Voice Over SLIP

Submitted by:	Brian Schmidt and Matthew Welsh
Date of Submission:	December 14, 2005
Course Name:	6.111 Introductory Digital Systems Laboratory

Abstract:

Inspired by the recent surge in VoIP popularity, we design and implement a phone system that is capable of packeting voice data using the Internet Protocol (IP) and transmitting it over a Serial Line Internet Protocol (SLIP) connection. A simple audio codec utilizing downsampling and upsampling provides audio data at a rate comparable to the maximum transmission rate possible over the serial connection.

Contents

List of Figures

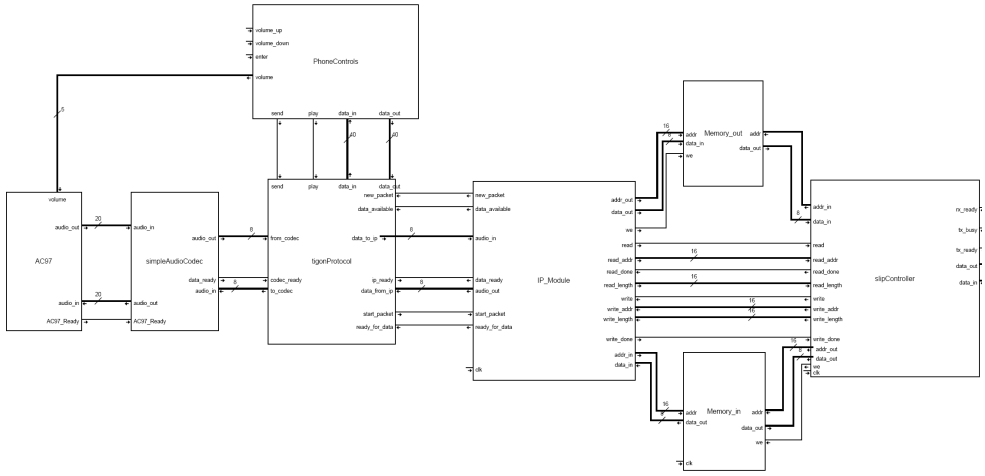

Figure 1: Block Diagram Layout of all Voice Over SLIP modules

1 Overview

The goal of our project was to implement a phone system capable of sending voice and connection data over a serial line internet protocol (SLIP) connection using internet protocol (IP) packets. We achieve this using several modules including an IP Encoder module, an IP Decoder module, a SLIP send module, a SLIP receive module, a phone controls module, and a Tigon protocol module. The Tigon protocol is a propriety protocol which we develop specifically for this project to send connection data along with voice data in IP packets. These modules are described in the Module Descriptions section of this report, and Figure 1 shows a graphical overview of how these various module interact.

Voice data from the AC97 built into the labkit is passed through the simple audio codec module which downsamples the data to 8-bits sampled at 8kHz. The audio data is then passed to the Tigon Protocol Encoder module which adds in 40 bits of data provided by the phone controls module. The Tigon Protocol Encoder passes the data to the IP encoder module which adds the appropriate header information for the data to be sent over a network. When enough data has been collected, the IP encoder module notifies the SLIP send module,

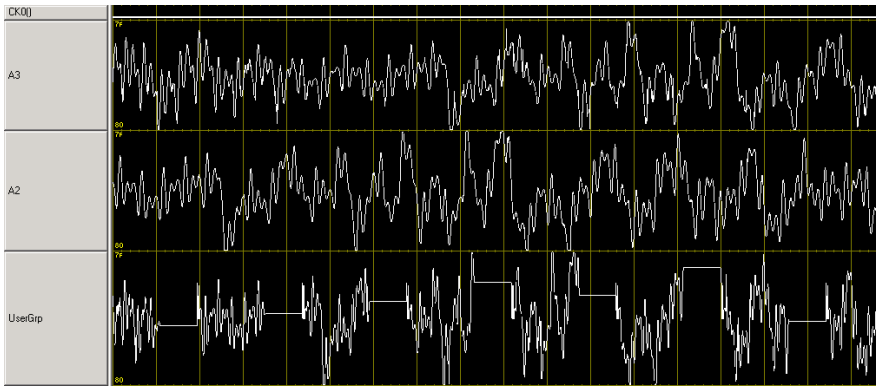

Figure 2: The middle waveform shows the audio data from the AC97, the bottom waveform shows the data compressed into IP packets, and the top waveform shows the audio data send to the AC97 on the receiving labkit.

which then transmits the data byte-by-byte over a serial line to another awaiting labkit.

