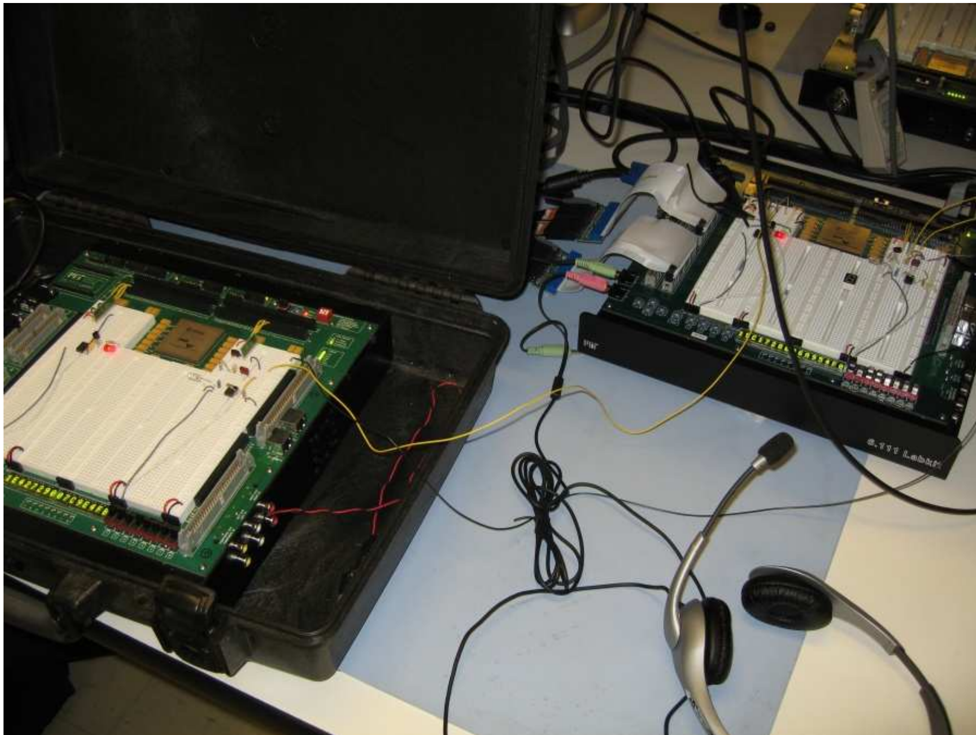
# Point-to-point Data Link, Collaborative Whiteboard, and Voice Conference

## MIT 6.111 Project Final Report


Michael F. Robbins — *Email:* mrobbins@mit.edu

Scott B. Ostler — *Email:* sostler@mit.edu


*December 13, 2006*




**Abstract**

A two-node communications system capable of transmitting arbitrary digital data over a noisy analog channel was implemented in the Verilog HDL and demonstrated on the Xilinx Virtex 2 series FPGA, and a full-duplex voice conference and shared VGA whiteboard were demonstrated over this channel. The link encoding was based on Orthogonal Frequency Division Multiplexing (OFDM), and bitwise redundancy effectively became redundancy in the frequency domain. The applications included the need for a high-speed off-chip framebuffer, as well as a packet engine that could tolerate the vastly different priority and quality requirements of both voice and drawing.

# Table of contents

**Table of contents** . . . . . 2

**List of figures** . . . . . 3

**1 Overview** . . . . . 4

**2 Modes of Operation** . . . . . 5

2.1 Local Mode . . . . . 5
2.2 Loopback Mode . . . . . 5
2.3 Link Mode . . . . . 5

**3 Module-Level Design and Implementation** . . . . . 6

3.1 Sine and Cosine Lookup Tables (Mike) . . . . . 6
3.2 Sinesample Module (Mike) . . . . . 7
3.3 IDFT Shifter Module (Mike) . . . . . 7
3.4 Discrete-Time Fourier Transform Module (Mike) . . . . . 7
3.5 Encoder and Decoder (Mike) . . . . . 9
3.6 Guard Detector, Threshold, and Synchronizer (Mike) . . . . . 9
3.7 Optical Components (Mike) . . . . . 10
3.8 Analog Front-End (Mike) . . . . . 11
3.9 Packet Engine, Transmit Side (Scott) . . . . . 14
3.10 Packet Engine, Recieve Side (Scott) . . . . . 14
3.11 Whiteboard (Scott) . . . . . 14
3.12 Pointer (Scott) . . . . . 16
3.13 SRAM Memory Interface and Framebuffer (Scott) . . . . . 16
3.14 Memory Router (Scott) . . . . . 17
3.15 Voice (Scott) . . . . . 17

**4 Testing and Results** . . . . . 18

**5 Discussion and Conclusions** . . . . . 20

**6 Appendix: Verilog Source Listing** . . . . . 21

6.1 adc_ad7476.v . . . . . 21
6.2 audio_ac97.v . . . . . 22
6.3 bin_threshold.v . . . . . 27
6.4 cos64.v . . . . . 27
6.5 dac_ad5399.v . . . . . 27
6.6 debounce.v . . . . . 28
6.7 delayN.v . . . . . 29

6.8 dft64.v . . . 29
6.9 display_16hex.v . . . 33
6.10 encoder_2of3.v . . . 36
6.11 finalproject.v . . . 37
6.12 frame_sync.v . . . 47
6.13 guard_detector.v . . . 48
6.14 memory_router.v . . . 49
6.15 omemhz_divider.v . . . 49
6.16 packet_engine.v . . . 50
6.17 pointer.v . . . 51
6.18 shifter.v . . . 53
6.19 sin64.v . . . 53
6.20 sinesample.v . . . 54
6.21 synchronize.v . . . 56
6.22 vram_display.v . . . 56
6.23 whiteboard.v . . . 57
6.24 xvga.v . . . 59
6.25 zbt_6111.v . . . 60

# List of figures

Block diagram and rough picture of information flow. This highlights the "LED-link" mode. . . . 6
Logic analyzer shows the output of the `dft64_magsq` module. Bins 0 (DC offset) and 2 are clearly on in this example. . . . 8
Photo of the LED-link receiver-side components, including the photodiode, dual op-amp, filters, and ADC. . . . 11
Schematic of the LED-link analog circuitry . . . 12
Photos of the hand-soldered DAC and ADC chips, with pin pitches of 0.50mm and 0.65mm respectively. . . . 13
Whiteboard in action, with visible pointer overlay. . . . 16
Oscilloscope of LED-link, showing transmitted LED current on top (yellow) and received photodiode light intensity on bottom (blue). High noise is clearly visible. . . . 19

# 1 Overview

We set out to construct a digital data link using LEDs and photodiodes to implement a communications channel in free space. A packet engine interfaces between the stream of data coming in over the channel to packets that are useful to specific applications. We constructed two simple applications to run over this protocol, namely a collaborative whiteboard with shared drawing space betwen the two stations, and a simultaneous voice conference.

A major part of this project was aimed at making a reliable digital datalink even in the face of contamination from many noise sources, such as ambient sunlight and indoor lighting. However, most noise sources are narrowband in nature, for example from line level lighting at 120Hz, or some fluorescent ballasts near 26kHz. For this reason, we chose to use Orthogonal Frequency Division Multiplexing (OFDM) as our digital encoding scheme. While this may sound complicated, it is in reality just using many on-off modulated carriers at once. A bit being sent over the chanenl may correspond to a bin in the frequency domain, and using the inverse FFT, we are able to construct a time domain signal to send. The narrowband noise will contaminate some of these bits, but using simple forward error correction techniques, we can tolerate a few bands of interference. The receiving end will take the FFT of the time-domain signal, and will be able to extract the various bits that were sent. When compared to simply sending the raw bitstream, this technique in theory provides resistance to narrowband noise and also to impulse ("popcorn") noise. This is a good use of the muscle of the FPGA, and demonstrates a task that a general purpose processor might not be able to handle in real time.

This kind of frequency-domain encoding requires special timing synchronization because of the random phase offset between stations. We devised a simple strategy with a frequency-domain "guard sequence" which will alert the receiver to the correct synchronization phase, and by transmitting each FFT length twice, we are guaranteed to have a signal that is properly phased on alternating groups of samples. (A simliar technique is done in the time domain over any asynchronous serial link, where the signal is edge triggered or oversampled to wait for a start bit.) While this technique cuts the channel capacity in half, it allows us to make a fairly simple way of getting synchronicity. In practice, this "guard sequence" phasing didn't always work in practice due to noise, and this is likely a good area for future improvement.

The whiteboard application required interfacing with framebuffers, overlays, and appropriate read and write timing to deal both with local draw request and remote ones arriving at any time over the data link. Ultimately, it demonstrated the kinds of bit error rates that might be present on the channel.

# 2 Modes of Operation

Although our goal is for two independent stations to interact over our communications channel, we had to do the majority of development using only one labkit. Therefore, to facilitate testing of our module's functionality during development, we specified three modes of operation, with only one mode requiring both labkits.

The three modes are as follows:

## 2.1 Local Mode

In local mode, there is only one labkit, and the application modules do not interact with the communications channel. Data words are still converted to the frequency domain and back, but through internal wires on the labkit. This "internal-link" mode was free of noise and thus was usually a good sanity test of the digital system.

## 2.2 Loopback Mode

In loopback mode, there is only one labkit, and the application modules transmit their data over the communications channel, back to themselves. The only difference between loopback mode and local mode is that loopback goes over a physical wire or optical path. This was at first realized with wires, but later with the LED and photodiode on the same labkit. This "LED-link" mode allowed tests of the robustness of the system. In general, this injected lots of noise into the system (as described later).

## 2.3 Link Mode

In link mode, there are two labkits, and the application modules in each communicate as peers. Because of signal strength and time constraints, this was only done with wires as the physical medium (hereafter designated "wire-link"). Three wires are reqired: ground, signal from kit A->B, and signal from kit B->A. No attempt was made to try an optical link between two labkits, but given sufficient extra time, such a test would not be difficult to conduct..

# 3 Module-Level Design and Implementation

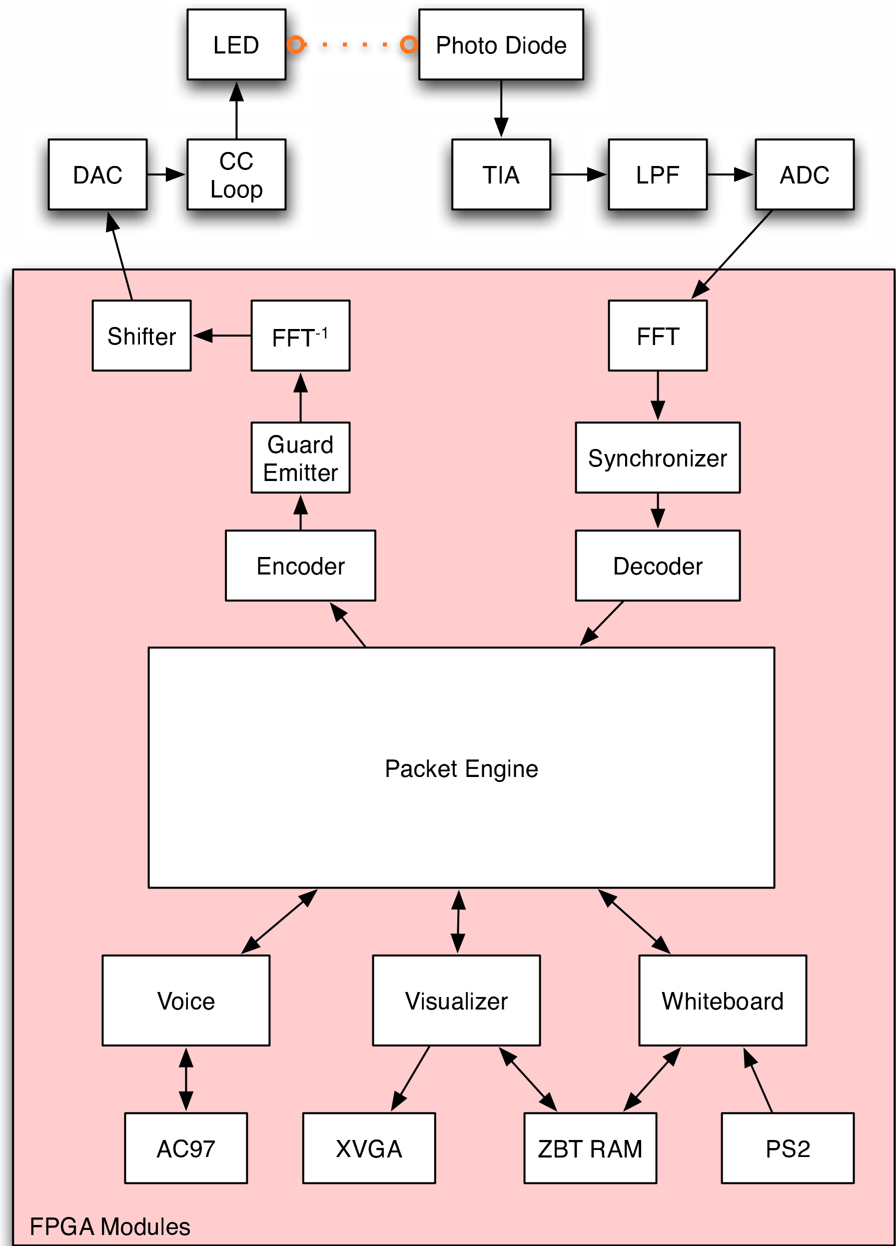



**Figure 1.** Block diagram and rough picture of information flow. This highlights the "LED-link" mode.

## 3.1 Sine and Cosine Lookup Tables (Mike)

A combinational 12-bit-wide lookup table for sine and cosine values was constructed using an external script. The bit width was chosen to match that of the DAC and ADC. For this 64-sample period wave, the `sin64` module only stores 32x12 bits by recognizing that $\sin(x) = -\sin(-x)$. A further factor-of-two size reduction could be realized by implementing the equation $\sin(x) = \sin(\pi - x)$.

### 3.2 Sinesample Module (Mike)

The `sinesample` module takes in a set of 32 bins (on/off) and calculates the IFFT value at a given index. It does this in 16 cycles by stepping over two bins at once.

