Real-time Visual Audio Composition

Tyler Hutchison, Bryan Newbold, Dimitri Turbiner
Professor: Chris Terman
TA: Ben Gelb

December 11, 2008

Abstract

We present a hardware design to convert captured images to an audio stream. There are a wealth of real time *software* implementations of the Fast Fourier Transform (FFT), but we use a Field Programable Gate Array (FPGA) and an NTSC video camera to converted captured images to clear audio output over a 0 to 10kHz bandwidth. Basic image processing operations such as inversion and thresholding enhance the definition of lines and shapes (such as those sketched by hand on a white-board) so that characteristic sounds are recognizable by ear after conversion to audio. Additionally, an easy to use XVGA GUI interface allows concurent frequency spectrum visualization. The implemented system is a Real-time Audio Visual Composition system with almost all of the previously proposed functionality.

Contents

List of Figures

Overview

Author: Tyler Hutchison

The Real-Time Audio Composition system allows for the conversion of visually encoded information into audio. A typical visual encoding program, such as Baudline, takes audio input and produces a scrolling visualization of the fast-Fourier transform (FFT). Our system could take one of these visualizations and produce the audio encoded in the image. A screenshot of a visual spectrograph is provided in Figure 10.

The two-dimensional spectrograph represents a series of frequency spectra which have been converted into lines. Magnitude is represented by intensity variation where white is the highest magnitude and black is the lowest.

In order to obtain an image from the world, a camera has been provided. Using the graphical user interface (GUI), users can capture single frames from the camera which serve as the visual spectrograph to interpreted and played back through headphones or speakers. A PS/2 mouse allows for control of the GUI system. Using the mouse, the user can interact with the buttons, the image itself, and a number of other features. An image of the GUI is shown in Figure 1.

The GUI allows for a number of functions. An edit function, regrettably never fully implemented due to time constraints, would allow for the direct editing of captured images by manually selecting an eight bit binary intensity value and applying that value to the image. By holding down the mouse button, multiple pixels could be edited by dragging the mouse. The Bode Plot in the lower right of the GUI functions similarly. Using the mouse, a user can adjust the attenuation of different frequencies in the image. Values range from 0 to 1, so no gain can be applied. If the user holds down the mouse button and drags the mouse within the bode plot, the 128 frequency domain taps will be adjusted depending on the userâs desire. The attenuation of particular frequencies will be represented by an intensity shift on the image as well, to provide feedback for the userâs attenuation adjustments. The time domain application of the bode plot also was never implemented, but the theoretical fading provides an interesting interface.

The five provided rectangular buttons are all relatively simple. If the user clicks a button, the button will activate and turn yellow. Another click deactivates the button. The top button, *Edit*, allows for user editing of the captured image which has been stored in memory. *Image* and *Overlay* store images in their private memories for display. *Enable*, enables the captured overlay image. The overlay image, if enabled, will play back through the audio channel as well as display as a red pixel shift on the image so the user can determine which particular areas have been modified by overlay, and which are from the original image. *Pause* ceases the playing of audio until it has been toggled again.

Another sprite on the image also will toggle pause, but only while the mouse button is being

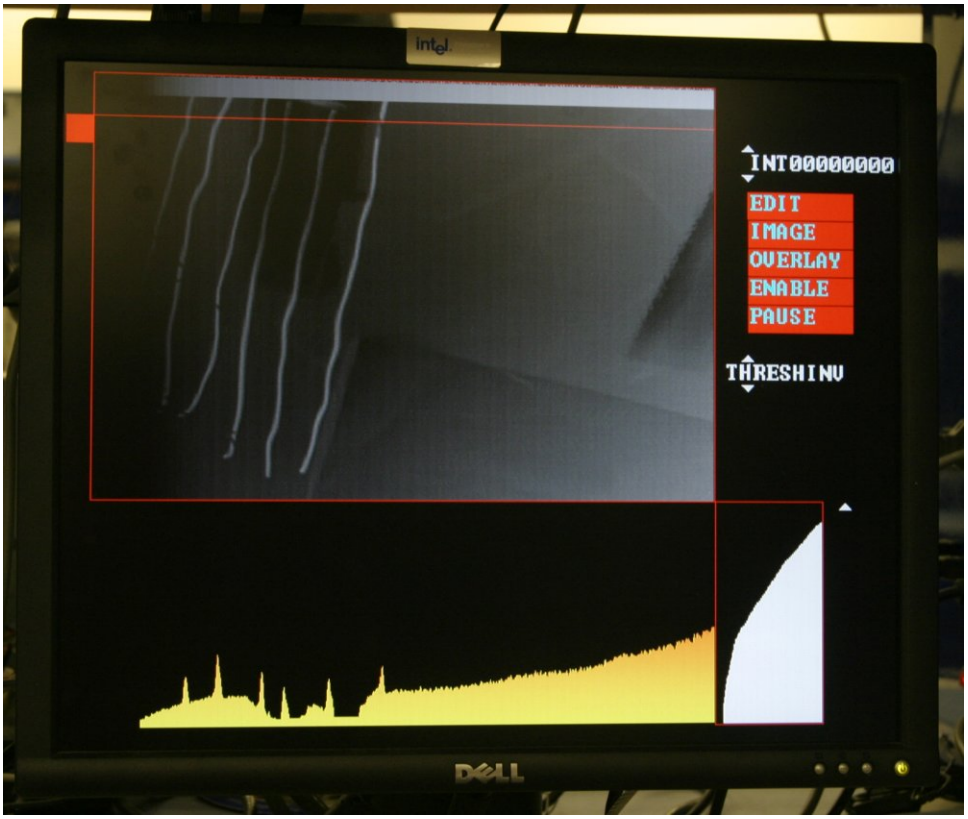

Figure 1: A photograph of the GUI implemented in the Real Time Audio Composition system. Each area of the screen will be explained in detail.
(Source: Team Member)

held down. One should notice the red block on the left side of the image with a thin red line cutting across the image. This scan bar represents the current line of the visual spectrograph that the audio channel is playing. When the block on the left side is selected and dragged by the mouse, the system will pause its play until the scan bar has been released. The scan bar can be dragged from the top of the image to the bottom to select a new location to start play. The dragging of the scan bar will also rapidly update the frequency spectrum display on the bottom of the screen. One will also notice that this spectrum updates while the system is playing as well. As mentioned above, the image merely represents line representations of many frequency spectra. The spectrum at the bottom is the magnitude versus frequency plot of the particular indexed line. Adjusting the bode plot will also be represented in the modification of this image.

None of the GUI manipulation would be possible without an intricate memory management system. The camera sends its output to the ADV7185 module which produces LLC data which can then be decoded from NTSC/PAL to YCrCb. Our monitor is functioning in RGB so this must be further decoded. In addition, as the system requires, an image should not be held in memory until an image capture has been requested. If image capture is requested, a frame is stored in a main memory, and if an overlay capture is requested, a frame is stored in a secondary memory. To avoid memory further memory problems, the display will not access memory during a store phase so there may be slight refresh delay. In addition to capturing an image, memory must be accessed by the audio output stage, the display stage, and overwritten in the case of a user edit. The GUI/Memory interface communicates using capture flags, edit flags, and display flags. Since the Audio/Memory interface operates at a much slower frequency (~24kHz), the data demand is much less.

The audio output stage also involves a number of stages. First, the frequency data is slightly modified and window to fill the pipelined buffer that leads to the Inverse FFT (IFFT). Memory must then be transformed to the time domain using the Inverse FFT (IFFT). To avoid aliasing from trailing data points inherent to the IFFT process, the data is low pass filtered. The bode plot values, stored in the GUI, would be theoretically sent through an IFFT to obtain time domain taps which can then be convolved with the time domain audio data to produce the audio signal. The AC97 audio chip in the FPGA runs at a clock speed of 48kHz so the audio samples must also be up-sampled to produce the final signal.

Audio Pipeline

Author: Bryan Newbold

The audio pipeline section of the project requests data from image memory, constructs appropriate frequency domain samples to feed into the inverse Fourier transform module, filters the resulting audio waveform, and passes samples to the digital-to-analog conversion (DAC) audio chip. Settings and parameters (including bode filter tap values) are accepted from the graphical user interface and labkit switches and buttons. The sampling rates and transform lengths at every stage were chosen to optimize frequency resolution and smoothness of playback when translating from the camera image to the range of human audio sensitivity.

The pipeline is driven by requests from the audio output chip; after initialization every link in the chain supplies a single sample at a time, one input for every output. Figure 2 is a block diagram showing the entire audio pipeline's modules and interconnecting wires; each stage will be discussed separately below.

Bandwidth and Timing

The entire audio pipeline runs synchronously at the roughly 65MHz clock rate specified for XVGA video output. The chain is driven by sample requests from the audio DAC chip. The AC97 audio codec specifies data rate of 48,000 samples/second, which allows the synthesis of signals up to 24kHz. The upper limit of human perception is around 20kHz, but most adults have difficultly discerning tones above 10kHz, so we decided to further cut our maximum frequency in half by up-sampling a 24,000 samples/second data stream (which synthesizes signals of up to ~10kHz) to the 48,000 samples/second request rate by repeating every sample once and applying a 31-tap filter to removing any resulting artifacts (not to be confused with the user selectable 128-tap bode filter).

This clock/sampling rate combination means that the system has more than 2700 clock cycles to deliver each sample, which is plenty of time to implement per-cycle filtering, interpolation, and any other calculations.

The inverse fast Fourier transform block requires an input vector with a power of two input length. Since the NTSC video camera has a 720 pixel horizontal resolution, an input vector with at least 1024 bins is required. Aliasing considerations mean this number must be doubled to 2048 (otherwise frequencies above 512 would alias back down to the lower frequencies). Because the IFFT module is a "Streaming/Pipelined" processing type, for every frequency domain value

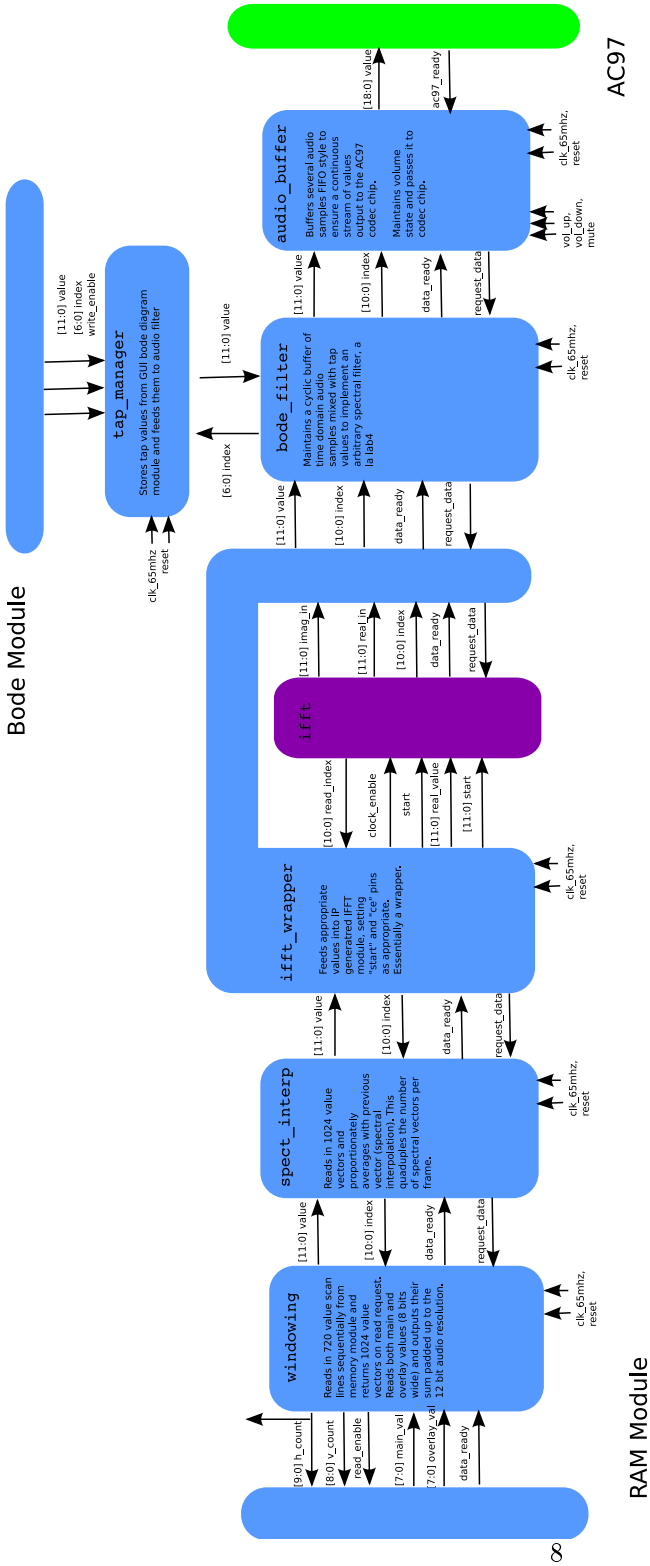

Figure 2: Audio Pipeline Block Diagram*(Source: Team Member)*

input, a time domain value is made available on the output. Because there is a many-value offset between the input and output samples, there is an initialization period at start up when the IFFT is streamed a sequence of values before any valid output is ready.

Windowing and Interpolation

The spectral interpolation module (aspect_interp) accepts full 2048 sample frequency domain vectors, corresponding to a single horizontal scan line of audio, and either passes the vector samples straight through or can double or quadruple repeat vectors, effectively halving the vertical scroll speed of audio processing. This option is toggled by switch #0 on the Labkit. The original intention was that neighboring lines would be blended together, smoothing transitions between vectors, but this feature was never functional.

The window module synthesizes these 2048 sample vectors by padding the 720 pixel scan line with zeros before and after, and requests the actual data samples from the memory modules. The window module also keeps track of the vertical offset of the audio pipeline; image pixels are read from memory one at a time as two 8-bit intensity values (one for "main", one for "overlay"). If the overlay_enable wire is active, then the main and overlay values are summed, otherwise only the main value is taken. The 9-bit combination is left shifted to fit into the positive range of the 12-bit frequency bin size.

IFFT Module

The inverse fast Fourier transform (IFFT) module was auto-generated using Xilinx FPGA design tools using the settings in Table 1. However, a full wrapper module was necessary to load samples in and out of the module, enable and disable processing, ensure sample index synchronization, etc. The ifft_wrapper module ensures that only the 12-bit real components are written and that both the 8-bit phase and 12-bit complex components are tied to 0 at input and ignored at output.

Setting	Value
Generator Version	Fast Fourier Transform 3.2
Input Length	2048 samples
Processing Mode	Pipelined
Sample bitwidth	12-bit signed
Output ordering	Natural
Clock enable pin	Enabled
Processing Stages	3 using BRAM

Table 1: IFFT Auto-generation Settings

The clock_enable pin on the IFFT module is used to control data flow: when a time-domain sample is requested from downstream, a frequency-domain sample is first requested from upstream and held at the real input of the IFFT. Then the clock_enable is pulled high for a single 65MHz clock cycle, causing the frequency-domain sample to shift in. This cycling

repeats until the data_valid pin indicates that there is a valid time-domain sample on the output side of the IFFT.

The greatest difficulty in getting the auto-generate IFFT module to work correctly was managing internal bit rescaling correctly. A number of different scaling modes and parameters can be specified, but the only configuration that worked for us was to return the full unscaled 23-bit signed integer, and let the user select which 11 bits should be selected for audio output (the most significant bit is always the sign bit in 2's complement form). For instances when only a handful of frequency bins are saturated and all other bins are zero, the magnitude of the IFFT output would be very low, and the top 10 bits of the output would be zeros. But in cases where a large square wave frequency vector was input, or, more commonly, there is broadband noise in the image data background, the IFFT output could be saturated up to the top two or three significant bits of data; selecting a window without the most significant bits results in essentially white noise output. The sliding window offset can be adjusted by the user with the left and right labkit buttons.

Bode Filtering

The bode filter stage was intended to apply user specified spectral filtering in the time domain: a frequency domain bode curve would be drawn by the user with the mouse in the GUI, then this curve would be inverse Fourier transformed to get 128 12-bit signed time domain tap values, which would be applied to the audio stream coming out of the main IFFT module. The tap_manager module would serve as a memory buffer for time-domain tap values, which would be generated and written to by the GUI, and read out by the bode_filter module in the audio pipeline. The bode_filter module stores a buffer of the last 128 12-bit signed time-domain audio samples, and passes out the dot product of this buffer with the tap values to implement the filtering. Only the most significant bits of the output sum would be passed on as valid values.

The bode filtering feature was never fully tested with actual tap values, but seemed to give the expected results when the first tap was specified all high (0xFF) and all other taps zero (0x00): this filter value simply passes through the most recent time-domain audio sample from the buffer.

AC97 Output

The AC97 codec, including command values, sample bursting, and parameter configuration, was mostly implemented using lab staff software (as credited in the source code). Aside from resizing and passing on the final audio samples and volume settings, the ac97_manager module up-samples the 24,000 samples per second to the 48,000 samples per second specified by the AC97 codec. This up-conversion is implemented by using each time-domain sample twice, then low pass filtering the output to remove the high frequency artifacts introduced. This low pass filter also conveniently filters out any noise or aliasing occurring in the frequencies above the 720 or so lowest frequencies actually specified by image data. Note that the up-sampling process also effectively halves the frequency corresponding to each bin in our frequency-domain spectral vectors, giving better frequency resolution in the lower, more human discernible regime.

The low pass filter is implemented almost exactly like in Lab 3, using 31 tap filters generated in matlab using the command:

```
round(fir1(30,.5)*(2^13))
```

Each tap is 14-bits signed, thus the scaling by 2^{13}. The parameter $W_n = 0.5$ specifies a cutoff frequency around 9kHz.

The actual AC97 output chip in the labkit is a National Semiconductor LM4550.

Graphic User Interface

Author: Tyler Hutchison

Display

The display is at a resolution of 1024 by 768 pixels. To provide the specified resolution at a refresh rate of 60 Hz, the VGA output is clocked at 65 MHz, where a single clock cycle increments the 11 bit counter, hcount, which holds the particular x position of the raster scan. To provide the proper sync timing to the monitor, hcount actually counts to 1343. Timing specifications for monitor display resolutions and refresh rates are extremely particular and strict. Similar to hcount, vcount holds the current y location. One detail that is important to note is that the upper left corner of the monitor is the origin so the raster scan increases in vcount as it moves down the screen. Also, vcount, again to provide the proper syncing, must count to 776. Many of the defined images are in simple 3 bit RGB format, where one bit represents either red, green or blue. The main image, however, is displayed in 8 bit grayscale, that is, each channel, red, green, and blue, have the same value. This modification provides for a higher contrast image which is imperative in providing an interpretable FFT.

PS/2 Mouse

The high level mouse module provides user input to the GUI. From the module, an 11 bit absolute x location, an 11 bit absolute y location, and a 3 bit button status (left, right, and middle buttons), can be utilized. There is approximately a 400 microsecond delay between byte packets to allow for completion so the transfer of three packets for the x, y, and button positions provides for a clock significantly less than the 65 MHz clock of the GUI. This still allows for smooth mouse scrolling. The updated PS/2 mouse module provided by the 6.111 staff has not been modified.

GUI Control

The GUI Control module provides the main control for the GUI interface. In the module, input checks are made depending on mouse location to provide for button toggling, edit intensity adjustment, image editing, bode plot editing, and scan bar drags. Figure 3 shows a simple figure displaying the sprite interactions with the GUI.

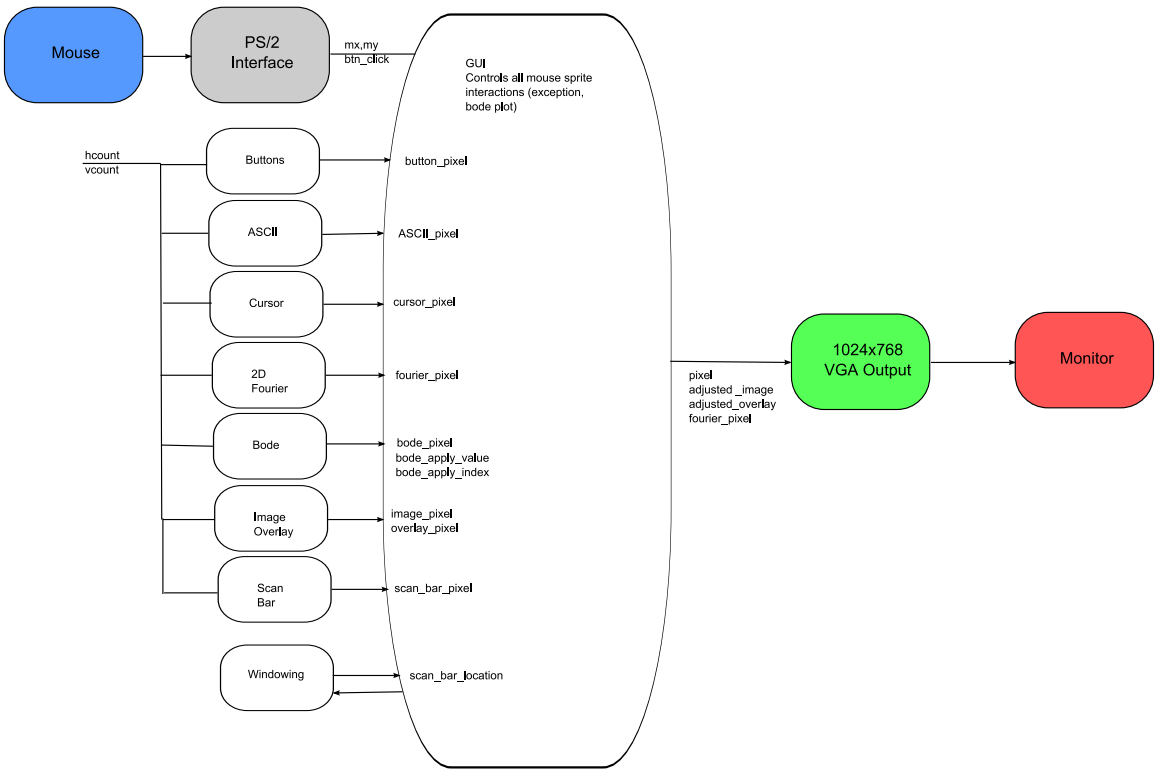

Figure 3: The many pixels being fed into the GUI which are OR-ed together to produce a pixel value at a particular location.
(Source: Team Member)

Based on location parameters for the various image types, the controller determines if the mouse location falls within the bounds of the image. Depending on the button input, different results can occur. I will discuss these interactions one at a time.

Intensity Arrows A mouse click will increment an 8 bit binary number if the top arrow is clicked, or decrement if the bottom arrow is clicked.

Edit Mode A mouse click will enable the edit mode function. Enabling edit mode provides for user modification of the ZBT main memory, in which the main image is stored. Clicking on the image at this point will modify individual pixels. If the mouse button is clicked, the 8 bit intensity value modified in the upper right (i.e. INT00000000) will be written to memory. To do so, an edit request flag will be set and the resulting pixel location and update value will be sent to a FIFO buffer to await memory availability. Memory management will be discussed in great detail later. If the user clicks and drags the mouse multiple pixels will be updated using a basic linear regression pattern. The multiple pixel locations and values will be stored in the FIFO for update. As previously noted, the functionality of the edit module was never fully realized.

Image A mouse click will enable image capture which sets write enable on memory to tell it to update the main ZBT ram in which the main image is stored.

Overlay A mouse click will enable overlay capture which is wired to memory to tell it to update the overlay ZBT ram in which the overlay image is stored.

Enable A mouse click will enable overlay viewing and playback. The red values of the RGB output will now contain 8 bit overlay pixel values added to the 8 bit main image pixel values with overflow detection.

Pause A mouse click will activate pause which wires to the audio output and stops the IFFT module. If pause is set low again, the audio output will begin at the current location of the scan bar.

Scan Bar The scan bar allows for an additional way to pause the system, as well as replay certain streams. At a basic level, the scan bar offers a reference to the user to see the point of the current audio playback. The current horizontal scan line of the audio playback is fed to the GUI window which, unless paused, updates the scan bars y location. If the system is paused, therefore, the scan bar ceases to move. Also, if the mouse scrolls over the scan bar block on the left side of the image and the mouse is clicked and held, the user may adjust the scan bar to any location in the image. The output system will finish the current line of play, and resume when the scan bar is unselected at the particular location that the scan bar sets. To do so, not only does the audio output communicate its 10 bit location with the scan bar, but the scan bar, if pause is set, communicates its 10 bit location back to audio.

Sprites

A number of simple sprites have been used to control many of the communication signals within the system and provide a comfortable interface for the user.

Buttons

Buttons are simple rectangles which are determined by an upper left coordinate on the screen designated by an initial (x,y) parameter. The only difference between a button and an ordinary rectangle is the change in pixel color when the button is activated. As noted above, the GUI controls the logic that determines if a button has been activated, and that activation signal controls a number of other modules.

Triangles

Triangles serve as both the mouse cursor and the intensity adjustment. By clicking the mouse, the color changes in the mouse cursor, but the intensity adjustors remain a solid color by simply setting the default color parameter and the clicking color parameter to be the same.

