;
			mag2 <= 0;
			mag3 <= 0;
			mag4 <= 0;
			mag5 <= 0;
			mag6 <= 0;
			mag7 <= 0;
			mag8 <= 0;
			mag9 <= 0;
			mag10 <= 0;
			mag11 <= 0;

			we <= 0;
		end
		else
		begin
			if (fft_dv)
			begin
			case (fft_index)
			34: mag0<= mag_in;
			33: begin
					we<=1;
					index_out <= 0;
					freq_out <= D<<bitshift;
					old_mag0 <= {5{switch[6]}};
					mag_out <= {5{switch[6]}};//
(mag_in+mag0);
					if  (switch[6] > old_mag0[0])
```

```
                              begin
                                    t0 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t0 <= t0 + 1;
                                    t_out <= t0 + 1;
                              end
                        end
                  36: mag1<= mag_in;
                  35: begin
                              we<=1;
                              index_out <= 1;
                              freq_out <= Eb<<bitshift;
                              old_mag1 <= (mag_in+mag1);
                              mag_out <=  0;//(mag_in+mag1);
                              if  ((mag_in + mag1) > (old_mag1 +
MAG_TOLERANCE))
                              begin
                                    t1 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t1 <= t1 + 1;

                                    t_out <= t1 + 1;
                              end
                        end

                  38: mag2<= 0;//mag_in;
                  37: begin
                              we<=1;
                              index_out <= 2;
                              freq_out <= E<<bitshift;
                              old_mag2 <= {5{switch[5]}};
                              mag_out <=
{5{switch[5]}};//(mag_in+mag2);
                              if  (switch[5]> old_mag2[0])
                              begin
                                    t2 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t2 <= t2 + 1;
                                    t_out <= t2 + 1;
                              end
                        end

                  39: mag3<= mag_in;
                  40: begin
                              we<=1;
                              index_out <= 3;

                              freq_out <= F<<bitshift;
```

```
                              old_mag3 <= (mag_in+mag3);
                              mag_out <=  0;//(mag_in+mag3);
                              if  ((mag_in + mag3) > (old_mag3 +
MAG_TOLERANCE))
                              begin
                                    t3 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t3 <= t3 + 1;
                                    t_out <= t3 + 1;
                              end
                        end

                  42:   begin
                              freq_out <= Gb<<bitshift;
                              we <= 1;
                              old_mag4 <= mag_in;
                              index_out <= 4;
                              mag_out <= 0;// 5'd16;//mag4;
                              if  (dbutton3) //(mag_in > (old_mag4 +
MAG_TOLERANCE))
                              begin
                                    t4 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t4 <= t4 + 1;
                                    t_out <= t4+1;
                              end
                        end

                  44: mag5<= mag_in;
                  45: begin
                              we<=1;
                              index_out <= 5;
                              freq_out <= G<<bitshift;
                              old_mag5 <= {5{switch[4]}};
                              mag_out <=
{5{switch[4]}};//(mag_in+mag5);
                              if  (switch[4] > old_mag5[0])
                              begin
                                    t5 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t5 <= t5 + 1;
                                    t_out <= t5 + 1;
                              end
                        end

                  48: mag6<= mag_in;
                  47: begin
                              we<=1;
```

```
                              index_out <= 6;
                              freq_out <= Ab<<bitshift;
                              old_mag6 <= (mag_in+mag6);
                              mag_out <=  0;//(mag_in+mag6);
                              if  ((mag_in + mag6) > (old_mag6 +
MAG_TOLERANCE))
                              begin
                                    t6 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t6 <= t6 + 1;
                                    t_out <= t6 + 1;
                              end
                        end

                  50: mag7<= mag_in;
                  51: begin
                              we<=1;
                              index_out <= 7;
                              freq_out <= A<<bitshift;
                              old_mag7 <= (mag_in+mag7);
                              mag_out <=  0;//(mag_in+mag7);
                              if  ((mag_in + mag7) > (old_mag7 +
MAG_TOLERANCE))
                              begin
                                    t7 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t7 <= t7 + 1;
                                    t_out <= t7 + 1;
                              end
                        end

