peq
An FPGA Parametric Equalizer

Joseph Colosimo

December 10, 2010

Abstract

Audio equalizers are important tools for adjusting the gains of various frequencies in an audio sample so that the sound output is more desirable. Typically, they are used to help offset unique differences in the frequency responses of speakers in order to more accurately replicate the original sound that was recorded.

This report describes the design and implementation of a parametric equalizer, a device that an important additional benefit over conventional equalizers: it allows the user to construct an audio filter of nearly any shape.

Contents

List of Figures

List of Tables

1

Overview

Audio equalizers are an important utility for both amateur audio enthusiasts and professional sound engineers alike.

A parametric equalizer provides the user with the ability to create an arbitrary filter for audio processing. It can be used for anything from improving the audio output quality on a specific set of speakers to removing known sources of noise to applying filter effects in a performance environment.

Parametric equalizers are usually implemented in software or firmware due to their complexity both in terms of their mathematics and the user interface requirements, but a few true commercial hardware parametric equalizers are available, such as the Behringer Ultracurve. [1] This device uses Digital Signal Processors (DSPs) to process an audio stream and provides the user with a variety of interfaces for tuning their desired gain curves.

This 6.111 final project consisted of designing and implementing a high-quality parametric equalizer on an FPGA that would allow the user to define an arbitrary gain curve and apply it to an incoming audio stream. The user designs a gain curve on his or her computer and sends it to the 6.111 Labkit via RS-232 Serial. The Labkit transforms an incoming audio from an AC'97 audio codec chip signal to the frequency domain, multiplies each frequency bin by its associated gain value from the gain curve, transforms the gain-corrected signal back to the time domain, and finally sends the new signal to the AC'97 audio chip for output to the headphone jack.

The first implementation of this design on the 6.111 Labkit successfully demonstrated its operability. Test filters were applied to audio streams and correctly modified the signals. However, the device has a prohibitive level of background static as a result of an incorrectly timed interface between the audio module and the FFT. Once the problem has been resolved, the device will be fully functional.

This report covers the design, construction, and testing of the parametric equalizer, describes potential solutions to the noise issue, and lists areas for further optimization and expansion of the device.

2

Description

1 Overall Architecture

The macroscopic architecture of the system is described in Figure 2.1. The core of the system consists of the FFT, gain curve applier, IFFT, and the gain curve block ram. To enter a gain curve, the user can select one of the available input methods, two of which are unimplemented at this time. Two smaller RAMs are used to interface between the main system and the display subsystem, which operates on a 65MHz clock instead of the system's standard 27MHz clock.

2 Audio Subsystem

Figure 2.2 describes the main part of this system, the audio manipulator that applies a gain curve to the audio stream.

2.1 AC'97 Layer

An AC'97 codec chip handles the multiplexing and conversion of audio data into a digital format. A provided module handles the serial communication with this chip and a wrapper layer simplifies the signal control. [2]

Essentially, only three signals are used between this layer and the rest of the system. The **ready** signal is asserted high for a single clock cycle when the AC'97 chip can send and receive data. This occurs at the sampling frequency, 48KHz. At this point, the signals **from_ac97_data** and **to_ac97_data** can be used. The former is a signed 20-bit output containing an audio sample and the latter is an input for a signed 20-bit value.

The AC'97 wrapper layer is essentially the same as the one used in the 6.111 spectrograph class example, [3] but with a few modifications. When Switch 2 on the Labkit is on, the module will forward all **from_ac_data** directly to **to_ac97_data** without going through the audio modification subsystem; otherwise, **to_ac97_data** is the output of the equalizer system.

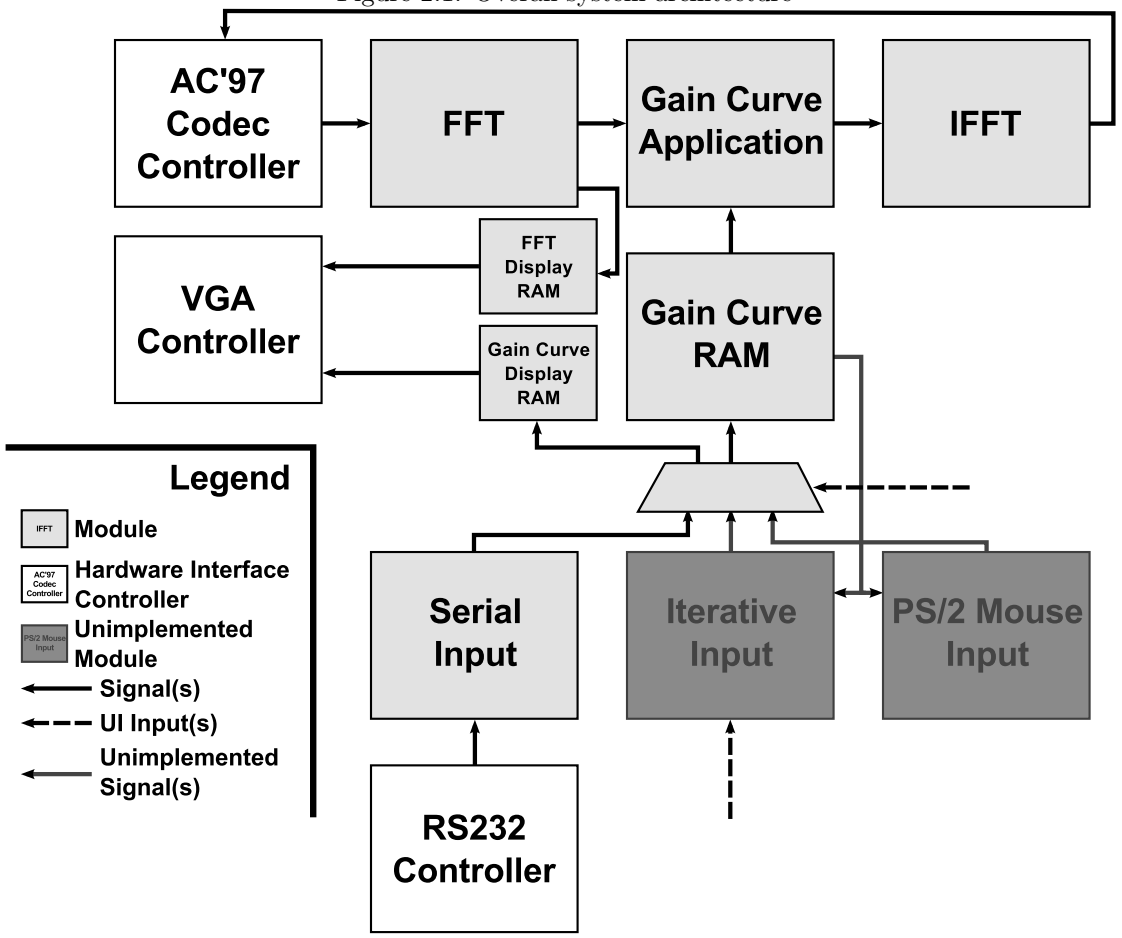
Secondly, when Switch 3 is on, **from_ac_data** is the output of a 750Hz sine wave that was generated and stored in memory. When it's off, Switch 6 controls whether to get input from the microphone (when the switch is off) or from the line-in RCA plugs.

2.2 FFT

The Fast Fourier Transform algorithm efficiently transforms a time-domain signal into the frequency domain. That is for an N-point FFT, it yields X_k from the input samples X_n in the following manner:

$$X_k[k] = \sum_{n=0}^{N-1} X_n \exp\left\{\frac{-2\pi j}{N} n \cdot k\right\} \tag{2.1}$$

Figure 2.1: Overall system architecture



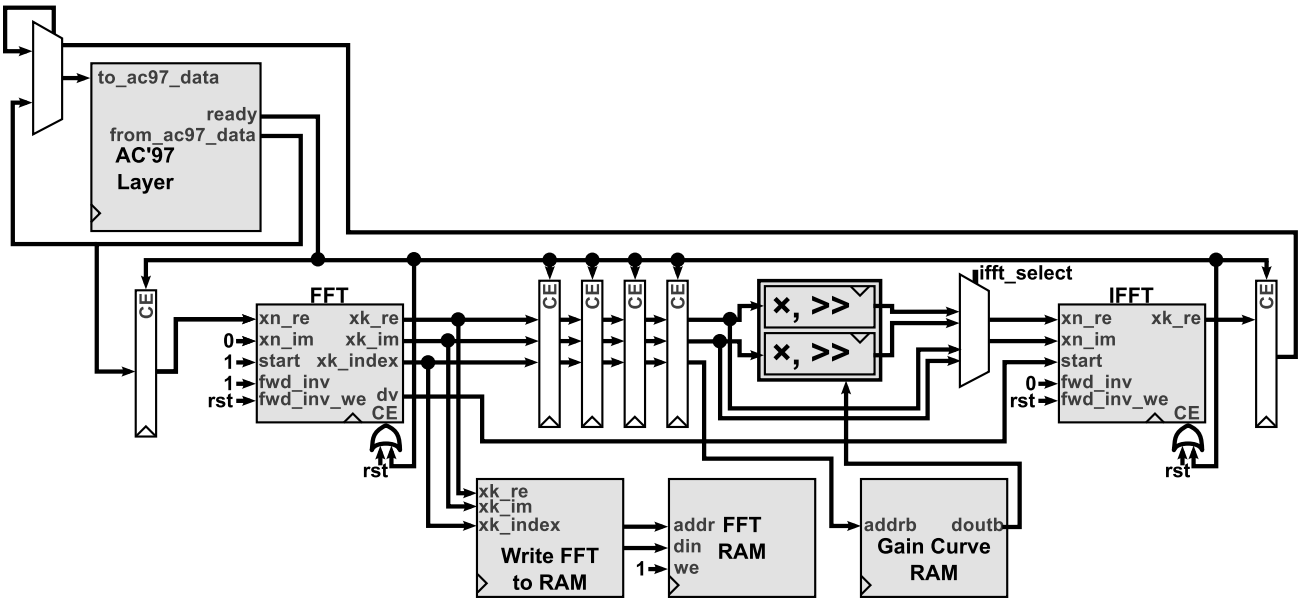
An N-point FFT generates N frequency "bins," values that hold some range of frequencies based on the sampling frequency. In this system, a 1024-point FFT was used and the incoming audio signal was 48KHz, so each bin held $\frac{48000}{1024} \approx 47$ Hz. In general, the lower the bin's frequency coverage, the better; therefore, it is advantageous to choose an FFT with as many points as possible.

The FFT module for this lab was generated with the Xilinx CoreGen wizard. The following options were chosen in the wizard:

- 1024 points
- pipelined/streaming mode
- 12-bit input
- 12-bit phase factor
- unscaled operation
- use clock enable (CE) pin
- use natural ordering

The data entering the FFT was a buffered **from_ac_data**. The pipelining of this signal, which was only allowed to change when **ready** was high, preventing garbage data from entering the FFT.

Figure 2.2: Audio subsystem



The **ready** signal was also extremely important when using the FFT. If the FFT processed data faster than the AC'97's clock, it would generate results too fast from the same set of input samples, and the effect would be as though the audio waveform had slowed down (yielding an audio waveform analogous to the result of applying pressure to a spinning vinyl on a turntable). Therefore, the clock enable pin was only high on **ready**.

The FFT also has a few other control signals. **start** is always enabled, **fwd_inv** is always high to indicate that the module is to perform a forward transform. This data is latched, so **fwd_inv_we** is high when the system performs its power-on reset and off otherwise.

The real and imaginary components of X_k are forwarded to the gain curve application module as well as a simple module that writes them to RAM for live display on the screen. This latter module uses the value from **xk_index** to determine the address in the FFT RAM. The **dv** (data valid) signal is necessary for correctly timing later modules.

2.3 Gain Curve Application

The outputs of the FFT (the real and imaginary components of X_k) is fed into a series of registers to delay the signals. The motivation for this delay will be discussed in 2.4.

After the real and imaginary components have been delayed, the resulting signals enter the gain curve application phase.

The index of the FFT output (which has also been equivalently delayed) is fed into the gain curve RAM and the real and imaginary components of the FFT are multiplied by the output of that RAM. This is a clocked operation, but it's clocked to the 27MHz system clock, so from the perspective of the far slower, 48KHz clock, the operation is instantaneous. A more accurate implementation would pipeline the result of the multiplication by a 48KHz clock, but the output would be indistinguishable from the user's perspective.

After the multiplication, the components are divided by 256, the maximum value of the gain curve (they are actually arithmetically shifted by 8, an equivalent operation). As a result, all frequency values are effectively multiplied by a gain less than one since the maximum gain is 255. This prevents "clipping," an effect where the output speaker voltage is pushed to its max and the speaker is overdriven.

The output of this system is sent to a multiplexer, where the user can use Switch 7 to select whether

to use the gain-corrected output of the FFT (off) or the pure output of the FFT (on) as the input into the inverse transform for the final processing phase.

2.4 IFFT

The inverse Fourier transform, as its name suggests, changes a frequency-domain signal into the time domain. Given a frequency domain signal of X_k, we an obtain the time domain signal X_n as follows:

$$X_n[n] = \frac{1}{N} \sum_{k=0}^{N-1} X_k \exp\left\{ \frac{2\pi j}{N} k \cdot n \right\} \tag{2.2}$$

It is important to note that the forward transform described in 2.2 and the inverse transform described here are nearly identical. The FFT algorithm can be easily modified into an IFFT by taking the complex conjugate of the inputs and applying a $\frac{1}{N}$ scaling factor. The Xilinx FFT module supports an inverse transform mode when **fwd_inv** is set to 0. It does, however, require more work to get the transform to operate correctly.

The same core generated in 2.2 was used to compute the IFFT. To simplify the system a little, consider the design case where a user wishes to redirect the output of an FFT into an IFFT. In an ideal scenario, the signal going into the FFT and the signal coming out of the IFFT will be exactly the same. Of course, because sampling is discrete, small errors will arise; this noise can be reduced by increasing the size of the FFT.

Because the FFT and IFFT are in unscaled mode, the input of the IFFT must be pre-multiplied by $\frac{1}{N}$, as the Xilinx core does not do this. This can be accomplished simply by performing a right arithmetic shift by $\lg(N)$.

Secondly, the IFFT shouldn't start until the FFT is producing valid output data. There are several output signals from the FFT that indicate the validity of the data, namely **dv**, **done**, and **edone**. The first of these is most effective because it goes high as soon as the FFT starts producing data and remains high thereafter. Therefore, we can feed **dv** of the FFT into **start** of the IFFT. This guarantees that the IFFT will always be synchronized to the start of the FFT.

The output of the FFT must be in natural ordering. Due to the way that the FFT algorithm functions, the frequency coefficients do not normally appear in order; a RAM and re-ordering algorithm must be instantiated in order to correct this problem.