One may note that for an N-point IDFT, $N^2$ multiplications are required, $N$ for each of $N$ samples. However, because we are running from the 27MHz clock and must output a new sample every $1\mu s$, we must do all of these in less than 27 cycles. Processing two bins at once brings us under this tining restriction.

### 3.3 IDFT Shifter Module (Mike)

Initial attempts were made at using the Xilinx IP Core FFT and IFFT modules, however there was significant frustration in making them perform as advertised. In response, the DTFT and IDFT were implemented manually. They are by no means "fast" fourier transforms, and are essentially just naive implementations of the appropriate Fourier series equations.

The `shifter` module steps through the 64 points of a frame, and runs the `sinesample` module for each one. It is essentially acting in a major/minor FSM role.

### 3.4 Discrete-Time Fourier Transform Module (Mike)

On the receiving side, the implementation of the DTFT is not nearly as straightforward as the inverse case. This is because the amount of data that must be processed for each bin value has increased from 32 1-bit values to 64 12-bit values. To handle this data, the module is constructed with two 64x12 Block RAMs (BRAMs), which take on alternating roles as time progresses. At any given moment in time, new samples wil be writing into one of the BRAMs, while they are (independently) read from the other BRAM and

multiplied by the appropriate phase factors to accumulate the real and imaginary parts of that bin's value.

The initiation of DFT processing comes whenever 64 samples have been read. However, once initiated, the processing (which must do 128 multiplies and accumulates for each of 32 bins) runs independently of the sample input (which is happening at 1MHz). Therefore, after this point, the data is no longer synchronous with he `onemhz_enable` signal. Instead, the `newbin` output tells downstream modules that new frequency content data is available, specified by the `xk_bin` number (0 to 31) and by the real and imaginary parts.

Since our communications scheme does not rely on signal phase or a whole constellation of frequency domain symbols, only the magnitude is important. And since we will simply be making threshold (on/off) decisions with the magnitude, we can equivalently do the same process with the magnitude squared. This way, we don't have to waste any time taking a square root. The computation of the magnitude squared occurs in the `dft64_magsq` helper module, which just adds a two cycle delay in doing its computation.

The image below shows a signal with a single frequency bin on. After going through the DAC, wire, and ADC, the signal has acquired a DC component, represented in bin 0, but otherwise, only the original frequency component (bin 2 in this case) is shown as having significant magnitude.

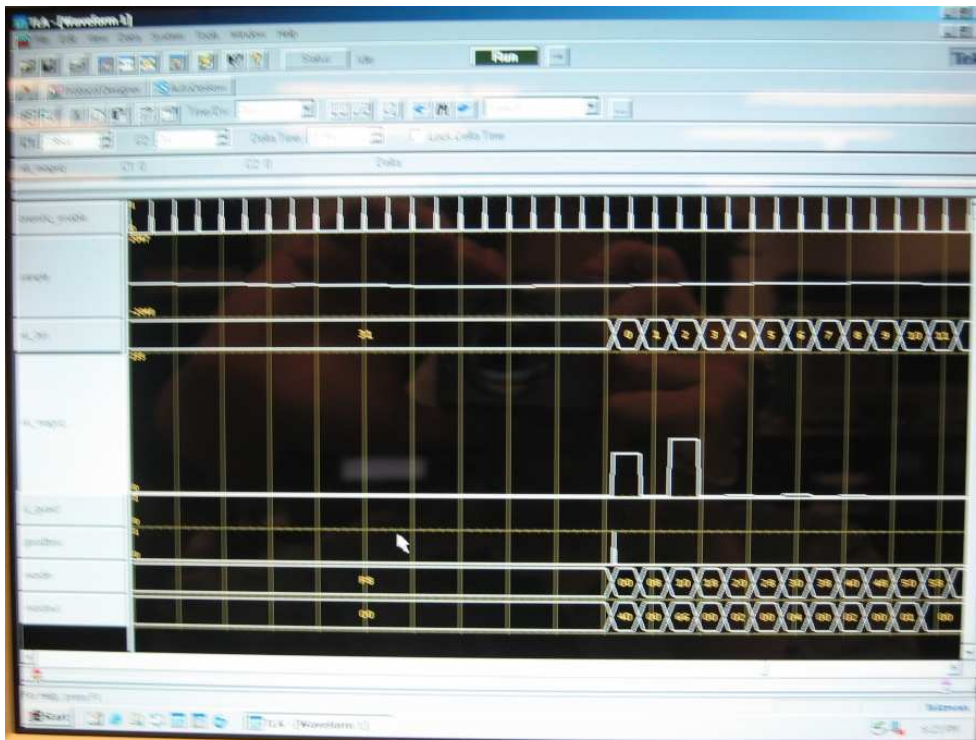



**Figure 2.** Logic analyzer shows the output of the `dft64_magsq` module. Bins 0 (DC offset) and 2 are clearly on in this example.

The completion of the Shifter (Inverse DFT) and forward DFT modules allows bits to be transmitted as frequency components.

### 3.5 Encoder and Decoder (Mike)

The fundamental frame of our system is a 9-bit word, which will be processed by the packet engine and routed appropriately. The signal redundancy is layered on top, and the encoder module is only aware that it needs to protect these 9 bits.

As we are currently considering using FFTs with a length of 64 time samples, we have 32 independent frequencies we can turn on or off. (This reduction from 64 to 32 has to do with the fact that we are limiting ourselves to real-valued functions and to magnitude only.) Thus, we can add considerable redundancy with the (32-9)=23 additional frequency bins. (Not all of these are available; the low frequencies were to be avoided due to 1/f noise, and some frequencies must be reserved for the guard signal.) Our simple coding was to place each of the 9 bits into 3 frequency bins. Thus, bit 0 was placed in bins 1, 10, and 19, and bit 1 was placed in bins 2, 11, and 20, and so on. (In about half of the cases, the sense is inverted, where the NOT of the bit was placed in that frequency bin. This was done to keep the total amplitude of the signal roughly constant.) Bins 28-30 are used for the guard signal, and bin 31 is always on, providing a reference level.

The decoder is based on a 2-of-3 majority decision box. As three bins are transmitted per bit, those three bins are examined post-threshold, and their majority vote decides whether that bin was on or off.

Thus, in absolute terms, we can only tolerate one "bin error" – a frequency bin that is mislabeled as on or off – because two which happen to affect the same bit would cause a wrongly corrected error (minimum Hamming distance is 2). However, if we are lucky and the multiple noise sources do not target those bins, we can tolerate a few more "bin errors."

### 3.6 Guard Detector, Threshold, and Synchronizer (Mike)

As described in the overview, we must avoid phase synchronization issues by some-

times inserting a special guard frame, which will be recognized by the guard detector and used by the frame synchronizer. The system allows two modes of operation, one where guard frame are manually sent and recieved to set the threshold, and another where they are automatically sent and recieved every few milliseconds. It was observed that with the LED-link, the automatic thresholding strategy was not successful. This requires furthher investigation.

The `guard_detector` module uses a sort of majority strategy similar to the decoder, in that it looks at the median strength of the guard values and compares it to the always-on frequency bin strength.

The `frame_sync` module keeps track of when these guards happen, and then appropriately discards alternating DFT frames. It also updates the `last_guard_strength` threshold when a guard frame is received.

The `bin_threshold` module has a relatively simple task, comparing the magnitude squared of a given bin to the threshold. It takes the stream of bins and magnitudes coming from the `dft64_magsq` module and outputs a 32-bit bin status after seeing a full DFT sequence.

### 3.7 Optical Components (Mike)

Our communications scheme is generally applicable to transmitting digital data over any noisy analog channel. While most communications systems today lie in the radio and microwave regions of the RF spectrum, we have chosen to use visible light as our medium of choice. This provides several advantages for this project, including easy development of analog components, relatively easy debugging, and easy ways to "distort" the channel and introduce noise.

For the transmitting end, we are using a simple red Light Emitting Diode (LED), a device which has a remarkably linear relationship between operating current and light intensity. It is also possible to modulate an LED quickly. A current control loop will be constructed to take the voltage output of the DAC (described later) and adjust the current going through the LED. A generic red LED (All Electronics Corp LED-1), like those easily obtained in the 6.002 lab or elsewhere, has been the used.

On the receiving end, a photodiode and transimpedance amplifier will measure the light received. A photodiode, when properly biased, will allow a reverse current to flow proportional to the intensity of light striking its sensor area. The transimpedance amplifier will convert this current into a voltage. After going through a simple RC highpass filter to attenuate ambient light (DC) and line voltage lighting (120Hz), this voltage will proceed into the analog front end. Several options are currently being considered for the photodiode and transimpedance amplifier. The Everlight EL-PD638C photodiode costs

about $0.30 and has a wide wavelength response (which would normally be bad for a communications system, but is useful to demonstrate noise). The transimpedance amplifier is a fairly simple op-amp circuit and is shown on the schematic later.

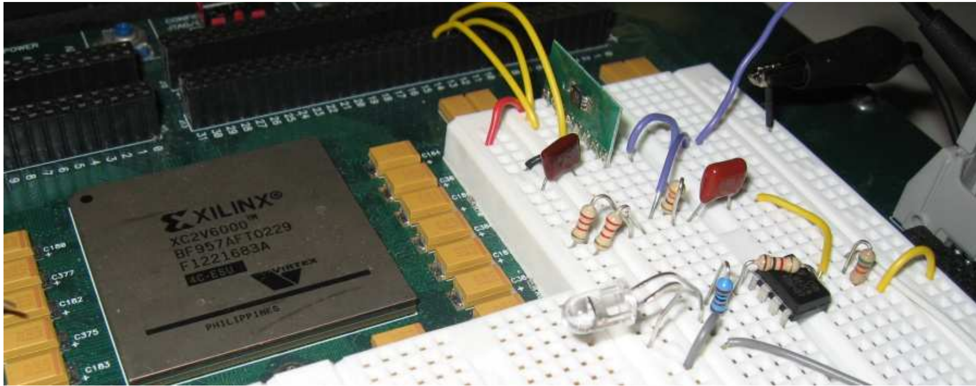



**Figure 3.** Photo of the LED-link receiver-side components, including the photodiode, dual op-amp, filters, and ADC.

Because of both time and signal strength constraints, the LED-link was used only onboard a single labkit, not between two stations. In this role, however, it was quite successful. The OFDM scheme was able to pick out the bits even with what the oscilloscope depicted as serious noise. The use of laser pointers to transmit the data further was investigated, but their non-linear transfer characteristics added an extra challenge that we chose not to pursue.

## 3.8 Analog Front-End (Mike)

As almost all of our signal processing will be done digitally, we needed a way to convert the analog waveforms to and from digital representations. We used a 1MS/s rate for both ADC and DAC functions, with 12 bits of resolution on each.

For the DAC, we used the Analog Devices AD5399. It was very easy to interface to over its serial bus and presented no difficulty.

For the ADC, we used the Analog Devices AD7476A. It had a maximum serial clock rate of 20MHz. This required some effort with Digital Clock Managers and signal synchronization on both ends to make everything work, but this was done and the module works reliably.

We considered the use of the second output of the AD5399 to implement an auto gain control, such that as the transmitter and receiver move apart, there are still enough bits of dynamic range to provide useful data. However, we did not in the end because we chose not to prioritize the physical channel. We planned to use the AD835 multiplier to accomplish gain control.

It should also be noted that the analog/digital bridge components are independent of the actual physical channel, and they were used both for the LED-link and the wire-link.