                  54: mag8<= mag_in;
                  53: begin
                              we<=1;
                              index_out <= 8;
                              freq_out <= Bb<<bitshift;
                              old_mag8 <= (mag_in+mag8);
                              mag_out <=  0;//(mag_in+mag8);
                              if  ((mag_in + mag8) > (old_mag8 +
MAG_TOLERANCE))
                              begin
                                    t8 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t8 <= t8 + 1;
                                    t_out <= t8 + 1;
                              end
                        end
```

```
                  56: mag9<= mag_in;
                  57: begin
                              we<=1;
                              index_out <= 9;
                              freq_out <= B<<bitshift;
                              old_mag9 <= (mag_in+mag9);
                              mag_out <= 0;// (mag_in+mag9);
                              if  ((mag_in + mag9) > (old_mag9 +
MAG_TOLERANCE))
                              begin
                                    t9 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t9 <= t9 + 1;
                                    t_out <= t9 + 1;
                              end
                        end

                  60: mag10<= mag_in;
                  59: begin
                              we<=1;
                              index_out <= 10;
                              freq_out <= C<<bitshift;
                              old_mag10 <= {5{switch[3]}};
                              mag_out <= {5{switch[3]}};//
(mag_in+mag10);
                              if  (switch[3] > old_mag10[0] )
                              begin
                                    t10 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t10 <= t10 + 1;
                                    t_out <= t10 + 1;
                              end
                        end

                  63:         begin
                              freq_out <= Db<<bitshift;
                              we <= 1;
                              old_mag11 <= mag_in;
                              index_out <= 11;
                              mag_out <=  0;//mag_in;
                              if  (mag_in > (old_mag11 +
MAG_TOLERANCE))
                              begin
                                    t11 <= 0;
                                    t_out <= 0;
                              end
                              else
                              begin
                                    t11 <= t11 + 1;
                                    t_out <= t11+1;
                              end
```

```
                              na_done <= 1; // check to put this in the
right case - YES- correct
                        end

                  default: we <= 0;
                  endcase
            end
            end
      end

endmodule


//-------------------------------------------------------------
//Synthesizer (adder) Module
//The Synthesizer module takes values in from DDS modules, scales
//them by given magnitudes, and outputs a signal to the ac97.

module synthesizer (clk, mag0, sine0, start_synth, sound_out, reset);

      input clk, start_synth, reset;
      input [16:0] sine0;
      input [4:0] mag0;

      output [19:0] sound_out;

      reg signed [19:0] sound;
      reg old_done;
      reg [3:0] count;

      wire wire_rdy = (~old_done && start_synth);


      always @(posedge clk)
      begin
            if (reset)
            begin
                  sound <= 0;
                  count<=11;
                  old_done <= 0;
            end
            else
            begin
                  old_done <= start_synth;
                  if (wire_rdy)
                  begin
                        count <= 0;
                        sound <= (sine0[16:2] & {15{mag0[4] | mag0[3] |
mag0[2]}});
                  end
                  else
                  begin
                        if(count < 15)
                        begin
                              count <= count + 1;
                              sound <= sound + (sine0[16:2] &
{15{mag0[4] | mag0[3] | mag0[2]}});
```

```
                        end
                    end

                end
        end

        assign sound_out = sound;//(count == 15) ? sound: sound_out;

endmodule

//Magnitude Memory Module
//This module stores 12 magnitudes and constantly outputs the magnitude
//that corresponds to the current DDS output

module mag_memory (clk, mag_in, index_in, mag_out, index_out);

   input clk;
   input [4:0] mag_in;
   input [3:0] index_in;
   input [3:0] index_out;
   output [4:0] mag_out;

   reg [4:0] mag [11:0];


   always @(posedge clk)
      mag [index_in] = mag_in;

   assign mag_out = mag [index_out];

endmodule


//Graph Module
//This module takes in magnitudes and their corresponding index and
generates a //bar graph on the monitor

module graph (vclock, reset, data0, data_index0,
                    hcount, vcount, hsync, vsync, blank, pixel);

   parameter LEFT = 8;
   parameter BOTTOM = 550;
   parameter COLOR = 3'b111;
   parameter DATA_LENGTH = 12;

   input vclock;  // 50MHz clock
   input reset;         // 1 to initialize module
   input [4:0] data0;   // data to graph

   input [3:0] data_index0; //index of data
   input [10:0] hcount; // horizontal index of current pixel (0..799)
   input [9:0] vcount; // vertical index of current pixel (0..599)
   input hsync;         // XVGA horizontal sync signal (active low)
   input vsync;         // XVGA vertical sync signal (active low)
   input blank;         // XVGA blanking (1 means output black pixel)