Once the data packets are received and process by the SLIP receive module on the companion labkit, the IP decoder module is notified, and it reads the packet data to the Tigon Protocol decoder module. The Tigon Decoder module feeds the first 40 bits of each packet to the phone controls module, which tells the Tigon decoder whether to send the audio data along to the simple audio codec module or simply ignore it. If passed to the simple audio codec, the audio data is then sent through the AC97 chip and played through headphones attached to the labkit. Figure 2 shows the audio data received from the AC97, the packetized audio data, and the resulting audio data output to the AC97.

2 Module Descriptions

2.1 Internet Protocol (IP) Encoder

The IP Encoder module is responsible for taking data from the audio codec module and repackaging in into a packet suitable for transmission over the internet. While our phone system does not communicate with any device connected to the internet, out intent was to

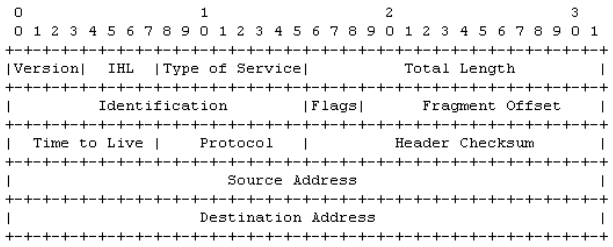

Figure 3: A typical IP packet generated by the IP Encoder Module

allow for this addition if time permitted.

IP packets provide data essential to routers sending data over the internet. This data includes source address, destination address, packet length, in addition to other fields which specify how the packet data is to be treated.

Figure 3, adapted from a similar table provided in RFC 791, shows the contents of a typical packet generated by the IP Encoder Module.

The IP Encoder module communicates with the SLIP Sender module by first writing packet data to a dual-port BRAM memory and then notifying the SLIP module of the start address and length of the written data. This permits the IP Encoder module to send arbitrarily large packets over the SLIP connection, which is necessary for sending any meaningful amount of data.

To allow for fine-tuned control of what data is sent in each packet, the IP Encoder module has a `start_packet` input that allows other modules to specify when the IP Encoder module should begin constructing a new packet. This permits the layering of protocols (such as TCP and UDP) which would be expected if the IP Encoder module were to be used in other applications. For our phone system, we layer a propriety protocol, which we name “Tigon”, that allows the phone controls module to send additional control instructions along with data packets. In our implementation, these additional controls range from “disconnect” to coordinates of the ball in a two-player pong game.

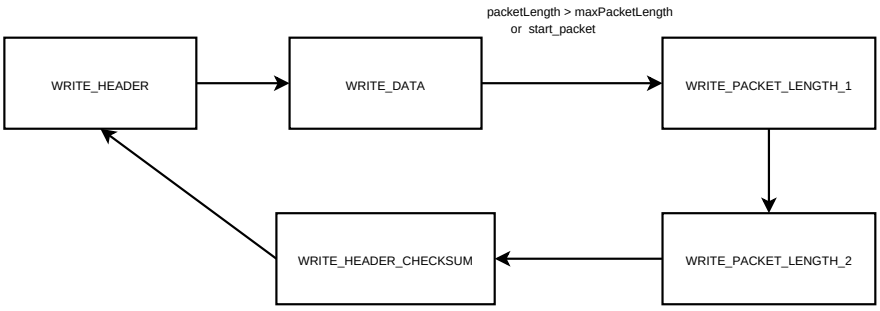

Figure 4: Major FSM of IP Encoder Module

In addition, the IP Encoder module provides a `ready_for_data` output which is high when the module is capable of accepting data to transmit. This signal goes low only during the time it takes the module to write the IP header for a new packet to memory.

The functionality of the IP Encoder module is implemented using two FSMs. The module's primary FSM, shown in Figure 4, transitions through four separate states. The FSM begins by writing a the data that comprises the IP header to memory. It then transitions into the WRITE_DATA state where it receives data from some external module and uses the data to fill the data of the IP packet. Once the IP Encoder module has either filled the packet or received instructions to begin a new packet, it transitions into `WRITE_PACKET_LENGTH_1` and then `WRITE_PACKET_LENGTH_2` where it writes the upper and lower 8 bits of the packet length, respectively, to the appropriate location in the header. Once this information is written, the module computes the header checksum as specified by the IP specifications and writes it to the appropriate location in the IP header. The process then repeats.