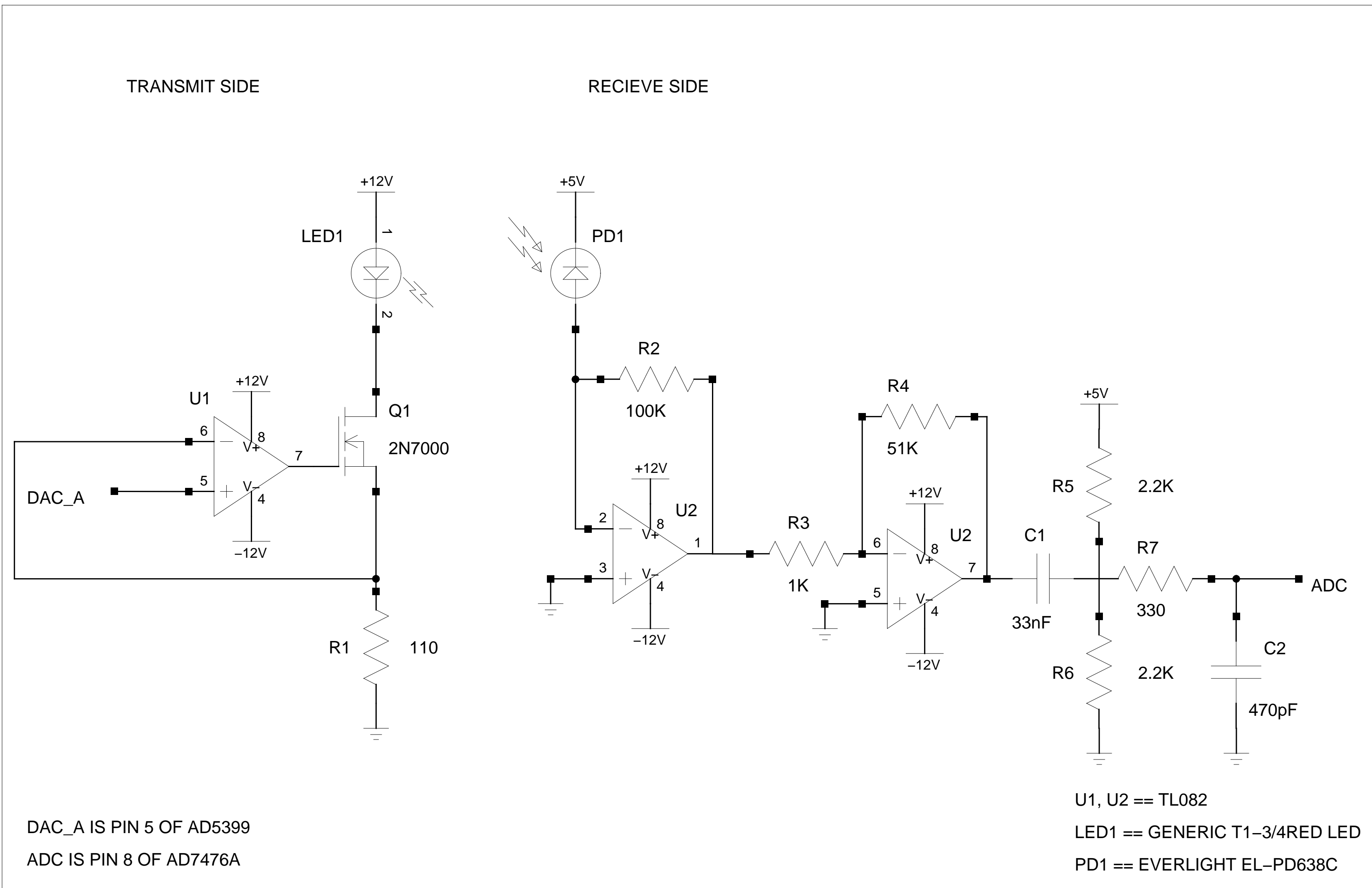



**Figure 4.** Schematic of the LED-link analog circuitry

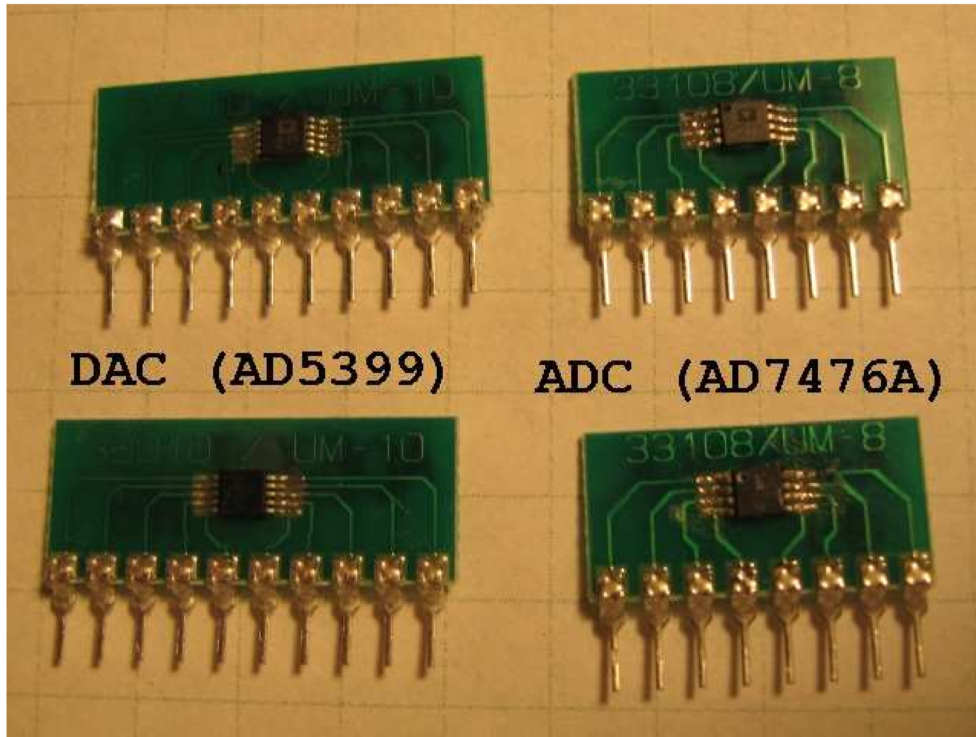



**Figure 5.** Photos of the hand-soldered DAC and ADC chips, with pin pitches of 0.50mm and 0.65mm respectively.

The schematic above shows the various components to the analog side. On the transmitting side, a current control loop is needed to control the current through the LED, because LEDs have a very linear relationship between current and light intensity. The op-amp can not source enough current by itself, so a n-FET is included in the feedback loop.

On the receiving side, one needs to have huge gain to turn nanoamps of photocurrent into volts, highpass to remove the DC level, and lowpass to avoid aliasing. All of these steps are accomplished by the circuit as shown.

Physically, the ADC and DAC chips were a hard to attach to their holder boards, and because solder mount components are increasingly dominating the market, it looks like there will be little alternative on the future. For the AD5399 DAC, adapter board

Digikey P/N 33010CA-ND ($3.22) was used for mounting, and for the AD7476A ADC, Digikey P/N 33108CA-ND ($2.80) was used. This was difficult and took several attempts (and burnt-out or over-soldered components).

### 3.9 Packet Engine, Transmit Side (Scott)

The Packet Engine transmits packetized units of information (called packets) from the application modules to the communications channel.

Application modules give packets to the Packet Module by giving single units of information (audio packets are 8 bit voice samples, drawing instruction packets are 26 bit structures containing pixel coordinates and a checksum). A control bit is appended to each packet, marking whether it is an audio or video packet, and then the packet is broken into 9 bit segments, each sent to the encoder in turn after an appropriate delay. Audio packets are sent immediately, because they fit into the 9 bits allowed for them. However, video packets must be broken in three, with each component being sent seperately.

Packets are prioritized by the following simple scheme: because audio packets are being sent constantly, they alone saturate the channel. Video packets are substantially more infrequent, but individually much more important; when both a video and audio packet are ready to be sent, the audio packet is dropped. Due to the low rate of video packets, this is largely inaudible.

### 3.10 Packet Engine, Recieve Side (Scott)

The Packet Engine receives frames from the Decoder module, and assembles those frames into packets, which are then dispatched to the appropriate Application module based on the high order bit of the assembled packet. Only video packets must be resassembled, as audio packets are contained within a single frame. We use a checksum to ensure the integrity of video packets, but that is encoded into the video packet itself, and checked later by the whiteboard module.

### 3.11 Whiteboard (Scott)

The Whiteboard module conducts an interactive drawing session between two stations. A user on one station can draw lines and shapes on their own station's display, by moving around a pointer with their labkit's directional buttons. When they wish to draw on the pixel that their mouse is under, they press their labkit's enter button. Their drawings are transmitted across the communications link and replicated on the other station's display. When desired, either use can press a clear-screen button, which erases both screens' content. In this way, two users can collaboratively draw a single picture.

The Whiteboard module maintains the position of a pointer (drawn by the Pointer module), and lets the user move that pointer around. When the enter button is held down, the area under the pointer is considered to have been drawn upon. At that point, an instruction corresponding to that action is formulated, and sent to one or both of the local station and the remote station. When a Whiteboard module receives a drawing instruction, it writes to the appropriate memory location of the backing ZBT framebuffer. Then, the next time the screen is redrawn, that pixel will show up as having been drawn upon. The clear screen functionality is handled by storing clear screen requests, and consumating them at the end of a screenredraw. When the screen is being cleared, the Whiteboard module outputs a high control signal, which is used by other modules to both clear the contents of memory and draw a blank screen.

The WHiteboard module also includes a checksum in the packets it sends, to help mitigate transmission errors. A simple 7 bit checksum is used, made by summing the x and y coordinates and taking the low 7 bits of the result. This could be much better, of course. Before drawing a remote packet, the included checksum is compared to the computed checksum of the packet's coordinate information; if they don't match, the packet is ignored.

How the module interets a given drawing instruction depends on the station's operating mode:

- In Local Mode, the Whiteboard module directs all drawing instructions into its drawing path. This means simply painting the palette with the user's mousestrokes, for the purpose of testing the drawing mechanism.

- In Loopback Mode, the Whiteboard module directs all drawing instructions o the Packet Engine. The Packet Engine then transmits the instruction over the communications channel. When the Packet Engine receives the drawing instruction as input, it dispatches that instruction to the Whiteboard module's drawing path.

- In Link Mode, the Whiteboard module combines the functionality of the other two operating modes; that is, it internally draws its own drawing instructions, while also sending them through the Packet Engine.

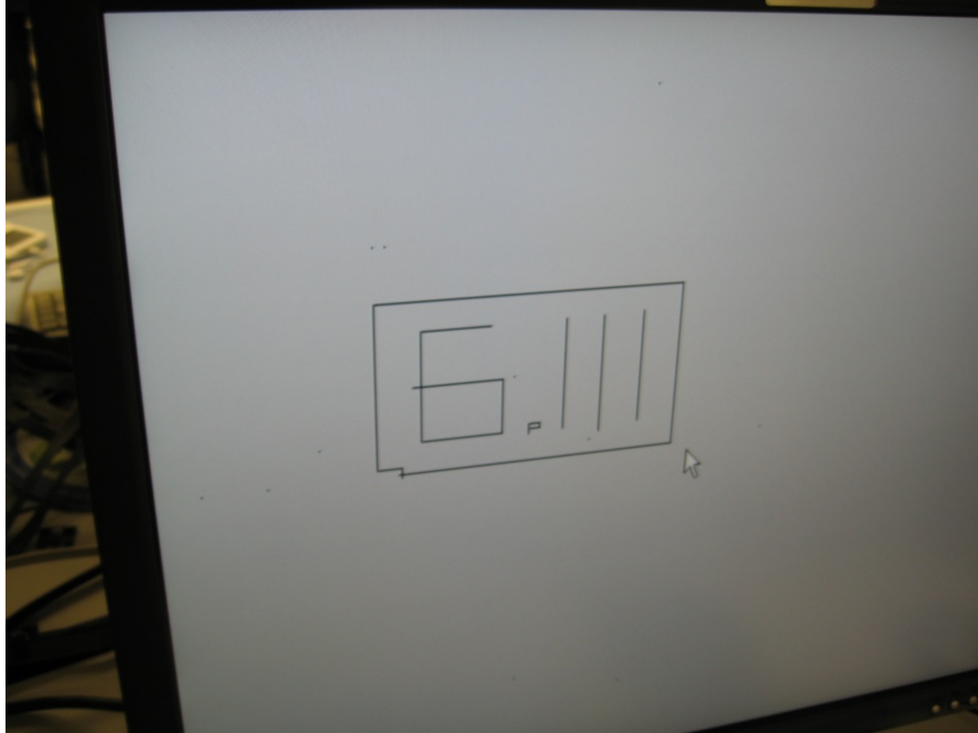


**Figure 6.** Whiteboard in action, with visible pointer overlay.

### 3.12 Pointer (Scott)

The Pointer module is responsible for drawing a black and white Pointer on the screen. It was implemented similarily to the Blob from Lab 5, except that its boundries wareere determined by referencing a lookup table, instead of mathematically computing the boundaries of a simple gemoetric shape. That is, for a given x-y coordinate within the pointer's bounds, we lookup whether that coordinate should be drawn, and if so, whether that coordinate is black or whitte. In this way, we can overlay a multicolor, non-square pointer over the framebuffer contents.

### 3.13 SRAM Memory Interface and Framebuffer (Scott)

The SRAM Memory Interface is responsible for reading pixel values from memory, and outputting them to the screen. It does this primarily using the staff-provided VRAM and XVGA modules, although modifications were made to both. When the screen is being drawn (i.e. blank is not high), hcount and ycount are used to compute a memory and byte address, using a simple simple scheme. (We store four pixels in a single 36bit memory word, so we use the low two bits of the coordinate's x-location to index into the memory word.) We delay all other video control signals, including the pointer output, to accomodate for the memory's read delay.

The framebuffer handles memory writes by making use of the staff-provided zbt_6.111 module, which simplifies writing values to memory. On top of that module, a pixel-addressing layer has been added, which translates mouse coordinates into memory addresses an byte indices. During a clear screen instruction, the frame buffer is only written to; the addresses are incremeted just as they would be during the normal frame buffer reading, but the write enable bit is turned on, and the RAM's data input is set to the screen's background color.

## 3.14 Memory Router (Scott)

Memory Router is a small module that multiplexes the ZBT RAM's writing wires between the screen clearing functionality, the local pixel drawing functionality, and the remote pixel drawing functionality. The module takes each of those signals as input, as well as a few control signals, and outputs one set of address, data byte select, and write enable wires to the ZBT RAM.

## 3.15 Voice (Scott)

The Voice module enables an audio link between two stations. A user on one station can speak into their microphone, and a digitally sampled recording of their voice will be transmitted across the communication link, and played back on the other station's headphones. In this way, two users can talk together.

The Voice module records and plays back digitized audio samples. Similar to Lab 3, the Voice module periodically samples the AC97's microphone output to record audio, and on playback outputs those recorded samples to the AC97's input. The module uses a near-ideal low-pass filter to remove harmonics from the output. The Voice Module connects to the Packet Engine, so that it can both send and receive audio packets over the station's link layer. It outputs an eight-bit sample to the Packet Engine, which appends a one-bit control signal marking the packet as an audio packet, then sends it over the channel. When it is recieved by the other Voice module, it is simply put out to the ASC97's output.

The path of a given audio sample depends on the station's operating mode:

- In Local Mode, the Voice module directs all recorded samples into its playback path. This means simply replaying the recorded samples, for the purpose of testing the recording and playback mechanisms.
- In both Loopback Mode and Link Mode, the Voice module directs all recorded samples through the Packet Engine. The Packet Engine then transmits the audio sample over the communication channel. When the Packet Engine receives an audio sample as input, the Engine gives it to the Voice module as a sample to be played. The Voice module then directs that sample to the playback path. Note that the only difference between these two modes is whether the Packet Engine gives the Voice module the samples that it recorded, or the samples that another Voice module recorded; this is not an important distinction to the Voice module.

# 4 Testing and Results

The clear lines of seperation in our project made it easy for each of us to build up slowly and use the real labkit as the primary test platform. With the exception of a small bit of the IDFT shifter code, only in-FPGA testing was performed.

There were initially lots of difficulties on both sides. For Mike, it was impossible to get a sensible answer out of Xilinx's FFT modules (soon replaced by his own implementations). For Scott, the PS/2 mouse was inconsistent (soon replaced by directional buttons), ande clock skew issues frequently plagued his video output (solved by substantially lowering the screen resolution). A substantial amount of time was lost to fighting the provided tools and resources (manually routing blocks, trying to hack around problems with others' modules). Looking back, we are glad that we made the decision to move away from unreliable components, and wished we had done so much sooner.