ASCII Display

An 8 bit ASCII ROM provides the font that is used in the GUI. The cstring_disp module, provided by the 6.111 staff and unmodified, sets the location of particular text. Depending on the location of the hcount, vcount raster scan the character x and y locations and the string to display are modified to write different strings in different locations. Since the largest string to be displayed is 11 characters, an 88 bit register holds the current string value which is to be displayed in a particular location. Each of the strings are hardcoded into memory using ASCII addresses. The exceptions are those of the intensity display and the image processing functionality. The intensity display would have been used by the edit mode to select specific intensity values for user selected pixels. The image processing display shows the current state of the image processing system. The state toggles between, NOPROC for no processing, THRESH for thresholding, and INV for inverse image capture. These particular image processing functions will be discussed further in the memory section. The ASCII ROM encodes for 12 vertical, 8 horizontal bit characters which cstring_disp doubles in dimension.

Fourier Display

The most important consideration concerning the Fourier display is that the values currently indexed by the scan bar will not be readily available when the raster scan reaches the Fourier display section. To avoid re-addressing the heavily used ZBT RAMs to display this image, the scan bar module takes the current 10 bit y location of the scan bar as an input. When the display?s vcount is at the location of the scan bar the 720, 8 bit values are stored in a BRAM so the display can be generated within the appropriate location. See Figure 4 below for a block diagram of the Fourier display including major inputs and outputs.

When the raster scan reaches the Fourier display, the memory is indexed by the hcount (offset by the x shift of the Fourier display) and, since the Fourier display is 255 pixels high, the representative y value is checked against the vcount location with respect to the top of the image. All pixels under this value are also drawn to attempt to display an image a bit smoother than a series of disjointed pixels. The color of the pixel is scaled from red (low) to yellow (high).

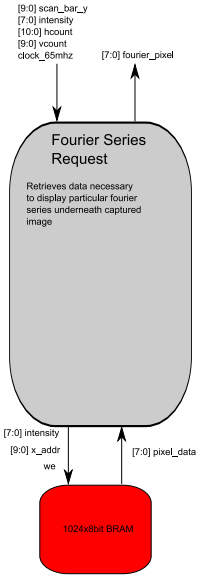

Figure 4: Block diagram for the two dimensional Fourier display including major input and output ports
(Source: Team Member)

This scaling means that, for Fourier pixels that should be drawn, the RGB values correspond to red at 255, green scaled depending on value, and blue 0.

Image and Overlay

The image and overlay viewing provides an interesting way in which the user can quickly modify the image without tedious edit mode adjustments. The display checks to see if the user wishes to display the overlay using the overlay enable wire. If it is enabled, the 8 bit image and overlay pixels are summed. If the value overflows the 8 bit buffer, the value is saturated at 255, otherwise the overlay is added to the red value of the VGA RGB output. Identical math is computed to check for overflow in the two-dimensional Fourier display as well.

Bode Display

The bode display works similarly to edit mode, a separate adjustment mode is not necessary. While the inputs and outputs of the bode function are relatively simple as shown in Figure 5, the implementation was challenging.

If the user simply clicks on the bode plot, the 10 bit x and 9 bit y values are recorded as well as the relative location within the bode plot corresponding to a new tap value. This tap value is stored in a 128 tap array, each 8 bits wide. The bode location itself is exactly 255 pixels high,

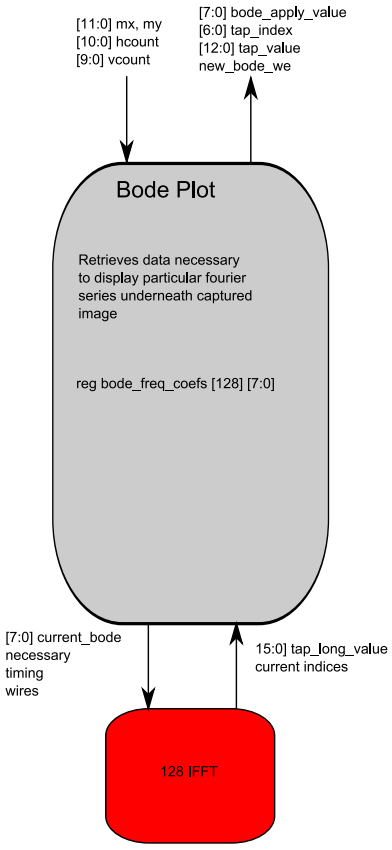

Figure 5: Block diagram of the bode plot module including major inputs and outputs *(Source: Team Member)*

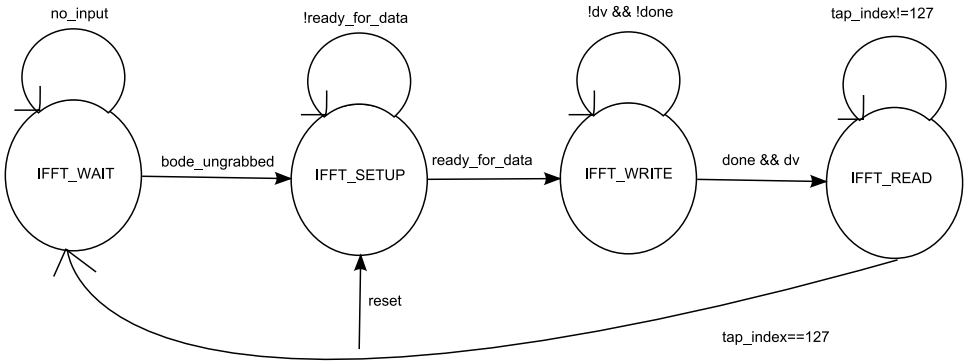

Figure 6: FSM describing the control of the various inputs and outputs of the IFFT to produce time domain tap values.
(Source: Team Member)

so locations directly correspond to the set value. If the user leaves the mouse button depressed as the mouse is moved, values are filled in around the mouse. We assumed that the user would want to be careful in drawing the bode function so values are filled in quite simply. Since the delta x and delta y components can be greater than one, the mouse may not actually record a value at each of the 128 taps, so the skipped tap values are simply assigned by a value hold method. This strategy means that the value at the current x index is applied to all the x indices in between delta x values until a new mouse location is updated. These frequency coefficients are fed back into the GUI module where they apply the specified attenuation on the image. The eight bit values, representing 1 to 0 are multiplied by the image and overlay pixel values and the most significant eight bits are selected to produce the displayed pixel data.

The Bode plot also computes the IFFT of the frequency domain coefficients so that the time domain taps could be convolved with the audio sequence to produce filtered data. While the communication between the bode tap manager and the bode plot module was never fully implemented, the IFFT output works to specification. A 128 tap IFFT is used which runs by radix-4 burst mode, natural order. This description means that values can be loaded whenever a ready for data flag is high, and as soon as the entire packet length is filled (128 values in our case), the IFFT is computed. Natural order refers to the ordering of bits on the output. Reverse ordering of bits is the natural mode of the IFFT, and since timing is not a huge concern at this stage, producing natural order bits is worth the time. The different stages of the IFFT input and output are controlled by an FSM as shown in Figure 6.

Unless just reset, at which the FSM is immediately clock enabled and put in the setup state so the time domain taps can be initialized, the FSM naturally idles in the wait state. With the edge detection of a button press release within the bode plot, the FSM enters the setup state since it can be assumed that a value within the bode plot has changed. As soon as the ready for data flag is set high, the FSM enters the write state. The values from the bode array are fed into the IFFT, lagging behind three cycles as designated by the datasheet. After all data has been entered, the FSM enters the read state. The resulting tap indices and values are

completely synchronous at this point and can be fed directly into the tap manager which will hold the resultant values.

Memory Management

Author: Dimitri Turbiner

The memory control module uses four interface modules to communicate with the Camera, GUI, and IFFT subsystems and two interface modules to control the physical SRAM chips on the LabKit. The camera_to_zbt interface module, keeps track of the sync signals coming from the Camera subsystem in order to compute the memory destination address. Incomming pixels are buffered and packed together into groups of four. When such a group of four is ready, the memory controller is signalled by camera_we. The input signals are all synchronized to the clock in the Camera subsystem – 27mHz, while the output signals have to be synchronized at the memory clock – 65mHz. Thus, the camera_to_zbt module has to perform clock resynchronization between the two subsystems it connects. The zbt_to_display, zbt_to_ifft, edit_to_zbt, modules all compute the memory destination address based on the requested pixel horizontal and vertical index (hcount and vcount), and when signalled by the memory controller that data is valid, pack or unpack four 8bit pixels to/from the 36 bit data bus.

The memory controller controls the two zbt ram access modules depending on which memory IO operations have been requested by the subsystems. The controller supports latching the various subsystem's data and address busses in order to guarantee minimum hold delays and to resolve simultaneous memory access requests. The latches control logic is built around a memory access priority hierarchy:

```
assign main_addr = camera_capture_request ?  camera_addr :
                   (display_read_request ?  display_addr :
                   (ifft_read_request ?  ifft_addr :
                   (edit_write_request ?  edit_addr :0)));
```

Camera Processing

The NTSC data stream is decoded by adv7185_decode_ntsc into a CCYR665 stream of pixels (Y,Cr,Cb 10bits each). A new pixel arrives with each posedge of pixel_ready. The 30bit YCrCb pixel value then converted into an 8bit pixel value by the image_processing module. The user selects in the GUI the desired conversion operations.

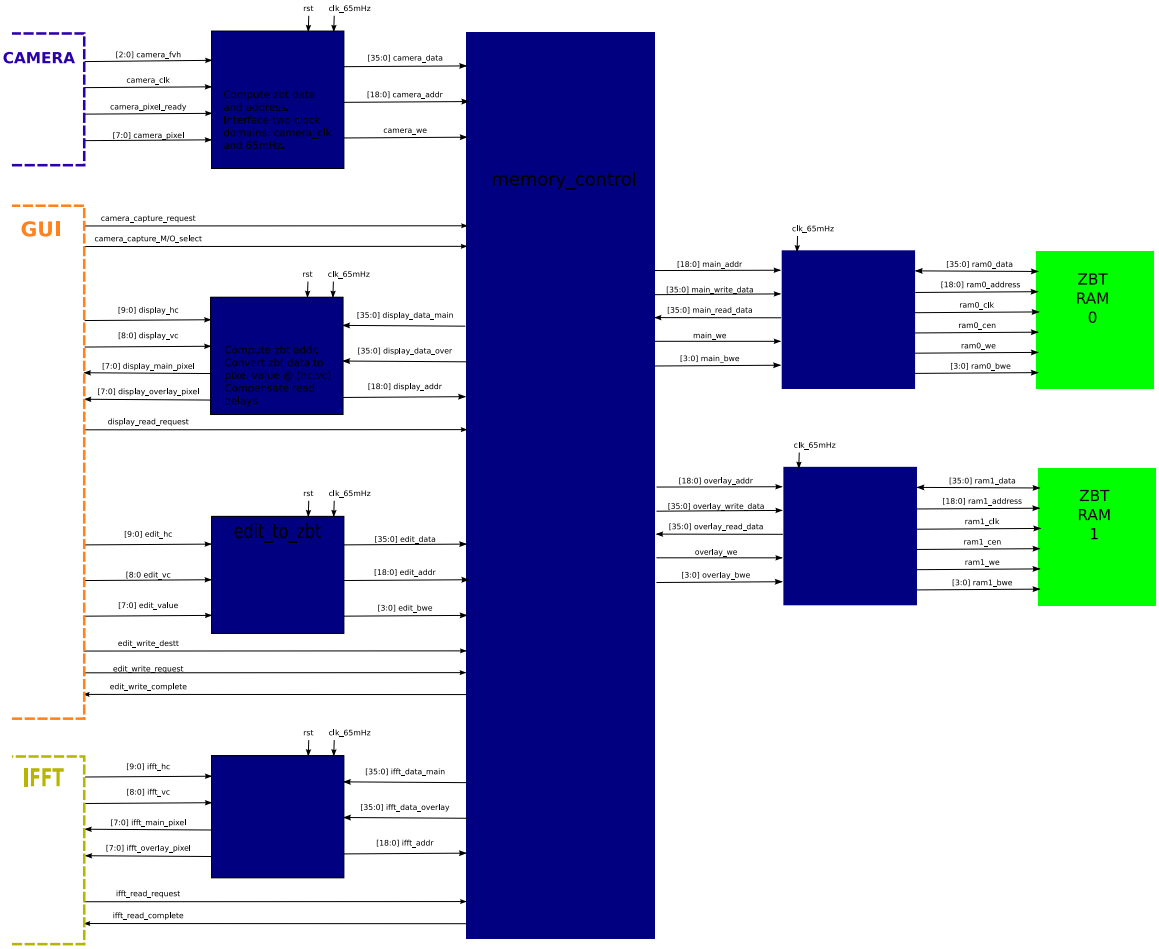

Figure 7: Memory Controller Block Diagram*(Source: Team Member)*

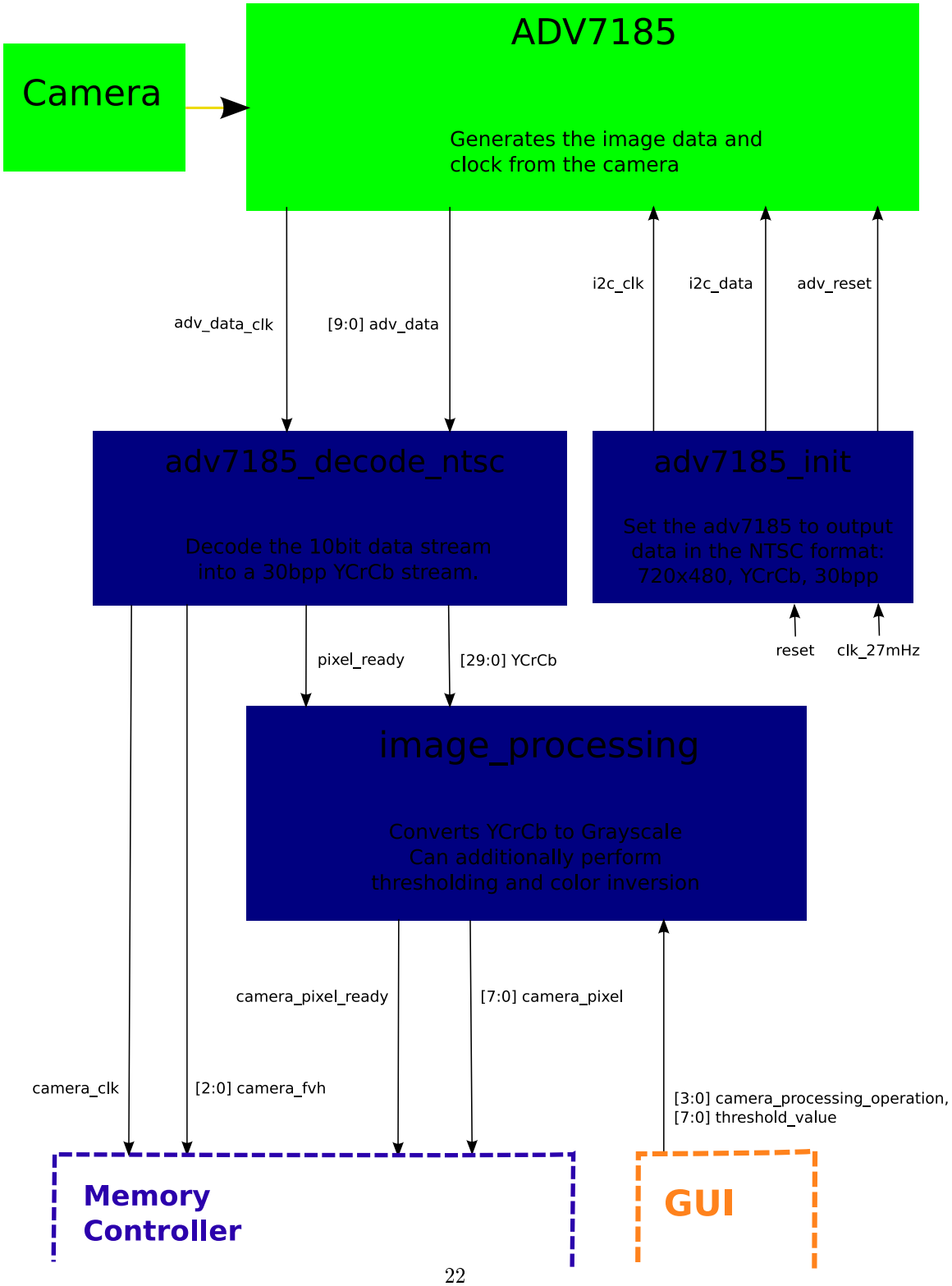

Figure 8: Camera Block Diagram*(Source: Team Member)*

Default	pixel_value=Y (simple grayscale)
invert:	pixel_value = 255-pixel_value
threshold	pixel_value= (pixel_value<threshold)? 0 : pixel_value
smooth threshold	like threshold, but with a smoother thresholding S-curve

Table 2: Available Processing Settings

The memory controller receives in addition to the 8bit pixel stream four sync signals:

- camera_pixel_ready signals a the arrival of a new pixel taking into account the added latency of pixel conversion,
- an end of horizontal line is signalled by h,
- an end of vertical line in signalled by,
- f indicates whether the pixel is part of an even or odd interlaced image field.

Testing and Debugging

Author: Tyler Hutchison and Bryan Newbold

The three components of the project, GUI, Audio, and Memory, were easily divided so that each component could be tested and debugged separately. This strategy allowed for comprehensive testing before the project approached total integration and long build times. Still, build times were noticeably slower than previous projects which was due to the large IFFT employed to convert the rows of image pixels into audio data.

Display

One of the first main goals was to get the different aspects of the display coordinating with each other. While the main image display relies on memory, each other component could easily function without memory input. First, simple toggle buttons were implemented with little problem. This first step was simple due to the fact that buttons hardly vary from the blobs used previously to generate a rectangle. Mouse interaction with the buttons was the next step. Although we considered this an easy step, it turned out to be a serious time sink early on due to problems with the PS/2 interface. During the normal operation of the mouse, it was seen on the hex display that the x and y address of the mouse would often jump wildly and even include erroneous button inputs. Since the PS/2 module generates its own clock, and must synchronize with the FPGA clock, we decided that it was most likely a timing issue inherent. This discovery was made with the help of Ben and Alex who also managed to discover a new mouse module which quickly integrated with the coded triangle mouse cursor. The triangles on the screen are also a simple sprite, but a couple of iterations were necessary to compensate for math errors in the calculation of the slope along the sides of the pointer. Text was then added to the buttons. An initial, hardcoded text placing module which used a unique .coe file to place text was written, but due to inefficient , messy logic computation, this was scrapped and previously existing code was used which proved to be much simpler to implement. This change proved to allow for the simple implementation of the future intensity values for edit mode and the modifiable image processing values. Despite early frustrations, a simple GUI was propagating signals from the GUI to the labkits LEDs.

The next step was to implement the more complex portions of the GUI. The two dimensional Fourier representation of particular scan lines turned out to be a much simpler process than expected. The 1024 addresses, 8 bit wide BRAM to store the particular indexed scan line worked perfectly on the first test. Trouble arose when the overlay image was added as well. Saturation was going to be a problem since the overlay data and the normal image data were

simply added together, but the logic to test for overflow caused the system to act rather finicky. This problem was solved by a simple, carefully clocked pipeline using a nine bit register to store the summation of the two eight bit values. In all, the quick implementation of the two dimensional Fourier representation allowed for the "fire" effect to be a sure addition, which also turned out to be a quick change. Though, to do so, the returned Fourier pixel had to be changed from 3 to 8 bits which slightly increased number of wires.

The bode plot turned out to be much more troublesome. It was not initially apparent that delta x values of the mouse ps/2 module would vary so greatly. The first implementation used single pixels to record the applied bode filter. Needless to say, this produced an ugly, uninterpretable series of values. After a test of a move and hold routine to create continuous lines between delta x values, the bode plot was interpretable, but, without slow mouse movement, didn't meet standards. Rather than computing exact values to place single pixels in an attempt to make a line, the plot looked much more appealing when all locations less than the value within the plot were given pixel values. After this change, the change was also made to the two dimensional Fourier transform for similar reasons. This gives the appearance of simple line connections within the plot and makes the plot look smoother to the user. More problems arose when the scaled bode plot was applied to the image. Since, at the time, memory was not integrated with the project, the image window was filled with white pixels so that the change across the image with the bode adjustment could be noticed. For still unknown reasons, the original attempt to apply the bode filter to the image failed. Since there are 128 separate bode frequency coefficients and 720 pixels in the image, it was considered reasonable to apply a single bode tap to every six horizontal pixels. Originally, a counter, reset whenever it reached six, would update the index of the bode filter and therefor apply a new multiplier to the pixel. This method failed. It seemed that the counter incremented the index far too often and the fading pattern repeated over and over. After relaying data to the hex display and counting pixels within the fade regions, no bug could be discovered, so eventually, the method was changed. Instead, a register simply held the current hcount value and whenever the value incremented six units beyond that, the index was updated. Though the method seems exactly the same as the old, the new method implemented perfectly on the first attempt. The only rational explanation as to why the original code didn't work was a minor flaw in how the counter was updated which may have been changed too often. An additional minor problem was also noticed after this implementation. When the pixel itself was updated by multiplying the stored bode value by the incoming pixel data, a great deal of scattering across the boundaries of the image was noticed. By pipelining the data within the clocked region, this problem was removed and smooth fading occurred. (Note: the unimplemented edit mode code uses only slightly different math than the bode plot draw function, and should theoretically work, but was never tested).

Audio Pipeline

A number of external tools were crucial for the development and debugging of the audio processing pipeline. An entire separate labkit module with a parameterized dummy sample generator replacing the camera and memory management and no GUI output. This allowed for debugging with consistent, simple, noise free spectral input. This labkit module was also wired to make full use of the hex display (to show bode tap index and both horizontal and vertical memory indexes) and the logic analyzer connections (all 4 16-bit connectors were used!). For

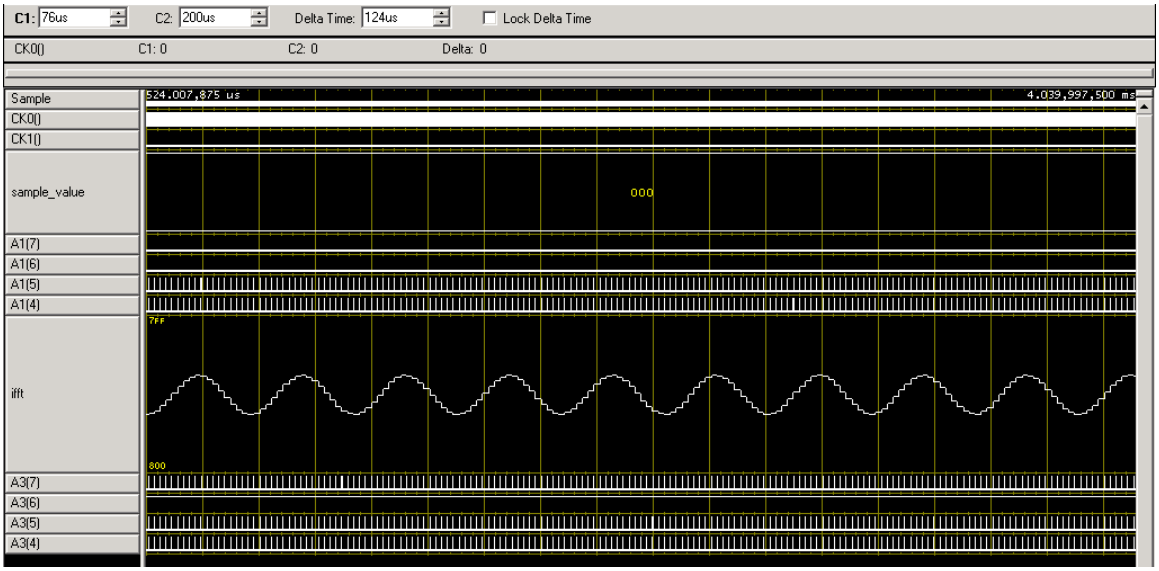

Figure 9: Pure sine waveform debugging on the logic analyzer. See text for description. *(Source: Team Member)*

instance, Figure 9 shows the IFFT module output of a simple sine wave; the clock-like ticks of channels above and below the sine waveform show the request and ready signals of the various other stages of the audio pipeline. Note that this section of the spectral waveform shows the all zero padding from the windowing module; the single non-zero sample_value is off screen to the left, and the dummy ram module request/ready pins are not toggled.

Another very important tool was a laptop running the free baudline[1] spectral analysis program. This software performs a very high quality real-time Fourier transform of microphone or line in audio samples and plots the results on screen as a scrolling spectrograph. This tool allowed analysis of audio output too complicated for our ears to make sense of, and ultimately allowed validation of our project by converting the sound coming out of speakers connected to the labkit and measured by the laptop microphone back into the original two dimensional image. See Figure 10 and the front page of this report for examples. A direct audio cable connection between the labkit headphone output and the laptop microphone input gave the best resolution and did not incorporate background lab noise into the spectrographs, but there is a strong aesthetic appeal to transmitting images by sound.

A last important testing tool was to compare the time-domain waveform generated by our system to the calculated waveform generated with the same frequency-domain input vector. Figure 11 shows the plot of a waveform generated in matlab using the command:

```
a=zeros(1024); a(1:21)=1000; figure(); plot(real(ifft(a)));
```

[1] see http://baudline.com.