   output [2:0] pixel;  // graph's pixel
```

```
   reg [4:0] data_buffer [DATA_LENGTH-1:0];
   reg [2:0] pixel;
   reg [7:0] count;

   always @(posedge vclock)
   begin
      if (reset)
            count <= 95;
      else
      begin
            pixel <= ((hcount>= LEFT) && (data_buffer[(hcount-
LEFT)>>3]<<2 > (BOTTOM - vcount)) &&
                  (vcount < BOTTOM) && ((hcount - LEFT) <
DATA_LENGTH<<3)) ? 3'b111: 3'b000;

            data_buffer [data_index0] <= data0;
      end
   end

endmodule

//Graphical User Interface Module
//This module displays the effects options on the screen and allows the
user to
//move between effects and adjust the effects characteristics with the
labkit //buttons

module gui(vclock,reset, up, down, left, right, enter,
            hcount,vcount,hsync,vsync,blank,
            phsync,pvsync,pblank,pixel_out, fx, intensity, speed,
            delay_speed, delay_int);

   input vclock;  // 65MHz clock
   input reset;         // 1 to initialize module
   input up, down, left, right, enter;  //labkit button controls
   input [10:0] hcount; // horizontal index of current pixel (0..799)
   input [9:0] vcount; // vertical index of current pixel (0..599)
   input hsync;         // XVGA horizontal sync signal (active low)
   input vsync;         // XVGA vertical sync signal (active low)
   input blank;         // XVGA blanking (1 means output black pixel)

   output phsync; // graph's horizontal sync
   output pvsync; // graph's vertical sync
   output pblank; // graph's blanking
   output [2:0] pixel_out;    // graph's pixel

   //output to fx module
   output [1:0] fx;
   output [4:0] speed;
   output [4:0] intensity;

   //output to delay module
   output [4:0] delay_speed, delay_int;

   reg [4:0] speed;
   reg [2:0] pixel_edge, pixel_hor, pixel_ver;
```

```
   //fx are:      pitch bending needs speed slider bar
   //       vibratto needs speed and intensity
   //       arpegiator needs speed
   //       delay needs timer and intensity


   wire [2:0] pixel0,pixel1, pixel2, pixel3, pixel4, pixel5,
      pixel6, pixel7, pixel8, pixel9, pixel10, pixel11,
      pixel12, pixel13;

   reg [2:0] color_bars [6:0];
   reg [2:0] color [6:0];
   reg [1:0] fx;//keeps track of which effect is 'on'
            //000 - bender
            //001 - vibratto
            //010 - arpegiator

   reg [2:0] control; //keeps track of which effect contol is being
adjusted
   reg [4:0] effect_intensity [6:0];  //5 bit intensity value, one for
each effect control

   reg old_left, old_right, old_up, old_down, old_enter;


   parameter SLIDER_HEIGHT = 10'd5;
   parameter SLIDER_WIDTH = 11'd16;
   parameter BAR_WIDTH = 11'd12;
   parameter BAR_HEIGHT = 10'd128 + SLIDER_HEIGHT;
   parameter BAR_TOP = 10'd100;

   wire [2:0] pixel_on, on1, on2, on3;
   reg [9:0] x;

   blob effect_on(x,10'd279,11'd32,10'd3,hcount,vcount,pixel_on,
3'b100);

   assign intensity = effect_intensity [2];
   assign delay_speed = effect_intensity [4];
   assign delay_int = effect_intensity [5];

   always @(posedge vclock)
   begin
      if (reset)
      begin
            color [0] <= 3'b001;
            color [1] <= 3'b001;
            color [2] <= 3'b001;
            color [3] <= 3'b001;
            color [4] <= 3'b001;
            color [5] <= 3'b001;
            color [6] <= 3'b001;
            color_bars[0]<= 3'b010;
            color_bars[1]<= 3'b010;
            color_bars[2]<= 3'b010;
```