2.2 IP Decoder

The IP Encoder receives data from the SLIP receiver module via an intermediate dual-port memory to which both the SLIP receiver module and the IP Decoder module have access. In a manner similar to the sending process of the IP Encoder module, the SLIP Receiver

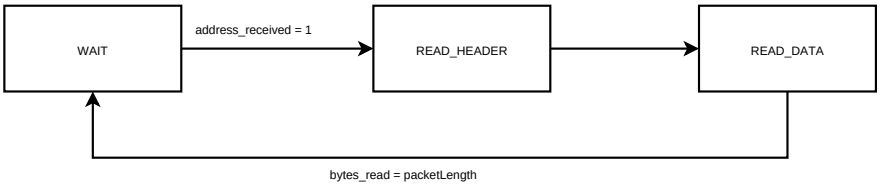

Figure 5: The major FSM of the IP Decoder module

indicates to the IP Decoder when it has a new packet and provides the packet's address in memory as well as its length. Using this provided information, the IP decoder module first reads important information from the IP packet header and then proceeds to read the data contained in the packet. In our implementation, this packet data is passes along to the Tigon protocol module at a rate set by the Tigon module using the `ip_ready` signal. Whenever the signal goes high, the IP decoder passes along another byte of data from the current IP packet.

Figure 5 shows a graphical representation of the IP decoder's main FSM.

Because it is possible for IP packets to arrive while the IP decoder module is still reading out data to the Tigon protocol decoder module, the IP Decoder module will buffer one packet if it is received while the IP Decoder is still reading packet data to the Tigon decoder module. Buffering more than one packet for this application is not practical because it would introduce a lag in the audio data being played. One the information for the most recent packet is buffered to be read after reading the current packet has completed. All other packets are simply dropped.

2.3 Tigon Protocol Encoder

The Tigon Protocol Encoder implements a proprietary protocol we developed to transfer addition call information along with audio data in the packets we transmit. While this protocol was simple in design, it proved the ability of the system stack protocols on top

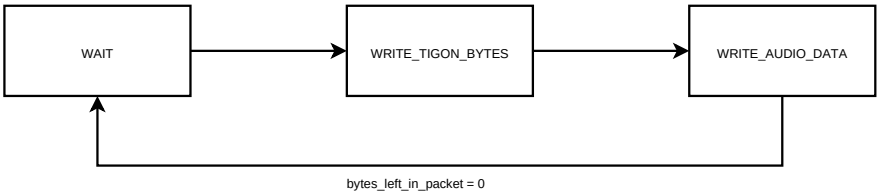

Figure 6: Major FSM of the Tigon Protocol Encoder module

of the internet protocol. In real world applications, IP packets are rarely sent without encapsulating another protocol. Common protocols sent using IP packets include UDP and TCP, which allow packets to be sent to specific ports on the destination computer or guarantee transmission of packets (both features the standard internet protocol fails to implement).

For the Tigon protocol we simply allocate 40 bits of data to the phone controls module so that it may send connection information over the SLIP connection. These 40 bits are written to the first 40 bits of the data field of each transmitted IP packet. The receiving side, the Tigon Protocol decoder takes this information and provides it to the phone control module.

The major FSM used to implement the Tigon Protocol Encoder, shown in Figure 6 module has three states. The module begins in a `WAIT` state until it receives a `send` signal from the phone control module. The Tigon encoder then sends the 40 data bits provided by the phone control module to the IP encoder module before sending audio data provided by the simple audio codec module to the IP encoder module.