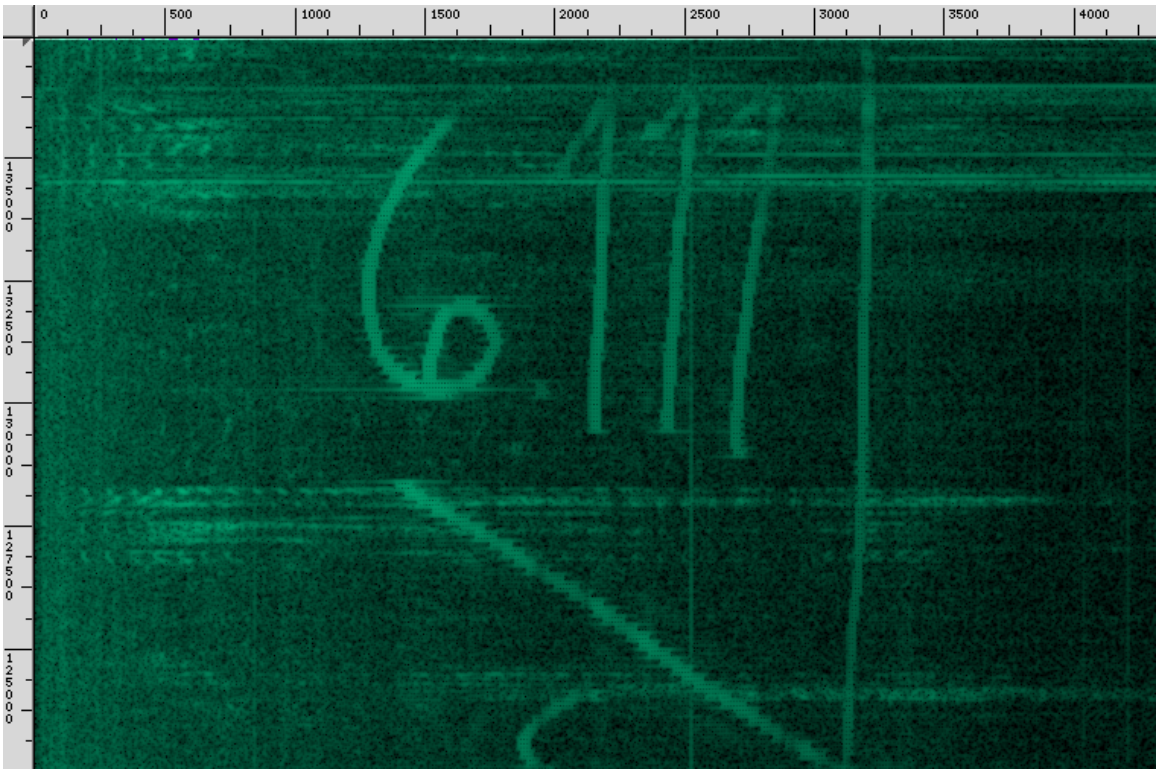

Figure 10: The course number written on a white-board is clearly visible in the spectrograph output; this spectrograph was captured using a speaker and microphone, not a directly wire connection.
(Source: Team Member)

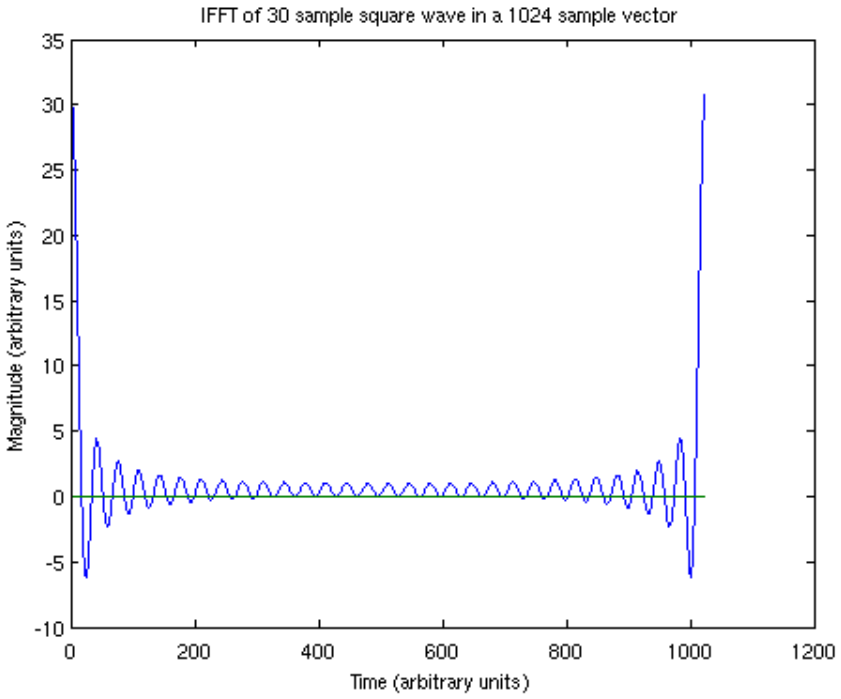

Figure 11: Matlab generated IFFT *(Source: Team Member)*

Figure 12 shows the corresponding time-domain waveform of labkit audio output when passed a 30 out of 1024 sample (not 2048 samples in this case) square wave. The squarewave shape can be seen in the lower, spectrogram portion of the image

A few other buttons and switches came in useful for debugging: Switch #7 disables the regular audio output and plays back a 750Hz tone as generated by the lab staff (very useful for sanity checking that headphones are plugged in and volume is reasonable!). The final low-pass filter and up-sampling can be disabled with Switch #3; disabling the up-sampling will result in audible frequency doubling and double the rate at which spectral samples are processed. The Enter button reset all modules. Switches #4-6 can be used to select different modes of the dummy ram module when that module is in use; some modes are a constant sine output, a 25 sample square wave, a pattern of 3 128 sample square waves, a linearly sweeping sine wave, and the swept sine against a noise background.

Integration

Integration, especially on the memory interface, posed the biggest problem for the group. Though naming conventions dictated by planned block diagrams were adhered to, additional

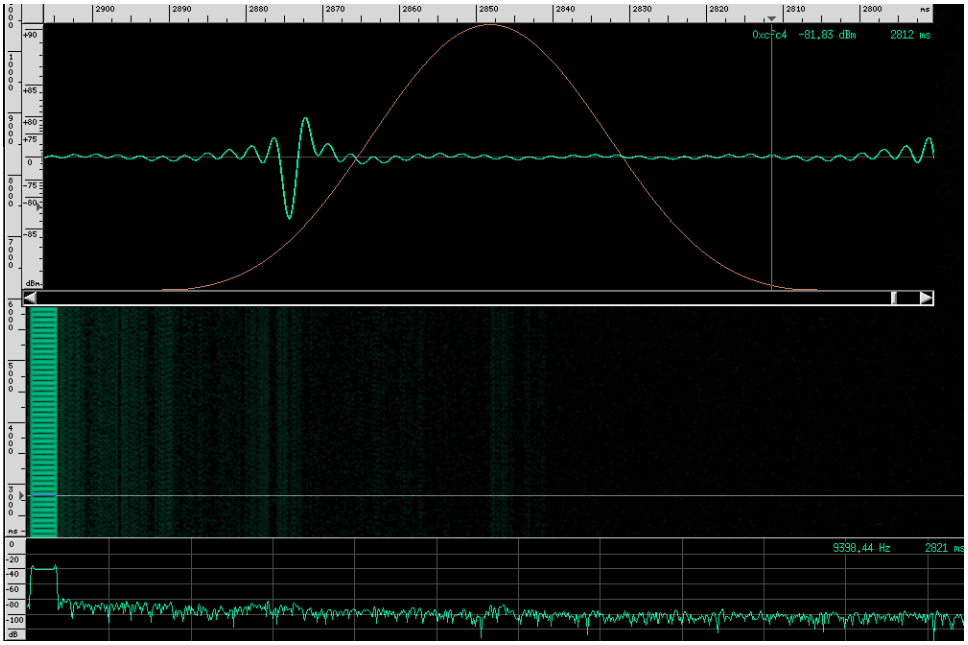

Figure 12: Time-domain waveform of labkit audio output when passed a 30 out of 1024 sample square wave. The squarewave shape can be seen in the lower, spectrogram portion of the image *(Source: Team Member)*

wires had to be passed based on how many functions evolved in the development. One minor fix existed between the audio playback and the pause system. Pause was simple enough, but the pause based on the scan bar wasn't thoroughly considered. The scan bar moves whenever the audio IFFT updates its relative audio vertical count. However, it was also intended that the scan bar can also force a pause and be dragged to an arbitrary location at which the audio will resume. This observation means that the scan bar and the audio playback must communicate their locations and which signal must be trusted at a given time. With the overall "pause" as a function of the scan bar grab and the pause button, this signal became the overriding wire which allowed for the scan line to communicate to the audio functionality. An additional problem that arose out of miscommunication was the use of the bode plot values. It was apparent that the bode plot updating was producing strange functionality, but it turned out that neither the bode plot nor the audio side computed the time domain taps of the frequency coefficients. Though the large IFFT used in the audio computation could have served to compute both tap values and the audio data, the additional thought and timing details that would have to go into such an implementation were considered not worth it due to project timeline constraints. Therefore, a small, additional 128 value wide IFFT was created using the Xilinx coregen software. Unlike the large audio IFFT, this IFFT used burst mode since the number of taps are relatively small and could produce data rather quickly by forcing an entire packet into the module. The bode IFFT was tested as an entirely new module and examined over the logic analyzer. Values forced into the bode frequency coefficient array yielded theoretically accurate results on the logic analyzer after a couple of short iterations over which the finite state machine of the IFFT computation was closely analyzed and appeared to have accurate state transitions. However, due to a shortage of time, the modules were never properly integrated

The integration between the memory and the GUI turned out to be relatively straightforward. This smooth transition was most likely due to the precedence that the image display has over the audio output. While audio integration had a number of problems as will be discussed, the image updates needs pixel values at precise times without using a great deal of extra memory to store large buffers. The memory controller, therefore, gives priority to memory read requests. Since the overlay image and the main image are captured similarly, both integrated quite smoothly. There were minor problems in saturation, as with the two dimensional Fourier representation, but solving it in one case lent itself to the other. Another problem, that again turned out to be more troublesome than was originally predicted was a band of noise around the edge of the camera image. It seemed that some data from the camera would have to be tossed or that perhaps there were addressing issues. The addressing issues will be discussed in the memory testing section, but the eventual solution became one of simple blanking of noisy pixels around the edges. This was extremely necessary since clean data in the lower frequencies was essential to considering the accuracy of the IFFT result.

Conclusion

Author: Tyler Hutchison

The implementation of a real time audio composition system has been described. Using the GUI interface, coupled with complex memory interactions, and robust audio output, one can use visual information to encode the frequency spectrum of an audio sample and play it back. A number of user customizable options exist. These include image overlay, and input image processing. Additionally, the user can easily track to any location in the sample to continue the audio playback which has been filtered and highly tested to achieve high output data which has been displayed using a direct FFT of audio stream. Many of the features also provide an eye-catching, appealing GUI. While the main functionality for additional customization exists, such as the bode application and direct image editing, these portions were not sufficiently debugged.

Due to time constraints, not all desired features were able to be implemented, but many of the components were designed and work to specification. Specific functionality that was not implemented was a more complex bode regression, user definable pole zero plot for another method to consider frequency filtering, bode plot time domain convolution, and the edit mode. Several other components were considered to improve the system as well. If video feed was displayed in the image display box during image capture, it would allow for easier centering and capturing of the image. This would be relatively simple to implement and may not require the use of additional memory if the ZBT used for main memory storage is dumped since it will be assumed that it is about to be overwritten.

The process of constructing and designing this project was an important learning experience. The pre-coding steps, while sometimes tedious, were important in solidifying ideas and provided for easier integration later. Also in construction, we quickly learned that, while some components seem trivial, the actual implementation turned out to be quite challenging because of timing constraints and unexpected results. Surprisingly, though a number of signals were required to operate, the memory management system led to less problems than were expected, most likely due to the low timing requirements of the audio output in comparison to the display.

Appendix: Code Listing

bnewbold_labkit.v

```
////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// Author: Nathan Ickes
//
// Edited: Nov 4, 2008, FTW
// Author: Bryan Newbold
//
////////////////////////////////////////////////////////////////////////////////
module labkit(
  // Remove comment from any signals you use in your design!
  // AC97
  output wire /*beep,*/ audio_reset_b, ac97_synch, ac97_sdata_out,
  input wire ac97_bit_clock, ac97_sdata_in,

  // VGA
  /*
  output wire [7:0] vga_out_red, vga_out_green, vga_out_blue,
  output wire vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync, vga_out_vs
  */
  // NTSC OUT
  /*
  output wire [9:0] tv_out_ycrcb,
  output wire tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
  output wire tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
  output wire tv_out_subcar_reset;
  */
  // NTSC IN
  /*
  input wire [19:0] tv_in_ycrcb,
  input wire tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef, tv_in_hff, tv
```

```
output wire tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, t
inout wire tv_in_i2c_data,
*/
// ZBT RAMS
/*
inout wire [35:0] ram0_data,
output wire [18:0] ram0_address,
output wire ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b,
output wire [3:0] ram0_bwe_b,
inout wire [35:0]ram1_data,
output wire [18:0]ram1_address,
output wire ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b,
output wire [3:0] ram1_bwe_b,
input  wire clock_feedback_in,
output wire clock_feedback_out,
*/
// FLASH
/*
inout wire [15:0] flash_data,
output wire [23:0] flash_address,
output wire flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b,
input wire flash_sts,
*/

// RS232
/*
output wire rs232_txd, rs232_rts,
input wire rs232_rxd, rs232_cts,
*/
// PS2
/*
input wire mouse_clock, mouse_data, keyboard_clock, keyboard_data,
*/
// FLUORESCENT DISPLAY
output wire disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b,
input wire disp_data_in,
output wire disp_data_out,
// BUTTONS, SWITCHES, LEDS
input wire button0,
//input wire button1,
//input wire button2,
//input wire button3,
input wire button_enter,
//input wire button_right,
//input wire button_left,
input wire button_down,
```

```
  input wire button_up,
  input wire [7:0] switch,
  output wire [7:0] led,
  // USER CONNECTORS, DAUGHTER CARD, LOGIC ANALYZER
  //inout wire [31:0] user1,
  //inout wire [31:0] user2,
  //inout wire [31:0] user3,
  //inout wire [31:0] user4,
  //inout wire [43:0] daughtercard,
  output wire [15:0] analyzer1_data, output wire analyzer1_clock,
  output wire [15:0] analyzer2_data, output wire analyzer2_clock,
  output wire [15:0] analyzer3_data, output wire analyzer3_clock,
  output wire [15:0] analyzer4_data, output wire analyzer4_clock,
  // SYSTEM ACE
  /*
  inout wire [15:0] systemace_data,
  output wire [6:0] systemace_address,
  output wire systemace_ce_b, systemace_we_b, systemace_oe_b,
  input wire systemace_irq, systemace_mpbrdy,
  */
  // CLOCKS
  //input wire clock1,
  //input wire clock2,
  input wire clock_27mhz
);
//==== bnewbold headers BEGIN ================================================

   // 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));

   // Reset Generation
   // A shift register primitive is used to generate an active-high reset
   // signal that remains high for 16 clock cycles after configuration finishes
   // and the FPGA's internal clocks begin toggling.
   wire reset;
        wire reset_button;
        wire initial_reset;
   SRL16 #(.INIT(16'hFFFF)) reset_sr(.D(1'b0), .CLK(clock_65mhz), .Q(initial_reset),
                                     .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
```

```
        debounce breset(.reset(reset_button),.clock(clock_65mhz),.noisy(~button0),.clean(reset
        assign reset = reset_button || initial_reset;
    // allow user to adjust volume; passed to audio_buffer in bnewbold_modules.v
    wire vup,vdown;
    debounce bup(.reset(reset),.clock(clock_65mhz),.noisy(~button_up),.clean(vup));
    debounce bdown(.reset(reset),.clock(clock_65mhz),.noisy(~button_down),.clean(vdown));

        wire [63:0] hex_data;
        // Shows current horizontal and vertial counts being read by audio pipeline
        display_16hex display_16hex1 (reset, clock_27mhz, hex_data,
                disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b, disp_data_out);
    // switch 7 for audio tone (debugging)
        // TODO: remove and tie low (1'b0)?
    wire tone_enable;
    debounce sw7(.reset(reset),.clock(clock_65mhz),.noisy(switch[7]),.clean(tone_enable));

    // switch 6,5 for dummy mode selection (debugging)
        // TODO: remove and tie low (2'b00)?
    wire [1:0] dummy_mode;
    debounce sw6(.reset(reset),.clock(clock_65mhz),.noisy(switch[6]),.clean(dummy_mode[1]));
        debounce sw5(.reset(reset),.clock(clock_65mhz),.noisy(switch[5]),.clean(dummy_mode[0]

    // switch 4,3 for filter enabling (debugging)
        // TODO: remove and tie high (2'b11)?
    wire bode_enable;
        wire filter_enable;
    debounce sw4(.reset(reset),.clock(clock_65mhz),.noisy(switch[4]),.clean(bode_enable));
        debounce sw3(.reset(reset),.clock(clock_65mhz),.noisy(switch[3]),.clean(filter_enable

//==== bnewbold headers END ================================================
//==== bnewbold BEGIN ======================================================

        // TODO: plug audio_v_count into GUI display
        wire [9:0] audio_h_count;
        wire [8:0] audio_v_count;
        wire audio_ram_re;
        wire [7:0] audio_main_value;
        wire [7:0] audio_overlay_value;
        wire audio_ram_ready;
        wire signed [11:0] pixel_value;
        wire [9:0] pixel_index;
        wire pixel_ready;
        wire pixel_re;

        // TODO: comment this out and replace above with actual ram modules/wires
        dummy_ram dummy_ram0(.clk_65mhz(clock_65mhz), .reset(reset),.h_count(audio_h_count),
```

```
        .v_count(audio_v_count), .ram_re(audio_ram_re), .main_value(audio_main_value)
        .overlay_value(audio_overlay_value), .ram_ready(audio_ram_ready),
        .dummy_mode(dummy_mode) );

windowing windowing0(.clk_65mhz(clock_65mhz), .reset(reset), .h_count(audio_h_count),
        .v_count(audio_v_count), .ram_re(audio_ram_re), .main_value(audio_main_value)
        .overlay_value(audio_overlay_value), .ram_ready(audio_ram_ready),
        .pixel_value(pixel_value), .pixel_index(pixel_index), .pixel_ready(pixel_ready
        .pixel_re(pixel_re) );
wire signed [11:0] sample_value;
wire [9:0] sample_index;
wire sample_ready;
wire sample_re;
wire [1:0] si_state;
wire [1:0] si_count;

spect_interp spect_interp0(.clk_65mhz(clock_65mhz), .reset(reset),
        .pixel_value(pixel_value), .pixel_index(pixel_index), .pixel_ready(pixel_ready
        .pixel_re(pixel_re), .sample_value(sample_value), .sample_index(sample_index)
        .sample_ready(sample_ready), .sample_re(sample_re), .state(si_state), .count(s
wire [9:0] ifft_read_index;
wire ifft_ce;
wire ifft_rfd;
wire signed [11:0] ifft_real_input;
wire signed [11:0] ifft_imag_input;
wire signed [11:0] later_sample_value;
wire [9:0] later_sample_index;
wire later_sample_ready;
wire later_sample_re;
wire ifft_fwd_inv_we;
wire signed [22:0] ifft_real_output;
//wire signed [22:0] ifft_imag_output;
wire [9:0] ifft_output_index;
wire ifft_output_ready;

// some of these ifft wires are unused
wire ifft_done, ifft_edone, ifft_busy;

ifft_wrapper ifft_wrapper0(.clk_65mhz(clock_65mhz), .reset(reset), .up_sample_value(sa
        .up_sample_index(sample_index), .up_sample_ready(sample_ready), .up_sample_re
        .ifft_read_index(ifft_read_index), .ifft_enable(ifft_ce),
        .ifft_rfd(ifft_rfd), .ifft_real_input(ifft_real_input), .ifft_imag_input(ifft_
        .ifft_real_output(ifft_real_output),
        //.ifft_imag_output(ifft_imag_output),
        .ifft_output_index(ifft_output_index),
        .ifft_output_ready(ifft_output_ready),
```

```
        .down_sample_value(later_sample_value), .down_sample_index(later_sample_index
        .ifft_fwd_inv_we(ifft_fwd_inv_we) );
ifft ifft0(.sclr(reset),
                          .ce(ifft_ce),                                          // clo
                          .fwd_inv_we(ifft_fwd_inv_we),// forward or inverse write ena
                          .rfd(ifft_rfd),                                // ready for c
                          .start(ifft_ce),                     // start (input)
                          .fwd_inv(1'b0),                                // forward=1 c
                          .dv(ifft_output_ready),              // data valid (output)
                          //.scale_sch_we(1'b0),                         // scaling wri
                          .done(ifft_done),                              // done (outpu
                          .clk(clock_65mhz),                             // clock (inpu
                          .busy(ifft_busy),                              // busy (outpu
                          .edone(ifft_edone),                  // will be done very s
                          //.scale_sch(10'h0),                 // scaling input  [9:0
                          .xn_re(ifft_real_input),      // real input component [11:0]
                          //.xk_im(ifft_imag_output),   // imaginary output component
                          .xn_index(ifft_read_index), // input index [9:0]
                          .xk_re(ifft_real_output),     // real output component [11:0
                          .xn_im(ifft_imag_input),      // imaginary input component
                          .xk_index(ifft_output_index)// output index [9:0]
                          );
// TODO: integrate this with actual GUI tap values
wire [6:0] new_tap_index;
wire  signed [11:0] new_tap_value;
wire new_tap_we;
wire [6:0] tap_index;
wire signed [11:0] tap_value;
tap_manager tap_manager0(.clk_65mhz(clock_65mhz), .reset(reset), .new_index(new_tap_in
        .new_value(new_tap_value), .new_we(new_tap_we), .tap_index(tap_index), .tap_va
wire  signed [11:0] filtered_value;
wire [9:0] filtered_index;
wire filtered_ready;
wire filtered_re;
bode_filter bode_filter0(.clk_65mhz(clock_65mhz), .reset(reset), .tap_index(tap_index
        .tap_value(tap_value), .sample_value(later_sample_value), .sample_index(later_
        .sample_re(later_sample_re), .filtered_value(filtered_value), .filtered_index
        .filtered_ready(filtered_ready), .filtered_re(filtered_re), .bode_enable(bode_
//wire [17:0] to_ac97_sample; // redeclared above
wire mute;
assign mute = 1'b0;
wire [17:0] to_ac97_sample;
audio_buffer audio_buffer0(.clk_65mhz(clock_65mhz), .reset(reset), .filtered_value(fil
        .filtered_index(filtered_index),        .filtered_ready(filtered_ready), .filt
        .sample(to_ac97_sample),
        //.ac97_ready(ac97_ready),
```

```
                .vol_up(vup), .vol_down(vdown), .mute(mute),
                .filter_enable(filter_enable), .tone_enable(tone_enable), .led_output(led),
                .audio_reset_b(audio_reset_b), .ac97_synch(ac97_synch), .ac97_sdata_out(ac97_s
                .ac97_bit_clock(ac97_bit_clock), .ac97_sdata_in(ac97_sdata_in) );

    //==== bnewbold END ========================================================

    //==== logic analyzer BEGIN ================================================
        // TODO: remove this section when integrated
        assign analyzer1_data[9:0] = ifft_real_input[11:2];
        assign analyzer1_data[15] = audio_ram_ready;
        assign analyzer1_data[14] = audio_ram_re;
        assign analyzer1_data[13] = pixel_ready;
        assign analyzer1_data[12] = pixel_re;
        assign analyzer1_data[11] = sample_ready;
        assign analyzer1_data[10] = sample_re;

        assign analyzer2_data[11:0] = to_ac97_sample[17:6];
        assign analyzer2_data[15] = ifft_ce;
        assign analyzer2_data[14] = ifft_rfd;
        assign analyzer2_data[13] = later_sample_re;
        assign analyzer2_data[12] = filtered_re;

        assign analyzer3_data[15:14] = 0;
        assign analyzer3_data[13:12] = si_count;
        assign analyzer3_data[11:10] = si_state;
        assign analyzer3_data[9:0] = sample_index;
        assign analyzer4_data[15:12] = 0;
        assign analyzer4_data[11:0] = filtered_value;

        assign analyzer1_clock = clock_65mhz;
        assign analyzer2_clock = clock_27mhz;
        assign analyzer3_clock = clock_65mhz;
        assign analyzer4_clock = clock_65mhz;

        assign hex_data[25:16] = audio_v_count;
        assign hex_data[9:0] = audio_h_count;

    //==== logic analyzer END ==================================================

    endmodule // labkit
```