Lastly, the inverse FFT needs three cycles after the start signal before it can accept data. Therefore, the output of the FFT needs to be appropriately delayed using the 48KHz **ready** signal as its clock. This is accomplished by using registers with a clock enable line set to **ready**.

The real output of the IFFT is fed into a register, also clocked to **ready**, which is then sent to **to_ac97_data**.

3 Display Subsystem

The display subsystem consists of a VGA controller and two modules that specify the value of a pixel given its location and data from RAM. All of these modules operate on a 65 MHz clock, created with the FPGA's DCM.

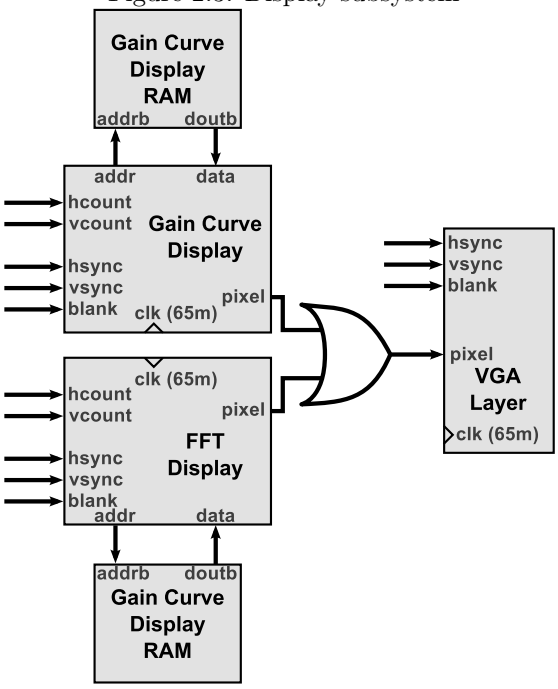
The VGA controller generates timing signals: the horizontal synchronizer **hsync**, vertical synchronizer **vsync**, and blanking signal **blank**. It also outputs the current position of the VGA scanner through **hcount**, a 11-bit value describing the current horizontal position, and **vcount**, a 10-bit value for the vertical position.

The screen resolution is 1024×768 and the screen is split into three sections to show the current gain curve, the absolute value of the FFT's real component, and the absolute value of the FFT's imaginary component.

3.1 Gain Curve Display

(see Appendix 6)

Figure 2.3: Display subsystem



The gain curve display module reads data from the gain curve display RAM by assigning the RAM address to the current horizontal value. It then paints a pixel a positive color if the current vertical value is between the bottom of the graph and the value of the gain curve at that address.

3.2 FFT Display

(see Appendix 5)

The FFT display shows the absolute values of the real and imaginary components of the FFT in the same manner as the gain curve display. To compute the absolute value of a component, the signal is tested to see if the most significant bit is 1. If it is, according to two's complement arithmetic, the signal's value is flipped and 1 is added to it.

4 Input Subsystem

This device provides a framework for developing many methods for the user to input a new gain curve. Currently, only the serial input method is functional, but the framework for an iterative input algorithm was built and a mouse-based input system was also planned.

4.1 Serial Input

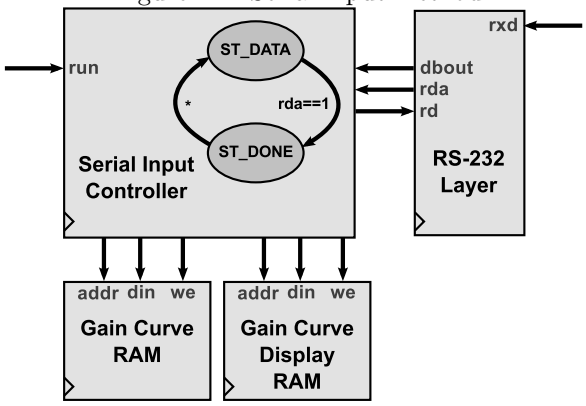
(see Appendix 7)

A user can connect a computer to the RS-232 port via a null-modem crossover cable and transfer a gain curve. A switch is toggled to start the process. Next, the user transmits a gain curve at 9600bps, with one stop bit and odd parity, and one byte per FFT bin. Therefore, the first byte that the user transfers will contain the gain for the first bin, the second will be for the second bin, etc.

Figure 2.4 describes the serial controller. The main module resets its internal state machine to its initial state (*ST_DATA*) and sets the RAM address counters to 0 when **run** is first asserted. The state machine interacts with the RS-232 protocol layer, grabbing a byte of data whenever one is available. When it receives

a byte, the controller stores it to the gain curve RAM as well as the gain curve display RAM. If the number of FFT points, N, is less than or equal to the display width, 1024, the controller increments the addresses of both the gain curve RAM and the display RAM. If N is greater than the display size, the controller will only update the address of the display RAM every $\frac{N}{1024}$ bytes. For example, for an 8192-point FFT, the display RAM's address will only be updated once every 8 bytes received. This is accomplished by keeping an **index** counter and only taking the top 10 for the display RAM address signal.

Figure 2.4: Serial input method



The RS-232 protocol layer was designed by Digilent Corporation for their Nexys2 Spartan-3E evaluation board [4] (see Appendix 12). This component receives data from the RX serial port and generates a byte by oversampling the data and looking for transitions between bits as well as checking the odd parity bit. It places this byte in a buffer.

A slight modification to the module was necessary to get the correct baud rate. The divisor is determined by dividing the clock in the following manner:

$$\text{divisor} = \frac{clockfreq}{\text{baud} \cdot 16 \cdot 2} \tag{2.3}$$

The Nexys2 board has a clock speed of 50MHz and the Labkit has a clock speed of 27MHz, so the divisor was changed to 89 to account for the change in clock frequency. Higher baud rates can be used for faster transfer speeds.

The interface between the gain curve input and this RS-232 layer simply reads data from a buffer and then asserts a signal indicating that it read that data, yielding a two-state state machine as shown in Figure 2.4 and described in Table 2.1. The **rd** signal is asserted when the serial control reads the byte from the RS-232 layer.

Table 2.1: Outputs of serial input state machine

State	**State Name**	**rd** (Completed Data Read)	**Other Actions**
0	ST_DATA	0	Save **dbout** to the gain curve and display RAMs
1	ST_DONE	1	Increment **index**

4.2 Incremental Input

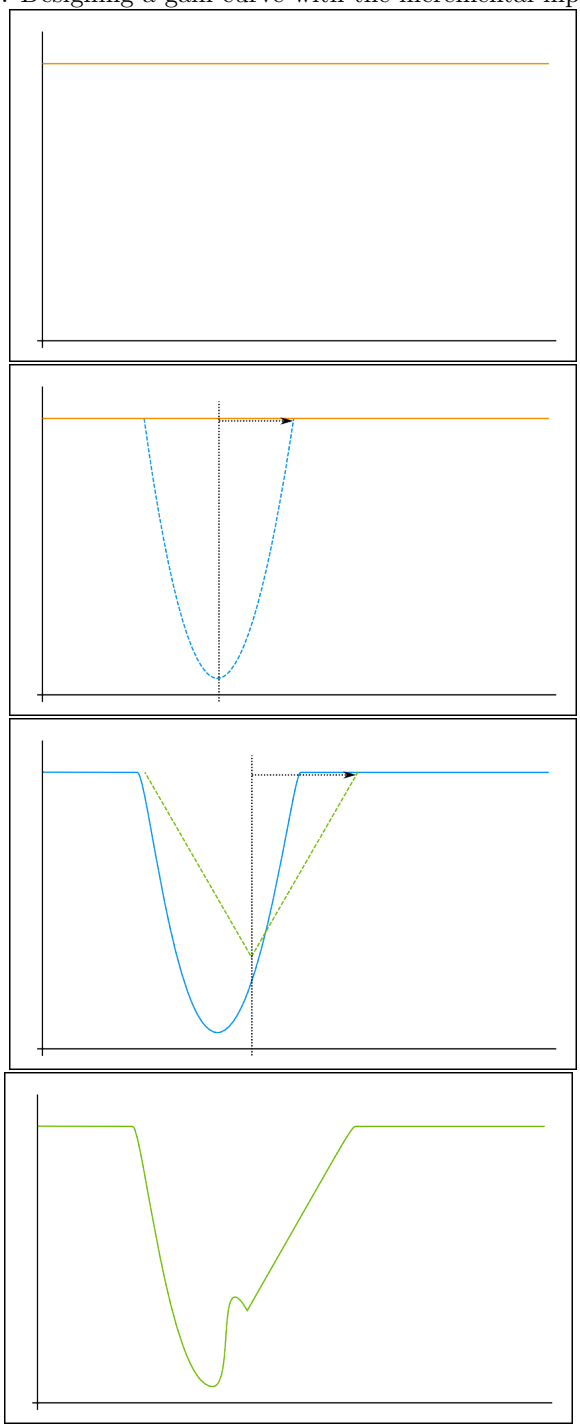
(see Appendix 8)

The incremental input system provides an interface for the user to design gain curve directly with the FPGA using buttons on the Labkit or perhaps a PS/2 keyboard. The concept behind making the input "incremental" comes from the parametric equalizer's overall design property as being an audio filter. If a

user finds that a sample of audio has, for example, noise at some specific frequencies, he can easily remove them by implementing "notch" filters that remove them.

The principle behind the incremental input system is that the user will make a filter of some shape, height, location, and bandwidth and place it on the gain curve. Then, the user will make another one and the two filters will be composed together smoothly, as shown in Figure 2.5.

Figure 2.5: Designing a gain curve with the incremental input method



Unfortunately, while some code was produced to start the design of this system (see Appendices 8, 9, and 10), it wasn't completed due to time constraints.

4.3 Mouse Input

A third possible input method would be to draw a gain curve with a mouse, just as one would draw with a piece of drawing software. This facilitates easy manipulation of the audio waveform and can be utilized by users who wish to experiment with audio effects.

5 Testing and Debugging

While the concept of the device itself was fairly simple, the implementation of the FFT and IFFT turned out to be somewhat complex than originally anticipated due to their low popularity and therefore relatively little documentation. The FFT module was instantiated in the same way that the class example's FFT was instantiated. [5], but for the IFFT module, it was necessary to introduce delays into the input signal and synchronize the start time with the output of the FFT. The Xilinx datasheet [6] did not document the core particularly well and only parenthetically mentioned the need for scaling and delay of the input signal into the IFFT.

To verify the operability of the FFT-gain-IFFT section, a 750Hz sine wave stored in memory was fed into the FFT. The gain curve was bypassed so that the IFFT would receive the scaled data of the FFT. At first, the FFT wasn't created with natural ordering enabled, so the output of the IFFT sounded nothing like the input. The effective result was that the output frequency bins did not match up with their expected input frequency bins of the IFFT.

The **done** signal of the FFT is asserted once every N computations, so this signal was used as a trigger on a logic analyzer. Also displayed on the analyzer were the outputs of the FFT, the magnitude of IFFT, and the FFT index. The expected result of feeding a 750Hz sine wave into the module was a single bin in the first half of the FFT with a large value (possibly with some surrounding harmonic bins containing very small values due to fact that only an infinitely large transform would have just a single bin with a nonzero value).

After ensuring that the FFT was correctly performing its computation and that the IFFT was being fed a scaled input, the magnitude of the IFFT output yielded very strange results. Essentially, the tone generated was a composition of two sine waves of equal magnitude and 90-degree phase shift. This problem was due to the fact that the IFFT input signal was not delayed correctly. To fix the problem, a variable delay system was introduced to determine exactly how far to delay the IFFT input. Once this delay was correctly applied, the sound output matched the sound input.

When switching from the sine tone to the output signals from AC'97, a full-spectrum noise (a "popping" or "clicking" sound) was introduced. Due to time constraints, the problem wasn't resolved, but it was narrowed down to having occurred somewhere between the AC'97 module and the FFT. It is likely that a timing issue causes the FFT input to temporarily become invalid, introducing noise into the calculation. The noise itself is not very noticeable on the magnitude output of the IFFT, but it can definitely be heard by the human ear.

3

Conclusion

1 Design Summary

By transforming a signal into the frequency domain, it is possible to apply unique gains for each resulting frequency bin and then transform it back into the time domain, allowing a user to apply arbitrary filters to an audio sample. An FPGA is especially efficient for this purpose since Xilinx and other companies offer powerful, low-latency cores for performing forward and inverse Fourier transforms.

By implementing this system in conjunction with user interfaces that provide the capability to set a "gain curve" — a vector of values to apply to the frequency bins, a parametric equalizer is produced.

The system implemented on the 6.11 Labkit allows the user to create gain curve on a computer and transfer it via the serial port. Audio from the microphone or the RCA line-in plugs is sent to the equalizer module and is sent out via the headphone jack.

Testing provided that the application of various audio filters resulted in signals that behaved as expected. For example, a high-pass filter applied to some music took out the bass frequencies, leaving a "tinny" sound for the purposes of demonstration.

2 Known Issues

A major lingering problem with the system was signal noise, likely caused by a problem with the AC'97 interface. The fact that the noise covered the full spectrum indicated that it was likely some sort of problem where data was becoming invalid for a short period of time, introducing noise into the FFT. A re-examination of the interface between the AC'97 and the FFT module is necessary to determine if this is indeed the case.

The FFT display should be switched to a spectrograph in order to display the audio signal in a more user-friendly manner. Also, doing this will free up a third of the screen for displaying a spectrograph of the data after it has been gain-corrected.

3 Areas for Improvement and Future Development

The two-port gain curve and gain curve display memories can be replaced by a single three-port memory since the data is the same, but is accessed by many modules. This would help to reduce the amount of block RAMs being used.

The system can be more-or-less duplicated to add the right stereo channel back (currently, it only uses the left channel). Moving to a larger FFT will improve the audio quality significantly because it increases the system's frequency resolution.

More specialized gain curves can be produced in the future to help correct for deficiencies in speakers and improve overall acoustic quality. The equalizer can also be used to produce effects filters that can be applied in real time if the serial module is modified to use a faster transfer speed.

The incremental and mouse input interfaces, while not crucial to the operation of the equalizer, would be helpful for users who don't want to attach a computer to the equalizer to reprogram it.

Other interesting uses for a parametric equalizer include an "auto-program" feature where directional microphones are placed in a room to simulate a user's ears and the system learns about the frequency response from all of the speakers in the room as they reach the users ears and adjust all the gain curves for each speaker to guarantee constant power to the user's ears.