```
            color_bars[3]<= 3'b010;
            color_bars[4]<= 3'b010;
            color_bars[5]<= 3'b010;
            color_bars[6]<= 3'b010;
            fx <= 0;
            effect_intensity[0] <= 5'b0;
            effect_intensity[1] <= 5'b0;
            effect_intensity[2] <= 5'b0;
            effect_intensity[3] <= 5'b0;
            effect_intensity[4] <= 5'b0;
            effect_intensity[5] <= 5'b0;
            effect_intensity[6] <= 5'b0;
            control <=1;
      end
      else
      begin
            old_left<=left;
            old_right <= right;
            old_up <= up;
            old_down <= down;
            old_enter <=enter;

            if (~old_left && left)
                  control <= (control == 0) ? 5 : control - 1;
            else if (~old_right && right)
                  control <= (control == 5) ? 0 : control + 1;

            if (~old_enter && enter)
            begin
                  case (control)
                  0: fx <= 0;
                  1: fx <= 1;
                  2: fx <= 1;
                  3: fx <= 2;
                  default: fx<=fx;
                  endcase
            end

            if (~old_up && up)
                  effect_intensity [control] <= effect_intensity
[control] + 1;

            if (~old_down && down)
                  effect_intensity [control] <= effect_intensity
[control] - 1;

            case (fx)
            2'b00: begin
                        speed <= effect_intensity[0];
                        x <= 10'd44;
                  end
            2'b01: begin
                        speed <= effect_intensity[1];
                        x <= 10'd150;
                  end
            2'b10: begin
                        speed <= effect_intensity[3];
```

```
                        x <= 10'd258;
                end
            endcase

          color[0] <= (control == 0) ? 3'b100: 3'b001;
            color[1] <= (control == 1) ? 3'b100: 3'b001;
            color[2] <= (control == 2) ? 3'b100: 3'b001;
            color[3] <= (control == 3) ? 3'b100: 3'b001;
            color[4] <= (control == 4) ? 3'b100: 3'b001;
            color[5] <= (control == 5) ? 3'b100: 3'b001;

            pixel_edge <= (hcount==0 | hcount==798 | vcount==0 |
vcount==599) ? 7 : 0;
            pixel_hor <= (vcount>400 & vcount <407) ? 3 : 0;
            pixel_ver <= (vcount > 400 & hcount > 397 & hcount<403) ?
3: 0;

       end
   end


//use sprite to generate slider bars

   blob bar_bend_speed(10'd57,BAR_TOP,BAR_WIDTH, BAR_HEIGHT,
hcount,vcount,pixel0, color_bars[0]);
   blob bar_vibratto_speed(10'd120,BAR_TOP,BAR_WIDTH,
BAR_HEIGHT,hcount,vcount,pixel1,color_bars[1]);
   blob bar_vibratto_intensity(10'd175,BAR_TOP,BAR_WIDTH,
BAR_HEIGHT,hcount,vcount,pixel2, color_bars[2]);
   blob bar_arpegiator_speed(10'd270,BAR_TOP,BAR_WIDTH,
BAR_HEIGHT,hcount,vcount,pixel3, color_bars[3]);
   blob bar_delay_length(10'd342,BAR_TOP,BAR_WIDTH,
BAR_HEIGHT,hcount,vcount,pixel5, color_bars[5]);
   blob bar_delay_intensity(10'd390,BAR_TOP,BAR_WIDTH,
BAR_HEIGHT,hcount,vcount,pixel6, color_bars[6]);

   blob bend_speed(11'd57+(BAR_WIDTH>>1)-(SLIDER_WIDTH>>1),
BAR_HEIGHT+BAR_TOP-SLIDER_HEIGHT-
(effect_intensity[0]<<2),SLIDER_WIDTH,SLIDER_HEIGHT,
hcount,vcount,pixel7, color[0]);
   blob vibratto_speed(11'd120+(BAR_WIDTH>>1)-
(SLIDER_WIDTH>>1),BAR_HEIGHT+BAR_TOP-SLIDER_HEIGHT-
(effect_intensity[1]<<2),SLIDER_WIDTH,SLIDER_HEIGHT,hcount,vcount,pixel
8,color[1]);
   blob vibratto_intensity(11'd175+(BAR_WIDTH>>1)-
(SLIDER_WIDTH>>1),BAR_HEIGHT+BAR_TOP-SLIDER_HEIGHT-
(effect_intensity[2]<<2),SLIDER_WIDTH,SLIDER_HEIGHT,hcount,vcount,pixel
9, color[2]);
   blob arpegiator_speed(11'd270+(BAR_WIDTH>>1)-
(SLIDER_WIDTH>>1),BAR_HEIGHT+BAR_TOP-SLIDER_HEIGHT-
(effect_intensity[3]<<2),SLIDER_WIDTH,SLIDER_HEIGHT,hcount,vcount,pixel
10, color[3]);
   blob delay_length(11'd342+(BAR_WIDTH>>1)-
(SLIDER_WIDTH>>1),BAR_HEIGHT+BAR_TOP-SLIDER_HEIGHT-
(effect_intensity[4]<<2),SLIDER_WIDTH,SLIDER_HEIGHT,hcount,vcount,pixel
12, color[4]);
```