bnewbold_modules.v

```
    `default_nettype none
    ////////////////////////////////////////////////////
```

```
// bnewbold_modules.v
// author: bryan newbold
// initial date: nov 16 2008
// my contributions for 6.111 final project
//
//////////////////////////////////////////////////
//
// rev1: 11/16/2008
//              first version, stubs
// rev2: 11/18/2008
//              skeletal progress, dummy_ram through audio
//    output pseudo functional
// rev3: 11/24/2008
//              OH BABY
// rev4: 12/03/2008
//              prep for integration
//
//////////////////////////////////////////////////
//----------------------------------------------------------------------------------
module dummy_ram(
        input wire clk_65mhz, reset,
        input wire [9:0] h_count,
        input wire [8:0] v_count,
        input wire ram_re,
        output reg signed [7:0] main_value,
        output wire signed [7:0] overlay_value,
        output reg ram_ready,
        input wire [2:0] dummy_mode
);

        assign overlay_value = main_value;
        always @(posedge clk_65mhz) begin
                if(ram_re) begin
                        ram_ready <= 1;
                        case(dummy_mode)
                                3'b000: main_value <= (h_count == {v_count, 1'b0}) ? 8'hFF : 8
                                3'b001: main_value <= (h_count == 10'd1) ? 8'hFF : 8'h00;
                                3'b010: main_value <= (h_count < 10'd25) ? 8'hFF : 8'h00;
                                3'b011: main_value <=  (h_count == 10'd600) ? 8'hFF : 8'h00;
                                3'b100: main_value <=  (h_count[7]) ? 8'hFF : 8'h00;
                                3'b101: main_value <=  (h_count < 10'd25) ? 8'hFF : 8'hA0;
                                3'b110: main_value <=  (h_count == 10'd720) ? 8'hFF : 8'h00;
                                3'b111: main_value <= (h_count == {v_count, 1'b0}) ? 8'hD0 : 8
                                default: main_value <= (h_count == 10'd1) ? 8'hFF : 8'h00;
                        endcase
                end else begin
```

```
                        main_value <= main_value;
                        ram_ready <= 0;
                end
        end
endmodule //dummy_ram
//----------------------------------------------------------------------------------------
module windowing(
        input wire clk_65mhz, reset,
        output reg [9:0] h_count,
        output reg [8:0] v_count,
        output reg ram_re,
        input wire [7:0] main_value,
        input wire [7:0] overlay_value,
        input wire ram_ready,

        output reg signed [11:0] pixel_value,
        input wire [10:0] pixel_index,
        output reg pixel_ready,
        input wire pixel_re,

        //trying to pause
        input wire pause,
        input wire [9:0] shifted_scan_y,
        input wire overlay_enable
);
        reg [1:0] state;
        parameter WAIT_REQ = 2'd0;
        parameter WAIT_RAM = 2'd1;
        parameter NO_RAM = 2'd2;

        always @(posedge clk_65mhz) begin
                if(reset == 1) begin
                        h_count <= 10'd0;
                        v_count <= 9'h0;
                        ram_re <= 1'b0;
                        pixel_value <= 12'h000;
                        pixel_ready <= 1'b0;
                        state <= WAIT_REQ;
                end else begin
                        case(state)
                        WAIT_REQ: begin
                                        pixel_ready <= 0;
                                        if((pixel_index >= 11'd8) && (pixel_index <= 11'd727)
                                                state <= pixel_re ? WAIT_RAM : WAIT_REQ;
                                                ram_re <= pixel_re;
                                                ////////////////////PAUSE/////////////////////
```

```
                                        if(pause && h_count==32)
                                                v_count <=shifted_scan_y;
                                        else begin
                                                h_count <= pixel_index - 11'd8;
                                                v_count <= pixel_re ? ((pixel_index =
                                        end
                                end else begin
                                        state <= pixel_re ? NO_RAM : WAIT_REQ;
                                        pixel_ready <= pixel_re ? 1'b1 : 1'b0;
                                        pixel_value <= 1'b0;
                                        ram_re <= 1'b0;
                                end
                        end
                WAIT_RAM: begin
                                state <= ram_ready ? WAIT_REQ : WAIT_RAM;
                                ram_re <= ~ram_ready;   // in case the first one gets
                                pixel_ready <= ram_ready;
                                if(overlay_enable)
                                        pixel_value[10:0] <= (main_value + overlay_va
                                else
                                        pixel_value[10:0] <= main_value << 3;
                        end
                NO_RAM: begin
                                state <= WAIT_REQ;
                                pixel_ready <= 1'b0;
                        end
                default: state <= WAIT_REQ;
                endcase
        end
    end
endmodule //windowing
//------------------------------------------------------------------------------
module spect_interp(
        input wire clk_65mhz, reset,

        input wire signed [11:0] pixel_value,
        output reg [10:0] pixel_index,
        input wire pixel_ready,
        output reg pixel_re,

        output reg signed [11:0] sample_value,
        input wire [10:0] sample_index,
        output reg sample_ready,
        input wire sample_re,
        output reg [1:0] state,
        output reg [1:0] count,
```

```
        input wire interp_enable
);

        reg signed [11:0] buf_in;
        wire signed [11:0] buf_out;
        reg buf_we;
        mybram #(.LOGSIZE(11),.WIDTH(12))
       line_buffer(.addr(sample_index),.clk(clk_65mhz),.we(buf_we),
                .din(buf_in),.dout(buf_out));

        parameter WAIT_REQ = 2'd0;
        parameter WAIT_PIXEL = 2'd1;
        parameter PASS_LAST = 2'd2;

        always @(posedge clk_65mhz) begin
                if(reset == 1) begin
                        pixel_re <= 1'b0;
                        sample_value <= 12'h000;
                        sample_ready <= 1'b0;
                        buf_we <= 1'b0;
                        buf_in <= 12'd0;
                        count <= 2'd0;
                end else begin
                        case(state)
                        WAIT_REQ: begin
                                        state <= sample_re ? ( (count == 2'd0) ? WAIT_PIXEL :
                                        pixel_re <= sample_re ? ( (count == 2'd0) ?  1'b1 : 1
                                        sample_ready <= 1'b0;
                                        sample_value <= sample_value;
                                        pixel_index <= sample_re ? sample_index : pixel_index
                                        buf_we <= 1'b0;
                                        count <= count;
                                end
                        WAIT_PIXEL: begin
                                        state <= pixel_ready ? WAIT_REQ : WAIT_PIXEL;
                                        pixel_re <= sample_re && !pixel_re;  // in case the f
                                        sample_ready <= pixel_ready;
                                        sample_value <= pixel_value; //(pixel_value >>> 1) +
                                        buf_in <= pixel_value;
                                        buf_we <= 1'b1;
                                        // go double fast!
                                        count <= pixel_ready ? ((sample_index == 11'd2047) ?
                                end
                        PASS_LAST: begin
                                        state <= WAIT_REQ;
                                        pixel_re <= 1'b0;
```

```
                                        sample_ready <= 1'b1;
                                        buf_we <= 1'b0;
                                        sample_value <= buf_out;
                                        // go double fast!
                                        count <= (sample_index == 11'd2047) ? (interp_enable
                        end
                        default: state <= WAIT_REQ;
                        endcase
                end
        end
endmodule // spect_interp
//------------------------------------------------------------------------------------
module ifft_wrapper(
        input wire clk_65mhz, reset,

        input wire signed [11:0] up_sample_value,
        output reg [10:0] up_sample_index,
        input wire up_sample_ready,
        output reg up_sample_re,

        input wire [10:0] ifft_read_index,
        input wire ifft_rfd,
        output reg signed [11:0] ifft_real_input,
        output reg signed [11:0] ifft_imag_input,

        output reg ifft_enable,
        input wire signed [23:0] ifft_real_output,
        input wire [10:0] ifft_output_index,
        input wire ifft_output_ready,

        output reg signed [11:0] down_sample_value,
        output reg [10:0] down_sample_index,
        output reg down_sample_ready,
        input wire down_sample_re,

        output reg ifft_fwd_inv_we,

        input wire shift_left,
        input wire shift_right
);

        // 3 bit shift control
        reg [3:0] shift_amount;
   reg old_shift_left,old_shift_right;
   always @ (posedge clk_65mhz) begin
     if (reset) shift_amount <= 4'd0;
```

```
  else begin
     if (shift_left & ~old_shift_left & shift_amount != 4'd15) shift_amount <= shift_amoun
     if (shift_right & ~old_shift_right & shift_amount != 4'd0) shift_amount <= shift_amou
     end
  old_shift_left <= shift_left;
  old_shift_right <= shift_right;
end
     reg [1:0] state;
     parameter WAIT_REQ = 2'd0;
     parameter WAIT_UPSTREAM = 2'd1;
     parameter WAIT_IFFT_REQ = 2'd2;
     parameter PUSH_IFFT = 2'd3;

     always @(posedge clk_65mhz) begin
             if(reset == 1) begin
                     ifft_enable <= 1'b1;    // so clear goes through
                     down_sample_value <= 12'h000;
                     down_sample_index <= 11'h000;
                     down_sample_ready <= 1'b0;
                     ifft_real_input <= 12'h000;
                     ifft_imag_input <= 12'h000;
                     state <= WAIT_REQ;
                     ifft_fwd_inv_we <= 1'b1;
             end else begin
                     ifft_fwd_inv_we <= 1'b0;
                     case(state)
                     WAIT_REQ: begin
                                     state <= down_sample_re ? WAIT_UPSTREAM : WAIT_REQ;
                                     ifft_enable <= 1'b0;
                                     down_sample_value <= down_sample_value;
                                     down_sample_index <= down_sample_index;
                                     down_sample_ready <= 1'b0;
                                     up_sample_re <= down_sample_re;
                                     up_sample_index <= ifft_read_index; //ifft_read_index
                             end
                     WAIT_UPSTREAM: begin
                                     state <= up_sample_ready ? (ifft_rfd ? PUSH_IFFT : WA
                                     ifft_enable <= up_sample_ready;
                                     down_sample_value <= down_sample_value;
                                     down_sample_index <= down_sample_index;
                                     down_sample_ready <= 1'b0;
                                     up_sample_re <= 1'b0;
                                     up_sample_index <= up_sample_index;

                                     ifft_real_input <= up_sample_value;
                                     ifft_imag_input <= 12'h000;
```

```
                        end
                    WAIT_IFFT_REQ: begin
                                state <= ifft_rfd ? PUSH_IFFT : WAIT_IFFT_REQ;
                                ifft_enable <= 1'b1;
                                down_sample_value <= down_sample_value;
                                down_sample_index <= down_sample_index;
                                down_sample_ready <= 1'b0;
                                up_sample_re <= 1'b0;
                                up_sample_index <= up_sample_index;
                        end
                    PUSH_IFFT: begin
                                state <= ifft_output_ready ? WAIT_REQ : WAIT_UPSTREAM
                                ifft_enable <= 1'b0;
                                // need sign bit, them selection of most significant b
                                down_sample_value <= ifft_real_output >>> shift_amoun
                                down_sample_index <= ifft_output_index;
                                down_sample_ready <= ifft_output_ready;
                                up_sample_re <= ifft_output_ready ? 1'b0 : 1'b1;
                                up_sample_index <= ifft_output_ready ? up_sample_inde
                        end
                    default: state <= WAIT_REQ;
                    endcase
            end
        end
endmodule // ifft_wrapper
//------------------------------------------------------------------------------------
// behaves as BRAM? 128x 12-bit taps
module tap_manager(
        input wire clk_65mhz, reset,

        input wire [6:0] new_index,
        input wire signed [11:0] new_value,
        input wire new_we,

        input wire [6:0] tap_index,
        output reg signed [11:0] tap_value
);
        reg signed [11:0] mem[127:0];
        always @(posedge clk_65mhz) begin
                if (reset) begin
                        tap_value <= 12'b011111111111;
                        mem[0] <= 12'b011111111111;
                        mem[1] <= 12'b000000000000;
                end else begin
                        if (new_we) mem[new_index] <= new_value;
                        else tap_value <= mem[tap_index];
```

```
                end
        end
endmodule // tap_manager
//----------------------------------------------------------------------------------
module bode_filter(
        input wire clk_65mhz, reset,

        output reg [6:0] tap_index,
        input wire signed [11:0] tap_value,

        input wire signed [11:0] sample_value,
        input wire [10:0] sample_index,
        input wire sample_ready,
        output reg sample_re,

        output wire signed [11:0] filtered_value,
        output reg [10:0] filtered_index,
        output reg filtered_ready,
        input wire filtered_re,

        input wire bode_enable,
        output reg [1:0] state
);
        parameter WAIT_REQ = 2'b00;
        parameter WAIT_SAMPLE = 2'b01;
        parameter COMPUTE = 2'b11;
        reg signed [26:0] out;
        reg signed [11:0] buffer[127:0];
        reg [6:0] buffer_index;
        assign filtered_value = bode_enable ? out[26:15] : sample_value;

        always @(posedge clk_65mhz) begin
                if (reset) begin
                        sample_re <= 1'b0;
                        filtered_index <= 10'h000;
                        filtered_ready <= 1'b0;
                        tap_index <= 7'h00;
                        buffer_index <= 7'h00;
                        out <= 27'h0000000;
                        state <= WAIT_REQ;
                end else begin
                        case(state)
                        COMPUTE: begin
                                        state <= (tap_index == 7'hFF) ? WAIT_REQ : COMPUTE;
                                        sample_re <= 1'b0;
                                        filtered_ready <= (tap_index == 7'hFF);
```

```
                                        filtered_index <= filtered_index;
                                        tap_index <= (tap_index + 1);
                                        out <= out + tap_value * buffer[buffer_index - tap_in
                                end
                        WAIT_REQ: begin
                                        state <= (filtered_re == 1'b1) ? WAIT_SAMPLE : WAIT_R
                                        sample_re <= filtered_re;
                                        filtered_ready <= 1'b0;
                                        filtered_index <= filtered_index;
                                        tap_index <= 7'h00;
                                        out <= out;
                                end
                        WAIT_SAMPLE: begin
                                        state <= (sample_ready == 1'b1) ? COMPUTE : WAIT_SAMP
                                        sample_re <= filtered_re && !sample_re;  // in case t
                                        filtered_ready <= 1'b0;
                                        filtered_index <= sample_index;
                                        tap_index <= 7'h00;
                                        buffer_index <= (sample_ready == 1'b1) ? (buffer_inde
                                        buffer[buffer_index] <= sample_value;
                                        out <= 27'h0000000;
                                end
                        default: state <= WAIT_REQ;
                        endcase
                end
        end
endmodule // bode_filter
//------------------------------------------------------------------------------
module audio_buffer(
        input wire clk_65mhz, reset,

        input wire signed [11:0] filtered_value,
        input wire [10:0] filtered_index,
        input wire filtered_ready,
        output reg filtered_re,

        output reg signed [17:0] sample,

        input wire vol_up, vol_down, mute, filter_enable, tone_enable,
        output wire [7:0] led_output,

   output wire audio_reset_b, ac97_synch, ac97_sdata_out,
   input wire ac97_bit_clock, ac97_sdata_in
);
        // 5 bit volume control
        reg [4:0] volume;
```

```
reg old_vup,old_vdown;
always @ (posedge clk_65mhz) begin
  if (reset) volume <= 5'd8;
  else begin
     if (vol_up & ~old_vup & volume != 5'd31) volume <= volume+1;
     if (vol_down & ~old_vdown & volume != 5'd0) volume <= volume-1;
  end
  old_vup <= vol_up;
  old_vdown <= vol_down;
end

     // AC97 driver
     reg [17:0] to_ac97_data;
     wire [17:0] from_ac97_data;
     wire ac97_ready;
audio_manager a(clk_65mhz, reset, volume, from_ac97_data, to_ac97_data, ac97_ready,
            audio_reset_b, ac97_sdata_out, ac97_sdata_in,
            ac97_synch, ac97_bit_clock);

// light up LEDs when recording, show volume during playback.
// led is active low
assign led_output = tone_enable ? ~{3'b000, volume} : ~{1'b0,7'hFF};

// test: playback 750hz tone, or loopback using incoming data
wire [19:0] tone;
tone750hz xxx(.clock(clk_65mhz),.ready(ac97_ready),.pcm_data(tone));

     // tone vs. sample selection
always @ (posedge clk_65mhz) begin
   if (ac97_ready) begin
                    // get here when we've just received new data from the AC97
                    if (mute) to_ac97_data <= 18'h00000;
                    else to_ac97_data <= tone_enable ? tone[19:2] : sample;
   end
end

     // actual filter stuff
     reg old_ready;
     reg reuse_last;
     reg filter_we;
     wire [25:0] filter_out;
     fir31 filter(.clock(clk_65mhz),.reset(reset),.ready(filtered_ready || filter_we),
             .x(filtered_value), .y(filter_out) );

     always @(posedge clk_65mhz) begin
             if(reset) begin
```

```
                        filtered_re <= 1'b1;
                        sample <= 18'h00000;
                        old_ready <= 1'b0;
                        reuse_last <= 1'b0;
                        filter_we <= 1'b0;
                end else begin
                        old_ready <= ac97_ready;
                        if(ac97_ready && !old_ready) begin
                                if(filter_enable) begin
                                        if(reuse_last == 1'b0) begin
                                                sample <= filter_out[25:8];
                                                filtered_re <= 1'b1;
                                                filter_we <= 1'b0;
                                                reuse_last <= 1'b1;
                                        end else begin
                                                sample <= filter_out[25:8];
                                                filter_we <= 1'b1;
                                                reuse_last <= 1'b0;
                                        end
                                end else begin
                                                sample <= filtered_value;
                                                reuse_last <= 1'b0;
                                                filtered_re <= 1'b1;
                                                filter_we <= 1'b0;
                                end
                        end else begin
                                sample <= sample;
                                filtered_re <= 1'b0;
                                filter_we <= 1'b0;
                        end
                end
        end
endmodule // audio_buffer
//////////////////////////////////////////////////////////////////////////////
//
// 31-tap FIR filter, 12-bit signed data, 14-bit signed coefficients.
// ready is asserted whenever there is a new sample on the X input,
// the Y output should also be sampled at the same time.  Assumes at
// least 32 clocks between ready assertions.  Note that since the
// coefficients have been scaled by 2**14, so has the output (it's
// expanded from 12 bits to 26 bits).  To get a 12-bit result from the
// filter just divide by 2**14, ie, use Y[25:14].
//
//////////////////////////////////////////////////////////////////////////////
module fir31(
  input wire clock,reset,ready,
```

```
  input wire signed [11:0] x,
  output reg signed [25:0] y
);
  reg [4:0] ci;
  wire signed [13:0] cv;
  coeffs31 c(.index(ci), .coeff(cv));

  reg last_ready;
  reg signed [11:0] b[31:0]; // buffer; larger than it has to be!
  reg [4:0] bi;
  reg [4:0] bi_last;

  always @(posedge clock) begin
    if (reset) begin
      y  <= 0;
      ci <= 0;
      bi <= 0;
      bi_last <= 31;
         end else begin
                 if (last_ready != ready) last_ready <= ready;
                 if ((last_ready != ready) && ready) begin
                        ci <= 0;
                        y <= 0;
                        bi <= bi + 1;
                        bi_last <= bi;
                        b[bi] <= x;
                 end else if (ci != 31) begin
                        ci <= ci+1;
                        y <= y + (cv * b[bi_last - ci]);
                 end else begin
                        ci <= ci;
                        y <= y;
                 end
          end
        end
endmodule // fir31
////////////////////////////////////////////////////////////////////////////////
//
// Coefficients for a 31-tap low-pass FIR filter with Wn=.5 (eg, 11kHz for a
// 44kHz sample rate?).  Since we're doing integer arithmetic, we've scaled
// the coefficients by 2**13 (note extra bit for signed!)
// Matlab command: round(fir1(30,.5)*(2^13))
//
////////////////////////////////////////////////////////////////////////////////
module coeffs31(
  input wire [4:0] index,
```

```
  output reg signed [13:0] coeff
);
  // tools will turn this into a 31x10 ROM
  always @(index)
    case (index)
      5'd0:  coeff = -14'sd14;
      5'd1:  coeff = 14'sd0;
      5'd2:  coeff = 14'sd24;
      5'd3:  coeff = 14'sd0;
      5'd4:  coeff = -14'sd55;
      5'd5:  coeff = 14'sd0;
      5'd6:  coeff = 14'sd115;
      5'd7:  coeff = 14'sd0;
      5'd8:  coeff = -14'sd219;
      5'd9:  coeff = 14'sd0;
      5'd10: coeff = 14'sd402;
      5'd11: coeff = 14'sd0;
      5'd12: coeff = -14'sd794;
      5'd13: coeff = 14'sd0;
      5'd14: coeff = 14'sd2586;
      5'd15: coeff = 14'sd4103;
      5'd16: coeff = 14'sd2586;
      5'd17: coeff = 14'sd0;
      5'd18: coeff = -14'sd794;
      5'd19: coeff = 14'sd0;
      5'd20: coeff = 14'sd402;
      5'd21: coeff = 14'sd0;
      5'd22: coeff = -14'sd219;
      5'd23: coeff = 14'sd0;
      5'd24: coeff = 14'sd115;
      5'd25: coeff = 14'sd0;
      5'd26: coeff = -14'sd55;
      5'd27: coeff = 14'sd0;
      5'd28: coeff = 14'sd24;
      5'd29: coeff = 14'sd0;
      5'd30: coeff = -14'sd14;
      default: coeff = 14'hXXX;
    endcase
endmodule // coeffs31
// identical to above, only W=.25
module coeffs31_lower(
  input wire [4:0] index,
  output reg signed [13:0] coeff
);
  // tools will turn this into a 31x10 ROM
  always @(index)
```

```
    case (index)
      5'd0:  coeff = -14'sd10;
      5'd1:  coeff = -14'sd17;
      5'd2:  coeff = -14'sd17;
      5'd3:  coeff = 14'sd0;
      5'd4:  coeff = 14'sd39;
      5'd5:  coeff = 14'sd81;
      5'd6:  coeff = 14'sd82;
      5'd7:  coeff = 14'sd0;
      5'd8:  coeff = -14'sd155;
      5'd9:  coeff = -14'sd297;
      5'd10: coeff = -14'sd285;
      5'd11: coeff = 14'sd0;
      5'd12: coeff = 14'sd562;
      5'd13: coeff = 14'sd1256;
      5'd14: coeff = 14'sd1831;
      5'd15: coeff = 14'sd2054;
      5'd16: coeff = 14'sd1831;
      5'd17: coeff = 14'sd1256;
      5'd18: coeff = 14'sd562;
      5'd19: coeff = 14'sd0;
      5'd20: coeff = -14'sd285;
      5'd21: coeff = -14'sd297;
      5'd22: coeff = -14'sd155;
      5'd23: coeff = 14'sd0;
      5'd24: coeff = 14'sd82;
      5'd25: coeff = 14'sd81;
      5'd26: coeff = 14'sd39;
      5'd27: coeff = 14'sd0;
      5'd28: coeff = -14'sd17;
      5'd29: coeff = -14'sd17;
      5'd30: coeff = -14'sd10;
      default: coeff = 14'hXXX;
    endcase
endmodule // coeffs31_lower
module dummy_coeffs31(
  input wire [4:0] index,
  output reg signed [13:0] coeff
);
  // tools will turn this into a 31x10 ROM
  always @(index)
    coeff <= (index == 4'b0000) ? 14'sd1 : 14'sd0;
endmodule // dummy_coeffs31
```