4

References

Notes

[1] *Behringer: DEQ2496* [Online]. Available:
`http://www.behringer.com/EN/Products/DEQ2496.aspx`
[2] *Lab 4 AC97 Modules* [Online]. Available:
`http://web.mit.edu/6.111/www/f2010/handouts/labs/lab4.v`
[3] *Spectrograph* [Online]. Available:
`http://web.mit.edu/6.111/www/f2007/handouts/fft.v`
[4] *RS232 interface reference component* [Online]. Available:
`https://www.digilentinc.com/Data/Documents/Reference%20Designs/RS232%20RefComp.zip`
[5]See [3]
[6] *Xilinx Fast Fourier Transform 7.1* [Online]. Available:
`http://www.xilinx.com/support/documentation/ip_documentation/xfft_ds260.pdf` (Note that version 3.2 was used in this system)

5

Appendices

1 Verilog Listing (top-level): labkit.v

```
/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * ********************** */

//
   ////////////////////////////////////////////////////////////////////////////

//
// 6.111 FPGA Labkit --- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
//
   ////////////////////////////////////////////////////////////////////////////

//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
```

```
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
//
   ////////////////////////////////////////////////////////////////////////

//
// Complete change history (including bug fixes)
//
// 2006-Mar-08: Corrected default assignments to "vga_out_red", "vga_out_green
   "
//              and "vga_out_blue". (Was 10'h0, now 8'h0.)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
//
   ////////////////////////////////////////////////////////////////////////


module labkit (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;

input   mouse_clock, mouse_data, 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 = 8'h0;
assign vga_out_green = 8'h0;
assign vga_out_blue = 8'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
assign led[3:1] = 3'b111;
// 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;

// 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


/* ************************************
 * PARAMETRIC EQUALIZER HARDWARE
   ************************************ */

 ////////////////////////////////
 // 65MHz clock (from lab 5)
 ////////////////////////////////
     // use FPGA's digital clock manager to produce a
     // 65MHz clock (actually 64.8MHz)
     wire clock_65mhz_unbuf,clock_65mhz;
     DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
     // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
     // 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_65mhz),.I(clock_65mhz_unbuf));


 ////////////////////////////////
 // power-on reset generation (from lab 5)
 ////////////////////////////////
     wire power_on_reset;    // remain high for first 16 clocks
     SRL16 reset_sr (.D(1'b0), .CLK(clock_27mhz), .Q(power_on_reset),
             .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
     defparam reset_sr.INIT = 16'hFFFF;

 ////////////////////////////////
 // debounce buttons (from many labs)
 ////////////////////////////////
     wire btn_rst, btn_add, btn_left, btn_right, btn_up, btn_down, btn_vup,
         btn_vdn;
```

```
    wire btn_a, btn_b, btn_c;
    debounce db1(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~
        button3),.clean(btn_rst));
    debounce db2(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~
        button_enter),.clean(btn_add));
    debounce db3(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~
        button_left),.clean(btn_left));
    debounce db4(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~
        button_right),.clean(btn_right));
    debounce db5(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~
        button_up),.clean(btn_up));
    debounce db6(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~
        button_down),.clean(btn_down));
    debounce db7(.reset(power_on_reset), .clock(clock_27mhz), .noisy(~
        button_right), .clean(btn_vup));
    debounce db8(.reset(power_on_reset), .clock(clock_27mhz), .noisy(~
        button_left), .clean(btn_vdn));
    debounce db9(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~
        button2),.clean(btn_a));
    debounce db10(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~
        button1),.clean(btn_b));
    debounce db11(.reset(power_on_reset),.clock(clock_27mhz),.noisy(~
        button0),.clean(btn_c));

////////////////////////////////
// vga controller (from lab 5)
////////////////////////////////
    // generate basic XVGA video signals
    wire [10:0] hcount;
    wire [9:0]  vcount;
    wire [2:0] pixel;
    wire hsync,vsync,blank;

    // pipelined syncs
    wire phsync,pvsync,pblank;

    xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
               .hsync(hsync),.vsync(vsync),.blank(blank));

    // VGA Output.  In order to meet the setup and hold times of the
    // AD7125, we send it ~clock_65mhz.
    assign vga_out_red = {8{pixel[2]}};
    assign vga_out_green = {8{pixel[1]}};
    assign vga_out_blue = {8{pixel[0]}};
    assign vga_out_sync_b = 1'b1;    // not used
    assign vga_out_blank_b = ~pblank;
    assign vga_out_pixel_clock = ~clock_65mhz;
    assign vga_out_hsync = phsync;
    assign vga_out_vsync = pvsync;

////////////////////////////////
// 16-digit hex display (from lab 3)
////////////////////////////////
    wire [15:0] hexdisplayinput;
```

```
    wire [15:0] hexdebug, hexdebug2;
    display_16hex hexdisplay(.reset(power_on_reset), .clock_27mhz(
        clock_27mhz),
        .data({hexdebug, hexdebug2, 16'h0000, hexdisplayinput}),
        .disp_blank(disp_blank), .disp_clock(disp_clock),
        .disp_rs(disp_rs), .disp_ce_b(disp_ce_b), .disp_reset_b(
            disp_reset_b),
        .disp_data_out(disp_data_out));

////////////////////////////////
// ac97 control module (from labkit audio test)
////////////////////////////////
    wire [3:0] vol;
    wire volume_up, volume_down;

    wire audio_ready;

    wire [15:0] from_ac97_data, to_ac97_data;

    fft_audio audio1 (
        .clock_27mhz(clock_27mhz),
        .reset(power_on_reset),
        .volume({vol, 1'b0}),
        .audio_in_data(from_ac97_data),
        .audio_out_data(to_ac97_data),
        .ready(audio_ready),
        .audio_reset_b(audio_reset_b),
        .ac97_sdata_out(ac97_sdata_out),
        .ac97_sdata_in(ac97_sdata_in),
        .ac97_synch(ac97_synch),
        .ac97_bit_clock(ac97_bit_clock),
        .mode_in(switch[3]),
        .mode_out(switch[2]),
        .insource(switch[6])
    );

    volume vol1 (power_on_reset, clock_27mhz, btn_vup, btn_vdn, vol);

////////////////////////////////
// main parametric equalizer module
////////////////////////////////
    wire [11:0] to_ac97_data_adj;
    wire [11:0] from_ac97_data_adj;

    pequalizer peq(
        .clk(clock_27mhz),
        .clk65(clock_65mhz),
        .rst(power_on_reset),

        // serial com
        .rs232_rxd(rs232_rxd),

        // buttons
        .btn_rst(btn_rst),
```

```
    .btn_add(btn_add),
    .btn_up(btn_up),
    .btn_down(btn_down),
    .btn_a(btn_a),
    .btn_b(btn_b),
    .btn_c(btn_c),
    .sw_inpmtd(switch[1:0]),

    // vga
    .hcount(hcount),
    .vcount(vcount),
    .hsync(hsync),
    .vsync(vsync),
    .blank(blank),
    .pixel(pixel),
    .phsync(phsync),
    .pvsync(pvsync),
    .pblank(pblank),

    // hex display
    .hex(hexdisplayinput),

    // ac97
    .from_ac97_data(from_ac97_data_adj),
    .to_ac97_data(to_ac97_data_adj),
    .audio_ready(audio_ready),

    .ifft_select(switch[7]),
    .fft_delay(4),
    .ifft_reg(switch[5]),

    //debug
    .debug(hexdebug),
    .debug2(hexdebug2),

    .dbg_fftidx(analyzer2_data),
    .dbg_fftdone(analyzer1_data),
    .dbg_fftout(analyzer3_data)
);

assign to_ac97_data = {to_ac97_data_adj, 4'b0000};
assign from_ac97_data_adj = from_ac97_data[15:16-12];

// debug
assign led[0] = rs232_rxd;
assign led[7:6] = switch[7:6];
assign led[5:4] = ~switch[5:4];

// Logic Analyzer
// analyzer1 -> fft index
// analyzer2 -> {ifft_done, fft_dv, fft_done}
// analyzer3 -> fft output
assign analyzer4_data = to_ac97_data;
assign analyzer1_clock = audio_ready;
```

```
        assign analyzer2_clock = audio_ready;
        assign analyzer3_clock = clock_27mhz;
        assign analyzer4_clock = clock_65mhz;
endmodule
```