```
   blob delay_intensity(11'd390+(BAR_WIDTH>>1)-
(SLIDER_WIDTH>>1),BAR_HEIGHT+BAR_TOP-SLIDER_HEIGHT-
(effect_intensity[5]<<2),SLIDER_WIDTH,SLIDER_HEIGHT,hcount,vcount,pixel
13, color[5]);

//code to display the text

   wire [50*8-1:0] cstring1 = "SPEED   SPEED INTENSITY    SPEED    TIME
INTENSITY";
   wire [47*8-1:0] cstring2 = "BENDER      VIBRATO       ARPEGIATOR
DELAY";
   wire [8*8-1:0] cstring3 = "SPECTRUM";
   wire [8*3-1:0] cstring4 = "WET";
   wire [8*3-1:0] cstring5 = "DRY";
   wire [8*3-1:0] cstring6 = "LOW";
   wire [8*4-1:0] cstring7 = "HIGH";

   wire [2:0] pixelc1, pixelc2, pixelc3, pixelc4, pixelc5, pixelc6,
pixelc7,
            pixelc8, pixelc9, pixelc10;

   char_string_display
char1(vclock,hcount,vcount,pixelc1,cstring1,11'd40,10'd250);
   defparam char1.NCHAR = 50; // number of 8-bit characters in cstring
   defparam char1.NCHAR_BITS = 6; // number of bits in NCHAR

   char_string_display
char2(vclock,hcount,vcount,pixelc2,cstring2,11'd35,10'd265);
   defparam char2.NCHAR = 47; // number of 8-bit characters in cstring
   defparam char2.NCHAR_BITS = 6; // number of bits in NCHAR

   char_string_display
char3(vclock,hcount,vcount,pixelc3,cstring3,11'd40,10'd410);

   char_string_display
char4(vclock,hcount,vcount,pixelc4,cstring3,11'd440,10'd410);

   char_string_display
char5(vclock,hcount,vcount,pixelc5,cstring5,11'd10,10'd410);
   defparam char5.NCHAR = 3;  // number of 8-bit characters in cstring
   defparam char5.NCHAR_BITS = 2; // number of bits in NCHAR

   char_string_display
char6(vclock,hcount,vcount,pixelc6,cstring4,11'd410,10'd410);
   defparam char6.NCHAR = 3;  // number of 8-bit characters in cstring

   defparam char6.NCHAR_BITS = 2; // number of bits in NCHAR

   char_string_display
char7(vclock,hcount,vcount,pixelc7,cstring6,11'd3,10'd570);
   defparam char7.NCHAR = 3;  // number of 8-bit characters in cstring
   defparam char7.NCHAR_BITS = 2; // number of bits in NCHAR

   char_string_display
char8(vclock,hcount,vcount,pixelc8,cstring6,11'd405,10'd570);
   defparam char8.NCHAR = 3;  // number of 8-bit characters in cstring
   defparam char8.NCHAR_BITS = 2; // number of bits in NCHAR
```

```
   char_string_display
char9(vclock,hcount,vcount,pixelc9,cstring7,11'd350,10'd570);
   defparam char9.NCHAR = 4;  // number of 8-bit characters in cstring
   defparam char9.NCHAR_BITS = 2; // number of bits in NCHAR

   char_string_display
char10(vclock,hcount,vcount,pixelc10,cstring7,11'd750,10'd570);
   defparam char10.NCHAR = 4; // number of 8-bit characters in cstring
   defparam char10.NCHAR_BITS = 2; // number of bits in NCHAR