bode_plot.v

```
//Written by Tyler Hutchison for 6.111
module bode_plot
        #(parameter LOC_X = 10'd762, LOC_Y=9'd491)
        (input reset,
        input clk,
        input [10:0] hcount,
        input [9:0] vcount,
        input [11:0] mx,my,
        input btn_click,
        input [6:0] bode_apply_index,
        output [7:0] bode_apply_value,
        output reg [2:0] pixel,
        output reg tap_bode_we,
        output [6:0] tap_index,
        output signed [11:0] tap_value,
        output [6:0] bode_index_check,
        output reg signed [8:0] current_bode
        );

        //for IFFT
        parameter IFFT_SETUP=2'b00;
        parameter IFFT_WRITE=2'b01;
        parameter IFFT_READ=2'b11;
        parameter IFFT_WAIT=2'b10;

        reg not_reset;
        reg [6:0] reset_counter;
        reg [7:0] bode_array [127:0] ;

        reg [11:0] prevmx; //see if we need to interpolate values

        reg line_grabbed;

        //for ifft
        reg ifft_we, sclr, ifft_on, bode_we,ce,bode_re,started,start,unload;
        wire done, busy, edone;
        wire [15:0] tap_value_long;
        reg last_grabbed;
        wire dv, ready_for_data;

        reg [1:0] ifft_state;

        wire ungrab_edge=(line_grabbed!=last_grabbed && !line_grabbed); //posedge edge detecti

        //for image application
```

```
        assign bode_apply_value=bode_array[bode_apply_index];

always @(posedge clk) begin
                if(reset) begin //reset just triggered get ready to reset all bode values
                        reset_counter<=7'b0;
                        not_reset<=1'b1;
                        line_grabbed<=0;
                        //turn ifft into inverse mode and enable writing
                        ifft_we<=1'b1;
                        sclr<=1'b1;
                        ifft_on<=1'b0;
                        ce<=0;
                        bode_we<=1;
                        bode_re<=1;
                        tap_bode_we<=1'b0;
                        start<=1;
                        unload<=1;
                        ifft_state<=IFFT_WAIT;
                end
                else if(not_reset) begin
                        sclr<=1'b0;
                        ifft_we<=1'b0;//we aren't writing to ifft_inv anymore
                        bode_array[reset_counter]<= 8'b11111111;
                        reset_counter <= reset_counter +1'b1;
                        if(reset_counter==7'b1111111) begin
                                not_reset<=1'b0;
                                ifft_state<=IFFT_SETUP; //initialize bode filter
                        end
                end

                //FSM for IFFT to convert bode coefficients to time domain taps
                //...we should have applied the frequency coefficients directly to the data
                //DUMB
                case(ifft_state)
                        IFFT_WAIT : begin
                                ce<=0;
                                tap_bode_we<=1'b0;
                                ifft_state<=ungrab_edge?IFFT_SETUP:IFFT_WAIT;
                        end
                        IFFT_SETUP: begin
                                ce<=1;
                                ifft_state<=(bode_index_check==0&&ready_for_data)?IFFT_WRITE:I
                        end
                        IFFT_WRITE: begin
                                if(bode_index_check>=2 || started) begin
                                        started<=1;
```

```
                        //notice extra 0 bit, bode array is actually unsigned
                        current_bode <= bode_index_check<2?{1'b0,bode_array[bo

                        ifft_state<=((bode_index_check+126)==127) ? IFFT_READ
                end
        end
        IFFT_READ: begin
                if(tap_index==127) begin
                        ifft_state<=IFFT_WAIT;
                end
                else if(dv) tap_bode_we<=1'b1;  //begin writing!
        end
        default: ifft_state<=IFFT_WAIT;
endcase

//see if mouse within the bode plot
if((mx>LOC_X && mx<(LOC_X+7'b1111111)) &&
        (my>LOC_Y && my<(LOC_Y+8'b11111111)) ||line_grabbed) begin
        ce<=0;
        //is user trying to grab the line
        if(btn_click && !line_grabbed) begin //did the line just get clicked?
                ce<=0; //shut off ifft
                line_grabbed<=1;
                bode_array[mx-LOC_X]<=8'd255-my+LOC_Y; //else calculate
                prevmx<=mx;
        end
        else if((~btn_click && line_grabbed)) begin //did the line just get un
                line_grabbed<=0;
        end
        else if(btn_click && line_grabbed) begin //output should wait until di
                if(prevmx != mx) begin //if mx has moved since beginning
                        if(prevmx>mx && prevmx >LOC_X-1) begin
                                prevmx<=prevmx-1'b1;
                                if(my>(LOC_Y+8'd255)) bode_array[prevmx-LOC_X]
                                else if(my<LOC_Y) bode_array[prevmx-LOC_X]<=8
                                else bode_array[prevmx-LOC_X]<=8'd255-my+LOC_Y
                        end
                        else if(prevmx<mx && prevmx<LOC_X+10'd128) begin
                                prevmx<=prevmx+1'b1;
                                if(my>(LOC_Y+8'd255)) bode_array[prevmx-LOC_X]
                                else if(my<LOC_Y) bode_array[prevmx-LOC_X]<=8
                                else bode_array[prevmx-LOC_X]<=8'd255-my+LOC_Y
                        end
                end
        end
end
```

```
                else line_grabbed<=0;
                last_grabbed<=line_grabbed;
        end
        //draw that plot! first outline it
        always @(hcount or vcount) begin
                if((hcount==(LOC_X-1)&& vcount>=(LOC_Y-1) && vcount<=(LOC_Y+9'b100000000)) ||
                        (vcount==(LOC_Y-1) && hcount>=(LOC_X-1) && hcount<=(LOC_X+8'b10000000
                        (hcount==(LOC_X+8'b10000000) && vcount>=(LOC_Y-1) && vcount<=(LOC_Y+9
                        (vcount ==(LOC_Y+9'b100000000) && hcount>=(LOC_X-1) && hcount<=(LOC_X-
                        pixel=3'b100;
                //check to see if we have to draw in a line point
                else if((hcount>=LOC_X && hcount<(LOC_X+7'b1111111)) &&
                                        (vcount>=LOC_Y && vcount<(LOC_Y+8'b11111111))) begin
                                        //index the bode values by distance from left edge
                                        //index values by distance from top edge
                                        if(bode_array[hcount-LOC_X]>=(8'd255-vcount+LOC_Y)) /
                                                pixel=3'b111;
                                        else
                                                pixel=3'b0;
                end
                else pixel=3'b0;
        end
        assign tap_value={tap_value_long[15],tap_value_long[7:0],3'b0}; //grab msb for sign
        //failed ifft module
        bodeifft tap_creator(.xn_re(current_bode),.xn_im(8'b0),.start(start),.unload(unload),
                                                                .sclr(sclr),.ce(ce),.clk(clk)
                                                                .rfd(ready_for_data),.busy(bus
endmodule
```

camera_to_zbt.v

```
//Modified by Dima Turbiner
//Inspired from:
//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to
// store 4 bytes of black-and-white intensity data from the NTSC
// video input.
//////////////////////////////////////////////////////////////////////////////
```

```
// Prepare data and address values to fill ZBT memory with NTSC camera data
module camera_to_zbt(clk, camera_clk, camera_fvh, camera_pixel_ready, camera_pixel,
                     camera_addr, camera_data, camera_we);
 //sw);
   input         clk;   // system clock
   input         camera_clk;    // video clock from camera
   input [2:0]   camera_fvh;
   input         camera_pixel_ready;
   input [7:0]   camera_pixel;
   output [18:0] camera_addr;
   output [35:0] camera_data;
   output        camera_we;     // write enable for camera to zbt
   parameter     COL_START = 10'b0;
   parameter     ROW_START = 10'b0;
   // here put the luminance data from the ntsc decoder into the ram
   // this is for 1024 x 768 XGA display
   reg [10:0]    col = 0;
   reg [10:0]    row = 0;
   reg [7:0]     vdata = 0;
   reg           vwe;
   reg           old_dv;
   reg           old_frame;     // frames are even / odd interlaced
   reg           even_odd;      // decode interlaced frame to this wire

   wire          frame = camera_fvh[2];
   wire          frame_edge = frame & ~old_frame;
   always @ (posedge camera_clk) //LLC1 is reference
     begin
        old_dv <= camera_pixel_ready;
        vwe <= camera_pixel_ready && !camera_fvh[2] & ~old_dv; // if data valid, write it
        old_frame <= frame;
        even_odd = frame_edge ? ~even_odd : even_odd;
        //****TEST: 1024 -> 720
        if (!camera_fvh[2])
          begin
            col <= camera_fvh[0] ? COL_START :
                   (!camera_fvh[2] && !camera_fvh[1] && camera_pixel_ready && (col < 1024)) ?
             row <= camera_fvh[1] ? ROW_START :
                    (!camera_fvh[2] && camera_fvh[0] && (row < 768)) ? row + 1 : row;
             vdata <= (camera_pixel_ready && !camera_fvh[2]) ? camera_pixel : vdata;
          end
    end
   // synchronize with system clock
   reg [10:0] x[1:0];
        reg [10:0] y[1:0];
   reg [7:0] data[1:0];
```

```
   reg       we[1:0];
   reg       eo[1:0];
   always @(posedge clk)
     begin
        {x[1],x[0]} <= {x[0],col};
        {y[1],y[0]} <= {y[0],row};
        {data[1],data[0]} <= {data[0],vdata};
        {we[1],we[0]} <= {we[0],vwe};
        {eo[1],eo[0]} <= {eo[0],even_odd};
     end
   // edge detection on write enable signal
   reg old_we;
   wire we_edge = we[1] & ~old_we;
   always @(posedge clk) old_we <= we[1];
   // shift each set of four bytes into a large register for the ZBT

   reg [31:0] mydata;
   always @(posedge clk)
     if (we_edge)
       mydata <= { mydata[23:0], data[1] };
   // compute address to store data in
   wire [18:0] myaddr = {1'b0, y[1][8:0], eo[1], x[1][9:2]};
   // update the output address and data only when four bytes ready
   reg [18:0] camera_addr;
   reg [35:0] camera_data;
   wire       camera_we = (we_edge & (x[1][1:0]==2'b00));
   always @(posedge clk)
     if ( camera_we )
       begin
          camera_addr <=  myaddr;
          camera_data <=  {4'b0,mydata};
       end

endmodule // camera_to_zbt
```

common.v

```
`default_nettype none
////////////////////////////////////////////////////
// common.v
// author: bryan newbold
// initial date: nov 16 2008
// shared modules for 6.111 final project
//
////////////////////////////////////////////////////
//
```

```
// rev1: 11/18/2008
//
////////////////////////////////////////////////////
//-------------------------------------------------------------------------------------
// Switch Debounce Module (from lab staff)
module debounce (
  input wire reset, clock, noisy,
  output reg clean
);
  reg [18:0] count;
  reg new;
  always @(posedge clock)
    if (reset) begin
      count <= 0;
      new <= noisy;
      clean <= noisy;
    end
    else if (noisy != new) begin
      // noisy input changed, restart the .01 sec clock
      new <= noisy;
      count <= 0;
    end
    else if (count == 270000)
      // noisy input stable for .01 secs, pass it along!
      clean <= new;
    else
      // waiting for .01 sec to pass
      count <= count+1;
endmodule
//-------------------------------------------------------------------------------------
// assemble/disassemble AC97 serial frames (from lab staff)
module ac97 (
  output reg ready,
  input wire [7:0] command_address,
  input wire [15:0] command_data,
  input wire command_valid,
  input wire [19:0] left_data,
  input wire left_valid,
  input wire [19:0] right_data,
  input wire right_valid,
  output reg [19:0] left_in_data, right_in_data,
  output reg ac97_sdata_out,
  input wire ac97_sdata_in,
  output reg ac97_synch,
  input wire ac97_bit_clock
);
```

```
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;
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 (from lab staff)
module ac97commands (
  input wire clock,
  input wire ready,
  output wire [7:0] command_address,
  output wire [15:0] command_data,
  output reg command_valid,
```

```
  input wire [4:0] volume,
  input wire [2:0] source
);
  reg [23:0] command;
  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
```

```
//------------------------------------------------------------------------------
// 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.
// (from lab staff)
module somebram #(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);
   // 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
endmodule
//------------------------------------------------------------------------------
// (from lab staff)
module audio_manager (
  input wire clock_27mhz,
  input wire reset,
  input wire [4:0] volume,
  output wire [17:0] audio_in_data,
  input wire [17:0] audio_out_data,
  output wire ready,
  output reg audio_reset_b,   // ac97 interface signals
  output wire ac97_sdata_out,
  input wire ac97_sdata_in,
  output wire ac97_synch,
  input wire ac97_bit_clock
);
  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;
  // wait a little before enabling the AC97 codec
  reg [9:0] reset_count;
  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(.ready(ac97_ready),
            .command_address(command_address),
            .command_data(command_data),
            .command_valid(command_valid),
            .left_data(left_out_data), .left_valid(1'b1),
            .right_data(right_out_data), .right_valid(1'b1),
            .left_in_data(left_in_data), .right_in_data(right_in_data),
            .ac97_sdata_out(ac97_sdata_out),
            .ac97_sdata_in(ac97_sdata_in),
            .ac97_synch(ac97_synch),
            .ac97_bit_clock(ac97_bit_clock));
  // ready: one cycle pulse synchronous with clock_27mhz
  reg [2:0] ready_sync;
  always @ (posedge clock_27mhz) ready_sync <= {ready_sync[1:0], ac97_ready};
  assign ready = ready_sync[1] & ~ready_sync[2];
  reg [17:0] out_data;
  always @ (posedge clock_27mhz)
    if (ready) out_data <= audio_out_data;
  assign audio_in_data = left_in_data[19:2];
  assign left_out_data = {out_data, 2'b00};
  assign right_out_data = left_out_data;
  // generate repeating sequence of read/writes to AC97 registers
  ac97commands cmds(.clock(clock_27mhz), .ready(ready),
                    .command_address(command_address),
                    .command_data(command_data),
                    .command_valid(command_valid),
                    .volume(volume),
                    .source(3'b000));     // mic
endmodule
//////////////////////////////////////////////////////////////////////////////////
// generate PCM data for 750hz sine wave (assuming f(ready) = 48khz)
// (from lab staff)
module tone750hz (
  input wire clock,
  input wire ready,
  output reg [19:0] pcm_data
);
   reg [8:0] index;
   initial begin
      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
//-----------------------------------------------------------------------------------
// run the hex display (duh)
// (from lab staff)
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
```

cstringdispunedit.v

```
//
// File:   cstringdisp.v
// Date:   24-Oct-05
// Author: I. Chuang, C. Terman
```

```
//
// Display an ASCII encoded character string in a video window at some
// specified x,y pixel location.
//
// INPUTS:
//
//   vclock       - video pixel clock
//   hcount       - horizontal (x) location of current pixel
//   vcount       - vertical (y) location of current pixel
//   cstring      - character string to display (8 bit ASCII for each char)
//   cx,cy        - pixel location (upper left corner) to display string at
//
// OUTPUT:
//
//   pixel        - video pixel value to display at current location
//
// PARAMETERS:
//
//   NCHAR        - number of characters in string to display
//   NCHAR_BITS   - number of bits to specify NCHAR
//
// pixel should be OR'ed (or XOR'ed) to your video data for display.
//
// Each character is 8x12, but pixels are doubled horizontally and vertically
// so fonts are magnified 2x.  On an XGA screen (1024x768) you can fit
// 64 x 32 such characters.
//
// Needs font_rom.v and font_rom.ngo
//
// For different fonts, you can change font_rom.  For different string
// display colors, change the assignment to cpixel.
//////////////////////////////////////////////////////////////////////////////
//
// video character string display
//
//////////////////////////////////////////////////////////////////////////////
module char_string_display (vclock,hcount,vcount,pixel,cstring,cx,cy);
   parameter NCHAR =11; // number of 8-bit characters in cstring
   parameter NCHAR_BITS = 4; // number of bits in NCHAR
   input vclock;        // 65MHz clock
   input [10:0] hcount; // horizontal index of current pixel (0..1023)
   input [9:0]  vcount; // vertical index of current pixel (0..767)
   output [2:0] pixel;  // char display's pixel
   input [NCHAR*8-1:0] cstring; // character string to display
   input [10:0] cx;
   input [9:0]  cy;
```

```
   // 1 line x 8 character display (8 x 12 pixel-sized characters)
   wire [10:0]  hoff = hcount-1-cx;
   wire [9:0]   voff = vcount-cy;
   wire [NCHAR_BITS-1:0] column = NCHAR-1-hoff[NCHAR_BITS-1+4:4];  // < NCHAR
   wire [2:0]   h = hoff[3:1];            // 0 .. 7
   wire [3:0]   v = voff[4:1];            // 0 .. 11
   // look up character to display (from character string)
   reg [7:0]  char;
   integer  n;
   always @*
     for (n=0 ; n<8 ; n = n+1 )         // 8 bits per character (ASCII)
       char[n] <= cstring[column*8+n];
   // look up raster row from font rom
   wire reverse = char[7];
   wire [10:0] font_addr = char[6:0]*12 + v;    // 12 bytes per character
   wire [7:0]  font_byte;
   font_rom f(font_addr,vclock,font_byte);
   // generate character pixel if we're in the right h,v area
   wire [2:0] cpixel = (font_byte[7 - h] ^ reverse) ? 7 : 0;
   wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*16)
                    & (vcount < cy + 24));
   wire [2:0] pixel = dispflag ? cpixel : 0;
endmodule
```

draw2d_fourier.v

```
//Tyler Hutchison: 6.111
module draw_2dfourier
        #(parameter LINE_LOC=0, //so we know where in the blob the 1 pixel scan line is
                                        IMAGE_LOC_X=40,
                                        TWOD_LOC_Y=490)
        (
                input clk,
                input reset,
                input [10:0] hcount,
                input [9:0] vcount, scan_bar_y,
                input [7:0] intensity,

                output reg [7:0] pixel);

                reg [7:0] int_in;
                reg [9:0] x_addr;
                wire [7:0] pixel_data;
                reg we;

                //is the current vcount the same as the scan bar location, store data and spi
```

```
                //we have to store the data since the scan bar might rest on a particular line
                //i.e. a pause or the fact that each line takes hundreds of miliseconds to ou
                always @(hcount or vcount) begin
                        //check to see if we should draw a line point here
                        if((hcount>=IMAGE_LOC_X && hcount<=(IMAGE_LOC_X+10'd720)) &&
                                        (vcount>=TWOD_LOC_Y && vcount <= (TWOD_LOC_Y+8
                                //should we draw a pixel at this point, or is the intensity di
                                if((8'b11111111+TWOD_LOC_Y-vcount)<=pixel_data) begin
                                        if((8'b11111111+TWOD_LOC_Y-vcount)==8'b0)
                                                pixel=8'b0;
                                        else
                                                pixel=(vcount-TWOD_LOC_Y-1); //output a point
                                end
                                else
                                        pixel=8'b0;
                        end
                        else pixel=8'b0;
                end
                always @(posedge clk) begin //stores scan line value in memory so it can be di
                        if(reset) begin
                                x_addr<=10'b0;
                                int_in<=8'b0;
                        end
                        else if(vcount==scan_bar_y+LINE_LOC && hcount>=IMAGE_LOC_X && hcount<=
                                we<=1;
                                x_addr<=hcount-IMAGE_LOC_X;
                                int_in<=intensity;
                        end
                        else begin
                                we<=0;
                                x_addr<=(hcount-IMAGE_LOC_X);
                        end
                end
        mybram #(.LOGSIZE(10),.WIDTH(8))
                twodfourier (.addr(x_addr), .clk(clk), .din(int_in),.dout(pixel_data),.we(we))
endmodule

//////////////////////////////////////////////////////////////////////////////
//
// 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.
//
//////////////////////////////////////////////////////////////////////////////
//from lab 4... used to store values indexed by the scan line
module mybram #(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);
      // 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
    endmodule
```

edit_to_zbt.v

```
//Dima Turbiner
module edit_to_zbt(
    input clk, reset, input [9:0]hcount, input [8:0]vcount, input [7:0]value,
    output [35:0]data, output [18:0]addr, output [3:0]bwe);

   assign bwe = 4'b0;
endmodule
```