2 Verilog Listing: peq.v

```
`default_nettype none

/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * peq.v                  *
 * main equalizer module  *
 * ********************** */

module pequalizer( // {{{
    input wire clk, clk65,
    input wire rst,

    // serial com
    input wire rs232_rxd,

    // buttons
    //input wire btn_rst, btn_add, btn_left, btn_up, btn_right, btn_down,
    input wire btn_rst, btn_add, btn_a, btn_b, btn_c, btn_up, btn_down,
    input wire [1:0] sw_inpmtd,

    // vga
    input wire [10:0] hcount,
    input wire [9:0] vcount,
    input wire hsync, vsync, blank,
    output wire [2:0] pixel,
    output wire phsync, pvsync, pblank,

    // hex
    output wire [15:0] hex,

    // ac97
    input wire  [11:0] from_ac97_data,
    output wire [11:0] to_ac97_data,
    input wire audio_ready,

    // controls
    input wire ifft_select,
    input wire [2:0] fft_delay,
    input wire ifft_reg,

    // debug
    //   (to hex display)
    output wire [15:0] debug,
    output wire [15:0] debug2,
```

```
    //   (to logic analyzer)
    output wire [15:0] dbg_fftdone,
    output wire [15:0] dbg_fftidx, dbg_ifftidx,
    output wire [15:0] dbg_fftout
); // }}}

    parameter LOGFFTSIZE = 10;        // 1024-point FFT
    parameter LOGDSPSIZE = 10;        // 1024-wide display

    parameter AUDIOWIDTH = 12;        // 12-bit audio
    parameter GCRVEWIDTH =  8;
    parameter DISPLWIDTH =  8;        // 8-bit gain curve display

    parameter FFTOUTSIZE = LOGFFTSIZE + AUDIOWIDTH + 1;       // size of fft
        output

    ////////////////////////////////
    // BLOCK RAMS (dual port)
    // instantiate RAM according to calculations in ram.v
    // gcurve is the gain curve RAM and gcdisp is the RAM for the display
    // fftram holds the current FFT
    // these RAMs have a read-only B port in addition to a read-write A port
    //////////////////////////////// {{{
        wire [LOGFFTSIZE-1:0] gcurve_addr_in, gcurve_addr_out;
        wire [LOGDSPSIZE-1:0] gcdisp_addr_in, gcdisp_addr_out;

        wire [GCRVEWIDTH-1:0] gcurve_din, gcurve_dout, gcurve_doutb;
        wire [DISPLWIDTH-1:0] gcdisp_din, gcdisp_dout, gcdisp_doutb;
        wire gcurve_we, gcdisp_we;

        blockram #(.LOGSIZE(LOGFFTSIZE), .WIDTH(GCRVEWIDTH)) gcurveram1
            (.clk(clk), .addr(gcurve_addr_in), .din(gcurve_din),
                .dout(gcurve_dout), .we(gcurve_we),
                .clkb(clk), .addrb(gcurve_addr_out), .doutb(gcurve_doutb));

        blockram #(.LOGSIZE(LOGDSPSIZE), .WIDTH(DISPLWIDTH)) gcdispram1
            (.clk(clk), .addr(gcdisp_addr_in), .din(gcdisp_din),
                .dout(gcdisp_dout), .we(gcdisp_we),
                .clkb(clk65), .addrb(gcdisp_addr_out), .doutb(gcdisp_doutb));

        ////////////////////
        // FFT DISPLAY RAM
        ////////////////////
        wire [LOGDSPSIZE-1:0] fftram_addr_in, fftram_addr_out;
        wire [2*DISPLWIDTH-1:0] fftram_din, fftram_dout, fftram_doutb;
        wire fftram_we;

        blockram #(.LOGSIZE(LOGDSPSIZE), .WIDTH(2*DISPLWIDTH)) fftram1
            (.clk(clk), .addr(fftram_addr_in), .din(fftram_din),
                .dout(fftram_dout), .we(fftram_we),
                .clkb(clk65), .addrb(fftram_addr_out), .doutb(fftram_doutb));
    // }}}
```

```
////////////////////////////////
// AUDIO IN -> FFT
// here, we take a pipelined, streaming FFT of from_ac97_data and place
// the output on fft_out_re and fft_out_im.  the fft is run only when
// audio_ready is high, since we shouldn't be doing anything when the ac97
// isn't sending out valid data
//
// fft_index is used to specify the location of the block ram to store the
// appropriate bin.  it's also sent out to a debug signal.
//
// fft_dv tells us when fft_out_* becomes valid and therefore is used to
// start the IFFT.  fft_done is asserted every N cycles and is used for
// debugging.
//
// lastly, we need to delay output of the FFT by 3 cycles when feeding it
// into the IFFT.
//////////////////////////////// {{{
    wire signed [FFTOUTSIZE-1:0] fft_out_im, fft_out_re;

    reg signed [FFTOUTSIZE-1:0] fft_out_re_1, fft_out_re_2, fft_out_re_3,
        fft_out_re_4, fft_out_re_5, fft_out_re_6, fft_out_re_7;
    reg signed [FFTOUTSIZE-1:0] fft_out_im_1, fft_out_im_2, fft_out_im_3,
        fft_out_im_4, fft_out_im_5, fft_out_im_6, fft_out_im_7;

    wire signed [FFTOUTSIZE-1:0] fft_out_re_final, fft_out_im_final;

    wire [LOGFFTSIZE-1:0] fft_index, ifft_index;

    reg [LOGFFTSIZE-1:0] fft_index_1, fft_index_2, fft_index_3,
        fft_index_4, fft_index_5, fft_index_6, fft_index_7;

    wire [LOGFFTSIZE-1:0] fft_index_final;

    wire fft_done, fft_dv;

    reg [AUDIOWIDTH-1:0] from_ac97_data_reg;
    always @ (posedge clk) begin
        if ( audio_ready || ifft_reg ) begin
            audio_ready_reg <= audio_ready;
            from_ac97_data_reg <= from_ac97_data;
        end
    end

    fft1024un fft(.clk(clk), .ce(rst | audio_ready),
        .xn_re(from_ac97_data_reg), .xn_im(12'b0), .start(1'b1),
        .fwd_inv(1'b1), .fwd_inv_we(rst), .xk_re(fft_out_re),
        .xk_im(fft_out_im), .xk_index(fft_index), .done(fft_done),
        .dv(fft_dv));

    // stupid debug tool to get the 3 cycles right
    always @ (posedge clk) begin
        // only change when audio_ready goes high
        if (audio_ready) begin
            // delay by 1
```

```
            fft_out_re_1 <= fft_out_re;
            fft_out_im_1 <= fft_out_im;
            fft_index_1 <= fft_index;
        // delay by 2
            fft_out_re_2 <= fft_out_re_1;
            fft_out_im_2 <= fft_out_im_1;
            fft_index_2 <= fft_index_1;
        // delay by 3
            fft_out_re_3 <= fft_out_re_2;
            fft_out_im_3 <= fft_out_im_2;
            fft_index_3 <= fft_index_2;
        // delay by 4
            fft_out_re_4 <= fft_out_re_3;
            fft_out_im_4 <= fft_out_im_3;
            fft_index_4 <= fft_index_3;
        // delay by 5
            fft_out_re_5 <= fft_out_re_4;
            fft_out_im_5 <= fft_out_im_4;
            fft_index_5 <= fft_index_4;
        // delay by 6
            fft_out_re_6 <= fft_out_re_5;
            fft_out_im_6 <= fft_out_im_5;
            fft_index_6 <= fft_index_5;
        // delay by 7
            fft_out_re_7 <= fft_out_re_6;
            fft_out_im_7 <= fft_out_im_6;
            fft_index_7 <= fft_index_6;
    end
end

// set the delay based on the fft_delay input
assign fft_out_re_final = (fft_delay == 0) ? fft_out_re   :
                          (fft_delay == 1) ? fft_out_re_1 :
                          (fft_delay == 2) ? fft_out_re_2 :
                          (fft_delay == 3) ? fft_out_re_3 :
                          (fft_delay == 4) ? fft_out_re_4 :
                          (fft_delay == 5) ? fft_out_re_5 :
                          (fft_delay == 6) ? fft_out_re_6 :
                          (fft_delay == 7) ? fft_out_re_7 : 0;

assign fft_out_im_final = (fft_delay == 0) ? fft_out_im   :
                          (fft_delay == 1) ? fft_out_im_1 :
                          (fft_delay == 2) ? fft_out_im_2 :
                          (fft_delay == 3) ? fft_out_im_3 :
                          (fft_delay == 4) ? fft_out_im_4 :
                          (fft_delay == 5) ? fft_out_im_5 :
                          (fft_delay == 6) ? fft_out_im_6 :
                          (fft_delay == 7) ? fft_out_im_7 : 0;

assign fft_index_final =  (fft_delay == 0) ? fft_index   :
                          (fft_delay == 1) ? fft_index_1 :
                          (fft_delay == 2) ? fft_index_2 :
                          (fft_delay == 3) ? fft_index_3 :
                          (fft_delay == 4) ? fft_index_4 :
```

```
                                (fft_delay == 5) ? fft_index_5 :
                                (fft_delay == 6) ? fft_index_6 :
                                (fft_delay == 7) ? fft_index_7 : 0;
// }}}


////////////////////////////////
// GAIN CURVE APPLICATION
// multiply each component of the fft by the associated value in the gain
// curve.  this takes one clock cycle, which won't affect the overall
// delay requirement of the IFFT input since audio_ready is asserted much
// more slowly than clk_27mhz
//
// after multiplying by the gain curve value, divide by the maximum gain
// curve value (2^GCRVEWIDTH)
//
// finally, the IFFT mandates that its input must be scaled down by the
// number of points in the FFT
//      since the IFFT and the FFT have the same number of points and the
//      output of the FFT has width W+N+1 where W is the input width and
//      N is lg(number of points), the IFFT should take
//          fft[W+N+1:0] >> N
//      however, note that we are in the special case where the number of
//      points of the FFT and the IFFT are equal, so we can just take the
//      top AUDIOWIDTH (i.e. the input size of the IFFT) points since that
//      will include both the N scaling factor (which alone will generate
//      an output width of AUDIOWIDTH+1) and the correction for the
//      disparity in input size (the extra bit)
//
//      the N scaling factor only becomes important if the IFFT has more
//      points than the FFT, but we aren't doing that here
//////////////////////////////// {{{
    reg signed [FFTOUTSIZE+GCRVEWIDTH-1:0] mult_re, mult_im;
    wire signed [FFTOUTSIZE-1:0] mult_re_shifted, mult_im_shifted;
    wire signed [AUDIOWIDTH-1:0] mult_re_out, mult_im_out;

    assign gcurve_addr_out = fft_index_final;

    always @ (posedge clk) begin
        mult_re <= fft_out_re_final * $unsigned(gcurve_doutb);
        mult_im <= fft_out_im_final * $unsigned(gcurve_doutb);
    end

    assign mult_re_shifted = mult_re >>> GCRVEWIDTH;
    assign mult_im_shifted = mult_im >>> GCRVEWIDTH;

    // ifft_select selects whether to use the just the output of the FFT
    // (1) or the value multiplied by the gain curve (0)
    assign mult_re_out = (ifft_select) ?
        fft_out_re_final[FFTOUTSIZE-1:FFTOUTSIZE-AUDIOWIDTH] :
            mult_re_shifted[FFTOUTSIZE-1:FFTOUTSIZE-AUDIOWIDTH];

    assign mult_im_out = (ifft_select) ?
        fft_out_im_final[FFTOUTSIZE-1:FFTOUTSIZE-AUDIOWIDTH] :
```

```
            mult_im_shifted[FFTOUTSIZE-1:FFTOUTSIZE-AUDIOWIDTH];
// }}}


////////////////////////////////
// AUDIO OUT
// we take the 12-bit, delayed mult_re_out and feed it into the IFFT with
// the appropriate control signal to start the module (data valid of the
// FFT)
//////////////////////////////// {{{
    wire signed [FFTOUTSIZE-1:0] ifft_out_re, ifft_out_im;
    wire ifft_done;

    fft1024un ifft(.clk(clk), .ce(rst | audio_ready),
        .xn_re(mult_re_out),
        .xn_im(mult_im_out),
        .start(fft_dv), .fwd_inv(1'b0),
        .fwd_inv_we(rst), .xk_re(ifft_out_re), .xk_im(ifft_out_im),
        .xk_index(ifft_index), .done(ifft_done));

    reg [AUDIOWIDTH-1:0] to_ac97_data_reg;

    assign to_ac97_data = to_ac97_data_reg;

    always @ (posedge clk) begin
        if ( audio_ready || ifft_reg ) begin
            to_ac97_data_reg <= ifft_out_re;
        end
    end
// }}}


////////////////////////////////
// FFT DISPLAY
// we store the absolute values of the real and imaginary components of
// the FFT output to a block RAM for display with the fftdisplay module.
//////////////////////////////// {{{
    reg fftram_we_reg;
    reg [LOGDSPSIZE-1:0] fftram_addr_in_reg;
    reg [2*DISPLWIDTH-1:0] fftram_din_reg;

    // just a stupid refactoring -- can be optimized out safely
    assign fftram_we = fftram_we_reg;
    assign fftram_addr_in = fftram_addr_in_reg;
    assign fftram_din = fftram_din_reg;

    always @ (posedge clk) begin
        fftram_we_reg <= 1;
        fftram_addr_in_reg <= fft_index[LOGFFTSIZE-1:LOGFFTSIZE-LOGDSPSIZE
            ];

        // store the absolute value of each signal
        case ({fft_out_re[FFTOUTSIZE-1], fft_out_im[FFTOUTSIZE-1]})
            2'b00:
```

```
                fftram_din_reg <=
                { fft_out_re[FFTOUTSIZE-1:FFTOUTSIZE-DISPLWIDTH],
                  fft_out_im[FFTOUTSIZE-1:FFTOUTSIZE-DISPLWIDTH] };
            2'b01:
                fftram_din_reg <=
                { fft_out_re[FFTOUTSIZE-1:FFTOUTSIZE-DISPLWIDTH],
                  ~fft_out_im[FFTOUTSIZE-1:FFTOUTSIZE-DISPLWIDTH] + 1 };
            2'b10:
                fftram_din_reg <=
                { ~fft_out_re[FFTOUTSIZE-1:FFTOUTSIZE-DISPLWIDTH] + 1,
                  fft_out_im[FFTOUTSIZE-1:FFTOUTSIZE-DISPLWIDTH] };
            2'b11:
                fftram_din_reg <=
                { ~fft_out_re[FFTOUTSIZE-1:FFTOUTSIZE-DISPLWIDTH] + 1,
                  ~fft_out_im[FFTOUTSIZE-1:FFTOUTSIZE-DISPLWIDTH] + 1 };
        endcase
    end
// }}}


////////////////////////////////
// FFT DEBUG SIGNALS
// these can be used by the logic analyzer
//////////////////////////////// {{{
    assign dbg_fftidx = fft_index;
    assign dbg_ifftidx = ifft_index;
    assign dbg_fftdone = {ifft_done, fft_dv, fft_done};
    assign dbg_fftout = fft_out_re[FFTOUTSIZE-1:FFTOUTSIZE-8];
// }}}


////////////////////////////////
// INPUT METHOD 0: RS-232 GAIN CURVE INPUT
// sw_inpmtd = 2 || 3
// switch sw_inpmtd off and then on two enter a new gain curve
//////////////////////////////// {{{
    wire [LOGFFTSIZE-1:0] gcurve_addr_ser;
    wire [LOGDSPSIZE-1:0] gcdisp_addr_ser;
    wire [GCRVEWIDTH-1:0] gcurve_din_ser;
    wire [DISPLWIDTH-1:0] gcdisp_din_ser;
    wire gcurve_we_ser, gcdisp_we_ser;

    gcserinp #(.LOGFFTSIZE(LOGFFTSIZE), .LOGDSPSIZE(LOGDSPSIZE),
        .AUDIOWIDTH(AUDIOWIDTH), .DISPLWIDTH(DISPLWIDTH),
        .GCRVEWIDTH(GCRVEWIDTH)) inp0
        (   .clk(clk),
            .rst(rst),

            .run(sw_inpmtd[1]),
            .rxd(rs232_rxd),

            .gcurve_addr(gcurve_addr_ser),
            .gcdisp_addr(gcdisp_addr_ser),
            .gcurve_din(gcurve_din_ser),
```

```
            .gcdisp_din(gcdisp_din_ser),
            .gcurve_we(gcurve_we_ser),
            .gcdisp_we(gcdisp_we_ser)
        );
// }}}


////////////////////////////////
// INPUT METHOD 1: GAIN CURVE MODIFIERS
// sw_inpmtd = 0
//
// this doesn't work correctly, so I had to scrap it for the time being
//////////////////////////////// {{{
    wire [LOGFFTSIZE-1:0] gcurve_addr_mod;
    wire [LOGDSPSIZE-1:0] gcdisp_addr_mod;
    wire [AUDIOWIDTH-1:0] gcurve_din_mod;
    wire [DISPLWIDTH-1:0] gcdisp_din_mod;
    wire gcurve_we_mod, gcdisp_we_mod;

    gcmodinp #(.LOGFFTSIZE(LOGFFTSIZE), .AUDIOWIDTH(AUDIOWIDTH), .
        DISPLWIDTH(DISPLWIDTH)) inp1
        (   .clk(clk),
            .rst(rst),
            .btn_rst(btn_rst),
            .btn_add(btn_add),
            //.btn_left(btn_left),
            .btn_up(btn_up),
            //.btn_right(btn_right),
            .btn_down(btn_down),
            .btn_a(btn_a),
            .btn_b(btn_b),
            .btn_c(btn_c),

            .hex(hex),

            .gcurve_addr(gcurve_addr_mod),
            .gcdisp_addr(gcdisp_addr_mod),
            .gcurve_din(gcurve_din_mod),
            .gcdisp_din(gcdisp_din_mod),
            .gcurve_dout(gcurve_dout),
            .gcdisp_dout(gcdisp_dout),      // not used
            .gcurve_we(gcurve_we_mod),
            .gcdisp_we(gcdisp_we_mod)
        );
// }}}


////////////////////////////////
// INPUT METHOD 2: MOUSE INPUT
// sw_inpmtd = 1
//
// this wasn't implemented yet
//////////////////////////////// {{{
    wire [LOGFFTSIZE-1:0] gcurve_addr_mse;
```

```
    wire [LOGDSPSIZE-1:0] gcdisp_addr_mse;
    wire [AUDIOWIDTH-1:0] gcurve_din_mse;
    wire [DISPLWIDTH-1:0] gcdisp_din_mse;
    wire gcurve_we_mse, gcdisp_we_mse;

    /*
    gcmseinp #(.LOGFFTSIZE(LOGFFTSIZE), .AUDIOWIDTH(AUDIOWIDTH), .
        DISPLWIDTH(DISPLWIDTH)) inp2
         (    .clk(clk),
              .rst(rst),

              // mouse events go here

              .gcurve_addr(gcurve_addr_mse),
              .gcdisp_addr(gcdisp_addr_mse),
              .gcurve_din(gcurve_din_mse),
              .gcdisp_din(gcdisp_din_mse),
              .gcurve_dout(gcurve_dout),
              .gcdisp_dout(gcdisp_dout),       // not used
              .gcurve_we(gcurve_we_mse),
              .gcdisp_we(gcdisp_we_mse),
         );
    */
// }}}


////////////////////////////////
// INPUT METHOD SELECTION
// suffix ser => input method 0 (serial)
// suffix mod => input method 1 (gain curve modifiers)
// suffix mse => input method 2 (mouse)
//////////////////////////////// {{{
    assign gcurve_addr_in = (sw_inpmtd == 0) ? gcurve_addr_mod :
                            (sw_inpmtd == 1) ? gcurve_addr_mse :
                               gcurve_addr_ser;
    assign gcdisp_addr_in = (sw_inpmtd == 0) ? gcdisp_addr_mod :
                            (sw_inpmtd == 1) ? gcdisp_addr_mse :
                               gcdisp_addr_ser;

    assign gcurve_din = (sw_inpmtd == 0) ? gcurve_din_mod :
                        (sw_inpmtd == 1) ? gcurve_din_mse : gcurve_din_ser
                           ;
    assign gcdisp_din = (sw_inpmtd == 0) ? gcdisp_din_mod :
                        (sw_inpmtd == 1) ? gcdisp_din_mse : gcdisp_din_ser
                           ;

    assign gcurve_we = (sw_inpmtd == 0) ? gcurve_we_mod :
                       (sw_inpmtd == 1) ? gcurve_we_mse : gcurve_we_ser;
    assign gcdisp_we = (sw_inpmtd == 0) ? gcdisp_we_mod :
                       (sw_inpmtd == 1) ? gcdisp_we_mse : gcdisp_we_ser;
// }}}


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

```
// VGA CONTROL
// two modules write to the display: the gain curve display and the FFT
// display
////////////////////////////////// {{{

wire [2:0] gcpxl, fftpxl;
assign pixel = gcpxl | fftpxl;

// no pipelining necessary right now
assign phsync = hsync;
assign pvsync = vsync;
assign pblank = blank;

gcdisplay #(.LOGFFTSIZE(LOGFFTSIZE), .AUDIOWIDTH(AUDIOWIDTH), .DISPLWIDTH(
    DISPLWIDTH)) gcdisp1
    (
        .clk(clk65),
        .rst(rst),

        .hcount(hcount),
        .vcount(vcount),
        .hsync(phsync),
        .vsync(pvsync),
        .blank(pblank),

        .pixel(gcpxl),

        .addr(gcdisp_addr_out),
        .data(gcdisp_doutb)
    );


fftdisplay #(.LOGFFTSIZE(LOGFFTSIZE), .LOGDSPSIZE(LOGDSPSIZE), .AUDIOWIDTH
    (AUDIOWIDTH), .DISPLWIDTH(DISPLWIDTH)) fftdisp1
    (
        .clk(clk65),
        .rst(rst),