Because there were really so few points of intersection between our work, integration went very smoothly. We do know of a few problems that need resolution:

- Attempting to use automatic guard frame detection on the LED-link results in garbage. It appears that random noise is triggering the guard frame detector. It may be the case that the algorithm itsel fneeds fixing.

- When drawing over the channel, only horizontal lines are received and drawn. There are few asymmetries between drawn lines, but we were unable to find this bug before our presentation.

- Periodic noise over the wire-link (two labkits). Might be caused by clock skew. Since we only had the second labkit for two days, it was hard to address this.

While we never measured a raw bit error rate, as the oscilliscope below shows, the LED-link resulted in serious noise injection, yet audio was still fine and draw commands were even being transmitted. However, the problem of automatic guard frame detection / threshold setting over the LED-link was not solved.

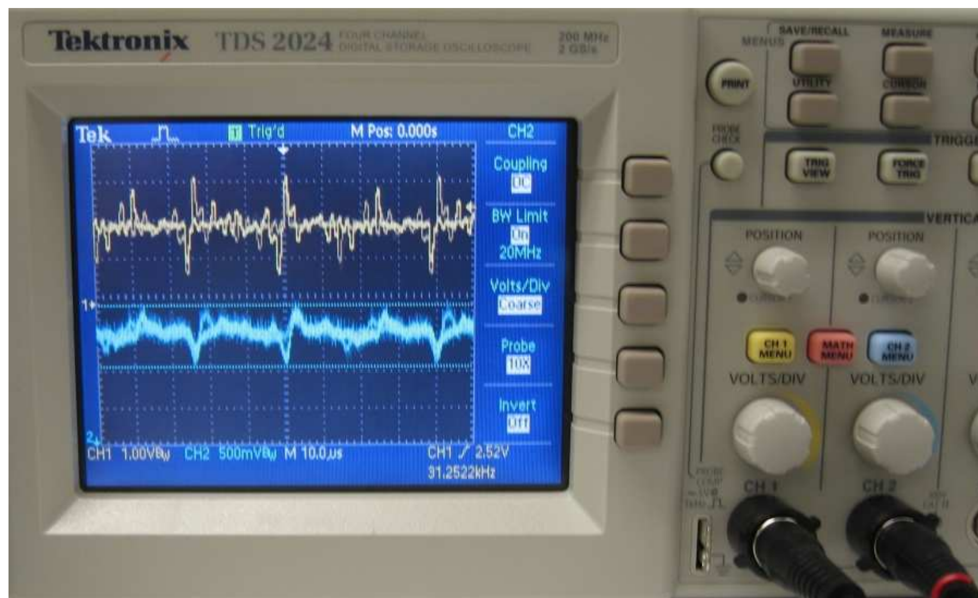



**Figure 7.** Oscilloscope of LED-link, showing transmitted LED current on top (yellow) and received photodiode light intensity on bottom (blue). High noise is clearly visible.

In general, local mode, loopback mode, and link mode were all demonstrated, albeit with some noise (to be expected with a project like this).

# 5 Discussion and Conclusions

We have built the system that we set out to create, and demonstrated its use in transmiting digital information at a significant rate, with an acceptable margin of reliability. Our final data rate was roughly 70.3 kilobits per second, and our system was capable of sending recognizable audio streams, as well as reasonably accurate Whiteboard drawing instructions. Although we did not have as much time as we would have liked to explore more interesting data transmission schemes, such as more advanced data encoding and protection, we nonetheless learned much about dealing with anlog noise in digital systems. Further, we believe it is a rich area of study for 6.111 projects, and are excited to think of what interesting directions future groups will explore.

# 6 Appendix: Verilog Source Listing

## 6.1 adc_ad7476.v

```
// adc_ad7476a module
// Michael F. Robbins, mrobbins@mit.edu
//
// Implements an incoming serial interface to the Analog
// Devices AD7476A analog to digital converter chip.
// The chip takes a voltage between 0 and 5V and outputs
// a 12-bit (unsigned) sample.
//
// Note that this chip is meant to run at a maximum of 20MHz serial clock
// but that we are pushing it up to 21MHz...
//
// Datasheet: http://www.analog.com/UploadedFiles/Data_Sheets/AD7476A_7477A_7478A.pdf
module adc_ad7476a(clk, start, data, done, cs, sclk, sdata, clock_40mhz_unbuf);
        input clk;      // assumes 27MHz clock in
        input start;
        output signed [11:0] data;
        output done;
        output cs; // active low
        output sclk;
        input sdata;

        output clock_40mhz_unbuf;

        // synthesize 20MHz clock
        // use FPGA's digital clock manager to produce a
   // 20MHz clock (actually 21.0MHz)
        // 40MHz clock is actually 42 MHz (27 / 9 * 14)
   wire clock_40mhz_unbuf,clock_20mhz;
   DCM vclk1(.CLKIN(clk),.CLKFX(clock_40mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 9
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 14
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37

        // divide the 40MHz clock by two
        reg clock_20mhz_unbuf;
        always @ (posedge clock_40mhz_unbuf)
                clock_20mhz_unbuf <= ~clock_20mhz_unbuf;
   BUFG vclk2(.O(clock_20mhz),.I(clock_20mhz_unbuf));

        reg running;
        assign sclk = !clock_20mhz || !running; // actually 21MHz...
        assign cs = !running;

        // 12-bit shift register for the input data
        reg [11:0] data_unsync;
        reg [4:0] bitnum; // counts through the 16 bits we have to read


        reg [2:0] start_27mhz_last;
        always @ (posedge clk)
                start_27mhz_last <= {start_27mhz_last[1:0], start};
        reg start_wait, start_sync, start_last;
        reg done_unsync;        // 20MHz done signal
        // SERIAL comm synchronized to 20MHz clock
        always @ (posedge clock_20mhz) begin
                // synchronize the start signal to the 20MHz clock domain
                start_wait <= start_27mhz_last[2] | start_27mhz_last[1];
                start_sync <= start_wait;
                start_last <= start_sync;
```

```
		if(start_sync && !start_last) begin
			bitnum <= 0;
			running <= 1;
		end
		else if(running) begin
			bitnum <= bitnum + 1;
			if(bitnum == 15) begin
				running <= 0;
			end
			// read a bit
			data_unsync <= {data_unsync[10:0], sdata};
		end
	end

	wire signed [12:0] data_unsync_signed = {1'b0, data_unsync};
	wire signed [12:0] data_unsync_twoscomp = data_unsync_signed - 13'sd2048;

	// DONE output synchronized to 27MHz clock
	reg done_wait1, done_wait2, done;
	reg [11:0] data;
	always @ (posedge clk) begin
		done_wait1 <= !running;
		done_wait2 <= done_wait1;
		done <= done_wait2 && !done_wait1;	// make sure that we only send out one "done" pulse
per sample

		if(done_wait1)
			data <= data_unsync_twoscomp[11:0];
	end
endmodule
```

## 6.2 audio_ac97.v

```
// BEGIN COPY FROM lab4.v

///////////////////////////////////////////////////////////////////////////////
//
// bi-directional monaural interface to AC97
//
///////////////////////////////////////////////////////////////////////////////

module lab4audio (clock_27mhz, reset, volume,
                  audio_in_data, audio_out_data, ready,
                  audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                  ac97_synch, ac97_bit_clock);

   input clock_27mhz;
   input reset;
   input [4:0] volume;
   output [7:0] audio_in_data;
   input [7:0] audio_out_data;
   output ready;

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

   wire [2:0] source;
   assign source = 0;      //mic

   wire [7:0] command_address;
   wire [15:0] command_data;
```

```
   wire command_valid;
   wire [19:0] left_in_data, right_in_data;
   wire [19:0] left_out_data, right_out_data;

   reg audio_reset_b;
   reg [9:0] reset_count;

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

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

   // ready: one cycle pulse synchronous with clock_27mhz
   reg [2:0] ready_sync;
   always @ (posedge clock_27mhz) begin
     ready_sync <= {ready_sync[1:0], ac97_ready};
   end
   assign ready = ready_sync[1] & ~ready_sync[2];

   reg [7:0] out_data;
   always @ (posedge clock_27mhz)
     if (ready) out_data <= audio_out_data;
   assign audio_in_data = left_in_data[19:12];
   assign left_out_data = {out_data, 12'b000000000000};
   assign right_out_data = left_out_data;

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

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

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

   input ac97_sdata_in;
   input ac97_bit_clock;
   output ac97_sdata_out;
   output ac97_synch;

   reg ready;

   reg ac97_sdata_out;
```

```
reg ac97_synch;

reg [7:0] bit_count;

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

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

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

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

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

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

   // Latch user data at the end of each frame. This ensures that the
   // first frame after reset will be empty.
   if (bit_count == 255)
     begin
        l_cmd_addr <= {command_address, 12'h000};
        l_cmd_data <= {command_data, 4'h0};
        l_cmd_v <= command_valid;
        l_left_data <= left_data;
        l_left_v <= left_valid;
        l_right_data <= right_data;
        l_right_v <= right_valid;
     end

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

```
          default: ac97_sdata_out <= 1'b0;
        endcase

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

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

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

      bit_count <= bit_count+1;

   end // always @ (posedge ac97_bit_clock)

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

endmodule

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

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

   reg [23:0] command;
   reg command_valid;

   reg [3:0] state;

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

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

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

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

      case (state)
        4'h0: // Read ID
          begin
             command <= 24'h80_0000;
             command_valid <= 1'b1;
          end
        4'h1: // Read ID
          command <= 24'h80_0000;
        4'h3: // headphone volume
          command <= { 8'h04, 3'b000, vol, 3'b000, vol };
        4'h5: // PCM volume
          command <= 24'h18_0808;
        4'h6: // Record source select
          command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
        4'h7: // Record gain = max
          command <= 24'h1C_0F0F;
        4'h9: // set +20db mic gain
          command <= 24'h0E_8048;
        4'hA: // Set beep volume
          command <= 24'h0A_0000;
        4'hB: // PCM out bypass mix1
          command <= 24'h20_8000;
        default:
          command <= 24'h80_0000;
      endcase // case(state)
   end // always @ (posedge clock)
endmodule // ac97commands

// END COPY from lab4.v

// lowpassfilter module
// Michael F. Robbins, mrobbins@mit.edu
//
// Implements a simple one-pole low-pass filter, intended for audio samples.
// It can be described as:
//    out[n+1] = (7/8)*out[n] + (1/8)*in[n]
// or as the z-transform:
//               (1/8)
//          H(z) = -----------
//             z - (7/8)
module lowpassfilter(clk, newsample, in, out, newsampleout);
        input clk, newsample;
        input signed [7:0] in;
        output signed [7:0] out;
        output newsampleout;

        reg signed [7:0] out;
        reg newsampleout;

        wire signed [11:0] outmul = out * 7;
        wire signed [11:0] inmul = in * 1;


        always @ (posedge clk) begin
                newsampleout <= 0;

                if(newsample) begin
                        newsampleout <= 1;

                        out <= (outmul) / 8 + (inmul / 8);
                end
```

```
        end
endmodule
```

## 6.3 bin_threshold.v

```
module bin_threshold(clk, threshold, newbin, xk_bin, xk_magsq, bins, newbins);
        input clk;
        input [23:0] threshold;
        input newbin;
        input [4:0] xk_bin;
        input [23:0] xk_magsq;
        output [31:0] bins;
        output newbins;

        reg [31:0] intbins;
        reg [31:0] bins;
        reg newbins;

        wire newbinval;
        assign newbinval = (xk_magsq >= threshold);

        always @ (posedge clk) begin
                newbins <= 0;
                if(newbin) begin
                        // build intbins bit by bit, as shift register, with new data
                        // added onto the end (we get bin 0 in first)
                        intbins <= {newbinval, intbins[31:1]};

                        if(xk_bin == 0) begin
                                // signal new set of bins out
                                bins <= intbins;
                                newbins <= 1'b1;
                        end
                end
        end
endmodule
```

## 6.4 cos64.v

```
// cos64 module
// Michael F. Robbins, mrobbins@mit.edu
//
// Stores a 12-bit (two's complement) value of a cosine lookup table.
// The function is:
//    value = 2047 * cos((pi/32) * addr)
// This module uses the sin64 lookup table and just shifts the address.
module cos64(addr, value);
        input [5:0] addr;
        output signed [11:0] value;

        wire [6:0] shiftedaddr;
        assign shiftedaddr = addr + 16;  // cos(x) = sin(x + pi/2)

        sin64 sintable(.addr(shiftedaddr[5:0]), .value(value));
endmodule
```

## 6.5 dac_ad5399.v

```
// dac_ad5399 module
```

```
// Michael F. Robbins, mrobbins@mit.edu
//
// Implements an outgoing serial interface to the Analog
// Devices AD5399 digital to analog converter chip.
// The chip takes a 12-bit, twos-complement data and can
// output to two analog channels.
//
// Datasheet: http://www.analog.com/UploadedFiles/Data_Sheets/AD5399.pdf
module dac_ad5399(clk, start, addr, data, cs, sclk, sdi);
	input clk;
	input start;
	input addr; //  0 == DAC A, 1 == DAC B
	input signed [11:0] data; // 12-bit twos-complement data

	// The 16 bit address word and its latched value.
	wire [15:0] dataword_tmp = {addr, 3'b0, data};
	reg [15:0] dataword;

	// bit counter (counts from 15 down to 0)
	reg [3:0] bitnum;

	// internal CS -- active high
	reg running;

	// outputs to the chip
	// Timing info: see datasheet figure 5.
	// The AD5399 clocks data in on the rising edge of SCLK,
	// and has a 5ns setup time (zero hold time).
	// Therefore, we'll invert the clock, and put out new data on its falling edge (our rising edge).
	output cs; // active low
	output sclk;
	output sdi;

	assign cs = ~running;
	assign sclk = ~clk;
	assign sdi = dataword[bitnum];



	always @ (posedge clk) begin
		if(start) begin
			// on start, read the top bit and move clock select low
			running <= 1;
			bitnum <= 15;
			// and shift in the 16-bit dataword
			dataword <= dataword_tmp;
		end
		else if(running) begin
			if(bitnum == 0)
				// after the last bit goes out, raise CS
			   running <= 0;
			else
				// read the next bit onto SDI
				bitnum <= bitnum - 1;
		end
	end
endmodule
```

## 6.6 debounce.v