finalproj_rev2.v

```
//////////////////////////////////////////////////
//TOP LEVEL MODULE FOR REAL TIME AUDIO COMPOSITION
//BRYAN NEWBOLD, TYLER HUTCHISON, DIMA TURBINER
//WIRING AND MODULES INSTANTIATED BELOW
//////////////////////////////////////////////////
`default_nettype none
///////////////////////////////////////////////////////////////////////////////
//
// 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)
//
// 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 final_proj   (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
                ac97_bit_clock,

                vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
                vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
                vga_out_vsync,
                tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
                tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
                tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,
                tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
                tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
                tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
                tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,
                ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
                ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,
                ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
                ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,
                clock_feedback_out, clock_feedback_in,
                flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
                flash_reset_b, flash_sts, flash_byte_b,
                rs232_txd, rs232_rxd, rs232_rts, rs232_cts,
                mouse_clock, mouse_data, keyboard_clock, keyboard_data,
                clock_27mhz, clock1, clock2,
                disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
                disp_reset_b, disp_data_in,
                button0, button1, button2, button3, button_enter, button_right,
                button_left, button_down, button_up,
                switch,
                led,

                user1, user2, user3, user4,

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

                analyzer1_data, analyzer1_clock,
                analyzer2_data, analyzer2_clock,
                analyzer3_data, analyzer3_clock,
                analyzer4_data, analyzer4_clock);
   output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
   input  ac97_bit_clock, ac97_sdata_in;

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

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

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

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

inout  [35:0] ram1_data;
output [18:0] ram1_address;
output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
output [3:0] ram1_bwe_b;
input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;
inout        mouse_clock, mouse_data;
     input 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
// 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'b1;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b1;
//assign tv_in_reset_b = 1'b0;
assign tv_in_clock = clock_27mhz;//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_clk = 1'b0;
   assign ram0_we_b = 1'b1;
   assign ram0_cen_b = 1'b0;    // clock enable
        assign ram0_bwe_b = 4'h0;
*/
/* enable RAM pins */
   assign ram0_ce_b = 1'b0;
   assign ram0_oe_b = 1'b0;
   assign ram0_adv_ld = 1'b0;
/**********/
/*
   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_clk = 1'b0;
        assign ram1_we_b = 1'b1;
   assign ram1_cen_b = 1'b1;
        assign ram1_bwe_b = 4'hF;
*/

   assign ram1_ce_b = 1'b0;
   assign ram1_oe_b = 1'b0;
        assign ram1_adv_ld = 1'b0;


   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input

   // Flash ROM
   assign flash_data = 16'hZ;
   assign flash_address = 24'h0;
   assign flash_ce_b = 1'b1;
   assign flash_oe_b = 1'b1;
   assign flash_we_b = 1'b1;
   assign flash_reset_b = 1'b0;
   assign flash_byte_b = 1'b1;
   // flash_sts is an input
   // RS-232 Interface
   assign rs232_txd = 1'b1;
   assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs
   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs
   // LED Displays
   //assign disp_blank = 1'b1;
   //assign disp_clock = 1'b0;
   //assign disp_rs = 1'b0;
```

```
//assign disp_ce_b = 1'b1;
//assign disp_reset_b = 1'b0;
//assign disp_data_out = 1'b0;
// disp_data_in is an input
// Buttons, Switches, and Individual LEDs
//lab3 assign led = 8'hFF;
// button0, button1, button2, button3, button_enter, button_right,
// button_left, button_down, button_up, and switches are inputs
// User I/Os
assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
assign user4 = 32'hZ;
// 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
// Logic Analyzer
assign analyzer2_data = 16'h0;
assign analyzer2_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;

//////////////////////////////////////////////////////////////////////
//
// final project
//
//////////////////////////////////////////////////////////////////////
// use FPGA's digital clock manager to produce a
// 65MHz clock (actually 64.8MHz)
wire clock_65mhz_unbuf,clock_65mhz,clock_50mhz_unbuf,clock_50mhz;
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
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
                .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
```

```
   defparam reset_sr.INIT = 16'hFFFF;
   // ENTER button is user reset
   wire reset,user_reset;
   debounce db1(.reset(power_on_reset),.clock(clock_65mhz),.noisy(~button_enter),.clean(user_
   assign reset = user_reset | power_on_reset;
           // allow user to adjust volume; passed to audio_buffer in bnewbold_modules.v
   wire vup,vdown;
   debounce bup(.reset(reset),.clock(clock_65mhz),.noisy(~button_up),.clean(vup));
   debounce bdown(.reset(reset),.clock(clock_65mhz),.noisy(~button_down),.clean(vdown));

   // allow user to adjust ifft output window; passed to ifft_wrapper in bnewbold_modules.v
   wire left_button,right_button;
   debounce bleft(.reset(reset),.clock(clock_65mhz),.noisy(~button_left),.clean(left_button))
   debounce bright(.reset(reset),.clock(clock_65mhz),.noisy(~button_right),.clean(right_butto
   // switch 7 for audio tone (debugging)
   wire tone_enable;
   debounce sw7(.reset(reset),.clock(clock_65mhz),.noisy(switch[7]),.clean(tone_enable));

   // switch 6,5,4 for dummy mode selection (debugging)
   wire [2:0] dummy_mode;
   debounce sw6(.reset(reset),.clock(clock_65mhz),.noisy(switch[6]),.clean(dummy_mode[2]));
        debounce sw5(.reset(reset),.clock(clock_65mhz),.noisy(switch[5]),.clean(dummy_mode[1]
        debounce sw4(.reset(reset),.clock(clock_65mhz),.noisy(switch[4]),.clean(dummy_mode[0]

   // switch 3,2 for filter enabling (debugging)
   wire bode_enable;
        wire filter_enable;
   debounce sw2(.reset(reset),.clock(clock_65mhz),.noisy(switch[2]),.clean(bode_enable));
        debounce sw3(.reset(reset),.clock(clock_65mhz),.noisy(switch[3]),.clean(filter_enable

   // switch 0 for interpolation enabling
   wire interp_enable;
   debounce sw0(.reset(reset),.clock(clock_65mhz),.noisy(switch[0]),.clean(interp_enable));

//==== bnewbold BEGIN ========================================================

        // TODO: plug audio_v_count into GUI display DONE
        wire [7:0]  ifft_main_pixel;    //***********
   wire [7:0]  ifft_overlay_pixel; //*  IFFT   *
        wire [9:0]  ifft_hc;
   wire [8:0]  ifft_vc;
        wire audio_ram_re;
        wire [7:0] audio_main_value;
        wire [7:0] audio_overlay_value;
        wire audio_ram_ready;
        wire signed [11:0] pixel_value;
```

```
     wire [10:0] pixel_index;
     wire pixel_ready_w;
     wire pixel_re;
     wire        ifft_read_request;  //******
wire        ifft_read_complete; //*IFFT*

     wire [9:0] shifted_scan_y; //to communicate with audio that user wants to shift scan l
     wire pause; //this is where ifft stage is initially frozen
     wire draw_overlay; //for use here and in the GUI

     windowing windowing0(.clk_65mhz(clock_65mhz), .reset(reset), .h_count(ifft_hc),
             .v_count(ifft_vc), .ram_re(ifft_read_request), .main_value(ifft_main_pixel),
             .overlay_value(ifft_overlay_pixel), .ram_ready(ifft_read_complete),
             .pixel_value(pixel_value), .pixel_index(pixel_index), .pixel_ready(pixel_ready
             .pixel_re(pixel_re),.pause(pause),.shifted_scan_y(shifted_scan_y),.overlay_ena
     wire signed [11:0] sample_value;
     wire [10:0] sample_index;
     wire sample_ready;
     wire sample_re;
     wire [1:0] si_state;
     wire [1:0] si_count;

     spect_interp spect_interp0(.clk_65mhz(clock_65mhz), .reset(reset),
             .pixel_value(pixel_value), .pixel_index(pixel_index), .pixel_ready(pixel_ready
             .pixel_re(pixel_re), .sample_value(sample_value), .sample_index(sample_index)
             .sample_ready(sample_ready), .sample_re(sample_re), .state(si_state), .count(s
             .interp_enable(interp_enable) );
     wire [10:0] ifft_read_index;
     wire ifft_ce;
     wire ifft_rfd;
     wire signed [11:0] ifft_real_input;
     wire signed [11:0] ifft_imag_input;
     wire signed [11:0] later_sample_value;
     wire [10:0] later_sample_index;
     wire later_sample_re;
     wire ifft_fwd_inv_we;
     wire signed [23:0] ifft_real_output;
     wire [10:0] ifft_output_index;
     wire ifft_output_ready;

     // some of these ifft wires are unused
     wire ifft_done, ifft_edone, ifft_busy;

     ifft_wrapper ifft_wrapper0(.clk_65mhz(clock_65mhz), .reset(reset), .up_sample_value(sa
             .up_sample_index(sample_index), .up_sample_ready(sample_ready), .up_sample_re
             .ifft_read_index(ifft_read_index), .ifft_enable(ifft_ce),
```

```
        .ifft_rfd(ifft_rfd), .ifft_real_input(ifft_real_input), .ifft_imag_input(ifft_
        .ifft_real_output(ifft_real_output),
        //.ifft_imag_output(ifft_imag_output),
        .ifft_output_index(ifft_output_index),
        .ifft_output_ready(ifft_output_ready),
        .down_sample_value(later_sample_value), .down_sample_index(later_sample_index
        .ifft_fwd_inv_we(ifft_fwd_inv_we),
        .shift_left(left_button), .shift_right(right_button) );
ifft ifft0(.sclr(reset),
                          .ce(ifft_ce),                                          // clo
                          .fwd_inv_we(ifft_fwd_inv_we),// forward or inverse write ena
                          .rfd(ifft_rfd),                                // ready for d
                          .start(ifft_ce),                       // start (input)
                          .fwd_inv(1'b0),                                // forward=1 d
                          .dv(ifft_output_ready),                // data valid (output)
                          .done(ifft_done),                              // done (outpu
                          .clk(clock_65mhz),                             // clock (inpu
                          .busy(ifft_busy),                              // busy (outpu
                          .edone(ifft_edone),                    // will be done very s
                          .xn_re(ifft_real_input),      // real input component [11:0]
                          .xn_index(ifft_read_index), // input index [9:0]
                          .xk_re(ifft_real_output),     // real output component [11:0
                          .xn_im(ifft_imag_input),      // imaginary input component
                          .xk_index(ifft_output_index)// output index [9:0]
                          );

// TODO: integrate this with actual GUI tap values DONE
wire [6:0] new_tap_index;
wire  signed [11:0] new_tap_value;
wire new_tap_we;
wire [6:0] tap_index;
wire signed [11:0] tap_value;
tap_manager tap_manager0(.clk_65mhz(clock_65mhz), .reset(reset), .new_index(new_tap_in
        .new_value(new_tap_value), .new_we(new_tap_we), .tap_index(tap_index), .tap_va
wire  signed [11:0] filtered_value;
wire [10:0] filtered_index;
wire filtered_ready;
wire filtered_re;
wire [1:0] filter_state;
bode_filter bode_filter0(.clk_65mhz(clock_65mhz), .reset(reset), .tap_index(tap_index
        .tap_value(tap_value), .sample_value(later_sample_value), .sample_index(later_
        .sample_re(later_sample_re),
        .sample_ready(ifft_output_ready), .filtered_value(filtered_value), .filtered_i
        .filtered_ready(filtered_ready), .filtered_re(filtered_re), .bode_enable(bode_
        .state(filter_state));
wire mute;
```

```
        assign mute = 1'b0;
        wire [17:0] to_ac97_sample;
        audio_buffer audio_buffer0(.clk_65mhz(clock_65mhz), .reset(reset), .filtered_value(fil
                .filtered_index(filtered_index),        .filtered_ready(filtered_ready), .filt
                .sample(to_ac97_sample),
                .vol_up(vup), .vol_down(vdown), .mute(mute),
                .filter_enable(filter_enable), .tone_enable(tone_enable), .led_output(led),
                .audio_reset_b(audio_reset_b), .ac97_synch(ac97_synch), .ac97_sdata_out(ac97_s
                .ac97_bit_clock(ac97_bit_clock), .ac97_sdata_in(ac97_sdata_in) );
//==== bnewbold END ========================================================
   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(.vclock(clock_65mhz),.hcount(hcount),.vcount(vcount),
              .hsync(hsync),.vsync(vsync),.blank(blank));
        // wire up to ZBT ram

   wire [35:0] main_write_data;
   wire [35:0] main_read_data;
   wire [18:0] main_addr;
   wire        main_we;
   wire [3:0]  main_bwe;

   zbt_6111 zbt_main(.clk(clock_65mhz), .cen(1'b1), .we(main_we), .bwe(main_bwe), .addr(main_a
                          .write_data(main_write_data), .read_data(main_read_data),
                     .ram_clk(ram0_clk), .ram_we_b(ram0_we_b), .ram_bwe_b(ram0_bwe_b),
                     .ram_address(ram0_address), .ram_data(ram0_data), .ram_cen_b(ram0_cen_b))


   wire [35:0] overlay_write_data;
   wire [35:0] overlay_read_data;
   wire [18:0] overlay_addr;
   wire        overlay_we;
   wire [3:0]  overlay_bwe;

   zbt_6111 zbt_overlay(.clk(clock_65mhz), .cen(1'b1), .we(overlay_we), .bwe(overlay_bwe), .ad
                     .write_data(overlay_write_data), .read_data(overlay_read_data),
                     .ram_clk(ram1_clk), .ram_we_b(ram1_we_b), .ram_bwe_b(ram1_bwe_b),
                     .ram_address(ram1_address), .ram_data(ram1_data), .ram_cen_b(ram1_cen_b))




   // adv7185 initialization module
```

```
adv7185init adv7185(.reset(reset), .clock_27mhz(clock_27mhz),
                    .source(1'b0), .tv_in_reset_b(tv_in_reset_b),
                    .tv_in_i2c_clock(tv_in_i2c_clock),
                    .tv_in_i2c_data(tv_in_i2c_data));

wire [29:0] ycrcb;   // video data (luminance, chrominance)
wire [2:0]  camera_fvh;      // sync for field, vertical, horizontal
wire        pixel_ready;     // data valid

ntsc_decode decode (.clk(tv_in_line_clock1), .reset(reset),
                                                              .tv_in_ycrcb(tv_in_ycrcb[19:1
                                                              .ycrcb(ycrcb), .f(camera_fvh[
                                                              .v(camera_fvh[1]), .h(camera_

//Image Processing
wire        camera_pixel_ready;
wire [7:0]  camera_pixel;
wire [3:0]  camera_processing_mode;  //*** GUI ***
wire threshup,threshdown;
     wire [7:0] threshold;

     debounce threshupclean(reset,clock_65mhz,button0,threshup);
     debounce threshdownclean(reset,clock_65mhz,button1,threshdown);
image_processing img_proc(.clk(tv_in_line_clock1), .reset(reset),
                          .ycrcb(ycrcb), .pixel_ready(pixel_ready),
                          .camera_pixel_ready(camera_pixel_ready),
                          .camera_pixel(camera_pixel),
                          .camera_processing_mode(camera_processing_mode),.threshup(threshu



wire [18:0] camera_addr;
wire [35:0] camera_data;
wire        camera_we;

camera_to_zbt cam_to_zbt(.clk(clock_65mhz), .camera_clk(tv_in_line_clock1), .camera_fvh(ca
                         .camera_pixel_ready(camera_pixel_ready), .camera_pixel(camera_pix
                         .camera_addr(camera_addr), .camera_data(camera_data), .camera_we(




wire [7:0]  display_main_pixel;    //***********
wire [7:0]  display_overlay_pixel; //*   GUI   *
wire [9:0]  display_hc;            //*         *
```

```
wire [8:0]  display_vc;            //***********

wire [18:0] display_addr;
wire [35:0] display_data_main;
wire [35:0] display_data_overlay;

zbt_to_display zbt_to_disp(.clk(clock_65mhz), .reset(reset), .hcount(display_hc), .vcount(d
                           .main_pixel(display_main_pixel), .overlay_pixel(display_overlay_
                           .addr(display_addr),
                           .data_main(display_data_main), .data_overlay(display_data_overla


wire [9:0]  edit_hc;            //***********
wire [8:0]  edit_vc;            //*   GUI   *
wire [7:0]  edit_value;          //***********

wire [18:0] edit_addr;
wire [35:0] edit_data;
wire [3:0]  edit_bwe;

edit_to_zbt ed_to_zbt(.clk(clock_65mhz), .reset(reset), .value(edit_value),
                      .hcount(edit_hc), .vcount(edit_vc),
                      .addr(edit_addr), .data(edit_data),
                      .bwe(edit_bwe));




wire [18:0] ifft_addr;
wire [35:0] ifft_data_main;
wire [35:0] ifft_data_overlay;

zbt_to_ifft zbt_to_ifft(.clk(clock_65mhz), .reset(reset), .hcount(ifft_hc), .vcount(ifft_vc
                        .main_pixel(ifft_main_pixel), .overlay_pixel(ifft_overlay_pixel),
                        .addr(ifft_addr),
                        .data_main(ifft_data_main), .data_overlay(ifft_data_overlay));




wire        camera_capture_request;//*******
wire        camera_capture_dest;   //*     *

wire        edit_write_request;   //*     *
wire        edit_write_dest;      //*     *
```

```
wire        edit_write_complete;  //*******
     wire display_read_request;
memory_control mem_control(.clk(clock_65mhz), .reset(reset),
                           .camera_data(camera_data), .camera_addr(camera_addr),
                           .camera_we(camera_we), .camera_capture_request(camera_capture_re
                           .camera_capture_dest(camera_capture_dest),
                           .display_data_main(display_data_main), .display_data_overlay(dis
                           .display_addr(display_addr), .display_read_request(display_read_
                           .edit_data(edit_data), .edit_addr(edit_addr), .edit_bwe(edit_bwe
                           .edit_write_request(edit_write_request), .edit_write_dest(edit_w
                           .edit_write_complete(edit_write_complete),
                           .ifft_data_main(ifft_data_main), .ifft_data_overlay(ifft_data_ov
                           .ifft_addr(ifft_addr), .ifft_read_request(ifft_read_request),
                           .ifft_read_complete(ifft_read_complete),

                           .main_addr(main_addr), .main_write_data(main_write_data),
                           .main_read_data(main_read_data), .main_we(main_we), .main_bwe(ma
                           .overlay_addr(overlay_addr), .overlay_write_data(overlay_write_d
                           .overlay_read_data(overlay_read_data), .overlay_we(overlay_we),
wire [2:0] pixel;
     wire [11:0] mx,my;
wire [2:0]  btn_click;
     wire no_ack;
     ps2_mouse_xy my_mouse(.clk(clock_65mhz), .reset(reset),
                                                      .ps2_clk(mouse_clock), .ps2_data(mouse
                                                      .mx(mx), .my(my), .btn_click(btn_click


     wire [63:0] hex_data = {threshold,1'b0,btn_click,4'b0,3'b0,no_ack,4'b0,mx,4'b0,my,4'b0
     // Shows current horizontal and vertial counts being read by audio pipeline
     display_16hex display_16hex1 (reset, clock_27mhz, hex_data,
            disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b, disp_data_out);

     wire [7:0] image_pixel, overlay_image_pixel,fourier_pixel;
     ////////////////////////////////////////////////////
     //GUI MODULE WIRING, MANY OF THE INPUTS AND OUTPUTS ARE DECLARED ABOVE
     //FOR OTHER MODULES OR SIMPLY AS CONVENIENCE
     ////////////////////////////////////////////////////

     GUI my_gui(.vclock(clock_65mhz),.reset(reset),
            .hcount(hcount),.vcount(vcount),
            .hsync(hsync),.vsync(vsync),.blank(blank), .display_main_pixel(display_main_pi
                                    .pixel(pixel),.image_pixel(image_pixel),
                                    .display_hc(display_hc),.display_vc(display_vc),
                                    .mx(mx),.my(my), .btn_click(btn_click),
                                    .display_read_request(display_read_request),
```

```
                                        .camera_capture_dest(camera_capture_dest),
                                        .camera_capture_request(camera_capture_request),
                                        .pause(pause),
                                        .draw_overlay(draw_overlay),
                                        .display_overlay_pixel(display_overlay_pixel),
                                        .overlay_image_pixel(overlay_image_pixel),
                                        .ifft_vc(ifft_vc), //controls scan bar
                                        .tap_index(new_tap_index),
                                        .tap_value(new_tap_value),
                                        .tap_bode_we(new_tap_we),
                                        .shifted_scan_y(shifted_scan_y),
                                        .camera_processing_mode(camera_processing_mode),
                                        .fourier_pixel(fourier_pixel)
                                        );

// switch[1:0] selects which video generator to use:
//  00: user's interface
//  01: 1 pixel outline of active video area (adjust screen controls)
//  10: color bars
wire [2:0] rgb;
     reg [7:0] red,green,blue;
     assign rgb = pixel;
     wire too_great=((overlay_image_pixel+image_pixel)>255);
     //pipeline delay will match with fourier at the bottom
     always @(posedge clock_65mhz) begin
             red<=(fourier_pixel!==8'b0)? 255:
                             (draw_overlay&&too_great) ? 255 :
                             (draw_overlay) ? ({8{rgb[2]}}|(overlay_image_pixel+image_pixe
                             ({8{rgb[2]}}|image_pixel);
             green<=(fourier_pixel==8'b0) ? {8{rgb[1]}} | image_pixel:

             blue<={8{rgb[0]}}|image_pixel;
     end

     //from provided ZBT Code
     wire    b,hs,vs;
delayN dn1(clock_65mhz,hsync,hs);    // delay by 3 cycles to sync with ZBT read
delayN dn2(clock_65mhz,vsync,vs);
delayN dn3(clock_65mhz,blank,b);
// VGA Output.  In order to meet the setup and hold times of the
// AD7125, we send it ~clock_65mhz.
     //vga out gets greyscale image_pixel if within image block
     //overlays, however, are drawn as additional red component up to max value (255)
     //test for >255 with too_great
     //a bit complicated
     //if sum greater than 255 and you want to draw the overlay, spit out max
```

```
        //if drawing overlay and not greater, sum them
        //if no request just draw image
        //otherwise just get our regular pixels
        //also for the "fire effect" of the fourier pixels at the bottom
        //to achieve "fire" green is modified and red is high,
   assign vga_out_red = red;
   assign vga_out_green = green;
   assign vga_out_blue = blue;
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_blank_b = ~b;
   assign vga_out_pixel_clock = ~clock_65mhz;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;
   assign analyzer1_data[15:10] = 6'b000000;
        assign analyzer1_data[9:0] = {filter_state,new_tap_index,new_tap_we};
   assign analyzer1_clock = clock_65mhz;

   assign analyzer3_data[15:8] = 8'b0000;
   assign analyzer3_data[7:0] = ifft_main_pixel;
   assign analyzer3_clock = clock_65mhz;
endmodule // finalproj
//////////////////////////////////////////////////////////////////////////////
//
// xvga: Generate XVGA display signals (1024 x 768 @ 60Hz)
// PROVIDED BY 6.111 LAB
//////////////////////////////////////////////////////////////////////////////
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
//////////////////////////////////////////////////////////////////////////////
// parameterized delay line : FROM 6.111 ONLINE CODE
module delayN(clk,in,out);
   input clk;
   input in;
   output out;
   parameter NDELAY = 3;
   reg [NDELAY-1:0] shiftreg;
   wire             out = shiftreg[NDELAY-1];
   always @(posedge clk)
     shiftreg <= {shiftreg[NDELAY-2:0],in};
endmodule // delayN
```