        .pixel(fftpxl),
        .hcount(hcount),
        .vcount(vcount),
        .hsync(phsync),
        .vsync(pvsync),
        .blank(pblank),

        .addr(fftram_addr_out),
        .data(fftram_doutb)
    );
// }}}

////////////////////////////////
// OTHER DEBUG SIGNALS
////////////////////////////////// {{{
    // send gain curve RAM addresses to the hex display
```

```
        assign debug = gcurve_addr_in;
        assign debug2 = gcdisp_addr_in;
    // }}}
```

endmodule

```
// vim: fdm=marker
```

3 Verilog Listing: ram.v

```
/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * ram.v                  *
 * dual port block RAM    *
 * ********************** */

/*
 * block RAM module (modified from lab 4)
 *
 * gain curve calculations:
 *    with a 8192-FFT, we need 8192 locations:
 *        LOGSIZE=13
 *
 *    we'll use an 8 bit value for the gain curve display:
 *        WIDTH=8
 *
 *    and a 12 bit value for the curve itself
 *        WIDTH=12
*/

//
   ////////////////////////////////////////////////////////////////////////////

//
// Verilog equivalent to a BRAM, tools will infer the right thing!
// number of locations = 1<<LOGSIZE, width in bits = WIDTH.
// default is a 16K x 1 memory.
//
//
   ////////////////////////////////////////////////////////////////////////////


module blockram
    #(parameter LOGSIZE=14, WIDTH=1)
    (input wire [LOGSIZE-1:0] addr,
        input wire clk,
        input wire [WIDTH-1:0] din,
        output reg [WIDTH-1:0] dout,
        input wire we,

        input wire clkb,
        input wire [LOGSIZE-1:0] addrb,
```

```
        output reg [WIDTH-1:0] doutb
    );

    // let the tools infer the right number of BRAMs
    (* ram_style = "block" *)
    reg [WIDTH-1:0] mem[(1<<LOGSIZE)-1:0];
    always @(posedge clk) begin
      if (we) mem[addr] <= din;
      dout <= mem[addr];
    end

    always @(posedge clkb) begin
        doutb <= mem[addrb];
    end

endmodule
```

4 Verilog Listing: xvga.v

```
/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * xvga.v                 *
 * xvga controller        *
 * ********************** */

/* from lab 5 */

//
   ////////////////////////////////////////////////////////////////////////////

//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
//
//
   ////////////////////////////////////////////////////////////////////////////


module xvga(input vclock,
            output reg [10:0] hcount,    // pixel number on current line
            output reg [9:0] vcount,     // line number
            output reg vsync,hsync,blank);

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   reg hblank,vblank;
   wire hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount == 1023);
   assign hsyncon = (hcount == 1047);
   assign hsyncoff = (hcount == 1183);
   assign hreset = (hcount == 1343);

   // vertical: 806 lines total
```

```
    // display 768 lines
    wire vsyncon,vsyncoff,vreset,vblankon;
    assign vblankon = hreset & (vcount == 767);
    assign vsyncon = hreset & (vcount == 776);
    assign vsyncoff = hreset & (vcount == 782);
    assign vreset = hreset & (vcount == 805);

    // sync and blanking
    wire next_hblank,next_vblank;
    assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
    assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
    always @(posedge vclock) begin
       hcount <= hreset ? 0 : hcount + 1;
       hblank <= next_hblank;
       hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

       vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
       vblank <= next_vblank;
       vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

       blank <= next_vblank | (next_hblank & ~hreset);
    end
endmodule
```

5 Verilog Listing: fftdisplay.v

```
/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * fftdisplay.v           *
 * display fft output     *
 * ********************** */

module fftdisplay #(parameter LOGFFTSIZE=0, LOGDSPSIZE=0, AUDIOWIDTH=0,
    DISPLWIDTH=0)
        (
            input wire clk,
            input wire rst,

            output wire [2:0] pixel,
            input wire [10:0] hcount,
            input wire [9:0] vcount,
            input wire hsync, vsync, blank,

            output wire [LOGDSPSIZE-1:0] addr,
            input wire [2*DISPLWIDTH-1:0] data
        );

    assign addr = hcount;

    reg [2:0] pxl1, pxl2;
    assign pixel = pxl1 | pxl2;
```

```
    always @ (posedge clk) begin
        if ( hcount <= 1023 && vcount >= 256 && vcount - 256 >= 255 - data
            [15:8] && vcount <= 511 )
            pxl1 <= 3'b001;
        else if ( vcount >= 256 && vcount <= 511 )
            pxl1 <= 3'b110;
        else
            pxl1 <= 3'b000;

        if ( hcount <= 1023 && vcount >= 512 && vcount - 512 >= 255 - data
            [7:0] && vcount <= 767 )
            pxl2 <= 3'b010;
        else if ( vcount >= 512 && vcount <= 767 )
            pxl2 <= 3'b101;
        else
            pxl2 <= 3'b000;

    end

endmodule
```

6 Verilog Listing: gcdisplay.v

```
/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * gcdisplay.v            *
 * gain curve display     *
 * ********************** */

module gcdisplay #(parameter LOGFFTSIZE=0, AUDIOWIDTH=0, DISPLWIDTH=0)
    (
        input wire clk,
        input wire rst,

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

        output reg [2:0] pixel,

        output wire [9:0] addr,
        input wire [DISPLWIDTH-1:0] data
    );

    assign addr = hcount[9:0];

    always @ (posedge clk) begin
        if ( hcount <= 1023 && vcount <= 255 - data || vcount >= 255 ) begin
            pixel <= 0;
        end else begin
            pixel <= 3;
        end
```

```
    end

endmodule
```

7 Verilog Listing: gcserinp.v

```
/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * gcserinp.v             *
 * get gain curve from ser *
 * ********************** */

module gcserinp #(parameter LOGFFTSIZE=0, LOGDSPSIZE=0, AUDIOWIDTH=0,
    DISPLWIDTH=0, GCRVEWIDTH=0)
    (   input clk,
        input rst,

        input run,
        input rxd,

        output wire [LOGFFTSIZE-1:0] gcurve_addr,
        output wire [LOGDSPSIZE-1:0] gcdisp_addr,
        output reg [GCRVEWIDTH-1:0] gcurve_din,
        output reg [DISPLWIDTH-1:0] gcdisp_din,
        output reg gcurve_we,
        output reg gcdisp_we,

        output wire [15:0] debug
    );

    reg [LOGFFTSIZE-1:0] index;
    reg working;

    assign gcurve_addr = index;
    assign gcdisp_addr = index >> (LOGFFTSIZE-LOGDSPSIZE);

    reg [7:0] serdata;
    reg rda;

    parameter ST_DATA = 0;
    parameter ST_DONE = 1;
    reg smcur, smnxt;

    wire rd;
    assign rd = ( smcur == ST_DATA ) ? 0 : ( smcur == ST_DONE ) ? 1 : 0;

    reg [7:0] dbout;
    reg txd, pe, fe, oe;
    wire tbe;

    Rs232RefComp rs232 (
        .RXD(rxd),
```

```
    .CLK(clk),
    .RST(rst),
    .DBOUT(serdata),
    .RDA(rda),
    .RD(rd),

    .TXD(txd),
    .DBIN(dbout),
    .TBE(tbe),
    .WR(1'b0),
    .PE(pe),
    .FE(fe),
    .OE(oe)
);

assign debug[7:0] = serdata;
assign debug[15] = smcur;
assign debug[14] = rda;
assign debug[13] = rd;
assign debug[12:8] = 0;

always @ (posedge clk) begin
    if (rst) begin
        gcurve_we <= 0;
        gcdisp_we <= 0;
        index <= 0;
    end else begin
        if (!run) begin
            index <= 0;
            gcurve_we <= 0;
            gcdisp_we <= 0;
            working <= 1;
            smcur <= ST_DATA;
        end else if (!working) begin
            gcurve_we <= 0;
            gcdisp_we <= 0;
        end else begin
            gcurve_we <= 1;
            gcdisp_we <= 1;
            smcur <= smnxt;

            if ( smcur == ST_DATA ) begin
                gcurve_din <= serdata;
                gcdisp_din <= serdata;
            end

            if ( smcur == ST_DONE ) begin
                if ( index == (1<<LOGFFTSIZE) - 1 ) begin
                    working <= 0;
                    smcur <= ST_DATA;
                end else begin
                    index <= index + 1;
                end
            end
```

```
            end
        end
    end

    always @ (*) begin
        case (smcur)
            ST_DATA:
                if (rda) smnxt = ST_DONE;
                else     smnxt = ST_DATA;

            ST_DONE:
                smnxt = ST_DATA;
        endcase
    end

endmodule
```

8 Verilog Listing: gcmodinp.v

```
/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * gcmodinp.v             *
 * gain curve modifier    *
 *   input method (not    *
 *   working)             *
 * ********************** */

module gcmodinp #(parameter LOGFFTSIZE=0, AUDIOWIDTH=0, DISPLWIDTH=0)
    (
        input wire clk,
        input wire rst,

        input wire btn_rst, btn_add, btn_a, btn_b, btn_c, btn_up, btn_down,

        output wire [15:0] hex,

        input wire [AUDIOWIDTH-1:0] gcurve_dout,
        input wire [DISPLWIDTH-1:0] gcdisp_dout,

        output wire [LOGFFTSIZE-1:0] gcurve_addr,
        output wire [9:0] gcdisp_addr,
        output wire [AUDIOWIDTH-1:0] gcurve_din,
        output wire [DISPLWIDTH-1:0] gcdisp_din,
        output wire gcurve_we, gcdisp_we,

        output wire busy
    );

    parameter ST_BINN = 0;
    parameter ST_WDTH = 1;
    parameter ST_GAIN = 2;
    parameter ST_RCMP_ST = 3;
```

```
parameter ST_RCMP_WT = 4;
parameter ST_RSET_ST = 5;
parameter ST_RSET_WT = 6;
parameter ST_DONE = 7;
reg [2:0] smcur, smnxt;

// controls
wire do_recompute, do_reset;
wire recompute_done, reset_done;

// storage elements to hold parameters
reg [LOGFFTSIZE-1:0] bin_num;
reg [LOGFFTSIZE-1-1:0] bin_width;
reg [AUDIOWIDTH-1:0] bin_gain;

////////////////////////////////
// HEX DISPLAY
////////////////////////////////
    assign hex = ( smcur == ST_BINN ) ? {3'b000, bin_num} :
                 ( smcur == ST_WDTH ) ? {4'b1010, bin_width} :
                 ( smcur == ST_GAIN ) ? {4'b1011, bin_gain} :
                 16'hFFFF;

////////////////////////////////
// BUSY SIGNAL
////////////////////////////////
    assign busy = ( smcur == ST_RCMP_ST || ST_RCMP_WT || ST_RSET_ST ||
       ST_RSET_WT || ST_DONE ) ? 1 : 0;

    wire [LOGFFTSIZE-1:0] gcurve_addr_rs, gcurve_addr_rc;
    wire [DISPLWIDTH:0] gcdisp_addr_rs, gcdisp_addr_rc;
    wire [AUDIOWIDTH-1:0] gcurve_din_rs, gcurve_din_rc;
    wire [DISPLWIDTH-1:0] gcdisp_din_rs, gcdisp_din_rc;
    wire gcurve_we_rc, gcurve_we_rs, gcdisp_we_rc, gcdisp_we_rs;

    assign gcurve_addr = ( smcur == ST_RSET_ST || smcur == ST_RSET_WT ) ?
       gcurve_addr_rs :
                         ( smcur == ST_RCMP_ST || smcur == ST_RCMP_WT ) ?
                            gcurve_addr_rc : 0;
    assign gcdisp_addr = ( smcur == ST_RSET_ST || smcur == ST_RSET_WT ) ?
       gcdisp_addr_rs :
                         ( smcur == ST_RCMP_ST || smcur == ST_RCMP_WT ) ?
                            gcdisp_addr_rc : 0;
    assign gcurve_din = ( smcur == ST_RSET_ST || smcur == ST_RSET_WT ) ?
       gcurve_din_rs :
                         ( smcur == ST_RCMP_ST || smcur == ST_RCMP_WT ) ?
                            gcurve_din_rc : 0;
    assign gcdisp_din = ( smcur == ST_RSET_ST || smcur == ST_RSET_WT ) ?
       gcdisp_din_rs :
                         ( smcur == ST_RCMP_ST || smcur == ST_RCMP_WT ) ?
                            gcdisp_din_rc : 0;
    assign gcurve_we = ( smcur == ST_RSET_ST || smcur == ST_RSET_WT ) ?
       gcurve_we_rs :
                         ( smcur == ST_RCMP_ST || smcur == ST_RCMP_WT ) ?
```

```
                                gcurve_we_rc : 0;
    assign gcdisp_we = ( smcur == ST_RSET_ST || smcur == ST_RSET_WT ) ?
        gcdisp_we_rs :
                            ( smcur == ST_RCMP_ST || smcur == ST_RCMP_WT ) ?
                                gcdisp_we_rc : 0;

    assign do_recompute = ( smcur == ST_RCMP_ST );
    assign do_reset = ( smcur == ST_RSET_ST );

////////////////////////////////
// GAIN CURVE RESET MODULE
////////////////////////////////
    gcreset #(.LOGFFTSIZE(LOGFFTSIZE), .AUDIOWIDTH(AUDIOWIDTH), .
        DISPLWIDTH(DISPLWIDTH)) rs1
        (
            .clk(clk),
            .rst(rst),

            .do_reset(do_reset),
            .reset_done(reset_done),

            .gcurve_addr(gcurve_addr_rs),
            .gcdisp_addr(gcdisp_addr_rs),
            .gcurve_din(gcurve_din_rs),
            .gcdisp_din(gcdisp_din_rs),
            .gcurve_we(gcurve_we_rs),
            .gcdisp_we(gcdisp_we_rs)
        );


////////////////////////////////
// RECOMPUTATION MODULE
////////////////////////////////
    gcrecomp #(.LOGFFTSIZE(LOGFFTSIZE), .AUDIOWIDTH(AUDIOWIDTH), .
        DISPLWIDTH(DISPLWIDTH)) rc1
        (
            .clk(clk),
            .rst(rst),

            .do_recompute(do_recompute),
            .recompute_done(recompute_done),

            .bin_num(bin_num),
            .bin_width(bin_width),
            .bin_gain(bin_gain),

            .gcurve_addr(gcurve_addr_rc),
            .gcdisp_addr(gcdisp_addr_rc),
            .gcurve_din(gcurve_din_rc),
            .gcdisp_din(gcdisp_din_rc),
            .gcurve_dout(gcurve_dout),
            .gcdisp_dout(gcdisp_dout),
            .gcurve_we(gcurve_we_rc),
            .gcdisp_we(gcdisp_we_rc)
```

```
        );