```
// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce (reset, clock, noisy, clean);
```

```
   parameter DELAY = 270000;   // .01 sec with a 27Mhz clock
   input reset, clock, noisy;
   output clean;

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

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

endmodule
```

## 6.7 delayN.v

```
// sostler

///////////////////////////////////////////////////////////////////////////////
// parameterized delay line

module delayN(clk,in,out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 4;

   reg [NDELAY-1:0] shiftreg;
   wire             out = shiftreg[NDELAY-1];

   always @(posedge clk)
     shiftreg <= {shiftreg[NDELAY-2:0],in};

endmodule // delayN
```

## 6.8 dft64.v

```
// dft64_magsq module
// Michael F. Robbins, mrobbins@mit.edu
//
//      Uses the dft64 module to compute the real/imaginary parts of the DFT,
// and just computes the magnitude squared.  Not really too much heavy
// lifting going on inside this module.
//
module dft64_magsq(clk, reset, sample, newsample, xk_bin, xk_magsq, newbin);
        input clk, reset, newsample;
        input [11:0] sample;
        output [4:0] xk_bin;
        output [32:0] xk_magsq;
```

```
	output newbin;

	wire [4:0] xk_binin;
	wire signed [15:0] xk_re;
	wire signed [15:0] xk_im;
	wire newbinin;
	dft64 dftin(.clk(clk), .reset(reset), .sample(sample),
					.newsample(newsample), .xk_bin(xk_binin), .xk_re(xk_re),
.xk_im(xk_im),
					.newbin(newbinin));

	// delay registers for pipeline stage 1
	reg newbin_t;
	reg signed [11:0] xk_re_t;
	reg signed [11:0] xk_im_t;
	reg [4:0] xk_bin_t;

	// compute magnitude in stage 2
	wire [31:0] xk_re_sq;
	assign xk_re_sq = xk_re_t*xk_re_t;
	wire [31:0] xk_im_sq;
	assign xk_im_sq = xk_im_t*xk_im_t;

	reg newbin;
	reg [4:0] xk_bin;
	reg [32:0] xk_magsq;

	always @ (posedge clk) begin
		// pipeline stage 1: read in inputs
		newbin_t <= newbinin;
		xk_bin_t <= xk_binin;
		xk_re_t <= xk_re;
		xk_im_t <= xk_im;

		// pipeline stage 2: output computation
		newbin <= newbin_t;
		xk_bin <= xk_bin_t;
		xk_magsq <= xk_re_sq + xk_im_sq;
	end
endmodule

// dft64 module
// Michael F. Robbins, mrobbins@mit.edu
//
// Takes in 64 samples, and then starts outputting the real and imaginary parts
// of their discrete-time Fourier series.  Inputs and outputs are all signed.
//
// Uses two BRAMs (each 64x12bits), writing new samples into one while
// doing the processing on the other.  The two memories are designated "0" and "1"
// (to not be confused with the two samples simultaneously processed, "A" and "B").
//
// While writing new samples, it uses one port of the BRAM, just incrementing
// the address and putting in the sample.
//
// When processing the bins, it uses both ports to work on two samples at once, and computes the
// real and imaginary contributions of each simultaneously.  (There are thus 4 multipliers
// in action at once.)  The two samples are referred to as "A" and "B".
//
module dft64(clk, reset, sample, newsample, xk_bin, xk_re, xk_im, newbin);
	input clk, reset, newsample;
	input signed [11:0] sample;
	output [4:0] xk_bin;
	output signed [15:0] xk_re;
	output signed [15:0] xk_im;
	output newbin;

	// bin register
```

```
        reg [4:0] bin;
        reg working; // 1 = we're currently reading & summing, 0 = just waiting for more samples

        // memories
        reg readmem;    // 0 = reading from mem0, 1 = reading from mem1
        wire [5:0] addr0;
        wire [5:0] addr0p1;
        assign addr0p1 = addr0 + 1;
        wire signed [11:0] d0a;
        wire signed [11:0] d0b;
        wire we0;
        bram64x12 mem0(.addra(addr0), .addrb(addr0p1), .clka(clk), .clkb(clk),
                                                .dina(sample), .dinb(12'b0), .douta(d0a), .doutb(d0b),
.wea(we0), .web(1'b0));
        wire [5:0] addr1;
        wire [5:0] addr1p1;
        assign addr1p1 = addr1 + 1;
        wire signed [11:0] d1a;
        wire signed [11:0] d1b;
        wire we1;
        bram64x12 mem1(.addra(addr1), .addrb(addr1p1), .clka(clk), .clkb(clk),
                                                .dina(sample), .dinb(12'b0), .douta(d1a), .doutb(d1b),
.wea(we1), .web(1'b0));

        // write enable lines
        assign we0 = !reset && (readmem == 1) && newsample;
        assign we1 = !reset && (readmem == 0) && newsample;

        // address lines
        reg [5:0] readaddr;
        reg [5:0] writeaddr;

        assign addr0 = (readmem==0) ? readaddr : writeaddr;
        assign addr1 = (readmem==1) ? readaddr : writeaddr;

        // data lines
        wire signed [11:0] data_a;
        wire signed [11:0] data_b;
        assign data_a = (readmem==0) ? d0a : d1a;
        assign data_b = (readmem==0) ? d0b : d1b;

        // sine table lookups
        wire [11:0] tableindex_a;
        wire signed [11:0] sin_a;
        wire signed [11:0] cos_a;
        assign tableindex_a = readaddr * bin;
        sin64 sintable_a(.addr(tableindex_a[5:0]), .value(sin_a));
        cos64 costable_a(.addr(tableindex_a[5:0]), .value(cos_a));
        wire [11:0] tableindex_b;
        wire signed [11:0] sin_b;
        wire signed [11:0] cos_b;
        assign tableindex_b = (readaddr+1) * bin;
        sin64 sintable_b(.addr(tableindex_b[5:0]), .value(sin_b));
        cos64 costable_b(.addr(tableindex_b[5:0]), .value(cos_b));

        // accumulators
        reg signed [29:0] accum_re;
        reg signed [29:0] accum_im;

        wire signed [29:0] next_accum_re;
        wire signed [29:0] next_accum_im;
        assign next_accum_re = accum_re + (cos_a * data_a) + (cos_b * data_b);
        assign next_accum_im = accum_im + (sin_a * data_a) + (sin_b * data_b);

        // computational logic
        reg [4:0] xk_bin;
        reg signed [15:0] xk_re;
```

```
reg signed [15:0] xk_im;
reg newbin;

always @ (posedge clk) begin
	if(reset) begin
		// reset
		readmem <= 1;
		readaddr <= 0;
		writeaddr <= 0;
		working <= 0;
		bin <= 0;
		xk_bin <= 0;
		xk_re <= 0;
		xk_im <= 0;
		accum_re <= 0;
		accum_im <= 0;
		newbin <= 1'b0;
	end
	else begin // not reset
		// write address lines
		if(newsample) begin
			writeaddr <= writeaddr + 1;
			if(writeaddr == 63) begin
				readmem <= ~readmem;
				bin <= 0;
				writeaddr <= 0;
				readaddr <= 0;
				working <= 1;
			end
		end

		// read addres lines
		if(working) begin // this guy goes through the samples for a given bin
			readaddr <= (readaddr==62) ? 0 : readaddr + 2;
			if(readaddr == 62 && bin == 31) working <= 0;
		end

		if(working) begin
			// working, i.e. we're computing a DFT.
			// readaddr is incremented above
			if(readaddr == 62) begin
				// This is the last pair of samples to read.
				// So, put final values out and clear accumulators.
				newbin <= 1'b1;
				xk_bin <= bin;
				bin <= bin + 1;
				// full scale would be
				//   xk_re <= next_accum_re[29:18];
				//   xk_im <= next_accum_im[29:18];
				// but we lose 5 bits because of the amplitude of the inverse FFT.
				// So for full generality, if this module is to be used elsewhere,
				// this should be changed.
				xk_re <= next_accum_re[26:11];
				xk_im <= next_accum_im[26:11];
				accum_re <= 0;
				accum_im <= 0;
			end
			else begin
				// This is not the final sample; just accumulate.
				accum_re <= next_accum_re;
				accum_im <= next_accum_im;
				newbin <= 1'b0;
			end
		end // if(working)
		else begin
			newbin <= 1'b0;
		end
```

```
                end // if(reset)
        end // always @ (posedge clk)
endmodule
```

## 6.9 display_16hex.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

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

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

   always @(posedge clock_27mhz)
     begin
        if (reset)
          begin
             count = 0;
             clock = 0;
          end
        else if (count == 26)
          begin
             clock = ~clock;
             count = 5'h00;
          end
        else
```

```
      count = count+1;
  end

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

assign dreset = (reset_count != 0);

assign disp_clock = ~clock;

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

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

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

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

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

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

      8'h03:
```

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

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

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

      8'h06:
        begin
           // Load the user's dot data into the dot reg, char by char
           disp_ce_b <= 1'b0;
           disp_data_out <= dots[dot_index]; // dot data from msb
           if (dot_index == 0)
             if (char_index == 0)
               state <= 5;                   // all done, latch data
             else
             begin
               char_index <= char_index - 1; // goto next char
               dot_index <= 39;
             end
           else
             dot_index <= dot_index-1;       // else loop thru all dots
        end

    endcase

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

```
    endcase

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

endmodule
```

## 6.10 encoder_2of3.v

```
// twoofthree module
// Michael F. Robbins, mrobbins@mit.edu
//
// combinational majority decision.
module twoofthree(a, b, c, out);
        input a, b, c;
        output out;

        assign out = (a&b) | (a&c) | (b&c);
endmodule

// encoder module
// Michael F. Robbins, mrobbins@mit.edu
//
// combinational word->bins module
module encoder(wordin, guard, bins);
        input [8:0] wordin;        // 9 bits in
        input guard;                    // 1 = guard frame, 0 = data frame
        output [31:0] bins;  // 31 frequency bins out

        // temporary encoding:
        // bit 0 -> bins 1, 10, 19
        // bit 8 -> bins 9, 18, 27
        // guard -> bins 28..30
        // always -> bin 31

        assign bins[0] = 1'b0;
        assign bins[9:1]   = {~wordin[8], wordin[7],~wordin[6], wordin[5],~wordin[4], wordin[3],~wordin[2],
wordin[1],~wordin[0]};
        assign bins[18:10] = { wordin[8],~wordin[7], wordin[6],~wordin[5], wordin[4],~wordin[3],
wordin[2],~wordin[1], wordin[0]};
        assign bins[27:19] = {~wordin[8], wordin[7],~wordin[6], wordin[5],~wordin[4], wordin[3],~wordin[2],
wordin[1],~wordin[0]};
        assign bins[30:28] = {guard, guard, guard};
        assign bins[31] = 1'b1;
endmodule
```

```
// decoder module
// Michael F. Robbins, mrobbins@mit.edu
//
// Combinational.  Bins -> 9 bit word.
module decoder(bins, guardout, wordout);
        input [31:0] bins;
        output guardout;
        output [8:0] wordout;

        // use 2 of 3 consensus decider.
        twoofthree bit0(~bins[0+1], bins[0+10],~bins[0+19], wordout[0]);
        twoofthree bit1( bins[1+1],~bins[1+10], bins[1+19], wordout[1]);
        twoofthree bit2(~bins[2+1], bins[2+10],~bins[2+19], wordout[2]);
        twoofthree bit3( bins[3+1],~bins[3+10], bins[3+19], wordout[3]);
        twoofthree bit4(~bins[4+1], bins[4+10],~bins[4+19], wordout[4]);
        twoofthree bit5( bins[5+1],~bins[5+10], bins[5+19], wordout[5]);
        twoofthree bit6(~bins[6+1], bins[6+10],~bins[6+19], wordout[6]);
        twoofthree bit7( bins[7+1],~bins[7+10], bins[7+19], wordout[7]);
        twoofthree bit8(~bins[8+1], bins[8+10],~bins[8+19], wordout[8]);

        twoofthree guard(bins[30], bins[29], bins[28], guardout);
endmodule
```

## 6.11 finalproject.v

```
// labkit module -- finalproject.v
// Scott Ostler and Mike Robbins
//
// This is the top-level module.
//
// Current inputs/outputs:
//      DAC chip (AD5399):
//                      _CS_            user3[31]
//                      SCLK            user3[30]
//                      SDATA           user3[29]
//              ADC chip (AD7476A):
//                      _CS_            user4[14]
//                      SCLK            user4[13]
//                      SDI             user4[12]
//              Buttons:
//                      up,down,left,right      (move pointer)
//                      enter           (draw pixel)
//                      3                       (clear screen)
//                      1                       (manually send guard frame)
//                      0                       (reset)
//              Switches:
//                      7                       (1 = use real ADC sample for input, 0 = use internal loopback)
//                      6                       (1 = use automatic guard frame sending, 0 = manual guard
frame)
//                      5                       (1 = wordin/wordout on analyzer4, 0 = dft64_magsq on
analyzer4)
//                      4                       (1 = surpress local drawing, 0 = don't surpress local drawing)
//                      3                       (used for manual IFFT mode)
//                      2                       (used for manual IFFT mode)
//                      1                       (used for manual IFFT mode)
//                      0                       (1 = manual IFFT mode, 0 = normal operation)


///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
```

```
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2006-Mar-08: Corrected default assignments to "vga_out_red", "vga_out_green"
//              and "vga_out_blue". (Was 10'h0, now 8'h0.)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
///////////////////////////////////////////////////////////////////////////////

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

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

               tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
               tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
```

```
            tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

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

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

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

            clock_feedback_out, clock_feedback_in,

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

            rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

            mouse_clock, mouse_data, keyboard_clock, keyboard_data,

            clock_27mhz, clock1, clock2,

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

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

            switch,

            led,

            user1, user2, user3, user4,

            daughtercard,

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

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

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

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

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

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

inout  [35:0] ram0_data;
output [18:0] ram0_address;
```

```
output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
output [3:0] ram0_bwe_b;

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

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

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

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

// Audio Input and Output
assign beep= 1'b0;
//assign audio_reset_b = 1'b0;
//assign ac97_synch = 1'b0;
//assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// VGA Output
//assign vga_out_red = 8'h0;
//assign vga_out_green = 8'h0;
//assign vga_out_blue = 8'h0;
//assign vga_out_sync_b = 1'b1;
//assign vga_out_blank_b = 1'b1;
//assign vga_out_pixel_clock = 1'b0;
//assign vga_out_hsync = 1'b0;
//assign vga_out_vsync = 1'b0;
```