2.4 Tigon Protocol Decoder

The Tigon Protocol Decoder module is used to decode data encoded using the Tigon Protocol Encoder module. The decoder module accepts data from the IP Decoder module and feed the first 40 bits of each packet to the phone controls module. The rest of the data is passed

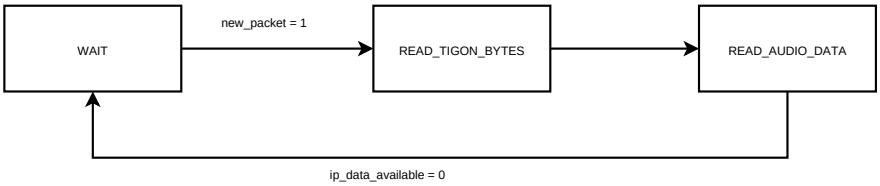

Figure 7: Tigon Protocol Decoder module FSM

along to the simple audio codec module to be played using the AC97 chip. Audio data is passed along only if the `play` signal provided by the phone controls module is high, but always provides 40 bits of audio data to the phone controls so that connect signals may be received from other phones.

The Tigon Protocol decoder has one major FSM, which represented graphically in Figure 7. The FSM begins in the `WAIT` state where it waits for the IP decoder module to indicate that it has a new packet available. The module then moves into the `READ_TIGON_BYTES` state where it reads the 5 bytes of data reserved for the protocol and provides them to the phone controls module. Finally, the FSM transitions into the `READ_AUDIO_DATA` state where it reads data to the simple audio codec module as long as the `play` signal from the phone controls module is high and the IP decoder module still has data available to be read.

2.5 SLIP Subsystem

The SLIP Subsystem is responsible for encapsulating IP packets using the SLIP protocol and sending them over the serial port. It is also responsible for de-encapsulating SLIP packets received over the serial port. The SLIP Subsystem is also responsible for serial transmission. The SLIP Subsystem takes as input IP packets from the IP Encoder and sends them over the serial line using the SLIP protocol. It also receives SLIP Encapsulated packets over the serial line and outputs the encased IP packets to the IP Decoder.

SLIP (RFC 1055) is an encapsulation protocol commonly used for point-to-point serial

connections using TCP/IP. The SLIP protocol defines two special characters: END and ESC. END is octal 300 (decimal 192) and ESC is octal 333 (decimal 219) not to be confused with the ASCII Escape character; for the purposes of this discussion, ESC will indicate the SLIP ESC character. To send a packet, a SLIP host simply starts sending the data in the packet. If a data byte is the same code as END character, a two byte sequence of ESC and octal 334 (decimal 220) is sent instead. If it the same as an ESC character, a two byte sequence of ESC and octal 335 (decimal 221) is sent instead. When the last byte in the packet has been sent, an END character is then transmitted. A simple addition that was implemented was to start as well as end packets with END. This flushes erroneous bytes caused by line noise.

The SLIP Subsystem uses many different modules. The modules will first be described separately then the interaction of them will be explained.

2.6 RS232

The RS232 module is responsible for the physical transmission of serial data over the rs232 port of the lab kit. The setup of this module is to send serial data at 115200 bits/sec using one start bit, two stop bits, and no parity. The RS232 module contains a few important modules:

2.6.1 Baud Gen

This module is responsible for creating a serial clock based on the specified baud rate. This clock signal is used in the transmitter module. It also creates a clock which is eight times as fast as the serial clock. This clock is used in the receive module.

2.6.2 Async Transmitter

This module is responsible for transmitting the actual serial data. It takes as input a byte of data (tx_data) and sends the eight bits of the data serially (one bit at a time) through the output txd at the rising edge of the serial clock (from the Baud Gen module). It sends a byte when the input tx_start is high. It also sets tx_busy high when it is sending. This module is clocked at 27MHz.

2.6.3 Async Receiver

This module is responsible for receiving the actual serial data. It takes as input the serial line rxd and a clock eight times as fast as the serial clock. The clock is eight times as fast so once rxd goes high (the start bit is received), the module has a few clock cycles to prepare to receive data. The module receives the data eight bits at a time and outputs the byte received to rx_data. It sets data_ready high for a clock cycle when it does this. This module is clocked at 27MHz.

2.7 SLIP Controller

The SLIP Controller contains a send and receive module which are responsible for encapsulating and de-encapsulating packet data for transmission over the serial line. The two SLIP Controller modules use shared BRAMs to both read IP packets from the IP Encoder and to store received IP packets for reading by the IP Decoder.