////////////////////////////////
// MASTER STATE MACHINE
// 0 BINN bin #
// 1 WDTH bin width
// 2 GAIN gain step
// 3 RCMP perform recomputation of gain curve
// 4 DONE reset values, return back to 0
////////////////////////////////
    always @ (posedge clk)
    begin
        if (rst) begin
            smcur <= ST_BINN;
            bin_num <= 0;
            bin_width <= 0;
            bin_gain <= 0;
        end else begin
             if ( smcur == ST_DONE ) begin
                 bin_num <= 0;
                 bin_width <= 0;
                 bin_gain <= 0;
             end else if ( btn_up ) begin
                 case (smcur)
                    ST_BINN:
                        bin_num <= bin_num + 1;
                    ST_WDTH:
                        bin_width <= bin_width + 1;
                    ST_GAIN:
                        bin_gain <= 8;
                endcase
             end else if ( btn_down ) begin
                 case (smcur)
                    ST_BINN:
                        bin_num <= bin_num - 1;
                    ST_WDTH:
                        bin_width <= bin_width - 1;
                    ST_GAIN:
                        bin_gain <= 16;
                 endcase
            end
            smcur <= smnxt;
       end
    end

    always @ (*)
    begin
       case (smcur)
           ST_RCMP_ST:
               smnxt = ST_RCMP_WT;

           ST_RCMP_WT:
               if (recompute_done) smnxt = ST_DONE;
               else smnxt = ST_RCMP_WT;
```

```
                ST_RSET_ST:
                    smnxt = ST_RSET_WT;

                ST_RSET_WT:
                    if (reset_done) smnxt = ST_DONE;
                    else smnxt = ST_RSET_WT;

                ST_DONE:
                    smnxt = ST_BINN;

                default:
                    if ( btn_a ) smnxt = ST_BINN;
                    else if ( btn_b ) smnxt = ST_WDTH;
                    else if ( btn_c ) smnxt = ST_GAIN;
                    else if ( btn_add ) smnxt = ST_RCMP_ST;
                    else if ( btn_rst ) smnxt = ST_RSET_ST;
                    else smnxt = smcur;
            endcase
        end

endmodule
```

9 Verilog Listing: gcrecomp.v

```
/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * gcrecomp.v             *
 * recurve gain curve (not *
 *   working)             *
 * ********************** */

module gcrecomp #(parameter LOGFFTSIZE=0, AUDIOWIDTH=0, DISPLWIDTH=0)
    (
        input wire clk,
        input wire rst,

        // controls
        input wire do_recompute,
        output reg recompute_done,

        input wire [LOGFFTSIZE-1:0] bin_num,
        input wire [LOGFFTSIZE-1-1:0] bin_width,
        input wire [AUDIOWIDTH-1:0] bin_gain,

        output wire [LOGFFTSIZE-1:0] gcurve_addr,
        output wire [9:0] gcdisp_addr,
        output reg [AUDIOWIDTH-1:0] gcurve_din,
        output reg [DISPLWIDTH-1:0] gcdisp_din,
        input wire [AUDIOWIDTH-1:0] gcurve_dout,
        input wire [DISPLWIDTH-1:0] gcdisp_dout,
        output reg gcurve_we, gcdisp_we
```

```
);

reg [LOGFFTSIZE-1:0] index;
reg [LOGFFTSIZE-1:0] bin_lo, bin_hi, bin_lo1, bin_hi1, bin_lo2, bin_hi2;

assign gcurve_addr = index;
assign gcdisp_addr = index >> (LOGFFTSIZE-10);

reg [AUDIOWIDTH-1:0] newwav;
reg [AUDIOWIDTH-1:0] gcurve_cur;

always @ (posedge clk) begin
    if (rst) begin
        recompute_done <= 1;
        gcurve_we <= 0;
        gcdisp_we <= 0;
        index <= 0;
        newwav <= 0;
    end else if (do_recompute) begin
        recompute_done <= 0;
        gcurve_we <= 0;
        gcdisp_we <= 0;
        index <= 0;
        newwav <= 1<<DISPLWIDTH - 1;
    end else if (!recompute_done) begin
        if ( index == 0 && gcurve_we == 0 ) begin
            // new gain curve update
            // -> set bin_lo and bin_hi (bin_hi > bin_lo or we don't
                proceed)

            if ( bin_num <= bin_width>>2 ) bin_lo2 <= 0;
            else bin_lo2 <= bin_num - bin_width>>2;

            if ( bin_num <= bin_width>>1 ) bin_lo1 <= 0;
            else bin_lo1 <= bin_num - bin_width>>1;

            if ( bin_num <= bin_width ) bin_lo <= 0;
            else bin_lo <= bin_num - bin_width;


            if ( (1<<LOGFFTSIZE)-1 - bin_width>>2 >= bin_num ) bin_hi2 <=
                (1<<LOGFFTSIZE)-1;
            else bin_hi2 <= bin_num + bin_width>>2;

            if ( (1<<LOGFFTSIZE)-1 - bin_width>>1 >= bin_num ) bin_hi1 <=
                (1<<LOGFFTSIZE)-1;
            else bin_hi1 <= bin_num + bin_width>>1;

            if ( (1<<LOGFFTSIZE)-1 - bin_width >= bin_num ) bin_hi <= (1<<
                LOGFFTSIZE)-1;
            else bin_hi <= bin_num + bin_width;

            gcurve_we <= 1;
            gcdisp_we <= 1;
```

```
        gcurve_cur <= gcurve_dout;

    end else begin
        // in active mode

        if ( !gcurve_we ) begin
            // in read phase, update gcurve_cur
            gcurve_cur <= gcurve_dout;
            gcurve_we <= 1;
            gcdisp_we <= 1;

        end else begin
            // in write mode, update the RAM

            /*
                ||   |    |      |    |   ||
            ----||--|---|-----|---|--||----
                lo  lo1  lo2    hi2  hi1  hi
            */

            if ( index > bin_lo && index < bin_num ) begin
                newwav <= newwav - bin_gain;
            end else if ( index > bin_num && index < bin_hi ) begin
                newwav <= newwav + bin_gain;
            end

            /*
            if ( index < bin_lo ) begin
                gcurve_din <= gcurve_cur;
                gcdisp_din <= (1<<DISPLWIDTH)-1 - ((gcurve_cur) >> (
                    AUDIOWIDTH-DISPLWIDTH));
            end else if ( index < bin_lo1 ) begin
                gcurve_din <= gcurve_cur >> 1 + newwav >> 1;
                gcdisp_din <= (1<<DISPLWIDTH)-1 - ((gcurve_cur >> 1 +
                    newwav >> 1) >> (AUDIOWIDTH-DISPLWIDTH));
            end else if ( index < bin_lo2 ) begin
                gcurve_din <= gcurve_cur >> 2 + newwav >> 1 + newwav
                    >> 2;
                gcdisp_din <= (1<<DISPLWIDTH)-1 - ((gcurve_cur >> 2 +
                    newwav >> 1 + newwav >> 2) >> (AUDIOWIDTH-
                    DISPLWIDTH));
            end else if ( index < bin_hi2 ) begin
                gcurve_din <= newwav;
                gcdisp_din <= (1<<DISPLWIDTH)-1 - ((newwav) >> (
                    AUDIOWIDTH-DISPLWIDTH));

            end else if ( index < bin_hi1 ) begin
                gcurve_din <= gcurve_cur >> 2 + newwav >> 1 + newwav
                    >> 2;
                gcdisp_din <= (1<<DISPLWIDTH)-1 - ((gcurve_cur >> 2 +
                    newwav >> 1 + newwav >> 2) >> (AUDIOWIDTH-
                    DISPLWIDTH));
            end else if ( index < bin_hi ) begin
                gcurve_din <= gcurve_cur >> 1 + newwav >> 1;
```

```
                        gcdisp_din <= (1<<DISPLWIDTH)-1 - ((gcurve_cur >> 1 +
                            newwav >> 1) >> (AUDIOWIDTH-DISPLWIDTH));
                    end else begin
                        gcurve_din <= gcurve_cur;
                        gcdisp_din <= (1<<DISPLWIDTH)-1 - ((gcurve_cur) >> (
                            AUDIOWIDTH-DISPLWIDTH));
                    end
                    */

                    gcurve_din <= newwav;
                    //gcdisp_din <= (newwav) >> (AUDIOWIDTH-DISPLWIDTH);
                    gcdisp_din <= newwav;

                    gcurve_we <= 0;
                    gcdisp_we <= 0;

                    if ( index == (1<<LOGFFTSIZE)-1 ) begin
                        recompute_done <= 1;
                        bin_lo <= 0;
                        bin_hi <= 0;
                        index <= 0;
                    end else begin
                        index <= index + 1;
                    end
                end
            end
        end
    end

endmodule
```

10 Verilog Listing: gcreset.v

```
/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * gcreset.v              *
 * reset gain curve       *
 * ********************** */

module gcreset #(parameter LOGFFTSIZE=0, AUDIOWIDTH=0, DISPLWIDTH=0)
    (
        input wire clk,
        input wire rst,

        // controls
        input wire do_reset,
        output reg reset_done,

        output wire [LOGFFTSIZE-1:0] gcurve_addr,
        output wire [9:0] gcdisp_addr,
        output reg [AUDIOWIDTH-1:0] gcurve_din,
        output reg [DISPLWIDTH-1:0] gcdisp_din,
```

```
        output reg gcurve_we, gcdisp_we
    );

    reg [LOGFFTSIZE-1:0] index;

    assign gcurve_addr = index;
    assign gcdisp_addr = index >> (LOGFFTSIZE-10); // divide by 8 -> 8192/8 =
        1024;

    always @ (posedge clk) begin
        if (rst) begin
            reset_done <= 1;
            gcurve_we <= 0;
            gcdisp_we <= 0;
            index <= 0;
        end else if (do_reset) begin
            reset_done <= 0;
            gcurve_we <= 1;
            gcdisp_we <= 1;
            index <= 0;
        end else if (!reset_done) begin
            gcurve_we <= 1;
            gcdisp_we <= 1;

            gcurve_din <= 1<<AUDIOWIDTH - 1;
            gcdisp_din <= 0;

            if ( index == (1<<LOGFFTSIZE)-1 ) begin
                reset_done <= 1;
            end else begin
                index <= index + 1;
            end
        end
    end

endmodule
```

11 Verilog Listing: ac97.v

```
/* ********************** *
 * Parametric Equalizer    *
 * J. Colosimo                    *
 * 6.111 - Fall '10               *
 * ac97.v                                  *
 * ac97 control module     *
 * ********************** */