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

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

// SRAMs
//assign ram0_data = 36'hZ;
//assign ram0_address = 19'h0;
//assign ram0_adv_ld = 1'b0;
//assign ram0_clk = 1'b0;
//assign ram0_cen_b = 1'b1;
//assign ram0_ce_b = 1'b1;
//assign ram0_oe_b = 1'b1;
//assign ram0_we_b = 1'b1;
//assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input


// Custom RAM assignment
assign ram0_ce_b = 1'b0;
assign ram0_oe_b = 1'b0;
assign ram0_adv_ld = 1'b0;


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

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

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

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

   // Buttons, Switches, and Individual LEDs
   //assign led = 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 analyzer1_data = 16'h0;
   assign analyzer1_clock = 1'b1;
   //assign analyzer2_data = 16'h0;
   //assign analyzer2_clock = 1'b1;
   assign analyzer3_data = 16'h0;
   assign analyzer3_clock = 1'b1;
   //assign analyzer4_data = 16'h0;
   //assign analyzer4_clock = 1'b1;

// ********
// FFTDEMO2
// ********

        // POWER-ON RESET (FROM LECTURE 8)
        wire power_on_reset, resetbutton, reset;
        SRL16 reset_sr (.D(1'b0), .CLK(clock_27mhz), .Q(power_on_reset),
                                        .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
        defparam reset_sr.INIT = 16'hFFFF;
        debounce resetbtn(1'b0, clock_27mhz, ~button0, resetbutton);
        assign reset = power_on_reset | resetbutton;

        // GUARD SYNC BUTTON
        wire guard_sync;
        debounce guardbtn(reset, clock_27mhz, ~button1, guard_sync);

        // ONEMHZ ENABLE
        wire onemhz_enable;
        onemhz_divider divider(clock_27mhz, reset, onemhz_enable);
```

```
     // ===============================================================================

     wire is_loopback = switch[4]; // if 1, supress local drawing

////////////////////////
// POINTER HANDLING  ///
////////////////////////
     wire video_clock = clock_27mhz;
// BUTTON SYNCH AND INVERT

wire button_left_syn, button_right_syn, button_up_syn, button_down_syn;
wire button_enter_syn, button_clear_syn;

synchronize left_syn(video_clock, ~button_left, button_left_syn);
synchronize right_syn(video_clock, ~button_right, button_right_syn);
synchronize up_syn(video_clock, ~button_up, button_up_syn);
synchronize down_syn(video_clock, ~button_down, button_down_syn);
synchronize enter_syn(video_clock, ~button_enter, button_enter_syn);
synchronize clear_syn(video_clock, ~button3, button_clear_syn);

// POINTER MOVEMENT

// 12 bits for easy hexdisplay output
wire [11:0] local_x, local_y;
wire frame_start;
wire clear_screen;
wire draw_pixel;
     wire draw_remote_pixel;
     wire [11:0] remote_x, remote_y;
     wire [26:0] draw_command_get, draw_command_put;

whiteboard wb(video_clock, reset, is_loopback, frame_start,
              button_left_syn, button_right_syn, button_up_syn, button_down_syn,
              button_enter_syn, button_clear_syn,
              clear_screen, draw_pixel, draw_remote_pixel,
              local_x, local_y,
                                       remote_x, remote_y,
                                       draw_command_get, draw_command_put);

////////////////////////
// VIDEO HANDLING    ///
////////////////////////

// XVGA SIGNALS

wire [10:0] hcount;
wire [9:0]  vcount;
wire hsync,vsync,blank;
xvga xvga1(video_clock,hcount,vcount,hsync,vsync,blank);

parameter RIGHT_BOUND = 639;
parameter BOTTOM_BOUND = 479;

// begin frame calculations at first non-drawn line
assign frame_start = (vcount == BOTTOM_BOUND + 1 && hcount == 0);

// RAM

wire [35:0] vram_write_data;
wire [35:0] vram_read_data;

wire [18:0] vram_read_addr;
wire [18:0] vram_write_addr;
wire [18:0] vram_addr = (blank) ? vram_write_addr : vram_read_addr;

wire        vram_we;
wire [1:0]  vram_write_byte;
```

```
   zbt_6111 zbt1(video_clock, 1'b1, vram_we, vram_addr, vram_write_byte,
                        vram_write_data, vram_read_data,
                        ram0_clk, ram0_we_b, ram0_address, ram0_data, ram0_cen_b, ram0_bwe_b);


   // VRAM READING
   wire [7:0] vr_pixel;
   vram_display vd1(reset,video_clock,hcount,vcount,
                 vr_pixel,
                 vram_read_addr,
                 vram_read_data);


   // VRAM WRITING
        memory_router mr(clear_screen, draw_pixel, draw_remote_pixel,
                                              local_x, local_y, remote_x, remote_y, hcount, vcount,
                                              vram_write_addr, vram_write_byte, vram_write_data, vram_we);

   // POINTER OVERLAY
   wire pnt_enable;
   wire pnt_pixel;   // 1 = white, 0 = black
   pointer pnt(local_x, local_y, hcount, vcount, pnt_pixel, pnt_enable);

   // DRAWING DELAY
   wire pnt_pixel_d; // delay pointer pixels to sync with ram pixels
   wire pnt_enable_d;
   wire blank_d;
   wire hsync_d;
   wire vsync_d;

   // delay by 3 cycles to sync with ZBT read
   delayN dn1(video_clock,hsync,hsync_d);
   delayN dn2(video_clock,vsync,vsync_d);
   delayN dn3(video_clock,blank,blank_d);
   delayN dn4(video_clock,pnt_pixel,pnt_pixel_d);
   delayN dn5(video_clock,pnt_enable,pnt_enable_d);

   wire [7:0] pixel;
   assign pixel = pnt_enable_d ? {8{pnt_pixel_d}} :
                              (clear_screen ? 8'd200 : vr_pixel);

   // VIDEO WIRES
   assign vga_out_red = pixel;
   assign vga_out_green = pixel;
   assign vga_out_blue = pixel;
   assign vga_out_sync_b = 1'b1;    // not used
   assign vga_out_pixel_clock = ~video_clock;
   assign vga_out_blank_b = ~blank_d;
   assign vga_out_hsync = hsync_d;
   assign vga_out_vsync = vsync_d;

        // ==============================================================================


        // AUDIO (FROM LAB 4)
        wire [7:0] from_ac97_data, to_ac97_data;
   wire ac97_ready;
        // fixed volume 5'd8
        // AC97 driver
   lab4audio a(clock_27mhz, reset, 5'd8, from_ac97_data, to_ac97_data, ac97_ready,
               audio_reset_b, ac97_sdata_out, ac97_sdata_in,
               ac97_synch, ac97_bit_clock);
        wire [7:0] from_ac97_lpf;
        wire newaudiosample;
        lowpassfilter lpf(.clk(clock_27mhz), .newsample(ac97_ready), .in(from_ac97_data), .out(from_ac97_lpf),
.newsampleout(newaudiosample));
```

```
	// INPUT BITS
	wire [8:0] wordin;
	packet_engine_transmit pet(.clk(clock_27mhz), .onemhz_enable(onemhz_enable),
									.audio_word({ 1'b0,
from_ac97_lpf}), .draw_packet_put(draw_command_put), .word(wordin));

	// ENCODER
	wire [31:0] bins, bins_from_encoder;
	assign bins = switch[0] ? {28'h0, switch[3:1], 1'b0} : bins_from_encoder; // for demoing the FFT/IFFT
	reg [6:0] tmp_slowcount;
	reg [4:0] tmp_framecount;
	always @ (posedge clock_27mhz)
		if(onemhz_enable) begin
			tmp_slowcount <= tmp_slowcount + 1;
			if(tmp_slowcount == 127) begin
				// a new FFT frame is going out
				tmp_framecount <= tmp_framecount + 1;
			end
		end
	wire send_guard = switch[6] ? (tmp_framecount == 0) : guard_sync;
	encoder enc(.wordin(wordin), .guard(send_guard), .bins(bins_from_encoder));
	//assign bins = {27'b0, switch[3:0], 1'b0};

	// FFTOUT SHIFTER
	wire [11:0] sampleout;
	wire newsample;
	shifter fftout(.clk(clock_27mhz), .reset(reset), .onemhz_enable(onemhz_enable),
					.bins(bins), .sample(sampleout), .newsample(newsample));


	// DAC output
	// wires to DAC module are on header user3:
	//		USER3[31] --> CS (pin 10)
	//	USER3[30] --> SCLK (pin 1)
	//		USER3[29] --> SDI (pin 2)
	assign user3[28:0] = 29'bZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;
	dac_ad5399 dac(.clk(clock_27mhz), .start(newsample), .addr(1'b0), .data(sampleout),
						.cs(user3[31]), .sclk(user3[30]), .sdi(user3[29]));


	// ADC input
	wire signed [11:0] adc_samplein;
	wire adc_newsample;
	assign user4[31:15] = 17'bZZZZZZZZZZZZZZZZZ;
	assign user4[10:0] = 12'bZZZZZZZZZZZZ;
	adc_ad7476a adc(.clk(clock_27mhz), .start(onemhz_enable), .data(adc_samplein),
					.done(adc_newsample), .cs(user4[14]), .sclk(user4[13]),
.sdata(user4[12]),
					.clock_40mhz_unbuf(user4[11]));

	// Select either the sample from the ADC or just internally (noise free)
	wire signed [11:0] samplein;
	assign samplein = switch[7] ? adc_samplein : sampleout;

	// FFT MAGSQ
	wire [4:0] xk_bin;
	wire [32:0] xk_magsq_raw;
	wire newbin;
	dft64_magsq dftin(.clk(clock_27mhz), .reset(reset), .sample(samplein),
				.newsample(onemhz_enable), .xk_bin(xk_bin), .xk_magsq(xk_magsq_raw),
.newbin(newbin));
	wire [23:0] xk_magsq = xk_magsq_raw[25:2];	// full scale would be [32:9], but

		 // we're limited in magnitude (found

		 // by trial and error) -- mrobbins
```

```
        // guard detector
        wire is_guard_real; // from guard detector
        wire is_guard_fake = guard_sync; // from button1
        wire is_guard = switch[6] ? is_guard_real : is_guard_fake;
        wire [23:0] guard_strength;
        guard_detector gd(.clk(clock_27mhz), .newbin(newbin), .xk_bin(xk_bin), .xk_magsq(xk_magsq),
                                        .is_guard(is_guard_real), .guard_strength(guard_strength));

        // fixed threshold
        wire [31:0] binstofs;
        wire newbinsout;
        wire [23:0] last_guard_strength;        // of known-good guard
        bin_threshold thrf(.clk(clock_27mhz), .threshold(last_guard_strength), .newbin(newbin),
                                                        .xk_bin(xk_bin), .xk_magsq(xk_magsq),
                                                        .bins(binstofs), .newbins(newbinsout));

        // frame synchronizer
        wire [31:0] binstodecoder;
        wire goodbins;
        frame_sync fs(.clk(clock_27mhz), .newbins(newbinsout), .bins(binstofs), .is_guard(is_guard),
                                  .guard_strength(guard_strength), .binsout(binstodecoder),
.goodbins(goodbins),
                                  .last_guard_strength(last_guard_strength));

        // decoder
        wire [8:0] wordout;
        wire guardout;
        decoder decode(.bins(binstodecoder), .guardout(guardout), .wordout(wordout));

        // receive-side packet engine
        wire [7:0] last_audio;
        packet_engine_receive per(.clk(clock_27mhz), .goodbins(goodbins), .word(wordout),
                                                                        .last_audio(last_audio),
.draw_packet(draw_command_get));

        // DEBUGGING OUTPUTS
        // hex data display
        wire [63:0] hexdata;
        display_16hex display1 (reset, clock_27mhz, hexdata,
                                                                        disp_blank, disp_clock, disp_rs,
disp_ce_b,
                                                                        disp_reset_b, disp_data_out);
//      assign hexdata[11:0] = sampleout;
//      assign hexdata[31:12] = 20'b0;
//      assign hexdata[55:32] = last_guard_strength;
//      assign hexdata[63:56] = 0;

        reg [26:0] temp_command_get;
        always @(posedge clock_27mhz) begin
                if (draw_command_get[26]) begin
                        temp_command_get <= draw_command_get;
                end
        end

        assign hexdata = {1'b0, draw_command_put, 8'b0000_0000, 1'b0, temp_command_get};

        // LEDs:
        //assign led = ~wordout[7:0];
        assign led = ~{button_enter_syn, button_up_syn, button_down_syn, 5'b0};


        // AC97: from packet engine...
        //wire [7:0] lpfout;
        //lowpassfilter lpf_out(.clk(clock_27mhz), .newsample(ac97_ready), .in(wordout[7:0]), .out(lpfout));
        assign to_ac97_data = last_audio;
```

```
        // LOGIC ANALYZER POD 2: received samples
        assign analyzer2_clock = clock_27mhz;
        assign analyzer2_data[11:0] = samplein;
        assign analyzer2_data[12] = onemhz_enable;
        assign analyzer2_data[13] = is_guard;
        assign analyzer2_data[14] = goodbins;
        assign analyzer2_data[15] = wordin[0];


        // LOGIC ANALYZER POD 4: fft magsq output OR wordin/wordout pairs.
        assign analyzer4_clock = wordout[0]; //  a hack, but we had to put it somewhere...
        wire [10:0] analyzer4_data_low;
        assign analyzer4_data_low = xk_magsq[23:13];
        wire [15:0] analyzer4_data_dft64 = {xk_bin, analyzer4_data_low}; // the bin# and the most significant
parts of the data.
        wire [15:0] analyzer4_data_words = {wordin[8:1], wordout[8:1]}; // high 8 bits only... sorry :(
        assign analyzer4_data = switch[5] ? analyzer4_data_words : analyzer4_data_dft64;
endmodule
```

## 6.12 frame_sync.v