//pixel value to be displayed
   assign pixel_out = (pixel0 | pixel1 | pixel2 | pixel3 | pixel4 |
pixel5
      | pixel6 | pixel7 | pixel8 | pixel9 | pixel10
      | pixel11 | pixel12 | pixel13
      | pixel_on | pixel_edge | pixel_hor | pixel_ver
      | pixelc1 | pixelc2 | pixelc3 | pixelc4| pixelc5 | pixelc6|
pixelc7| pixelc8| pixelc9| pixelc10);
                                                          //text

   assign phsync = hsync;
   assign pvsync = vsync;
   assign pblank = blank;

endmodule


//Blob module
//Builds a rectangle with specified parameters

module blob(x,y,w,h,hcount,vcount,pixel, color);

   input [9:0] x; //x is left
   input [10:0] hcount;
   input [9:0] y,vcount;  //y is top
   input [10:0] w; //width
   input [9:0] h;  //height
   input [2:0] color;

   output [2:0] pixel;

   reg [2:0] pixel;

   always @ (x or y or hcount or vcount)
   begin
      if ((hcount >= x && hcount < (x+w)) &&
      (vcount >= y && vcount < (y+h)))
      pixel = color;
      else pixel = 0;
   end

endmodule


//Wet Graph Data Module
```

```
//This module takes the wet data and places the magnitudes in the
correct index //bin to be displayed on the wet spectrum graph

module wet_graph_data(clk, freq_in, bitshift, mag_in, D, Eb, E, F, Gb,
G, Ab, A, Bb, B, C, Db,  mag_out, index_out);

   input clk;
   input [25:0] freq_in;
   input [3:0] bitshift;
   input [4:0] mag_in;

   input [11:0] D, Eb, E, F, Gb, G, Ab, A, Bb, B, C, Db;  //note
frequencies

   output [4:0] mag_out;
   output [3:0] index_out;

   reg [25:0] freq_mid;
   reg [5:0] bitshift_mid;
   reg [4:0] mag_mid;

   reg [6:0] index_out;
   reg [4:0] mag_out;

   always @(posedge clk)
   begin



      index_out <= {(freq_in<<bitshift > (Bb-1)),                 //bit
3
            ((freq_in<<bitshift > Gb-1) && (freq_in<Bb)),  //bit 2
            (((freq_in<<bitshift > (E-1)) && (freq_in<<bitshift < Gb))
|| ((freq_in<<bitshift > (Ab-1))&& (freq_in<<bitshift < Bb)) ||
(freq_in<<bitshift > C)), //bit 1
            (((freq_in<<bitshift > (Eb-1))&& (freq_in<<bitshift < E))
|| ((freq_in<<bitshift > (F-1))&& (freq_in<<bitshift < Gb)) ||
((freq_in<<bitshift > (G-1))&& (freq_in<<bitshift < Ab)) ||
((freq_in<<bitshift > (A-1)) && (freq_in<<bitshift < Bb)) ||
((freq_in<<bitshift > (B-1)) && (freq_in<<bitshift < C)) ||
(freq_in<<bitshift > (Db-1)))};
            mag_out <= mag_in;
   end
endmodule



//FX module
//This module takes the effect selection data from the GUI and modifies
the
//incoming frequencies accordingly and outputs the effected frequency

module fx (clk, freq_in, mag_in, index_in, t_in, freq_out,
                        mag_out, index_out, we_in, we_out, fx,
fx_speed, fx_int);

   input clk;
```

```
   input [25:0] freq_in;
   input [4:0] mag_in;
   input [3:0] index_in;
   input [7:0] t_in;                // time since start of note
   input [1:0] fx;                        // fx type from user
   input [4:0] fx_speed;         // rate of change of output
frequencies
   input [4:0] fx_int;           // distance between frequency
increments
   input we_in;

   output [25:0] freq_out;
   output [4:0] mag_out;
   output [3:0] index_out;
   output we_out;

//////////////// FX  Select///////////////////////
   wire [3:0] bindex_out;
   wire [3:0] vindex_out;
   wire [3:0] aindex_out;
   wire [3:0] hindex_out;
   wire [25:0] bfreq_out;
   wire [25:0] vfreq_out;
   wire [25:0] afreq_out;
   wire [25:0] hfreq_out;
   wire bwe_out;
   wire vwe_out;
   wire awe_out;
   initial we_out = 0;
   reg we_out;
   reg [25:0] freq_out;
   reg [3:0] index_out;
   reg [4:0] mag_out;

   always @ (posedge clk) begin
            mag_out <= mag_in;
        case (fx)
          2'b00: begin                                  //pitch bender
                freq_out <= bfreq_out;
                index_out <= bindex_out;
                we_out <= bwe_out;
                end
          2'b01: begin                                  //vibrato
             freq_out <= vfreq_out;
                index_out <= vindex_out;
                we_out <= vwe_out;
                end
          2'b10: begin                                  //arpeggiator
             freq_out <= afreq_out;
                index_out <= aindex_out;
                   we_out <= awe_out;
                end
          2'b11: begin                                  //harmonizer
             freq_out <= hfreq_out;
                index_out <= hindex_out;
                end
         endcase
```