/*
 * this module was mostly copied from the fft.v example, but extra signals
 * were added to control the volume and the input source (which can either by
 * the microphone or the line in).  in addition, a 750Hz done signal can be
 * selected as an input source
 */
```

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

//
// bi-directional mono interface to AC97
//
//
   ////////////////////////////////////////////////////////////////////////


module fft_audio (clock_27mhz, reset, volume,
                  audio_in_data, audio_out_data, ready,
                  audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                  ac97_synch, ac97_bit_clock, mode_in, mode_out, insource);
// {{{

   input clock_27mhz;
   input reset;
   input [4:0] volume;
   output [15:0] audio_in_data;
   input [15:0] audio_out_data;
   output ready;

   //ac97 interface signals
   output audio_reset_b;
   output ac97_sdata_out;
   input ac97_sdata_in;
   output ac97_synch;
   input ac97_bit_clock;

   input mode_in, mode_out;
   input insource;

   wire [2:0] source;
   assign source = {insource, 2'b00};

   wire [7:0] command_address;
   wire [15:0] command_data;
   wire command_valid;
   wire [19:0] left_in_data, right_in_data;
   wire [19:0] left_out_data, right_out_data;

   reg audio_reset_b;
   reg [9:0] reset_count;

   //wait a little before enabling the AC97 codec
   always @(posedge clock_27mhz) begin
      if (reset) begin
         audio_reset_b = 1'b0;
         reset_count = 0;
      end else if (reset_count == 1023)
        audio_reset_b = 1'b1;
      else
        reset_count = reset_count+1;
```

```
    end

    wire ac97_ready;
    ac97 ac97(ac97_ready, command_address, command_data, command_valid,
              left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
              right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
              ac97_bit_clock);

    // generate two pulses synchronous with the clock: first capture, then
       ready
    reg [2:0] ready_sync;
    always @ (posedge clock_27mhz) begin
      ready_sync <= {ready_sync[1:0], ac97_ready};
    end
    assign ready = ready_sync[1] & ~ready_sync[2];


    // generate 750Hz sine wave
    wire [19:0] sine_data;
    tone750hz t7h (.clock(clock_27mhz), .ready(ready), .pcm_data(sine_data));


    reg [15:0] out_data;
    always @ (posedge clock_27mhz)
      if (ready) out_data <= audio_out_data;
    assign audio_in_data = (mode_in) ? sine_data[19:4] : left_in_data[19:4];
    assign left_out_data = (mode_out) ? {audio_in_data, 4'b0000} : {out_data,
       4'b0000};
    assign right_out_data = left_out_data;

    // generate repeating sequence of read/writes to AC97 registers
    ac97commands cmds(clock_27mhz, ready, command_address, command_data,
                      command_valid, volume, source);
endmodule
// }}}

// assemble/disassemble AC97 serial frames
module ac97 (ready,
             command_address, command_data, command_valid,
             left_data, left_valid,
             right_data, right_valid,
             left_in_data, right_in_data,
             ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);
// {{{

    output ready;
    input [7:0] command_address;
    input [15:0] command_data;
    input command_valid;
    input [19:0] left_data, right_data;
    input left_valid, right_valid;
    output [19:0] left_in_data, right_in_data;

    input ac97_sdata_in;
```

```
input ac97_bit_clock;
output ac97_sdata_out;
output ac97_synch;

reg ready;

reg ac97_sdata_out;
reg ac97_synch;

reg [7:0] bit_count;

reg [19:0] l_cmd_addr;
reg [19:0] l_cmd_data;
reg [19:0] l_left_data, l_right_data;
reg l_cmd_v, l_left_v, l_right_v;
reg [19:0] left_in_data, right_in_data;

initial begin
   ready <= 1'b0;
   // synthesis attribute init of ready is "0";
   ac97_sdata_out <= 1'b0;
   // synthesis attribute init of ac97_sdata_out is "0";
   ac97_synch <= 1'b0;
   // synthesis attribute init of ac97_synch is "0";

   bit_count <= 8'h00;
   // synthesis attribute init of bit_count is "0000";
   l_cmd_v <= 1'b0;
   // synthesis attribute init of l_cmd_v is "0";
   l_left_v <= 1'b0;
   // synthesis attribute init of l_left_v is "0";
   l_right_v <= 1'b0;
   // synthesis attribute init of l_right_v is "0";

   left_in_data <= 20'h00000;
   // synthesis attribute init of left_in_data is "00000";
   right_in_data <= 20'h00000;
   // synthesis attribute init of right_in_data is "00000";
end

always @(posedge ac97_bit_clock) begin
   // Generate the sync signal
   if (bit_count == 255)
     ac97_synch <= 1'b1;
   if (bit_count == 15)
     ac97_synch <= 1'b0;

   // Generate the ready signal
   if (bit_count == 128)
     ready <= 1'b1;
   if (bit_count == 2)
     ready <= 1'b0;

   // Latch user data at the end of each frame. This ensures that the
```

```
    // first frame after reset will be empty.
    if (bit_count == 255)
      begin
         l_cmd_addr <= {command_address, 12'h000};
         l_cmd_data <= {command_data, 4'h0};
         l_cmd_v <= command_valid;
         l_left_data <= left_data;
         l_left_v <= left_valid;
         l_right_data <= right_data;
         l_right_v <= right_valid;
      end

    if ((bit_count >= 0) && (bit_count <= 15))
      // Slot 0: Tags
      case (bit_count[3:0])
        4'h0: ac97_sdata_out <= 1'b1;       // Frame valid
        4'h1: ac97_sdata_out <= l_cmd_v;    // Command address valid
        4'h2: ac97_sdata_out <= l_cmd_v;    // Command data valid
        4'h3: ac97_sdata_out <= l_left_v;   // Left data valid
        4'h4: ac97_sdata_out <= l_right_v; // Right data valid
        default: ac97_sdata_out <= 1'b0;
      endcase

    else if ((bit_count >= 16) && (bit_count <= 35))
      // Slot 1: Command address (8-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;

    else if ((bit_count >= 36) && (bit_count <= 55))
      // Slot 2: Command data (16-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;

    else if ((bit_count >= 56) && (bit_count <= 75))
      begin
         // Slot 3: Left channel
         ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
         l_left_data <= { l_left_data[18:0], l_left_data[19] };
      end
    else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
         ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
    else
      ac97_sdata_out <= 1'b0;

    bit_count <= bit_count+1;

end // always @ (posedge ac97_bit_clock)

always @(negedge ac97_bit_clock) begin
    if ((bit_count >= 57) && (bit_count <= 76))
      // Slot 3: Left channel
      left_in_data <= { left_in_data[18:0], ac97_sdata_in };
    else if ((bit_count >= 77) && (bit_count <= 96))
      // Slot 4: Right channel
      right_in_data <= { right_in_data[18:0], ac97_sdata_in };
```

```
   end

endmodule
// }}}

// issue initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                     command_valid, volume, source);
// {{{

   input clock;
   input ready;
   output [7:0] command_address;
   output [15:0] command_data;
   output command_valid;
   input [4:0] volume;
   input [2:0] source;

   reg [23:0] command;
   reg command_valid;

   reg [3:0] state;

   initial begin
      command <= 4'h0;
      // synthesis attribute init of command is "0";
      command_valid <= 1'b0;
      // synthesis attribute init of command_valid is "0";
      state <= 16'h0000;
      // synthesis attribute init of state is "0000";
   end

   assign command_address = command[23:16];
   assign command_data = command[15:0];

   wire [4:0] vol;
   assign vol = 31-volume;  // convert to attenuation

   always @(posedge clock) begin
      if (ready) state <= state+1;

      case (state)
        4'h0: // Read ID
          begin
             command <= 24'h80_0000;
             command_valid <= 1'b1;
          end
        4'h1: // Read ID
          command <= 24'h80_0000;
        4'h3: // headphone volume
          command <= { 8'h04, 3'b000, vol, 3'b000, vol };
        4'h5: // PCM volume
          command <= 24'h18_0808;
        4'h6: // Record source select
```

```
            command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
          4'h7: // Record gain = max
            command <= 24'h1C_0F0F;
          4'h9: // set +20db mic gain
            command <= 24'h0E_8048;
          4'hA: // Set beep volume
            command <= 24'h0A_0000;
          4'hB: // PCM out bypass mix1
            command <= 24'h20_8000;
          default:
            command <= 24'h80_0000;
        endcase // case(state)
    end // always @ (posedge clock)
endmodule // ac97commands
// }}}

module volume (reset, clock, up, down, vol, disp);
// {{{
    input reset, clock, up, down;
    output [3:0] vol;
    output [39:0] disp;

    reg [3:0] vol;
    reg [39:0] disp;
    reg old_up, old_down;

    always @(posedge clock)
      if (reset)
        begin
            vol <= 0;
            old_up <= 0;
            old_down <= 0;
        end
      else
        begin
            if ((up == 1) && (old_up == 0) && (vol < 15))
              vol <= vol+1;
            else if ((down == 1) && (old_down == 0) && (vol > 0))
              vol <= vol-1;
            old_up <= up;
            old_down <= down;
        end

    always @(vol)
      case (vol[3:1])
        0: disp <= { 5{8'b00000000}};
        1: disp <= { 5{8'b01000000}};
        2: disp <= { 5{8'b01100000}};
        3: disp <= { 5{8'b01110000}};
        4: disp <= { 5{8'b01111000}};
        5: disp <= { 5{8'b01111100}};
        6: disp <= { 5{8'b01111110}};
        7: disp <= { 5{8'b01111111}};
      endcase
```

```
endmodule
// }}}

//
   ////////////////////////////////////////////////////////////////////////////

//
// generate PCM data for 750hz sine wave (assuming f(ready) = 48khz)
//
//
   ////////////////////////////////////////////////////////////////////////////


module tone750hz (clock, ready, pcm_data);
// {{{
   input clock;
   input ready;
   output [19:0] pcm_data;

   reg [8:0] index;
   reg [19:0] pcm_data;

   initial begin
      // synthesis attribute init of old_ready is "0";
      index <= 8'h00;
      // synthesis attribute init of index is "00";
      pcm_data <= 20'h00000;
      // synthesis attribute init of pcm_data is "00000";
   end

   always @(posedge clock) begin
      if (ready) index <= index+1;
   end

   // one cycle of a sinewave in 64 20-bit samples
   always @(index) begin
      case (index[5:0])
        6'h00: pcm_data <= 20'h00000;
        6'h01: pcm_data <= 20'h0C8BD;
        6'h02: pcm_data <= 20'h18F8B;
        6'h03: pcm_data <= 20'h25280;
        6'h04: pcm_data <= 20'h30FBC;
        6'h05: pcm_data <= 20'h3C56B;
        6'h06: pcm_data <= 20'h471CE;
        6'h07: pcm_data <= 20'h5133C;
        6'h08: pcm_data <= 20'h5A827;
        6'h09: pcm_data <= 20'h62F20;
        6'h0A: pcm_data <= 20'h6A6D9;
        6'h0B: pcm_data <= 20'h70E2C;
        6'h0C: pcm_data <= 20'h7641A;
        6'h0D: pcm_data <= 20'h7A7D0;
        6'h0E: pcm_data <= 20'h7D8A5;
        6'h0F: pcm_data <= 20'h7F623;
```

```
            6'h10: pcm_data <= 20'h7FFFF;
            6'h11: pcm_data <= 20'h7F623;
            6'h12: pcm_data <= 20'h7D8A5;
            6'h13: pcm_data <= 20'h7A7D0;
            6'h14: pcm_data <= 20'h7641A;
            6'h15: pcm_data <= 20'h70E2C;
            6'h16: pcm_data <= 20'h6A6D9;
            6'h17: pcm_data <= 20'h62F20;
            6'h18: pcm_data <= 20'h5A827;
            6'h19: pcm_data <= 20'h5133C;
            6'h1A: pcm_data <= 20'h471CE;
            6'h1B: pcm_data <= 20'h3C56B;
            6'h1C: pcm_data <= 20'h30FBC;
            6'h1D: pcm_data <= 20'h25280;
            6'h1E: pcm_data <= 20'h18F8B;
            6'h1F: pcm_data <= 20'h0C8BD;
            6'h20: pcm_data <= 20'h00000;
            6'h21: pcm_data <= 20'hF3743;
            6'h22: pcm_data <= 20'hE7075;
            6'h23: pcm_data <= 20'hDAD80;
            6'h24: pcm_data <= 20'hCF044;
            6'h25: pcm_data <= 20'hC3A95;
            6'h26: pcm_data <= 20'hB8E32;
            6'h27: pcm_data <= 20'hAECC4;
            6'h28: pcm_data <= 20'hA57D9;
            6'h29: pcm_data <= 20'h9D0E0;
            6'h2A: pcm_data <= 20'h95927;
            6'h2B: pcm_data <= 20'h8F1D4;
            6'h2C: pcm_data <= 20'h89BE6;
            6'h2D: pcm_data <= 20'h85830;
            6'h2E: pcm_data <= 20'h8275B;
            6'h2F: pcm_data <= 20'h809DD;
            6'h30: pcm_data <= 20'h80000;
            6'h31: pcm_data <= 20'h809DD;
            6'h32: pcm_data <= 20'h8275B;
            6'h33: pcm_data <= 20'h85830;
            6'h34: pcm_data <= 20'h89BE6;
            6'h35: pcm_data <= 20'h8F1D4;
            6'h36: pcm_data <= 20'h95927;
            6'h37: pcm_data <= 20'h9D0E0;
            6'h38: pcm_data <= 20'hA57D9;
            6'h39: pcm_data <= 20'hAECC4;
            6'h3A: pcm_data <= 20'hB8E32;
            6'h3B: pcm_data <= 20'hC3A95;
            6'h3C: pcm_data <= 20'hCF044;
            6'h3D: pcm_data <= 20'hDAD80;
            6'h3E: pcm_data <= 20'hE7075;
            6'h3F: pcm_data <= 20'hF3743;
        endcase // case(index[5:0])
    end // always @ (index)
endmodule
// }}}

// vim fdm=marker
```