GUI.v

```
//Tyler Hutchison : 6.111
`default_nettype none
//////////////////////////////////////////////////////////////////////////////
//
// THE GUI
//
//////////////////////////////////////////////////////////////////////////////
module GUI (
   input vclock,        // 65MHz clock
   input reset,         // 1 to initialize module
   input [10:0] hcount, // horizontal index of current pixel (0..1023)
   input [9:0]  vcount, // vertical index of current pixel (0..767)
   input hsync,         // XVGA horizontal sync signal (active low)
   input vsync,         // XVGA vertical sync signal (active low)
   input blank,         // XVGA blanking (1 means output black pixel)
        input [11:0] mx,
        input [11:0] my,
        input [2:0] btn_click,
```

```
     input [7:0] display_overlay_pixel,
     input [7:0] display_main_pixel,
     input [8:0] ifft_vc,

     output reg edit_activated,
output reg [2:0] pixel,
     output reg [7:0] image_pixel,
     output reg display_read_request,
     output reg [9:0] display_hc,
     output reg [8:0] display_vc,
     output camera_capture_dest,
     output camera_capture_request,
     output reg draw_overlay,
     output pause,
     output reg [7:0] overlay_image_pixel,
     output tap_bode_we,
     output [6:0] tap_index,
     output [11:0] tap_value,
     output [9:0] shifted_scan_y,
     output reg [3:0] camera_processing_mode,
     output [7:0] fourier_pixel
);
     //LOCATION OF CAPTURED IMAGE AND OVERLAY BOX
     parameter IMAGE_LOC_X=11'd40;
     parameter IMAGE_LOC_Y=10;
     parameter LINE_LOC=0;
     parameter SCAN_BAR_DIM=32;

     //IMAGE PROCESSING PARAMETERS
     parameter THRESH={8'd84, 8'd72,8'd82,8'd69,8'd83,8'd72};
     parameter INVERT={8'd73,8'd78,8'd86};
     parameter NOPROC={8'd78,8'd79,8'd80,8'd82,8'd79,8'd67};
     parameter SOFT={8'd83,8'd79,8'd70,8'd84};
     //button edge detection
     reg btn_last;
     wire btn_edge=(btn_click[2]==1 && btn_last==0);

     reg [7:0] int_val; //keep track of intensity used in edit functionality
     wire [2:0] edit_pixel,image_capture_pixel,overlay_pixel,overlay_enable_pixel,
             pause_pixel,text_pixel,bode_pixel,scan_pixel,mouse_pixel,upper_int_pixel,lowe
             upper_proc_pixel,lower_proc_pixel;
     reg capture_image, capture_overlay, suggest_pause, force_pause; //edit_activated
     assign pause = (suggest_pause|force_pause);
     //suggest pause is a soft pause allowed when scan bar is being dragged

     reg [7:0] edit_value;
```

```
reg edit_write_request;

reg [9:0] edit_hc;
reg [3:0] half_pixelone,half_pixeltwo;
reg [8:0] edit_vc;
reg [10:0] scan_line_x;
reg [9:0] scan_line_y;
reg track; //to let scan bar know that it should track the mouse pointer
reg next_row; //to let the scan bar know when to move as determined by audio stage

//for text display
reg [9:0] cy;
reg [10:0] cx;
reg [87:0] cstring;

//for applying bode plot adjustments to the diagram for visual reinforcement
reg [2:0] c_counter;
reg [6:0] bode_apply_index;
wire [7:0] bode_apply_value;

//this 16 bit register holds the multiplication of the bode plot value with the image
//to display the adjusted value, we only take the most significant 8 bits to
//round from a float to a inteeger
reg [15:0] adjusted_pixel;
reg [15:0] adjusted_overlay; //overlay must be seperate since only added to blue spec
reg [7:0] raw_image_pixel;
reg [2:0] image_outline;
reg [10:0] prev_hcount; //weird problems with counter... so just holding value and wa

//these registers are used to draw lines in the edit mode and ship data to the FIFO i
//of the memory
reg [9:0] prev_edit_x;
reg [8:0] prev_edit_y;

//for fourier display at the bottom, same code used in labkit
//should perhaps pass a wire at some point
reg [7:0] fourier_int;
reg [8:0] overflow_check;
reg too_great;

//camera_capture_dest: high, main image, low, overlay
assign camera_capture_dest=~capture_image;
assign camera_capture_request=capture_image | capture_overlay;
assign shifted_scan_y=scan_line_y-IMAGE_LOC_Y; //if shifted need to communicate with a
```

```
//extra shifts beyond image loc_x are to ignore noise blanking
always @(hcount or vcount) begin
        //outline plot
        if((hcount==(IMAGE_LOC_X+23)&& vcount>=(IMAGE_LOC_Y) && vcount<=(IMAGE_LOC_Y+9
                (vcount==(IMAGE_LOC_Y) && hcount>=(IMAGE_LOC_X+23) && hcount<=(IMAGE_L
                (hcount==(IMAGE_LOC_X+10'd720) && vcount>=(IMAGE_LOC_Y) && vcount<=(IM
                (vcount ==(IMAGE_LOC_Y+10'd480) && hcount>=(IMAGE_LOC_X+23) && hcount<
                image_outline=3'b100;
                display_read_request=0;
                overlay_image_pixel=8'b0;
                image_pixel=8'b0;
        end
        //if within the image bounds
        else if((hcount>=IMAGE_LOC_X+24 && hcount<(IMAGE_LOC_X+10'd720)) &&
                (vcount>=IMAGE_LOC_Y && vcount<IMAGE_LOC_Y+9'd480)) begin
                display_hc=hcount-IMAGE_LOC_X;
                display_vc=vcount-IMAGE_LOC_Y;
                display_read_request=1; //get from memory
                image_outline=3'b0;
                image_pixel=adjusted_pixel[15:8]; //most significant 8 bits of the si
                overlay_image_pixel=adjusted_overlay[15:8];
        end
        else begin //don't do anything otherwise
                display_read_request=0;
                image_pixel=8'b0;
                overlay_image_pixel=8'b0;
                image_outline=3'b0;
        end
end

always @(posedge vclock) begin
        adjusted_pixel<=8'b11111111*bode_apply_value;
        //below here is all mouse checking code including button clicking, edit mode,
        if(reset) begin
                c_counter<=0;
                edit_activated<=0;
                capture_image<=0;
                draw_overlay<=0;
                capture_overlay<=0;
                suggest_pause<=0;
                force_pause<=0;
                scan_line_x<=IMAGE_LOC_X-SCAN_BAR_DIM; //reset scan bar
                scan_line_y<=IMAGE_LOC_Y-LINE_LOC;
                track<=0;
                int_val<=8'b0;
                camera_processing_mode<=4'b0;
```

```
        half_pixelone<=3'b0;
        half_pixeltwo<=3'b0;
        pixel<=3'b0;
end
////////////////////////////////////////////////////
//////////////////////////BUTTON CONTROL////////////////
////////////////////////////////////////////////////
//intensity arrows width 16 for now
else if(btn_edge && ((mx>=10'd800 && mx<=10'd808 && my>=10'd75+mx-10'd800 && m
                        (mx>=10'd792 && mx<=10'd800 && my>=10'd75+10'd800-mx &
        int_val<=int_val+1;
else if(btn_edge && ((mx>=10'd800 && mx<=10'd808 && my<=(10'd118+10'd800-mx) &
                        (mx>=10'd792 && mx<=10'd800 && my<=(10'd118+mx-10'd80
        int_val<=int_val-1;
//image processing mode arrows
else if(btn_edge && ((mx>=10'd800 && mx<=10'd808 && my>=10'd320+mx-10'd800 &&
                        (mx>=10'd792 && mx<=10'd800 && my>=10'd320+10'd800-mx
        camera_processing_mode<=camera_processing_mode+1;
else if(btn_edge && ((mx>=10'd800 && mx<=10'd808 && my<=(10'd363+10'd800-mx) &
                        (mx>=10'd792 && mx<=10'd800 && my<=(10'd363+mx-10'd80
        camera_processing_mode<=camera_processing_mode-1;
//edit (800,130)
else if((mx>=10'd800 && mx<= (10'd800+10'd128)) && (my>=9'd130 && my<=(9'd130+
        edit_activated<=1^edit_activated;
//imagecapture (800,163)
else if((mx>=10'd800 && mx<=(10'd800+10'd128)) && (my>=9'd163 && my<=(9'd162+9
        capture_image<=1^capture_image;
        capture_overlay<=0;
end
//overlay (800,196)
else if((mx>=10'd800 && mx<=(10'd800+10'd128)) && (my>=9'd196 && my<=(9'd196+9
        capture_overlay<=1^capture_overlay;
        capture_image<=0;
end
//enable (800,229)
else if((mx>=10'd800 && mx<=(10'd800+10'd128)) && (my>=9'd229 && my<=(9'd229+9
        draw_overlay<=1^draw_overlay;
//pause(800,262)
else if((mx>=10'd800 && mx<=(10'd800+10'd128)) && (my>=9'd262 && my<=(9'd262+9
        force_pause<=1^force_pause;
//should the scan bar track the mouse? button should still be down
//or should scan bar move due to next audio sample being played
// it is assumed that this flag will never be high if the system is paused
////////////////////////////////////////////////////////////////////////
//////////////////////////////SCAN BAR CONTROL//////////////////////////
////////////////////////////////////////////////////////////////////////
```

```
if(track && btn_click[2]) begin
                if(my<=(IMAGE_LOC_Y-LINE_LOC)) scan_line_y<=IMAGE_LOC_Y-LINE_L
                else if(my>=(IMAGE_LOC_Y+LINE_LOC+9'd480)) scan_line_y<=IMAGE_
                else scan_line_y<=my;

                suggest_pause<=1;
end
//is the mouse within the scan blob on the left of the image and being clicked
else if((mx>=scan_line_x && mx<=(scan_line_x+SCAN_BAR_DIM)) &&
                        (my>=scan_line_y && my<=(scan_line_y+SCAN_BAR_DIM)) &&
                        track<=1;
                        suggest_pause<=1;
end
else if(!pause && (ifft_vc+IMAGE_LOC_Y)) //if it's just going! and... audio ha
        scan_line_y<=ifft_vc+IMAGE_LOC_Y;//offset since ifft_vc goes 0-479
else begin
        suggest_pause<=0;
        track<=0; //if no more button input, shut down the tracking
end

//////////////////////////////////////////////////////////////
//////////////////////////ADJUST DISPLAY PIXELS/////////////////
//////////////////////////////////////////////////////////////

if(hcount==IMAGE_LOC_X-2) begin //if just =
        prev_hcount<=IMAGE_LOC_X-1;
        bode_apply_index<=0;
end
else if(hcount==(prev_hcount+6)) begin //if moved 6 pixels
        prev_hcount<=hcount;
        bode_apply_index<=bode_apply_index+1'b1;
end
adjusted_pixel<=bode_apply_value*display_main_pixel;
adjusted_overlay<=bode_apply_value*display_overlay_pixel;
overflow_check <= adjusted_pixel[15:8] + adjusted_overlay[15:8];
too_great <= overflow_check[8]; //if msb of overflow_check is high
if(draw_overlay && too_great)
        fourier_int<=255;
else if(draw_overlay)
        fourier_int<=overflow_check[7:0];
else
        fourier_int<=image_pixel;
//////////////////////////////////////////////////////////////
//////////////////////////EDIT MODE/////////////////////////////
//////////////////////////////////////////////////////////////
//is the GUI in edit mode and should we update pixels
```

```
//since dx,dy can change by a large amount we need to interpolate
//between these changes, this math will be noted in detail
if(edit_activated) begin
        if((mx>=IMAGE_LOC_X && mx<=(IMAGE_LOC_X+10'd720)) &&
                (my>=IMAGE_LOC_Y && my<=(IMAGE_LOC_Y+9'd480)) && btn_click[2]
                        edit_write_request<=1;
                        edit_hc<=mx-IMAGE_LOC_X;
                        edit_vc<=my-IMAGE_LOC_Y;
                        edit_value<=int_val; //only need to update intensity
                        prev_edit_x<=mx;
                        prev_edit_y<=my;
        end
        else if(edit_write_request && btn_click[2]) begin //if editting has al
                if(prev_edit_x != mx || prev_edit_y != my) begin //if the mous
                        if(my==prev_edit_y) begin//horizontal line, prev_edit_
                                if(prev_edit_x>mx) prev_edit_x<=prev_edit_x-1
                                else prev_edit_x<=prev_edit_x+1'b1;
                        end
                        else if(mx==prev_edit_x) begin //vertical line prev_ed
                                if(prev_edit_y>my) prev_edit_y<=prev_edit_y-1
                                else prev_edit_y<=prev_edit_y+1'b1;
                        end
                        else begin //line of arbitrary slope! uh oh!
                                if(prev_edit_x>mx && prev_edit_y>my) begin
                                        prev_edit_x<=prev_edit_x+1'b1;
                                        prev_edit_y<=prev_edit_y+1'b1;
                                end
                                else if(prev_edit_x>mx && prev_edit_y<my) begi
                                        prev_edit_x<=prev_edit_x+1'b1;
                                        prev_edit_y<=prev_edit_y-1'b1;
                                end
                                else if(prev_edit_x<mx && prev_edit_y>my) begi
                                        prev_edit_x<=prev_edit_x-1'b1;
                                        prev_edit_y<=prev_edit_y+1'b1;
                                end
                                else if(prev_edit_x<mx && prev_edit_y<my) begi
                                        prev_edit_x<=prev_edit_x-1'b1;
                                        prev_edit_y<=prev_edit_y-1'b1;
                                end
                        end
                        edit_hc<=prev_edit_x-IMAGE_LOC_X;
                        edit_vc<=prev_edit_y-IMAGE_LOC_Y;
                end
        end
        else edit_write_request<=0;
end
```

```
else edit_write_request<=0;

//used to edge detect the button
btn_last<=btn_click[2];

////////////////////////////////////////////////////////
//////////////////////////STRING DISPLAY////////////////
////////////////////////////////////////////////////////
//get those strings displaying!
if(hcount>800) begin
//intensity
        if(vcount>10'd84 & vcount<10'd107) begin
                cstring<={8'd73,8'd78,8'd84,
                8'd48+int_val[7],8'd48+int_val[6],8'd48+int_val[5],
                8'd48+int_val[4],8'd48+int_val[3],8'd48+int_val[2],
                8'd48+int_val[1],8'd48+int_val[0]}; //"INT00000000" or whateve
                cy<=10'd84;
        end
//edit
        else if(vcount>10'd130 & vcount<10'd162)
        begin
                cstring<={8'd69,8'd68,8'd73,8'd84,56'b0}; //"EDIT" then NULL
                cy<=10'd131;
        end
//image
        else if(vcount>10'd163 & vcount<10'd195) begin
                cstring<={8'd73,8'd77,8'd65,8'd71,8'd69,48'b0};//"IMAGE" then
                cy<=10'd164;
        end
//overlay
        else if(vcount>10'd196 & vcount<10'd228) begin
                cstring<={8'd79,8'd86,8'd69,8'd82,8'd76,8'd65,8'd89,32'b0};//'
                cy<=10'd197;
        end
//enable
        else if(vcount>10'd229 & vcount<10'd261) begin
                cstring<={8'd69,8'd78,8'd65,8'd66,8'd76,8'd69,40'b0};//"ENABL
                cy<=10'd230;
        end
//pause
        else if(vcount>10'd262 & vcount<10'd294) begin
                cstring<={8'd80,8'd65,8'd85,8'd83,8'd69,48'b0}; //"PAUSE"
                cy<=10'd263;
        end
//image processing status
        else if(vcount>10'd329 & vcount<10'd355) begin
```

```
                        cx<=11'd740;
                        cstring<=(camera_processing_mode[0] && camera_processing_mode
                                        (~camera_processing_mode[0] && camera
                                        (camera_processing_mode[0] && camera_
                                        (~camera_processing_mode[0]&&camera_p
                                        (camera_processing_mode[0])

                        cy<=10'd329;
                end
                else begin
                        cx<=11'd800;
                end
        end
        //too many sprites, pipeline!
        half_pixelone<=mouse_pixel|edit_pixel|image_capture_pixel|overlay_pixel|overla
        half_pixeltwo<=|scan_pixel|upper_int_pixel|lower_int_pixel|image_outline|upper
        pixel<=half_pixelone|half_pixeltwo;
end

        ////////////////////////////////////////////////////////
        ////////////////////////SPRITE DECLARATIONS/////////////
        ////////////////////////////////////////////////////////
        button #(.WIDTH(128),.HEIGHT(32),.COLOR(3'b100),.COLOR_CLICKED(3'b110),.LOCATI
edit_button(.reset(reset),.hcount(hcount),.vcount(vcount),
        .pixel(edit_pixel), .activated(edit_activated));

        button #(.WIDTH(128),.HEIGHT(32),.COLOR(3'b100),.COLOR_CLICKED(3'b110),.LOCATI
image_capture_button(.reset(reset),.hcount(hcount),.vcount(vcount),
        .pixel(image_capture_pixel),.activated(capture_image));

        button #(.WIDTH(128),.HEIGHT(32),.COLOR(3'b100),.COLOR_CLICKED(3'b110),.LOCATI
overlay_button(.reset(reset),.hcount(hcount),.vcount(vcount),
        .pixel(overlay_pixel),.activated(capture_overlay));

        button #(.WIDTH(128),.HEIGHT(32),.COLOR(3'b100),.COLOR_CLICKED(3'b110),.LOCATI
overlay_enable_button(.reset(reset),.hcount(hcount),.vcount(vcount),
        .pixel(overlay_enable_pixel),.activated(draw_overlay));

        button #(.WIDTH(128),.HEIGHT(32),.COLOR(3'b100),.COLOR_CLICKED(3'b110),.LOCATI
pause_button(.reset(reset),.hcount(hcount),.vcount(vcount),
        .pixel(pause_pixel),.activated(pause));

        triangle #(.DIM(8),.COLOR(3'b111),.CLICK_COLOR(3'b001))
mouse_pointer(.x(mx),.y(my),.hcount(hcount),.vcount(vcount), .btn_click(btn_click[2]),
        .pixel(mouse_pixel));
```

```
                triangle #(.DIM(8),.COLOR(3'b111),.CLICK_COLOR(3'b111))
                upper_intensity_arrow(.x(11'd800),.y(11'd75),.hcount(hcount),.vcount(vcount),
                        .pixel(upper_int_pixel));

                flipped_triangle #(.DIM(8),.COLOR(3'b111))
                lower_intensity_arrow(.x(11'd800),.y(11'd110),.hcount(hcount),.vcount(vcount)
                        .pixel(lower_int_pixel));

                triangle #(.DIM(8),.COLOR(3'b111),.CLICK_COLOR(3'b111))
                upper_proc_arrow(.x(11'd800),.y(11'd320),.hcount(hcount),.vcount(vcount),.btn
                        .pixel(upper_proc_pixel));

                flipped_triangle #(.DIM(8),.COLOR(3'b111))
                lower_proc_arrow(.x(11'd800),.y(11'd355),.hcount(hcount),.vcount(vcount),
                        .pixel(lower_proc_pixel));

                scan_bar #(.WIDTH(32),.HEIGHT(32),.LINE_LOC(0),.COLOR(3'b100))
                                myscan_bar(.reset(reset),.x(scan_line_x),.y(scan_line_y),.hcou


                draw_2dfourier #(.LINE_LOC(0),.IMAGE_LOC_X(IMAGE_LOC_X),.TWOD_LOC_Y(490))
                my_2dplot(.clk(vclock),.reset(reset),.hcount(hcount),.vcount(vcount),
                        .scan_bar_y(scan_line_y),.intensity(fourier_int),.pixel(fourier_pixel

                bode_plot #(.LOC_X(762),.LOC_Y(491))
                my_bode (.clk(vclock),.reset(reset),.hcount(hcount),.vcount(vcount),.mx(mx),.m
                                        .pixel(bode_pixel),.tap_bode_we(tap_bode_we),.bode_app
                                        .bode_apply_value(bode_apply_value),.tap_index(tap_ind


        char_string_display my_text(vclock,hcount,vcount,text_pixel,cstring,cx,cy);

endmodule
```

image_processing.v

```
//Dima Turbiner
module image_processing(clk, reset, ycrcb, pixel_ready, camera_pixel_ready, camera_pixel, came
    input clk;
    input reset;
    input [29:0] ycrcb;
    input pixel_ready;
    output camera_pixel_ready;
    output [7:0] camera_pixel;
    input [3:0] camera_processing_mode;
         input threshup,threshdown;
```

```
            output reg [8:0] threshold;

            reg threshup_last,threshdown_last;
            wire threshup_edge=(threshup!=threshup_last && threshup);
            wire threshdown_edge=(threshdown!=threshdown_last && threshdown);

            reg camera_pixel_ready;
            wire [7:0] inv_pixel = camera_processing_mode[0] ? ~ycrcb[29:22] : ycrcb[29:22];
            wire [7:0] thrsh_pixel = camera_processing_mode[1] ? ((inv_pixel < threshold)? 8'b0
            assign camera_pixel = thrsh_pixel;

           always @(posedge clk) begin
                   if(reset) threshold<=8'b0;
                   else if(threshup_edge) threshold<=threshold+10;
                   else if(threshdown_edge) threshold<=threshold-10;
                   threshup_last<=threshup;
                   threshdown_last<=threshdown;
                   camera_pixel_ready <= pixel_ready;
           end




endmodule
```

memory_control.v

```
//Dima Turbiner
module memory_control(clk, reset, camera_data, camera_addr, camera_we, camera_capture_request
        display_data_main, display_data_overlay, display_addr, display_read_request, edit_data
        edit_write_dest, edit_write_complete, ifft_data_main, ifft_data_overlay, ifft_addr, i
        main_addr, main_write_data, main_read_data, main_we, main_bwe, overlay_addr, overlay_
        overlay_we, overlay_bwe);

   input clk;
   input reset;

   input [35:0] camera_data;
   input [18:0] camera_addr;
   input        camera_we;
   input        camera_capture_request;
   input        camera_capture_dest;  //store to main or overlay

   output [35:0] display_data_main;
   output [35:0] display_data_overlay;
   input [18:0]  display_addr;
```

```
input         display_read_request;

input [35:0]  edit_data;
input [18:0]  edit_addr;
input [3:0]   edit_bwe;
input         edit_write_request;
input         edit_write_dest;  //store to main or overlay
output        edit_write_complete; //because the write is asynchronous
output [35:0] ifft_data_main;
output [35:0] ifft_data_overlay;
input [18:0]  ifft_addr;
input         ifft_read_request;
output        ifft_read_complete;

output [18:0] main_addr;
output [35:0] main_write_data;
input [35:0]  main_read_data;
output        main_we;
output [3:0]  main_bwe;

output [18:0] overlay_addr;
output [35:0] overlay_write_data;
input [35:0]  overlay_read_data;
output        overlay_we;
output [3:0]  overlay_bwe;




     // code to write pattern to ZBT memory
reg [31:0]   count;
always @(posedge clk) count <= reset ? 0 : count + 1;
wire [18:0]  vram_addr2 = count[0+18:0];
wire [35:0]  vpat = {4{count[3+4:4],4'b0}};

     assign main_addr = camera_capture_request ? camera_addr :
                                    (display_read_request ? display_addr :
                                    (ifft_read_request ? ifft_addr :
                                    (edit_write_request ? edit_addr :0)));

     assign main_bwe = (edit_write_request &&
                             !camera_capture_request && !display_read_request &&
                             !ifft_read_request) ? edit_bwe : 4'h0;

     assign main_write_data = camera_capture_request ? camera_data :
                                                                    (edit_write_request ?
```

```
assign main_we = camera_capture_request ? (camera_capture_dest? 0 : camera_we) :
                                          (edit_write_request && !camera_capture_reques
                                          && !ifft_read_request) ? (edit_write_dest ? 0



assign overlay_addr = camera_capture_request ? camera_addr :
                                (display_read_request ? display_addr :
                                (ifft_read_request ? ifft_addr :
                                (edit_write_request ? edit_addr :0)));
assign overlay_bwe = (edit_write_request &&
                        !camera_capture_request && !display_read_request &&
                        !ifft_read_request) ? edit_bwe : 4'h0;
assign overlay_write_data = camera_capture_request ? camera_data :
                                                                (edit_write_request ?
assign overlay_we = camera_capture_request ? (camera_capture_dest? camera_we :0) :
                                          (edit_write_request && !camera_capture_reques
                                          && !ifft_read_request) ? (edit_write_dest ? 1



assign display_data_main = (display_read_request && !camera_capture_request) ? main_re
assign display_data_overlay = (display_read_request && !camera_capture_request) ? ove


assign ifft_data_main = main_read_data;
assign ifft_data_overlay = overlay_read_data;


// verify if ifft read data and addt have been stable for at least 2 cycles:
reg [3:0] ifft_clocks;
always @(posedge clk) begin
        if(ifft_read_request && (!camera_capture_request) && (!display_read_request))
                ifft_clocks <= (ifft_clocks==4'd3)? ifft_clocks : (ifft_clocks+1);
        end
        //if ifft read operation got interrupted
        else begin
                ifft_clocks <= 0;
        end
end

assign ifft_read_complete = (ifft_clocks==4'd3);



//verify if edit WE has been stable for at least one cycle:
```

```
        reg [3:0] edit_clocks;
        always @(posedge clk) begin
                if(edit_write_request && !ifft_read_request && !camera_capture_request && !di
                        edit_clocks <= (edit_clocks==4'd2) ? edit_clocks : (edit_clocks+1);
                else
                        edit_clocks <= 0;
        end


        assign edit_write_complete = (edit_clocks==4'd2);


endmodule
```

ntsc2zbt.v

```
//
// File:   ntsc2zbt.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Example for MIT 6.111 labkit showing how to prepare NTSC data
// (from Javier's decoder) to be loaded into the ZBT RAM for video
// display.
//
// The ZBT memory is 36 bits wide; we only use 32 bits of this, to
// store 4 bytes of black-and-white intensity data from the NTSC
// video input.
//////////////////////////////////////////////////////////////////////////////
// Prepare data and address values to fill ZBT memory with NTSC data
module ntsc_to_zbt(clk, vclk, fvh, dv, din, ntsc_addr, ntsc_data, ntsc_we, sw);
   input         clk;   // system clock
   input         vclk;  // video clock from camera
   input [2:0]   fvh;
   input         dv;
   input [7:0]   din;
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output        ntsc_we;       // write enable for NTSC data
   input         sw;            // switch which determines mode (for debugging)
   parameter     COL_START = 10'd30;
   parameter     ROW_START = 10'd30;
   // here put the luminance data from the ntsc decoder into the ram
   // this is for 1024 x 768 XGA display
   reg [9:0]     col = 0;
   reg [9:0]     row = 0;
```

```
reg [7:0]     vdata = 0;
reg           vwe;
reg           old_dv;
reg           old_frame;     // frames are even / odd interlaced
reg           even_odd;      // decode interlaced frame to this wire

wire          frame = fvh[2];
wire          frame_edge = frame & ~old_frame;
always @ (posedge vclk) //LLC1 is reference
  begin
     old_dv <= dv;
     vwe <= dv && !fvh[2] & ~old_dv; // if data valid, write it
     old_frame <= frame;
     even_odd = frame_edge ? ~even_odd : even_odd;
     if (!fvh[2])
       begin
          col <= fvh[0] ? COL_START :
                 (!fvh[2] && !fvh[1] && dv && (col < 1024)) ? col + 1 : col;
          row <= fvh[1] ? ROW_START :
                 (!fvh[2] && fvh[0] && (row < 768)) ? row + 1 : row;
          vdata <= (dv && !fvh[2]) ? din : vdata;
       end
  end
// synchronize with system clock
reg [9:0] x[1:0],y[1:0];
reg [7:0] data[1:0];
reg       we[1:0];
reg       eo[1:0];
always @(posedge clk)
  begin
     {x[1],x[0]} <= {x[0],col};
     {y[1],y[0]} <= {y[0],row};
     {data[1],data[0]} <= {data[0],vdata};
     {we[1],we[0]} <= {we[0],vwe};
     {eo[1],eo[0]} <= {eo[0],even_odd};
  end
// edge detection on write enable signal
reg old_we;
wire we_edge = we[1] & ~old_we;
always @(posedge clk) old_we <= we[1];
// shift each set of four bytes into a large register for the ZBT

reg [31:0] mydata;
always @(posedge clk)
  if (we_edge)
    mydata <= { mydata[23:0], data[1] };
```

```
   // compute address to store data in
   wire [18:0] myaddr = {1'b0, y[1][8:0], eo[1], x[1][9:2]};
   // alternate (256x192) image data and address
   wire [31:0] mydata2 = {data[1],data[1],data[1],data[1]};
   wire [18:0] myaddr2 = {1'b0, y[1][8:0], eo[1], x[1][7:0]};
   // update the output address and data only when four bytes ready
   reg [18:0] ntsc_addr;
   reg [35:0] ntsc_data;
   wire       ntsc_we = sw ? we_edge : (we_edge & (x[1][1:0]==2'b00));
   always @(posedge clk)
     if ( ntsc_we )
       begin
          ntsc_addr <= sw ? myaddr2 : myaddr;   // normal and expanded modes
          ntsc_data <= sw ? {4'b0,mydata2} : {4'b0,mydata};
       end

endmodule // ntsc_to_zbt
```