```
   end


//////////////// FX Modules ///////////////

   bender bend(clk, freq_in, bfreq_out, index_in, bindex_out, t_in,
bwe_out);
   vibrato vib(clk, freq_in, vfreq_out, index_in, vindex_out, fx_int,
t_in, vwe_out);
   arpeggiator arp(clk, freq_in, afreq_out, index_in, aindex_out, t_in,
awe_out);
endmodule


//////////////// Pitch Bender //////////////////
//The pitch bender gradually returns increase frequencies with time

module bender(clk, fin, fout, iin, iout, tcount, weout);
   input clk;
   input [25:0] fin;
   input [3:0] iin;
   input [7:0] tcount;
   output [25:0] fout;
   output [3:0] iout;
   output weout;

   wire [18:0] mstep = fin >> 7; // want to increase fout by ~ 0.8% per
step

   assign fout = fin + tcount*mstep;         // fout will increase
linearly with time

   reg [25:0] oldfin;
   always @ (posedge clk)
      oldfin <= fin;
   assign weout = (fin != oldfin);  // write to dds module on new input
(no delay on output)

   assign iout = iin;
endmodule


///////////////// Vibrato //////////////////////
//The vibrato fluctuates the output frequency around the input
frequency
module vibrato(clk, f_in, f_out, i_in, i_out, int, t_count, we_out);
   input clk;
   input [25:0] f_in;
   input [3:0] i_in;
   input [4:0] int;
   input [7:0] t_count;

   output [25:0] f_out;
   output [3:0] i_out;
   //output [1:0] tmod;
   //output [5:0] tinc;
```

```
   //output [25:0] mstep;
   //output [7:0] intstep;
   output we_out;

   wire [7:0] math_thing = (11 - (int>>2));
   wire [7:0] math_thing2 = (18 - (int>>1));


   reg [25:0] oldfin;
   always @ (posedge clk)
      oldfin <= f_in;
   assign we_out = (f_in != oldfin);  // write to dds module on new
input (no delay on output)

   wire [1:0] tmod = t_count % 4;
   wire [1:0] intmod = int % 4;
   wire [25:0] mstep;
   assign mstep = (intmod == 0 | intmod == 1) ? f_in >> math_thing :
                  (intmod == 2 | intmod == 3) ? (f_in >> math_thing) +
(f_in >>math_thing2):
                           f_in;

   assign f_out = (tmod == 0) ? f_in:
                             (tmod == 1) ? f_in + mstep:
                           (tmod == 2) ? f_in:
                            f_in - mstep;

   assign i_out = i_in;

endmodule

////////////////// Arpeggiator //////////////////////
//This module cycles through different frequencies (the 3rd, 5th
octave, etc)
//of the input frequency

module arpeggiator(clk, fin, fout, iin, iout, tcount, weout);
   input clk;
   input [25:0] fin;
   input [3:0] iin;
   input [7:0] tcount;

   output [25:0] fout;
   output [3:0] iout;
   output weout;

   wire [25:0] third = fin + (fin>>2);
   wire [25:0] fifth = fin + (fin>>1);
   wire [25:0] octave = (fin<<1);
   wire [25:0] tenth = (fin<<1) + (fin>>1);

   wire [3:0] tmod = tcount % 8;

   assign fout = (tmod == 0) ? fin:
                 (tmod == 1) ? third:
                          (tmod == 2) ? fifth:
                          (tmod == 3) ? octave:
```

```
                            (tmod == 4) ? tenth:
                            (tmod == 5) ? octave:
                            (tmod == 6) ? fifth:
                             third;

   reg [25:0] oldfin;
   always @ (posedge clk)
      oldfin <= fin;
   assign weout = (fin != oldfin);  // write to dds module on new input
(no delay on output)
   assign iout = iin;

endmodule
```