12 VHDL Listing: rs232.vhd

```
----------------------------------------------------------------------------------
--  RS232RefCom.vhd
----------------------------------------------------------------------------------
-- Author:  Dan Pederson
--          Copyright 2004 Digilent, Inc.
----------------------------------------------------------------------------------
-- Description:         This file defines a UART which tranfers data from
--                              serial form to parallel form and vice versa.
----------------------------------------------------------------------------------
-- Revision History:
--  07/15/04 (Created) DanP
--       02/25/08 (Created) ClaudiaG: made use of the baudDivide constant
--
                    in the Clock Dividing Processes
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Rs232RefComp is
    Port (
                TXD     : out std_logic         := '1';
        RXD     : in  std_logic;
        CLK     : in  std_logic;
                                --Master Clock
                DBIN    : in  std_logic_vector (7 downto 0);    --Data Bus in
                DBOUT : out std_logic_vector (7 downto 0);      --Data Bus out
                RDA     : inout std_logic;
                                --Read Data Available
                TBE     : inout std_logic       := '1';                 --
                    Transfer Bus Empty
                RD              : in  std_logic;
                                  --Read Strobe
                WR              : in  std_logic;
                                  --Write Strobe
                PE              : out std_logic;
                                  --Parity Error Flag
                FE              : out std_logic;
                                  --Frame Error Flag
                OE              : out std_logic;
                                  --Overwrite Error Flag
                RST             : in  std_logic := '0');        --Master Reset
end Rs232RefComp;

architecture Behavioral of Rs232RefComp is
```

```
-------------------------------------------------------------------------
-- Component Declarations
-------------------------------------------------------------------------

-------------------------------------------------------------------------
--  Local Type Declarations
-------------------------------------------------------------------------

        --Receive state machine
        type rstate is (
                strIdle,                        --Idle state
                strEightDelay,  --Delays for 8 clock cycles
                strGetData,             --Shifts in the 8 data bits, and
                    checks parity
                strCheckStop            --Sets framing error flag if Stop bit
                    is wrong
        );

        type tstate is (
                sttIdle,                        --Idle state
                sttTransfer,    --Move data into shift register
                sttShift                        --Shift out data
                );

        type TBEstate is (
                stbeIdle,
                stbeSetTBE,
                stbeWaitLoad,
                stbeWaitWrite
                );


-------------------------------------------------------------------------
-- Signal Declarations
-------------------------------------------------------------------------

        constant baudDivide : std_logic_vector(7 downto 0) := "01011000";
                --Baud Rate dividor, set now for a rate of 9600.
```

```
    signal rdReg     :  std_logic_vector(7 downto 0) := "00000000";
                        --Receive holding register
    signal rdSReg    :  std_logic_vector(9 downto 0) := "1111111111";
                        --Receive shift register
    signal tfReg     :  std_logic_vector(7 downto 0);
                                                      --Transfer holding
        register
    signal tfSReg    :  std_logic_vector(10 downto 0)          := "
       11111111111";       --Transfer shift register
    signal clkDiv    :  std_logic_vector(8 downto 0) := "000000000";
              --used for rClk
    signal rClkDiv :  std_logic_vector(3 downto 0)  := "0000";
                           --used for tClk
    signal ctr       :  std_logic_vector(3 downto 0) := "0000";
                                 --used for delay times
    signal tfCtr     :  std_logic_vector(3 downto 0) := "0000";
                           --used to delay in transfer
    signal rClk      :  std_logic := '0';
                           --Receiving Clock
    signal tClk      :  std_logic;
                                 --Transfering Clock
    signal dataCtr :  std_logic_vector(3 downto 0)  := "0000";
              --Counts the number of read data bits
    signal parError:  std_logic;
                                 --Parity error bit
    signal frameError: std_logic;
                                 --Frame error bit
    signal CE               :  std_logic;
                                          --Clock enable for the
       latch
    signal ctRst     :  std_logic := '0';
    signal load      :  std_logic := '0';
    signal shift     :  std_logic := '0';
    signal par       :  std_logic;
signal tClkRST       :  std_logic := '0';
    signal rShift    :  std_logic := '0';
    signal dataRST :  std_logic := '0';
    signal dataIncr:  std_logic := '0';

    signal strCur    :  rstate       := strIdle;
              --Current state in the Receive state machine
    signal strNext   :  rstate;
                                 --Next state in the Receive state
       machine
    signal sttCur    :  tstate := sttIdle;
              --Current state in the Transfer state machine
    signal sttNext :  tstate;
                                 --Next state in the Transfer staet
        machine
    signal stbeCur :  TBEstate := stbeIdle;
    signal stbeNext:  TBEstate;

-------------------------------------------------------------------------

-- Module Implementation
```

```
begin
        frameError <= not rdSReg(9);
        parError <= not ( rdSReg(8) xor (((rdSReg(0) xor rdSReg(1)) xor (
            rdSReg(2) xor rdSReg(3))) xor ((rdSReg(4) xor rdSReg(5)) xor (
            rdSReg(6) xor rdSReg(7)))) );
        DBOUT <= rdReg;
        tfReg <= DBIN;
        par <=  not ( ((tfReg(0) xor tfReg(1)) xor (tfReg(2) xor tfReg(3)))
            xor ((tfReg(4) xor tfReg(5)) xor (tfReg(6) xor tfReg(7))) );

--Clock Dividing Functions--

        process (CLK, clkDiv)
                --set up clock divide for rClk
            begin
                    if (Clk = '1' and Clk'event) then
                            if (clkDiv = baudDivide) then
                                    clkDiv <= "000000000";
                            else
                                    clkDiv <= clkDiv +1;
                            end if;
                    end if;
            end process;

        process (clkDiv, rClk, CLK)
                --Define rClk
        begin
                if CLK = '1' and CLK'Event then
                        if clkDiv = baudDivide then
                                rClk <= not rClk;
                        else
                                rClk <= rClk;
                        end if;
                end if;
        end process;

        process (rClk)
                                --set up clock divide for tClk
            begin
                    if (rClk = '1' and rClk'event) then
                            rClkDiv <= rClkDiv +1;
                    end if;
            end process;

        tClk <= rClkDiv(3);
                                    --define tClk

        process (rClk, ctRst)
                --set up a counter based on rClk
            begin
                    if rClk = '1' and rClk'Event then
                            if ctRst = '1' then
```

```
                                ctr <= "0000";
                        else
                                ctr <= ctr +1;
                        end if;
                end if;
        end process;

process (tClk, tClkRST)
                --set up a counter based on tClk
        begin
                if (tClk = '1' and tClk'event) then
                        if tClkRST = '1' then
                                tfCtr <= "0000";
                        else
                                tfCtr <= tfCtr +1;
                        end if;
                end if;
        end process;

--This process controls the error flags--
process (rClk, RST, RD, CE)
        begin
                if RD = '1' or RST = '1' then
                        FE <= '0';
                        OE <= '0';
                        RDA <= '0';
                        PE <= '0';
                elsif rClk = '1' and rClk'event then
                        if CE = '1' then
                                FE <= frameError;
                                OE <= RDA;
                                RDA <= '1';
                                PE <= parError;
                                rdReg(7 downto 0) <= rdSReg (7 downto
                                    0);
                        end if;
                end if;
        end process;

--This process controls the receiving shift register--
process (rClk, rShift)
        begin
                if rClk = '1' and rClk'Event then
                        if rShift = '1' then
                                rdSReg <= (RXD & rdSReg(9 downto 1));
                        end if;
                end if;
        end process;

--This process controls the dataCtr to keep track of shifted values--
process (rClk, dataRST)
        begin
                if (rClk = '1' and rClk'event) then
                        if dataRST = '1' then
```

```
                        dataCtr <= "0000";
                    elsif dataIncr = '1' then
                        dataCtr <= dataCtr +1;
                    end if;
                end if;
        end process;

    --Receiving State Machine--
    process (rClk, RST)
        begin
            if rClk = '1' and rClk'Event then
                if RST = '1' then
                    strCur <= strIdle;
                else
                    strCur <= strNext;
                end if;
            end if;
        end process;

    --This process generates the sequence of steps needed receive the data

    process (strCur, ctr, RXD, dataCtr, rdSReg, rdReg, RDA)
        begin
            case strCur is

                when strIdle =>
                    dataIncr <= '0';
                    rShift <= '0';
                    dataRst <= '0';

                    CE <= '0';
                    if RXD = '0' then
                        ctRst <= '1';
                        strNext <= strEightDelay;
                    else
                        ctRst <= '0';
                        strNext <= strIdle;
                    end if;

                when strEightDelay =>
                    dataIncr <= '0';
                    rShift <= '0';
                    CE <= '0';

                    if ctr(2 downto 0) = "111" then
                        ctRst <= '1';
                        dataRST <= '1';
                        strNext <= strGetData;
                    else
                        ctRst <= '0';
                        dataRST <= '0';
                        strNext <= strEightDelay;
                    end if;
```

```
                        when strGetData =>
                                CE <= '0';
                                dataRst <= '0';
                                if ctr(3 downto 0) = "1111" then
                                        ctRst <= '1';
                                        dataIncr <= '1';
                                        rShift <= '1';
                                else
                                        ctRst <= '0';
                                        dataIncr <= '0';
                                        rShift <= '0';
                                end if;

                                if dataCtr = "1010" then
                                        strNext <= strCheckStop;
                                else
                                        strNext <= strGetData;
                                end if;

                        when strCheckStop =>
                                dataIncr <= '0';
                                rShift <= '0';
                                dataRst <= '0';
                                ctRst <= '0';

                                CE <= '1';
                                strNext <= strIdle;

                end case;

        end process;

--TBE State Machine--
process (CLK, RST)
        begin
                if CLK = '1' and CLK'Event then
                        if RST = '1' then
                                stbeCur <= stbeIdle;
                        else
                                stbeCur <= stbeNext;
                        end if;
                end if;
        end process;

--This process gererates the sequence of events needed to control the
    TBE flag--
process (stbeCur, CLK, WR, DBIN, load)
        begin

                case stbeCur is

                        when stbeIdle =>
                                TBE <= '1';
                                if WR = '1' then
```

```
                                        stbeNext <= stbeSetTBE;
                                else
                                        stbeNext <= stbeIdle;
                                end if;

                        when stbeSetTBE =>
                                TBE <= '0';
                                if load = '1' then
                                        stbeNext <= stbeWaitLoad;
                                else
                                        stbeNext <= stbeSetTBE;
                                end if;

                        when stbeWaitLoad =>
                                if load = '0' then
                                        stbeNext <= stbeWaitWrite;
                                else
                                        stbeNext <= stbeWaitLoad;
                                end if;

                        when stbeWaitWrite =>
                                if WR = '0' then
                                        stbeNext <= stbeIdle;
                                else
                                        stbeNext <= stbeWaitWrite;
                                end if;
                        end case;
                end process;

    --This process loads and shifts out the transfer shift register--
    process (load, shift, tClk, tfSReg)
            begin
                    TXD <= tfsReg(0);
                    if tClk = '1' and tClk'Event then
                            if load = '1' then
                                    tfSReg (10 downto 0) <= ('1' & par &
                                        tfReg(7 downto 0) &'0');
                            end if;
                            if shift = '1' then

                                    tfSReg (10 downto 0) <= ('1' & tfSReg
                                        (10 downto 1));
                            end if;
                    end if;
            end process;

    --  Transfer State Machine--
    process (tClk, RST)
            begin
                    if (tClk = '1' and tClk'Event) then
                            if RST = '1' then
                                    sttCur <= sttIdle;
                            else
                                    sttCur <= sttNext;
```

```
                        end if;
                    end if;
                end process;

        -- This process generates the sequence of steps needed transfer the
            data--
        process (sttCur, tfCtr, tfReg, TBE, tclk)
                begin

                    case sttCur is

                        when sttIdle =>
                                tClkRST <= '0';
                                shift <= '0';
                                load <= '0';
                                if TBE = '1' then
                                        sttNext <= sttIdle;
                                else
                                        sttNext <= sttTransfer;
                                end if;

                        when sttTransfer =>
                                shift <= '0';
                                load <= '1';
                                tClkRST <= '1';
                                sttNext <= sttShift;


                        when sttShift =>
                                shift <= '1';
                                load <= '0';
                                tClkRST <= '0';
                                if tfCtr = "1100" then
                                        sttNext <= sttIdle;
                                else
                                        sttNext <= sttShift;
                                end if;
                    end case;
                end process;

end Behavioral;
```

13 Verilog Listing: debounce.v

```
/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * debounce.v             *
 * debounce module        *
 * ********************** */

/* this is the button debounce module from many labs throughout the course */
```

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

//
// Button Debounce Module
//
//
   ////////////////////////////////////////////////////////////////////////////


module debounce (reset, clock, noisy, clean);

   input reset, clock, noisy;
   output clean;

   reg [18:0] count;
   reg new, clean;

   always @(posedge clock)
     if (reset)
       begin
          count <= 0;
          new <= noisy;
          clean <= noisy;
       end
     else if (noisy != new)
       begin
          new <= noisy;
          count <= 0;
       end
     else if (count == 270000)
       clean <= new;
     else
       count <= count+1;

endmodule
```

14 Verilog Listing: display_16hex.v

```
/* ********************** *
 * Parametric Equalizer   *
 * J. Colosimo            *
 * 6.111 - Fall '10       *
 * display_16hex.v        *
 * 16-digit hex display   *
 * ********************** */

//
   ////////////////////////////////////////////////////////////////////////////

//
// 6.111 FPGA Labkit --- Hex display driver
//
// File:    display_16hex.v
```

```
// Date:    24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
//
   ////////////////////////////////////////////////////////////////////////


module display_16hex (reset, clock_27mhz, data,
                      disp_blank, disp_clock, disp_rs, disp_ce_b,
                      disp_reset_b, disp_data_out);

   input reset, clock_27mhz;     // clock and reset (active high reset)
   input [63:0] data;            // 16 hex nibbles to display

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

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

   //
      ////////////////////////////////////////////////////////////////////////

   //
   // Display Clock
   //
   // Generate a 500kHz clock for driving the displays.
   //
   //
      ////////////////////////////////////////////////////////////////////////


   reg [4:0] count;
   reg [7:0] reset_count;
   reg clock;
   wire dreset;

   always @(posedge clock_27mhz)
     begin
        if (reset)
          begin
             count = 0;
             clock = 0;
          end
        else if (count == 26)
          begin
```

```
            clock = ~clock;
            count = 5'h00;
          end
        else
          count = count+1;
    end

  always @(posedge clock_27mhz)
    if (reset)
      reset_count <= 100;
    else
      reset_count <= (reset_count==0) ? 0 : reset_count-1;

  assign dreset = (reset_count != 0);

  assign disp_clock = ~clock;

  //
     ////////////////////////////////////////////////////////////////////////

  //
  // Display State Machine
  //
  //
     ////////////////////////////////////////////////////////////////////////


  reg [7:0] state;              // FSM state
  reg [9:0] dot_index;          // index to current dot being clocked out
  reg [31:0] control;           // control register
  reg [3:0] char_index;         // index of current character
  reg [39:0] dots;              // dots for a single digit
  reg [3:0] nibble;             // hex nibble of current character

  assign disp_blank = 1'b0; // low <= not blanked

  always @(posedge clock)
    if (dreset)
      begin
          state <= 0;
          dot_index <= 0;
          control <= 32'h7F7F7F7F;
      end
    else
      casex (state)
        8'h00:
          begin
            // Reset displays
            disp_data_out <= 1'b0;
            disp_rs <= 1'b0; // dot register
            disp_ce_b <= 1'b1;
            disp_reset_b <= 1'b0;
            dot_index <= 0;
            state <= state+1;
```

```
    end

  8'h01:
    begin
      // End reset
      disp_reset_b <= 1'b1;
      state <= state+1;
    end

  8'h02:
    begin
      // Initialize dot register (set all dots to zero)
      disp_ce_b <= 1'b0;
      disp_data_out <= 1'b0; // dot_index[0];
      if (dot_index == 639)
        state <= state+1;
      else
        dot_index <= dot_index+1;
    end

  8'h03:
    begin
      // Latch dot data
      disp_ce_b <= 1'b1;
      dot_index <= 31;          // re-purpose to init ctrl reg
      disp_rs <= 1'b1; // Select the control register
      state <= state+1;
    end

  8'h04:
    begin
      // Setup the control register
      disp_ce_b <= 1'b0;
      disp_data_out <= control[31];
      control <= {control[30:0], 1'b0}; // shift left
      if (dot_index == 0)
        state <= state+1;
      else
        dot_index <= dot_index-1;
    end

  8'h05:
    begin
      // Latch the control register data / dot data
      disp_ce_b <= 1'b1;
      dot_index <= 39;          // init for single char
      char_index <= 15;         // start with MS char
      state <= state+1;
      disp_rs <= 1'b0;          // Select the dot register
    end

  8'h06:
    begin
      // Load the user's dot data into the dot reg, char by char
```

```
            disp_ce_b <= 1'b0;
            disp_data_out <= dots[dot_index]; // dot data from msb
            if (dot_index == 0)
              if (char_index == 0)
                state <= 5;                   // all done, latch data
              else
              begin
                char_index <= char_index - 1; // goto next char
                dot_index <= 39;
              end
            else
              dot_index <= dot_index-1;       // else loop thru all dots
        end

    endcase

always @ (data or char_index)
  case (char_index)
    4'h0:             nibble <= data[3:0];
    4'h1:             nibble <= data[7:4];
    4'h2:             nibble <= data[11:8];
    4'h3:             nibble <= data[15:12];
    4'h4:             nibble <= data[19:16];
    4'h5:             nibble <= data[23:20];
    4'h6:             nibble <= data[27:24];
    4'h7:             nibble <= data[31:28];
    4'h8:             nibble <= data[35:32];
    4'h9:             nibble <= data[39:36];
    4'hA:             nibble <= data[43:40];
    4'hB:             nibble <= data[47:44];
    4'hC:             nibble <= data[51:48];
    4'hD:             nibble <= data[55:52];
    4'hE:             nibble <= data[59:56];
    4'hF:             nibble <= data[63:60];
  endcase

always @(nibble)
  case (nibble)
    4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
    4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
    4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
    4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
    4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
    4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
    4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
    4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
    4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
    4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
    4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
    4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
    4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
    4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
    4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
    4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
```

```
    endcase

endmodule
```

15 Python Listing: makewave.py

```
#!/usr/bin/env python

from math import *

def f(x):
    # 750hz notch
    #if x in range(12,40) or x in range(20,60): return 0;
    #else: return 255;

    # quadratic notch
    #return (1*(x-16))**2

    # high pass
    #return x**2

    # like am radio
    """
    if x in range(5,400):
        return -(1/60.*(x-115))**2+255
    else:
        return 0
    """

    # antialiasing
    if x < 32: return 0.5 * x**2;
    elif x < 256: return 80*log( - x + 256 );
    else: return 0

    #_ = 10
    #if x > _ : return 0;
    #else: return 255*


###############

import string, sys

N = 1024;
y = [];

for x in xrange(N):
    # make a symmetric graph
    if x <= N/2:
        _ = f(x);
    else:
        _ = f(N-x);

    y.append(str( max( 0, min( 255, int(round(_)) ) ) ));
```

```
_ = string.join(y, "\n")

if len(sys.argv) < 2:
    print _;
else:
    f = open(sys.argv[1], 'w');
    f.write(_);
    f.close();
```

16 Python Listing: xfer.py

```
#!/usr/bin/env python2

import serial, sys

def convert(d):
    return chr(int(d));

if len(sys.argv) < 2:
    sys.exit(1);

if len(sys.argv) == 3:
    f = open(sys.argv[2], 'r');
    data = f.readlines();
    print ":: read data from", sys.argv[2];
else:
    data = sys.stdin.readlines();
    print ":: read data from stdin"

if len(data) == 1: data = data[0].split(' ');

cxn = serial.Serial(sys.argv[1], 9600, parity=serial.PARITY_ODD);
#cxn = serial.Serial(sys.argv[1], 9600);
print ":: opened", cxn.portstr;

tx = "";
for d in data:
    tx += convert(d);

cxn.write(tx);

print "done"
sys.exit(0);
```