```
// frame_sync module
// Michael F. Robbins, mrobbins@mit.edu
//
// Our "phase detection" system uses guard frames to tell which of alternating
// frames (64 samples long) we should look at.  When a guard packet comes across,
// we have to decide whether to change phase or not.  We also have to decide whether
// we should update the bin detection threshold value (last_guard_strength).
//
// When ever we recieve an in-phase frame, we pass its bins on to the decoder.
//
module frame_sync(clk, newbins, bins, is_guard, guard_strength, binsout, goodbins, last_guard_strength);
        input clk, newbins;
        input [31:0] bins;
        input is_guard;
        input [23:0] guard_strength;

        output [31:0] binsout;
        output goodbins;
        output [23:0] last_guard_strength;
        reg [31:0] binsout;
        reg goodbins;
        reg [23:0] last_guard_strength;

        reg phase;      // 0 when in-phase, 1 when out-of-phase
        reg last_was_guard;     // was the last frame a guard frame?

        always @ (posedge clk) begin
                goodbins <= 0;
                if(newbins) begin
                        phase <= ~phase;                // phase flips on every set of bins

                        if(phase == 0) begin    // pass on in-phase bins
                                binsout <= bins;
                                goodbins <= 1;
                        end

                        if(is_guard) begin
                                if(last_was_guard) begin
                                        if(last_guard_strength >= guard_strength)
                                                phase <= 0;     // last one was stronger, so ignore this guy
                                        else begin
                                                phase <= 1; // this one was stronger, so take phase control
                                                last_guard_strength <= guard_strength;
```

```
                                        end
                                end
                                else begin
                                        phase <= 1;     // we just received an in-phase frame, so the next one
is out-of-phase
                                        last_guard_strength <= guard_strength;
                                end
                        end

                        last_was_guard <= is_guard;
                end
        end
endmodule
```

## 6.13 guard_detector.v

```
// guard_detector module
// Michael F. Robbins, mrobbins@mit.edu
//
// A fairly simple strategy to detect whether or not this frame is a guard frame.
// Snoops the outputs from the dft64_magsq and after listening to the mag^2 values
// of bins 28 through 31, issues its decision.
//
// To be considered a guard frame, the median magnitude (midguard) of the
// guard frequencies (28-30) must be greater than 1/2 of the value of bin 31 (always on).
//
// If this is a guard frame, the "guard strength" (to be used as a threshold)
// is set to 3/4 of the value of the median magnitude (midguard).
//
// Note that this is suceptible to noise that targets bin #31, and makes our system
// vulnerable to noise (either constructive or destructive) on this bin.
//
module guard_detector(clk, newbin, xk_bin, xk_magsq, is_guard, guard_strength);
        input clk, newbin;
        input [4:0] xk_bin;             // listening to the output of dft64_magsq
        input [23:0] xk_magsq;

        output is_guard;
        output [23:0] guard_strength;   // =3/4 of the middle guard bin value

        reg [23:0] m28; // magnitude^2 of guard1
        reg [23:0] m29; // magnitude^2 of guard2
        reg [23:0] m30; // magnitude^2 of guard3
        reg [23:0] m31; // magnitude^2 of always on frequency

        reg [23:0] midguard;    // middle (median) guard value
        always @ (m28, m29, m30) begin
                if((m28 >= m29 && m28 <= m30) | (m28 >= m30 && m28 <= m29))
                        midguard = m28;
                else if((m29 >= m28 && m29 <= m30) | (m29 >= m30 && m29 <= m28))
                        midguard = m29;
                else
                        midguard = m30;
        end

        wire [23:0] qtrcutoff = (m31 >> 1); // divide "always on" bin by two

        reg is_guard;
        //assign is_guard = (midguard >= qtrcutoff);
        assign guard_strength = 3*(midguard / 4);

        always @ (posedge clk) begin
                if(newbin) begin
```

```
			case(xk_bin)				// bring in the data as we listen to
it crossing the wires...
				28:	m28 <= xk_magsq;
				29:	m29 <= xk_magsq;
				30:	m30 <= xk_magsq;
				31:	m31 <= xk_magsq;
			endcase
		end

		if(xk_bin == 31 && !newbin)	// make our decision after we've heard everything.
				is_guard <= (midguard >= qtrcutoff);
	end
endmodule
```

## 6.14 memory_router.v

```
// sostler
//
// multiplexes local drawing, remote drawing, and screen clearing wires to ZBT module

module memory_router(clear_screen, draw_pixel, draw_remote_pixel,
					local_x, local_y, remote_x, remote_y, hcount, vcount,
					vram_write_addr, vram_write_byte, vram_write_data, vram_we);

	input clear_screen;
	input draw_pixel, draw_remote_pixel;
	input [11:0] local_x, local_y, remote_x, remote_y;
	input [10:0] hcount;
	input [9:0] vcount;

	output [35:0] vram_write_data;
   output [18:0] vram_write_addr;
   output        vram_we;
   output [1:0]  vram_write_byte;


	wire [18:0] pointer_mem_addr = {1'b0, local_y[9:0], local_x[9:2]};
   wire [1:0]  pointer_mem_byte = local_x[1:0];
   wire [18:0] remote_mem_addr  = {1'b0, remote_y[9:0], remote_x[9:2]};
   wire [1:0]  remote_mem_byte  = remote_x[1:0];
   wire [18:0] screen_mem_addr  = {1'b0, vcount[9:0], hcount[9:2]};
   wire [1:0]  screen_mem_byte  = hcount[1:0];

   assign vram_write_addr = clear_screen ? screen_mem_addr : (draw_remote_pixel ? remote_mem_addr :
pointer_mem_addr);
   assign vram_write_byte = clear_screen ? screen_mem_byte : (draw_remote_pixel ? remote_mem_byte :
pointer_mem_byte);
   assign vram_write_data = clear_screen ? {4{9'd200}} : {36{0}};
   assign vram_we         = clear_screen || draw_pixel || draw_remote_pixel;

endmodule
```

## 6.15 omemhz_divider.v

```
// onemhz_divider module
// Michael F. Robbins, mrobbins@mit.edu
//
// Given a 27MHz input clock, outputs high one cycle every 27 cycles.
// Used to trigger modules which should be started at 1MHz.
module onemhz_divider(clk, reset, div);
	input clk;		// assumes a 27MHz input clock
```

```
	input reset;	// resets the counter
	output div;		// the divided trigger

	reg [4:0] counter;

	assign div = (counter == (27-1));

	always @ (posedge clk)
		counter <=  reset ? 0 :
						div ? 0 :
						counter + 1;
endmodule
```

## 6.16 packet_engine.v

```
// sostler
//
// packet_engine (transmit and recieve) modules
//
// Audio words pass straight through.
// However, draw commands (bunches of three words)
// are allowed to bunch up until they're ready to send to the whiteboard.

module packet_engine_transmit(clk, onemhz_enable, audio_word, draw_packet_put, word);
	input clk, onemhz_enable;
	input [8:0] audio_word;
	input [26:0] draw_packet_put;

	output [8:0] word;
	reg [8:0] word;

	reg [6:0] slowcounter; // counts to 127 so we put out a new word

	reg [26:0] draw_packet_int;
	reg [1:0] draw_packet_index;
	reg [8:0] draw_packet_word;
	always @ (draw_packet_index, draw_packet_int)
		case(draw_packet_index)
			0:		draw_packet_word = draw_packet_int[26:18];
			1:		draw_packet_word = draw_packet_int[17:9];
			2:		draw_packet_word = draw_packet_int[8:0];
			default: draw_packet_word = 9'bZ;
		endcase

	always @ (posedge clk) begin
		// catch incoming draw packets
		if(draw_packet_put[26])
			draw_packet_int <= draw_packet_put;

		// deal with words going out
		if(onemhz_enable) begin
			slowcounter <= slowcounter + 1;

			if(slowcounter == 127) begin
				// put new word out
				if(draw_packet_int[26]) begin
					draw_packet_index <= draw_packet_index + 1;
					word <= draw_packet_word;
					if(draw_packet_index == 2) begin
						// clear the old draw packet
						draw_packet_int <= 0;
						draw_packet_index <= 0;
					end
				end
```

```
                                else // just put audio packet out
                                        word <= audio_word;
                        end
                end
        end
endmodule

module packet_engine_receive(clk, goodbins, word, last_audio, draw_packet);
        input clk;
        input goodbins;
        input [8:0] word;
        output [7:0] last_audio;
        output [26:0] draw_packet;

        reg [7:0] last_audio;
        reg [26:0] draw_packet;

        reg [26:0] draw_packet_int;
        reg [1:0] draw_packet_index;
        always @ (posedge clk) begin
                draw_packet <= {27{1'b0}};
                if(goodbins) begin
                        if(draw_packet_index == 0) begin
                                // we're not assembling a draw packet
                                if(word[8] == 1) begin
                                        // start draw packet assembly
                                        draw_packet_int[26:18] <= word;
                                        draw_packet_index <= 1;
                                end
                                else begin
                                        // just a boring old audio word
                                        last_audio <= word[7:0];
                                end
                        end
                        else if(draw_packet_index == 1) begin
                                // we're already assembling a draw packet (middle word)
                                draw_packet_int[17:9] <= word;
                                draw_packet_index <= 2;
                        end
                        else if(draw_packet_index == 2) begin
                                // we're already assembling a draw packet       (last word)
                                draw_packet_int[8:0] <= word;
                                draw_packet_index <= 0;
                                draw_packet <= draw_packet_int;
                        end
                        else // we shouldnt be here
                                draw_packet_index <= 0;

                end
        end
endmodule
```

## 6.17 pointer.v

```
// sostler

// The pointer module is a graphics widget that draws an outlined pointer on the screen.
// It outputs a 1-bit pixel value, as well as an enable value for transprency.

module pointer(x, y, hcount, vcount, pixel, enable);

   parameter WIDTH = 12;
   parameter HEIGHT = 21;

   // [x,y] are the coordinates of the mouse
```

```
// [hcount,ycount] are the coordinates of the pixel being drawn
input [11:0] x;
input [10:0] hcount;
input [11:0] y;
input [9:0]  vcount;

output pixel;           // 1 = white, 0 = black
output enable;          // 1 = draw pixel, 0 = do not draw pixel

wire [11:0] x_rel;
wire [11:0] y_rel;

assign x_rel = (hcount >= x) ? hcount - x : WIDTH;
assign y_rel = (vcount >= y) ? vcount - y : HEIGHT;

reg [11:0] rowmask;
always @ (y_rel)
   case(y_rel)
      00:      rowmask = 12'b100000000000;
      01:      rowmask = 12'b110000000000;
      02:      rowmask = 12'b111000000000;
      03:      rowmask = 12'b111100000000;
      04:      rowmask = 12'b111110000000;
      05:      rowmask = 12'b111111000000;
      06:      rowmask = 12'b111111100000;
      07:      rowmask = 12'b111111110000;
      08:      rowmask = 12'b111111111000;
      09:      rowmask = 12'b111111111100;
      10:      rowmask = 12'b111111111110;
      11:      rowmask = 12'b111111111111;
      12:      rowmask = 12'b111111110000;
      13:      rowmask = 12'b111111110000;
      14:      rowmask = 12'b111001111000;
      15:      rowmask = 12'b110001111000;
      16:      rowmask = 12'b100000111100;
      17:      rowmask = 12'b000000111100;
      18:      rowmask = 12'b000000011110;
      19:      rowmask = 12'b000000011110;
      20:      rowmask = 12'b000000001100;
      default: rowmask = 12'b000000000000;
   endcase

reg [11:0] rowfill;
always @ (y_rel)
   case(y_rel)
      00:      rowfill = 12'b100000000000;
      01:      rowfill = 12'b110000000000;
      02:      rowfill = 12'b101000000000;
      03:      rowfill = 12'b100100000000;
      04:      rowfill = 12'b100010000000;
      05:      rowfill = 12'b100001000000;
      06:      rowfill = 12'b100000100000;
      07:      rowfill = 12'b100000010000;
      08:      rowfill = 12'b100000001000;
      09:      rowfill = 12'b100000000100;
      10:      rowfill = 12'b100000000010;
      11:      rowfill = 12'b100000011111;
      12:      rowfill = 12'b100010010000;
      13:      rowfill = 12'b100110010000;
      14:      rowfill = 12'b101001001000;
      15:      rowfill = 12'b110001001000;
      16:      rowfill = 12'b100000100100;
      17:      rowfill = 12'b000000100100;
      18:      rowfill = 12'b000000010010;
      19:      rowfill = 12'b000000010010;
      20:      rowfill = 12'b000000001100;
      default: rowfill = 12'b000000000000;
```

```
      endcase

   assign enable = (x_rel < WIDTH) ? rowmask[11 - x_rel] : 0;
   assign pixel = ~rowfill[11 - x_rel];

endmodule
```

## 6.18 shifter.v

```
// shifter module
// Michael F. Robbins, mrobbins@mit.edu
//
// Takes a set of bins (32 bits wide) in and generates the
// 64 time-domain samples corresponding to turning those frequency
// components on and off.
// Clocks out samples based on the onemhz_enable.

module shifter(clk, reset, onemhz_enable, bins, sample, newsample);
        input clk, reset, onemhz_enable;
        input [31:0] bins;

        output signed [11:0] sample;
        output newsample;
        reg signed [11:0] sample;
        reg newsample;

        reg [31:0] intbins;     // internal copy of the bins (so they can otherwise be changed)

        wire done;
        wire [11:0] curval;
        reg [5:0] counter;

        // the sinesample does all the heavy lifting...
        sinesample sine(.clk(clk), .start(onemhz_enable), .bins(intbins), .index(counter), .done(done),
.sample(curval));

        always @ (posedge clk) begin
                newsample <= 1'b0;
                if(reset) begin
                        // reset
                        sample <= 0;
                        newsample <= 1'b0;
                        intbins <= 0;
                        counter <= 0;
                end
                else begin
                        // not reset
                        if(onemhz_enable) begin
                                sample <= curval;                               // output this one, and
                                newsample <= 1'b1;                      // start computing the next sample
                                counter <= counter + 1;
                                if(counter == 63)                               // we're done this frame; get
new bins
                                        intbins <= bins;
                        end
                end
        end
endmodule
```

## 6.19 sin64.v