ps2_mousenew.v

```
//Consistently functioning mouse code, Thanks Ben and TAs involved!
// ps2_mouse_xy gives a high-level interface to the mouse, which
// keeps track of the "absolute" x,y position (within a parameterized
// range) and also returns button presses.
module ps2_mouse_xy(clk, reset, ps2_clk, ps2_data, mx, my, btn_click,error_no_ack);

   input clk, reset;
   inout ps2_clk, ps2_data;     // data to/from PS/2 mouse
   output [11:0] mx, my;        // current mouse position, 12 bits
   output [2:0]  btn_click;     // button click: Left-Middle-Right
   output error_no_ack;
   // module parameters
   parameter     MAX_X = 1023;
   parameter     MAX_Y = 767;

   // low level mouse driver

   wire [8:0]    dx, dy;
   wire [2:0]    btn_click;
   wire          data_ready;
   wire          error_no_ack;
   wire [1:0]    ovf_xy;
   wire          streaming;

//   original 6.111 fall 2005 Verilog - appears to be buggy  so it has been
//   commented out.
```

```
//   ps2_mouse m1(clk,reset,ps2_clk,ps2_data,dx,dy,ovf_xy, btn_click,
//               data_ready,streaming);
//
// using ps2_mouse Verilog from Opencore
// divide the clk by a factor of two sot that it works with 65mhz and the original timing
// parameters in the open core source.
// if the Verilog doesn't work the user should update the timing parameters. This  Verilog as
// 65Mhz clock; seems to work with 32.5mhz without problems. GPH  11/23/2008 with
// assist from BG
ps2_mouse_interface
   #(.WATCHDOG_TIMER_VALUE_PP(26000),
   .WATCHDOG_TIMER_BITS_PP(15),
   .DEBOUNCE_TIMER_VALUE_PP(246),
   .DEBOUNCE_TIMER_BITS_PP(8))
   m1(
  .clk(clk),
  .reset(reset),
  .ps2_clk(ps2_clk),
  .ps2_data(ps2_data),
  .x_increment(dx),
  .y_increment(dy),
  .data_ready(data_ready),
  .read(1'b1),  // force a read
  .left_button(btn_click[2]),
  .right_button(btn_click[0]),  // rx_read_o
  .error_no_ack(error_no_ack)
  );

//  error_no_ack  not used
   // Update "absolute" position of mouse

   reg [11:0]  mx, my;
   wire        sx = dx[8];              // signs
   wire        sy = dy[8];
   wire [8:0]  ndx = sx ? {0,~dx[7:0]}+1 : {0,dx[7:0]}; // magnitudes
   wire [8:0]  ndy = sy ? {0,~dy[7:0]}+1 : {0,dy[7:0]};

   always @(posedge clk) begin
      mx <= reset ? 0 :
            data_ready ? (sx ? (mx>ndx ? mx - ndx : 0)
                          : (mx < MAX_X - ndx ? mx+ndx : MAX_X)) : mx;
      // note Y is flipped for video cursor use of mouse
      my <= reset ? 0 :
            data_ready ? (sy ? (my < MAX_Y - ndy ? my+ndy : MAX_Y)
                          : (my>ndy ? my - ndy : 0))  : my;
//          data_ready ? (sy ? (my>ndy ? my - ndy : 0)
```

```
//                          : (my < MAX_Y - ndy ? my+ndy : MAX_Y)) : my;
   end

endmodule
//-----------------------------------------------------------------------------
//
// Author: John Clayton
// Date  : April 30, 2001
// Update: 6/06/01 copied this file from ps2.v (pared down).
// Update: 6/07/01 Finished initial coding efforts.
// Update: 6/09/01 Made minor changes to state machines during debugging.
//                 Fixed errors in state transitions.  Added state to m2
//                 so that "reset" causes the mouse to be initialized.
//                 Removed debug port.
//
//
//
//
//
// Description
//-----------------------------------------------------------------------------
// This is a state-machine driven serial-to-parallel and parallel-to-serial
// interface to the ps2 style mouse.  The state diagram for part of the
// m2 state machine was obtained from the work of Rob Chapman, as published
// at:
// www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1998_w/mouse_notes.html
//
//
// Some aspects of the mouse interface are not implemented (e.g, verifying
// the FA response code from the mouse when enabling streaming mode.)
// However, the mouse interface was designed so that "hot plugging" a mouse
// into the connector should cause the interface to send the F4 code to the
// mouse in order to enable streaming.  By this means, the mouse begins to
// operate, and no reset pulse should be needed.
//
// Similarly, there is a "watchdog" timer implemented, so that during periods
// of inactivity, the bit_count is cleared to zero.  Therefore, the effects of
// a bad count value are corrected, and internal errors of that type are not
// propagated into subsequent packet receive operations.
//
// To enable the streaming mode, F4 is sent to the mouse.
// The mouse responds with FA to acknowledge the command, and then enters
// streaming mode at the default rate of 100 packets per second (transmission
// of packets ceases when the activity at the mouse is not longer sensed.)
//
// There are additional commands to change the sampling rate and resolution
```

```
// of the mouse reported data.  Those commands are not implemented here.
// (E8,XX = set resolution 0,1,2,3)
// (E7 = set scaling 2:1)
// (E6 = reset scaling)
// (F3,XX = set sampling rate to XX packets per second.)
//
// At this time I do not know any of the command related to using the
// wheel of a "wheel mouse."
//
// The packets consists of three bytes transmitted in sequence.  The interval
// between these bytes has been measured on two different mice, and found to
// be different.  On the slower (older) mouse it was approximately 345
// microseconds, while on a newer "wheel" mouse it was approximately 125
// microseconds.  The watchdog timer is designed to cause processing of a
// complete packet when it expires.  Therefore, the watchdog timer must last
// for longer than the "inter-byte delay" between bytes of the packet.
// I have set the default timer value to 400 usec, for my 49.152 MHz clock.
// The timer value and size of the timer counter is settable by parameters,
// so that other clock frequencies and settings may be used.  The setting for
// the watchdog timeout is not critical -- it only needs to be greater than
// the inter-byte delay as data is transmitted from the mouse, and no less
// than 60usec.
//
// Each "byte" of the packet is transmitted from the mouse as follows:
//
// 1 start bit, 8 data bits, 1 odd parity bit, 1 stop bit. == 11 bits total.
// (The data bits are sent LSB first)
//
// The data bits are formatted as follows:
//
// byte 0: YV, XV, YS, XS, 1, 0, R, L
// byte 1: X7..X0
// byte 2: Y7..Y0
//
// Where YV, XV are set to indicate overflow conditions.
//       XS, YS are set to indicate negative quantities (sign bits).
//        R,  L are set to indicate buttons pressed, left and right.
//
//
//
// The interface to the ps2 mouse (like the keyboard) uses clock rates of
// 30-40 kHz, dependent upon the mouse itself.  The mouse generates the
// clock.
// The rate at which the state machine runs should be at least twice the
// rate of the ps2_clk, so that the states can accurately follow the clock
// signal itself.  Four times oversampling is better.  Say 200kHz at least.
```

```
// In order to run the state machine extremely fast, synchronizing flip-flops
// have been added to the ps2_clk and ps2_data inputs of the state machine.
// This avoids poor performance related to slow transitions of the inputs.
//
// Because this is a bi-directional interface, while reading from the mouse
// the ps2_clk and ps2_data lines are used as inputs.  While writing to the
// mouse, however (which is done when a "packet" of less than 33 bits is
// received), both the ps2_clk and ps2_data lines are sometime pulled low by
// this interface.  As such, they are bidirectional, and pullups are used to
// return them to the "high" state, whenever the drivers are set to the
// high impedance state.
//
// Pullups MUST BE USED on the ps2_clk and ps2_data lines for this design,
// whether they be internal to an FPGA I/O pad, or externally placed.
// If internal pullups are used, they may be fairly weak, causing bounces
// due to crosstalk, etc.  There is a "debounce timer" implemented in order
// to eliminate erroneous state transitions which would occur based on bounce.
// Parameters are provided to configure the debounce timer for different
// clock frequencies.  2 or 3 microseconds of debounce should be plenty.
// You may possibly use much less, if your pullups are strong.
//
// A parameters is provided to configure a 60 microsecond period used while
// transmitting to the mouse.  The 60 microsecond period is guaranteed to be
// more than one period of the ps2_clk signal.
//
//
//-----------------------------------------------------------------------------------
`resetall
`timescale 1ns/100ps
`define TOTAL_BITS   33  // Number of bits in one full packet
module ps2_mouse_interface (
  clk,
  reset,
  ps2_clk,
  ps2_data,
  left_button,
  right_button,
  x_increment,
  y_increment,
  data_ready,       // rx_read_o
  read,             // rx_read_ack_i
  error_no_ack
  );
// Parameters
// The timer value can be up to (2^bits) inclusive.
parameter WATCHDOG_TIMER_VALUE_PP = 26000; // Number of sys_clks for 400usec.
```

```
parameter WATCHDOG_TIMER_BITS_PP  = 15;    // Number of bits needed for timer
parameter DEBOUNCE_TIMER_VALUE_PP = 186;   // Number of sys_clks for debounce
parameter DEBOUNCE_TIMER_BITS_PP  = 8;     // Number of bits needed for timer
// State encodings, provided as parameters
// for flexibility to the one instantiating the module.
// In general, the default values need not be changed.
// There are three state machines: m1, m2 and m3.
// States chosen as "default" states upon power-up and configuration:
//    "m1_clk_h"
//    "m2_wait"
//    "m3_data_ready_ack"
parameter m1_clk_h = 0;
parameter m1_falling_edge = 1;
parameter m1_falling_wait = 3;
parameter m1_clk_l = 2;
parameter m1_rising_edge = 6;
parameter m1_rising_wait = 4;
parameter m2_reset = 14;
parameter m2_wait = 0;
parameter m2_gather = 1;
parameter m2_verify = 3;
parameter m2_use = 2;
parameter m2_hold_clk_l = 6;
parameter m2_data_low_1 = 4;
parameter m2_data_high_1 = 5;
parameter m2_data_low_2 = 7;
parameter m2_data_high_2 = 8;
parameter m2_data_low_3 = 9;
parameter m2_data_high_3 = 11;
parameter m2_error_no_ack = 15;
parameter m2_await_response = 10;
parameter m3_data_ready = 1;
parameter m3_data_ready_ack = 0;
// I/O declarations
input clk;
input reset;
inout ps2_clk;
inout ps2_data;
output left_button;
output right_button;
output [8:0] x_increment;
output [8:0] y_increment;
output data_ready;
input read;
output error_no_ack;
reg left_button;
```

```
reg right_button;
reg [8:0] x_increment;
reg [8:0] y_increment;
reg data_ready;
reg error_no_ack;
// Internal signal declarations
wire watchdog_timer_done;
wire debounce_timer_done;
wire packet_good;
reg [`TOTAL_BITS-1:0] q;  // Shift register
reg [2:0] m1_state;
reg [2:0] m1_next_state;
reg [3:0] m2_state;
reg [3:0] m2_next_state;
reg m3_state;
reg m3_next_state;
reg [5:0] bit_count;      // Bit counter
reg [WATCHDOG_TIMER_BITS_PP-1:0] watchdog_timer_count;
reg [DEBOUNCE_TIMER_BITS_PP-1:0] debounce_timer_count;
reg ps2_clk_hi_z;     // Without keyboard, high Z equals 1 due to pullups.
reg ps2_data_hi_z;    // Without keyboard, high Z equals 1 due to pullups.
reg clean_clk;        // Debounced output from m1, follows ps2_clk.
reg rising_edge;      // Output from m1 state machine.
reg falling_edge;     // Output from m1 state machine.
reg output_strobe;    // Latches data data into the output registers
//-----------------------------------------------------------------------------
// Module code
assign ps2_clk = ps2_clk_hi_z?1'bZ:1'b0;
assign ps2_data = ps2_data_hi_z?1'bZ:1'b0;
// State register
always @(posedge clk)
begin : m1_state_register
  if (reset) m1_state <= m1_clk_h;
  else m1_state <= m1_next_state;
end
// State transition logic
always @(m1_state
         or ps2_clk
         or debounce_timer_done
         or watchdog_timer_done
         )
begin : m1_state_logic
  // Output signals default to this value, unless changed in a state condition.
  clean_clk <= 0;
  rising_edge <= 0;
  falling_edge <= 0;
```

```
  case (m1_state)
    m1_clk_h :
      begin
        clean_clk <= 1;
        if (~ps2_clk) m1_next_state <= m1_falling_edge;
        else m1_next_state <= m1_clk_h;
      end

    m1_falling_edge :
      begin
        falling_edge <= 1;
        m1_next_state <= m1_falling_wait;
      end

    m1_falling_wait :
      begin
        if (debounce_timer_done) m1_next_state <= m1_clk_l;
        else m1_next_state <= m1_falling_wait;
      end
    m1_clk_l :
      begin
        if (ps2_clk) m1_next_state <= m1_rising_edge;
        else m1_next_state <= m1_clk_l;
      end
    m1_rising_edge :
      begin
        rising_edge <= 1;
        m1_next_state <= m1_rising_wait;
      end
    m1_rising_wait :
      begin
        clean_clk <= 1;
        if (debounce_timer_done) m1_next_state <= m1_clk_h;
        else m1_next_state <= m1_rising_wait;
      end
    default : m1_next_state <= m1_clk_h;
  endcase
end
// State register
always @(posedge clk)
begin : m2_state_register
  if (reset) m2_state <= m2_reset;
  else m2_state <= m2_next_state;
end
// State transition logic
always @(m2_state
```

```
          or q
          or falling_edge
          or rising_edge
          or watchdog_timer_done
          or bit_count
          or packet_good
          or ps2_data
          or clean_clk
          )
begin : m2_state_logic
  // Output signals default to this value, unless changed in a state condition.
  ps2_clk_hi_z <= 1;
  ps2_data_hi_z <= 1;
  error_no_ack <= 0;
  output_strobe <= 0;
  case (m2_state)

    m2_reset :    // After reset, sends command to mouse.
      begin
        m2_next_state <= m2_hold_clk_l;
      end
    m2_wait :
      begin
        if (falling_edge) m2_next_state <= m2_gather;
        else m2_next_state <= m2_wait;
      end
    m2_gather :
      begin
        if (watchdog_timer_done && (bit_count == `TOTAL_BITS))
          m2_next_state <= m2_verify;
        else if (watchdog_timer_done && (bit_count < `TOTAL_BITS))
          m2_next_state <= m2_hold_clk_l;
        else m2_next_state <= m2_gather;
      end
    m2_verify :
      begin
        if (packet_good) m2_next_state <= m2_use;
        else m2_next_state <= m2_wait;
      end
    m2_use :
      begin
        output_strobe <= 1;
        m2_next_state <= m2_wait;
      end
    // The following sequence of 9 states is designed to transmit the
    // "enable streaming mode" command to the mouse, and then await the
```

```
// response from the mouse.  Upon completion of this operation, the
// receive shift register contains 22 bits of data which are "invalid"
// therefore, the m2_verify state will fail to validate the data, and
// control will be passed into the m2_wait state once again (but the
// mouse will then be enabled, and valid data packets will ensue whenever
// there is activity on the mouse.)
m2_hold_clk_l :
  begin
    ps2_clk_hi_z <= 0;   // This starts the watchdog timer!
    if (watchdog_timer_done && ~clean_clk) m2_next_state <= m2_data_low_1;
    else m2_next_state <= m2_hold_clk_l;
  end
m2_data_low_1 :
  begin
    ps2_data_hi_z <= 0;  // Forms start bit, d[0] and d[1]
    if (rising_edge && (bit_count == 3))
      m2_next_state <= m2_data_high_1;
    else m2_next_state <= m2_data_low_1;
  end
m2_data_high_1 :
  begin
    ps2_data_hi_z <= 1;  // Forms d[2]
    if (rising_edge && (bit_count == 4))
      m2_next_state <= m2_data_low_2;
    else m2_next_state <= m2_data_high_1;
  end
m2_data_low_2 :
  begin
    ps2_data_hi_z <= 0;  // Forms d[3]
    if (rising_edge && (bit_count == 5))
      m2_next_state <= m2_data_high_2;
    else m2_next_state <= m2_data_low_2;
  end
m2_data_high_2 :
  begin
    ps2_data_hi_z <= 1;  // Forms d[4],d[5],d[6],d[7]
    if (rising_edge && (bit_count == 9))
      m2_next_state <= m2_data_low_3;
    else m2_next_state <= m2_data_high_2;
  end
m2_data_low_3 :
  begin
    ps2_data_hi_z <= 0;  // Forms parity bit
    if (rising_edge) m2_next_state <= m2_data_high_3;
    else m2_next_state <= m2_data_low_3;
  end
```

```
    m2_data_high_3 :
      begin
        ps2_data_hi_z <= 1;  // Allow mouse to pull low (ack pulse)
        if (falling_edge && ps2_data) m2_next_state <= m2_error_no_ack;
        else if (falling_edge && ~ps2_data)
          m2_next_state <= m2_await_response;
        else m2_next_state <= m2_data_high_3;
      end
    m2_error_no_ack :
      begin
        error_no_ack <= 1;
        m2_next_state <= m2_error_no_ack;
      end
    // In order to "cleanly" exit the setting of the mouse into "streaming"
    // data mode, the state machine should wait for a long enough time to
    // ensure the FA response is done being sent by the mouse.  Unfortunately,
    // this is tough to figure out, since the watchdog timeout might be longer
    // or shorter depending upon the user.  If the watchdog timeout is set to
    // a small enough value (less than about 560 usec?) then the bit_count
    // will get reset to zero by the watchdog before the FA response is
    // received.  In that case, bit_count will be 11.
    // If the bit_count is not reset by the watchdog, then the
    // total bit_count will be 22.
    // In either case, when this state is reached, the watchdog timer is still
    // running and it is best to let it expire before returning to normal
    // operation.  One easy way to do this is to check for the bit_count to
    // reach 22 (which it will always do when receiving a normal packet) and
    // then jump to "verify" which will always fail for that time.
    m2_await_response :
      begin
        if (bit_count == 22) m2_next_state <= m2_verify;
        else m2_next_state <= m2_await_response;
      end
    default : m2_next_state <= m2_wait;
  endcase
end
// State register
always @(posedge clk)
begin : m3_state_register
  if (reset) m3_state <= m3_data_ready_ack;
  else m3_state <= m3_next_state;
end
// State transition logic
always @(m3_state or output_strobe or read)
begin : m3_state_logic
  case (m3_state)
```

```
    m3_data_ready_ack:
          begin
            data_ready <= 1'b0;
            if (output_strobe) m3_next_state <= m3_data_ready;
            else m3_next_state <= m3_data_ready_ack;
          end
    m3_data_ready:
          begin
            data_ready <= 1'b1;
            if (read) m3_next_state <= m3_data_ready_ack;
            else m3_next_state <= m3_data_ready;
          end
    default : m3_next_state <= m3_data_ready_ack;
  endcase
end
// This is the bit counter
always @(posedge clk)
begin
  if (reset) bit_count <= 0;  // normal reset
  else if (falling_edge) bit_count <= bit_count + 1;
  else if (watchdog_timer_done) bit_count <= 0;  // rx watchdog timer reset
end
// This is the shift register
always @(posedge clk)
begin
  if (reset) q <= 0;
  else if (falling_edge) q <= {ps2_data,q[`TOTAL_BITS-1:1]};
end
// This is the watchdog timer counter
// The watchdog timer is always "enabled" to operate.
always @(posedge clk)
begin
  if (reset || rising_edge || falling_edge) watchdog_timer_count <= 0;
  else if (~watchdog_timer_done)
    watchdog_timer_count <= watchdog_timer_count + 1;
end
assign watchdog_timer_done = (watchdog_timer_count==WATCHDOG_TIMER_VALUE_PP-1);
// This is the debounce timer counter
always @(posedge clk)
begin
  if (reset || falling_edge || rising_edge) debounce_timer_count <= 0;
//  else if (~debounce_timer_done)
  else debounce_timer_count <= debounce_timer_count + 1;
end
assign debounce_timer_done = (debounce_timer_count==DEBOUNCE_TIMER_VALUE_PP-1);
// This is the logic to verify that a received data packet is "valid"
```

```
// or good.
assign packet_good = (
                          (q[0]  == 0)
                       && (q[10] == 1)
                       && (q[11] == 0)
                       && (q[21] == 1)
                       && (q[22] == 0)
                       && (q[32] == 1)
                       && (q[9]  == ~^q[8:1])    // odd parity bit
                       && (q[20] == ~^q[19:12])  // odd parity bit
                       && (q[31] == ~^q[30:23])  // odd parity bit
                       );
// Output the special scan code flags, the scan code and the ascii
always @(posedge clk)
begin
  if (reset)
  begin
    left_button <= 0;
    right_button <= 0;
    x_increment <= 0;
    y_increment <= 0;
  end
  else if (output_strobe)
  begin
    left_button <= q[1];
    right_button <= q[2];
    x_increment <= {q[5],q[19:12]};
    y_increment <= {q[6],q[30:23]};
  end
end
endmodule
//`undefine TOTAL_BITS
```

sprites.v

```
//Tyler Hutchison: 6.111
//////////////////////
//button is just like blob, from lab 5
//but toggles when clicked
//////////////////////
module button
   #(parameter WIDTH = 64,      // default width: 64 pixels
               HEIGHT = 32,     // default height: 64 pixels
               COLOR = 3'b111,          // default color: white
                                        COLOR_CLICKED=3'b100, //default activated color : red
                                        LOCATION_X = 11'd0,
```

```
                                        LOCATION_Y = 11'd0) //parameterize location
   (
               input reset,
               input [10:0] hcount,
    input [9:0] vcount,
         input activated,
    output reg [2:0] pixel);

        //hcount runs at 65mhz so its fast!
   always @ (hcount or vcount) begin
               //button color default : white/red
               if ((hcount > LOCATION_X && hcount < (LOCATION_X+WIDTH)) && //if within the bu
         (vcount >= LOCATION_Y && vcount < (LOCATION_Y+HEIGHT)))
                        pixel = activated ? COLOR_CLICKED : COLOR;
      else pixel = 3'b0;
   end

endmodule
module triangle
   #(parameter DIM = 8,        // base/2, base/2=height
               COLOR = 3'b111,  // default color: white
                                        CLICK_COLOR=3'b100) //default color on click (only le
   (input [10:0] hcount,
    input [9:0] vcount,
         input [11:0] x,y,
         input btn_click,
    output reg [2:0] pixel);
   always @ (x or y or hcount or vcount) begin
               //outline, since base = height, slope to top of triangle is 1
               if (((hcount >= x && hcount < (x+DIM)) && //first two lines for \ side of tri
                        (vcount >= (y+hcount-x) && vcount < (y+DIM))) ||
                        ((hcount >=(x-DIM) && hcount < x) && //for / side of the triangle
                        (vcount >=(y-hcount+x) && vcount < (y+DIM))))
                        pixel = btn_click ? CLICK_COLOR : COLOR; //if clicked output the clic
      else pixel = 3'b0;
   end
endmodule
module flipped_triangle
   #(parameter DIM = 8,        // base/2, base/2=height
               COLOR = 3'b111  // default color: white
                                        )
   (input [10:0] hcount,
    input [9:0] vcount,
         input [11:0] x,y,
    output reg [2:0] pixel);
   always @ (x or y or hcount or vcount) begin
```

```
                //outline, since base = height, slope to top of triangle is 1
                if (((hcount >= x && hcount < (x+DIM)) && //first two lines for \ side of tria
                        (vcount >= y && vcount < (y+DIM-hcount+x))) ||
                        ((hcount >=(x-DIM) && hcount < x) && //for / side of the triangle
                        (vcount >=y && vcount < (y+DIM+hcount-x))))
                        pixel = COLOR; //if clicked output the click color
      else pixel = 3'b0;
   end
endmodule
module scan_bar //very similar to button, but can move
   #(parameter WIDTH = 32,      // default width: 32 pixels for grab blob
               HEIGHT = 32,     // default height: 32 pixels for grab blob
                                        LINE_LOC=0, //offset from top of the grab blob
               COLOR = 3'b100)          // default color: red
   (
                input reset,
                input [10:0] x,hcount,
    input [9:0] y,vcount,
    output reg [2:0] pixel);
        always @ (x or y or hcount or vcount) begin
                if ((hcount > x && hcount < (x+WIDTH)) && //if within the button make it a col
                        (vcount >= y && vcount < (y+HEIGHT)))
                        pixel =COLOR;
                else if(vcount==(y+LINE_LOC) && hcount>=x && hcount < (x+WIDTH+10'd720)) //dra
                        pixel =COLOR;
      else pixel = 3'b0;
        end
endmodule
```

zbt_6111.v

```
//*****TODO:
//    * test BWE (need to invert?)
//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Simple ZBT driver for the MIT 6.111 labkit, which does not hide the
// pipeline delays of the ZBT from the user.  The ZBT memories have
// two cycle latencies on read and write, and also need extra-long data hold
// times around the clock positive edge to work reliably.
//
////////////////////////////////////////////////////////////////////////////////
// Ike's simple ZBT RAM driver for the MIT 6.111 labkit
//
```

```
// Data for writes can be presented and clocked in immediately; the actual
// writing to RAM will happen two cycles later.
//
// Read requests are processed immediately, but the read data is not available
// until two cycles after the intial request.
//
// A clock enable signal is provided; it enables the RAM clock when high.
module zbt_6111(clk, cen, we, bwe, addr, write_data, read_data,
                  ram_clk, ram_we_b, ram_bwe_b, ram_address, ram_data, ram_cen_b);
   input clk;                   // system clock
   input cen;                   // clock enable for gating ZBT cycles
   input we;                    // write enable (active HIGH)
   input [3:0] bwe;                     // byte write enable select
   input [18:0] addr;           // memory address
   input [35:0] write_data;     // data to write
   output [35:0] read_data;     // data read from memory
   output        ram_clk;       // physical line to ram clock
   output        ram_we_b;      // physical line to ram we_b
   output [3:0]  ram_bwe_b;     // physical line to ram bwe_b
   output [18:0] ram_address;   // physical line to ram address
   inout [35:0]  ram_data;      // physical line to ram data
   output        ram_cen_b;     // physical line to ram clock enable

   // clock enable (should be synchronous and one cycle high at a time)
   wire          ram_cen_b = ~cen;

   // create delayed ram_we signal: note the delay is by two cycles!
   // ie we present the data to be written two cycles after we is raised
   // this means the bus is tri-stated two cycles after we is raised.
   reg [1:0]   we_delay;
   always @(posedge clk)
     we_delay <= cen ? {we_delay[0],we} : we_delay;

   // create two-stage pipeline for write data
   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk)
     if (cen)
       {write_data_old2, write_data_old1} <= {write_data_old1, write_data};
   // wire to ZBT RAM signals
   assign      ram_we_b = ~we;
   assign      ram_clk = ~clk;     // RAM is not happy with our data hold
                                   // times if its clk edges equal FPGA's
                                   // so we clock it on the falling edges
                                   // and thus let data stabilize longer
   assign      ram_address = addr;
```

```
   assign      ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};
   assign      read_data = ram_data;


   assign ram_bwe_b = bwe;


endmodule // zbt_6111
```

zbt_to_display.v

```
//Dima Turbiner
module zbt_to_display(input clk, reset,
                      input [9:0]hcount, input [8:0]vcount,
                      input [35:0]data_main, input [35:0]data_overlay,
                                output reg [7:0]main_pixel, output reg [7:0]overlay_pixel,
                      output [18:0]addr);
   //**** CHECK THIS ADDRESSING
   assign addr = {2'b0, vcount, hcount[9:2]};
   wire [1:0] hc4 = hcount[1:0];

   reg [35:0]    main_data_latched;
   reg [35:0]    last_main_data;

   reg [35:0]    overlay_data_latched;
   reg [35:0]    last_overlay_data;
   always @(posedge clk) begin
      last_main_data <= (hc4==2'd3) ? main_data_latched : last_main_data;
      last_overlay_data <= (hc4==2'd3) ? overlay_data_latched : last_overlay_data;
        end


   always @(posedge clk) begin
        main_data_latched <= (hc4==2'd1) ? data_main : main_data_latched;
      overlay_data_latched <= (hc4==2'd1) ? data_overlay : overlay_data_latched;
        end
   always @*            // each 36-bit word from RAM is decoded to 4 bytes
     case (hc4)
       2'd3:
         begin
            main_pixel = last_main_data[7:0];
            overlay_pixel = last_overlay_data[7:0];
         end
       2'd2:
         begin
```

```
            main_pixel = last_main_data[7+8:0+8];
            overlay_pixel = last_overlay_data[7+8:0+8];
          end
        2'd1:
          begin
            main_pixel = last_main_data[7+16:0+16];
            overlay_pixel = last_overlay_data[7+16:0+16];
          end
        2'd0:
          begin
            main_pixel = last_main_data[7+24:0+24];
            overlay_pixel = last_overlay_data[7+24:0+24];
          end
      endcase
endmodule
```

zbt_to_ifft.v

```
//Dima Turbiner
module zbt_to_ifft(input clk, reset, input [9:0]hcount, input [8:0]vcount,
                                                  input [35:0]data_main, input [35:0]data_overla
                                                  output reg [7:0]main_pixel, output reg [7:0]o
   //**** CHECK THIS ADDRESSING
   assign        addr = {2'b0, vcount, hcount[9:2]};

   wire [1:0]    hc4 = hcount[1:0];

   reg [35:0]    main_data_latched;
   reg [35:0]    last_main_data;

   reg [35:0]    overlay_data_latched;
   reg [35:0]    last_overlay_data;


   always @(posedge clk) begin
      last_main_data <= (hc4==2'd3) ? main_data_latched : last_main_data;
      last_overlay_data <= (hc4==2'd3) ? overlay_data_latched : last_overlay_data;
        end


   always @(posedge clk) begin
       main_data_latched <= (hc4==2'd1) ? data_main : main_data_latched;
     overlay_data_latched <= (hc4==2'd1) ? data_overlay : overlay_data_latched;
        end

   always @*            // each 36-bit word from RAM is decoded to 4 bytes
```

```
      case (hc4)
        2'd3:
          begin
             main_pixel = last_main_data[7:0];
             overlay_pixel = last_overlay_data[7:0];
          end
        2'd2:
          begin
             main_pixel = last_main_data[7+8:0+8];
             overlay_pixel = last_overlay_data[7+8:0+8];
          end
        2'd1:
          begin
             main_pixel = last_main_data[7+16:0+16];
             overlay_pixel = last_overlay_data[7+16:0+16];
          end
        2'd0:
          begin
             main_pixel = last_main_data[7+24:0+24];
             overlay_pixel = last_overlay_data[7+24:0+24];
          end
      endcase
endmodule
```