```
// sin64 module
```

```
// Michael F. Robbins, mrobbins@mit.edu
//
// Stores a 12-bit (two's complement) value of a sine lookup table.
// The function is essentially:
//    value = 2047 * sin((pi/32) * addr)
module sin64(addr, value);
        input [5:0] addr;
        output signed [11:0] value;

        reg signed [11:0] posval;
        wire signed [11:0] negval;

        always @ (addr) begin
                case (addr[4:0])
                0:      posval = 0;
                1:      posval = 201;
                2:      posval = 399;
                3:      posval = 594;
                4:      posval = 783;
                5:      posval = 965;
                6:      posval = 1137;
                7:      posval = 1299;
                8:      posval = 1447;
                9:      posval = 1582;
                10:     posval = 1702;
                11:     posval = 1805;
                12:     posval = 1891;
                13:     posval = 1959;
                14:     posval = 2008;
                15:     posval = 2037;
                16:     posval = 2047;
                17:     posval = 2037;
                18:     posval = 2008;
                19:     posval = 1959;
                20:     posval = 1891;
                21:     posval = 1805;
                22:     posval = 1702;
                23:     posval = 1582;
                24:     posval = 1447;
                25:     posval = 1299;
                26:     posval = 1137;
                27:     posval = 965;
                28:     posval = 783;
                29:     posval = 594;
                30:     posval = 399;
                31:     posval = 201;
                endcase
        end

        // Flip sign for indices 32..63 without explicitly storing them in the ROM.
        // (We could do something similar noting the symmetry between codes 1->15 and 31->17,
        // but this works for now.)
        assign negval = -posval;
        assign value = addr[5] ? negval : posval;
endmodule
```

## 6.20 sinesample.v

```
// sinesample module
// Michael F. Robbins, mrobbins@mit.edu
//
// Computes a single sample of the IFFT.
// In 16 cycles, adds up the 32 different possible sine waves which are on/off as
// definted by the bins input.  Does two multiply/accumulates per cycle.
```

```
//
// The bins are done in even/odd pairs.  That is, the first cycle looks at
// frequency bins 0 and 1, the 2nd cycle looks at bins 2 and 3, and so on.
// This could be made more efficient by looking at the symmetry between certain
// frequency bands, but this gets the job done in a way that is easily understood.
//
module sinesample(clk, start, bins, index, done, sample);
  input clk;            // 27MHz clock input
  input start;          // restart calculation signal (synchronous)
  input [31:0] bins;    // FFT^(-1) bins, where bin i corresponds to sin(2*pi/64*index*n)
  input [5:0] index;    // position in the sine wave
  output done;          // signal that sample is valid
  output signed [11:0] sample;  // the 12 high-order bits of the calculation result

  reg done;
  reg signed [16:0] curval;     // current sum
  assign sample = curval[15:4];  // FULL SCALE WOULD BE [16:5], but because
                                                                                    // of our encoder
choice, we can get an extra bit.
                                                                                    //   -- mrobbins, Dec
10 2006
  reg [31:0] intbins;   // internal copy of bins, read in on the start signal
  reg [4:0] binindex1;  // bin index: 0, 2, 4, ... 30
  reg [4:0] binindex2;  // bin index: 1, 3, 5, ... 31

  // instantiate two sine lookup tables
  wire signed [11:0] data1;
  wire signed [11:0] data2;
  reg [10:0] addr1; // can be at most 63*31
  reg [10:0] addr2; // can be at most 63*31
  sin64 table1(.addr(addr1[5:0]), .value(data1));
  sin64 table2(.addr(addr2[5:0]), .value(data2));

  always @ (posedge clk) begin
    if(start) begin
      // start of a new calculation
      done <= 0;
      curval <= 0;
      intbins <= bins;
      binindex1 <= 0;
      binindex2 <= 1;
      addr1 <= 0;
      addr2 <= index;
    end
    else if(done) begin
      // done; don't touch things
    end
    else begin
      // running the calculation
      binindex1 <= binindex1 + 2;       // table1 will do bins 0, 2, 4, 6... 30
      binindex2 <= binindex2 + 2;       // table2 will do bins 1, 3, 5, 7... 31

      addr1 <= addr1 + 2*index;
      addr2 <= addr2 + 2*index;         // the difference is in the starting addr2...

               // both frequency components are always looked up from the tables,
               // but depending on the bins, 0, 1, or two of them are added to the accumulator.
      curval <= curval + ((intbins[binindex1] && intbins[binindex2]) ? data1 + data2 :
                         (intbins[binindex1]) ? data1 :
                         (intbins[binindex2]) ? data2 : 0);

      if(binindex1 == 30) begin         // again, we end on bin 30 for table1.  see above.
        // the last step; signal done
        done <= 1'b1;
      end
    end
  end
```

```
endmodule
```

## 6.21 synchronize.v

```
// pulse synchronizer
module synchronize(clk,in,out);
  parameter NSYNC = 2;  // number of sync flops.  must be >= 2
  input clk;
  input in;
  output out;

  reg [NSYNC-2:0] sync;
  reg out;

  always @ (posedge clk)
  begin
    {out,sync} <= {sync[NSYNC-2:0],in};
  end
endmodule
```

## 6.22 vram_display.v

```
// sostler

// changes made: added byte indexing

///////////////////////////////////////////////////////////////////////////////
// generate display pixels from reading the ZBT ram
// note that the ZBT ram has 2 cycles of read (and write) latency
//
// We take care of that by latching the data at an appropriate time.
//
// Note that the ZBT stores 36 bits per word; we use only 32 bits here,
// decoded into four bytes of pixel data.

module vram_display(reset,clk,
                    hcount,vcount,vr_pixel,
                              vram_addr,
                    vram_read_data);

   input reset, clk;

   input  [10:0]  hcount;
   input  [9:0]   vcount;

   output [7:0]   vr_pixel;

   output [18:0]  vram_addr;
   input  [35:0]  vram_read_data;

   wire [18:0]  vram_addr = {1'b0, vcount, hcount[9:2]};

   wire [1:0]      pixel_index = hcount[1:0]; // block 0 is pixel 0

   reg [7:0]       vr_pixel;
   reg [35:0]      vr_data_latched;
   reg [35:0]      last_vr_data;

   always @(posedge clk)
     last_vr_data <= (pixel_index==2'd3) ? vr_data_latched : last_vr_data;
```

```
   always @(posedge clk)
     vr_data_latched <= (pixel_index==2'd1) ? vram_read_data : vr_data_latched;

   // pixel alignment:
   // 0--------  9-------- 18-------- 27--------
   // 01234567x  01234567x  01234567x  01234567x

   always @(last_vr_data or pixel_index)
      case(pixel_index)
         0: vr_pixel = last_vr_data[7:0];
         1: vr_pixel = last_vr_data[16:9];
         2: vr_pixel = last_vr_data[25:18];
         3: vr_pixel = last_vr_data[34:27];
         default: vr_pixel = 8'bxxxxxxxx;
      endcase

endmodule // vram_display
```

## 6.23 whiteboard.v

```
// sostler

module whiteboard(video_clock, reset, is_loopback, draw_local_pixel,
                  button_left, button_right, button_up, button_down,
                  button_enter, button_clear,
                  clear_screen, local_pixel, remote_pixel,
                  local_x, local_y,
                                          remote_x, remote_y,
                                          draw_command_in, draw_command_out);

   parameter RIGHT_BOUND = 639;
   parameter BOTTOM_BOUND = 479;

   input video_clock, reset, is_loopback, draw_local_pixel;
        input button_left, button_right, button_up, button_down, button_enter, button_clear;

   output clear_screen; // the screen should be cleared
   output local_pixel;  // if a local pixel should be drawn (only high @ draw_local_pixel)
        output remote_pixel; // if a remote pixel should be drawn (only high @ draw_local_pixel + 1 cycle)

   output [11:0] local_x, local_y;   // pointer coordinates
        output [11:0] remote_x, remote_y; // remote draw coordinates

        input  [26:0] draw_command_in;
        output [26:0] draw_command_out;

        wire [11:0] clear_x = {12'b001_111_111_111};
        wire [11:0] clear_y = {12'b000_111_111_111};
        wire [12:0]     clear_checksum = clear_x + clear_y;

   reg clear_screen;                                          // if we are currently clearing the screen
        reg clear_screen_request;               // if we have a recieved a clear screen request

        wire local_clear = (!is_loopback && button_clear); // if in loopback, ignore local clear requests


        wire local_draw  = button_enter && draw_local_pixel;

        reg [11:0] local_x, local_y;            // coordinates of local pointer

        // store remote draw requests until draw_remote_pixel
        reg       draw_request;                         // if we have recieved a remote draw command
        reg [11:0] remote_x, remote_y;  // coordinates of remote draw command
        reg               draw_remote_pixel;    // is high one clock after draw_local_pixel
```

```
	// INCOMING PACKET HANDLING

	// break incoming packets into components
	wire [11:0] request_x		= {2'b0, draw_command_in[18:9]};
	wire [11:0] request_y		= {3'b0, draw_command_in[8:0]};

	// packet checksum calculation
	wire [6:0]  request_checksum = draw_command_in[25:19];
	wire [12:0] request_sum_tmp  = request_x + request_y;
	wire        is_good_checksum = (request_checksum == request_sum_tmp[6:0]);

	// determine request type (draw request, clear request, or garbage)
	wire		remote_request = draw_command_in[26]; // if msb of request is 1
	wire			clear_request = (request_x == clear_x && request_y == clear_y); // special
coordinates indicate clear request
	wire			remote_clear = (clear_request && is_good_checksum);


	// OUTGOING PACKET HANDLING
	wire [12:0] local_checksum = local_x + local_y;
	assign draw_command_out = button_clear ? {draw_local_pixel, clear_checksum[6:0], clear_x[9:0],
clear_y[8:0]} :

{local_draw, local_checksum[6:0], local_x[9:0], local_y[8:0]};

	// PIXEL DRAWING

   assign local_pixel = !is_loopback && local_draw;
	assign remote_pixel = draw_remote_pixel && draw_request;



   always @(posedge video_clock) begin

		draw_remote_pixel <= draw_local_pixel;

		if (remote_request && !clear_request && is_good_checksum) begin
			draw_request <= 1;
			remote_x <= request_x;
			remote_y  <= request_y;
		end

      if (reset) begin
         local_x <= 0;
         local_y <= 0;
         clear_screen_request <= 1;
         clear_screen <= 1;
			draw_request <= 0;
      end

      else if (draw_local_pixel) begin

			// at beginning of frame, process clear_screen requests
			clear_screen_request <= 0;
         clear_screen <= (clear_screen_request | local_clear | remote_clear);

			// POINTER MOVEMENT

         if (button_left && !button_right && local_x > 0) begin
            local_x <= local_x - 1;
         end

         else if (button_right && !button_left && local_x < RIGHT_BOUND) begin
            local_x <= local_x + 1;
         end
```

```
        if (button_up && !button_down && local_y > 0) begin
           local_y <= local_y - 1;
        end

        else if (button_down && !button_up && local_y < BOTTOM_BOUND) begin
           local_y <= local_y + 1;
        end

     end

     else if (local_clear || remote_clear) begin
        clear_screen_request <= 1;
     end

		else if (draw_remote_pixel) begin
			draw_request <= 0; // clear draw request once we draw it
		end

  end

endmodule
```

## 6.24 xvga.v

```
// sostler: parameterized timing values

////////////////////////////////////////////////////////////////////////////////
// xvga: Generate XVGA display signals

module xvga(vclock,hcount,vcount,hsync,vsync,blank);

   // TIMING PARAMETERS

   parameter h_active = 640;
   parameter h_active_z = h_active - 1; // zero index for beauty tee hee
   parameter h_front_porch = 16;
   parameter h_sync_pulse = 96;
   parameter h_back_porch = 48;

   parameter v_active = 480;
   parameter v_active_z = v_active - 1; // ibid
   parameter v_front_porch = 11;
   parameter v_sync_pulse = 2;
   parameter v_back_porch = 31;

   input        vclock;
   output [10:0] hcount;
   output [9:0]  vcount;
   output           vsync;
   output           hsync;
   output           blank;

   reg      hsync,vsync,hblank,vblank,blank;
   reg [10:0] hcount;    // pixel number on current line
   reg [9:0]  vcount;    // line number

   wire      hsyncon,hsyncoff,hreset,hblankon;

   assign hblankon = (hcount == h_active_z);
   assign hsyncon  = (hcount == h_active_z + h_front_porch);
   assign hsyncoff = (hcount == h_active_z + h_front_porch + h_sync_pulse);
   assign hreset   = (hcount == h_active_z + h_front_porch + h_sync_pulse + h_back_porch);
```

```
   wire   vsyncon,vsyncoff,vreset,vblankon;

   assign vblankon = hreset & (vcount == v_active_z);
   assign vsyncon =  hreset & (vcount == v_active_z + v_front_porch);
   assign vsyncoff = hreset & (vcount == v_active_z + v_front_porch + v_sync_pulse);
   assign vreset =   hreset & (vcount == v_active_z + v_front_porch + v_sync_pulse + v_back_porch);

   // 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
```

## 6.25 zbt_6111.v

```
//
// 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, addr, write_byte,
                write_data, read_data,
                          ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b, ram_bwe_b);

   input         clk;                   // system clock
   input         cen;                   // clock enable for gating ZBT cycles
   input         we;                       // write enable (active HIGH)
   input  [18:0] addr;             // memory address
   input  [1:0]  write_byte;  // which byte to write to
   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 [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
   output [3:0]  ram_bwe_b;   // physical line to ram byte write enable
```

```
   reg    [3:0]  ram_bwe_b;

   // represent pixel 0 in byte position 0
   always @(write_byte)
      case(write_byte)
               3: ram_bwe_b = ~4'b1000;
               2: ram_bwe_b = ~4'b0100;
               1: ram_bwe_b = ~4'b0010;
               0: ram_bwe_b = ~4'b0001;
      endcase

   // 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;

endmodule // zbt_6111
```