2.7.1 SlipSend

This module is responsible for reading IP packets from RAM and encapsulating them with the SLIP protocol for transmission over the serial line. When the input read goes high SlipSend will begin the process of sending a packet. It will start at the first location of the

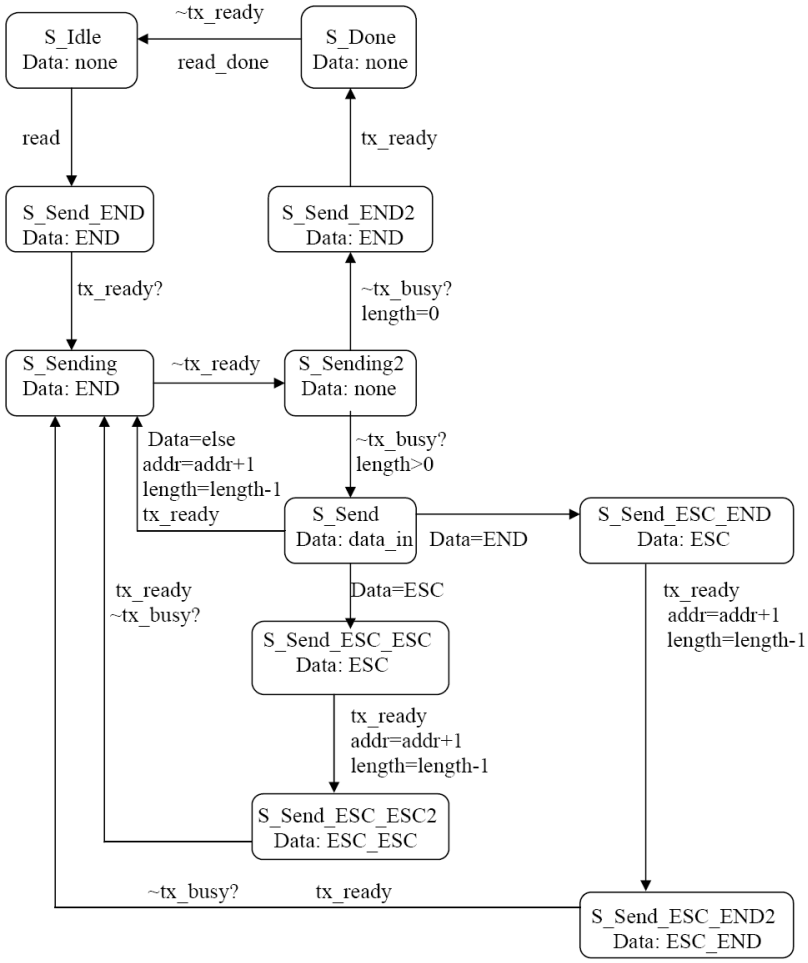

Figure 8: Graphical representation of the SlipSend module FSM

packet in RAM (read_addr) and send every byte in RAM until it reaches the length of the packet (read_length). The sending of each byte uses a finite state machine that implements the SLIP protocol described above. In summary: END characters are sent at the beginning and end of every packet. Also, if the special characters ESC or END are encountered in a packet, they are changed to ESC followed by ESC_ESC and ESC followed by ESC_END respectively. See Figure 8 for the FSM. This module is clocked at 27MHz.

2.7.2 SlipRec

This module is responsible for receiving SLIP encapsulations over the serial line and writing the enclosed IP packet to the RAM to be read by the IP Decoder. When rx_ready is high, a byte of data is ready to be read. A finite state machine is used to implement the SLIP protocol. In summary: After SlipRec receives an END character it knows it is starting a new packet. It will store each byte of data in the RAM until it receives another END character. Also, if it receives an ESC character it will replace it and the next byte with the equivalent character according to the specification described above. Once an END character is received, SlipRec will inform the IP Decoder of the first address of the packet and its length and set write high for one clock cycle. See Figure 9 for the FSM. This module is clocked at 27MHz.

3 Testing and Debugging

3.1 IP Encoder

While we initially planned to verify the correctness of the IP packets generated by the IP Encoder module by parsing packets sent to a computer, this task proved to be more complicated than we expected. Establishing a network connection in Windows (even a simple SLIP connection) requires authentication and handshaking protocols that we were not prepared to implement. Instead we compared the data sent by the encoder (viewed using both a logic analyzer and on a computer using a serial data logger) to packets captured using a standard packet sniffer. Although we could not replicate packets generated by the IP Encoder module on the computer exactly, we were able to compare packets generate by the IP Encoder module with those generated by a compute and verify that they were in the same format.

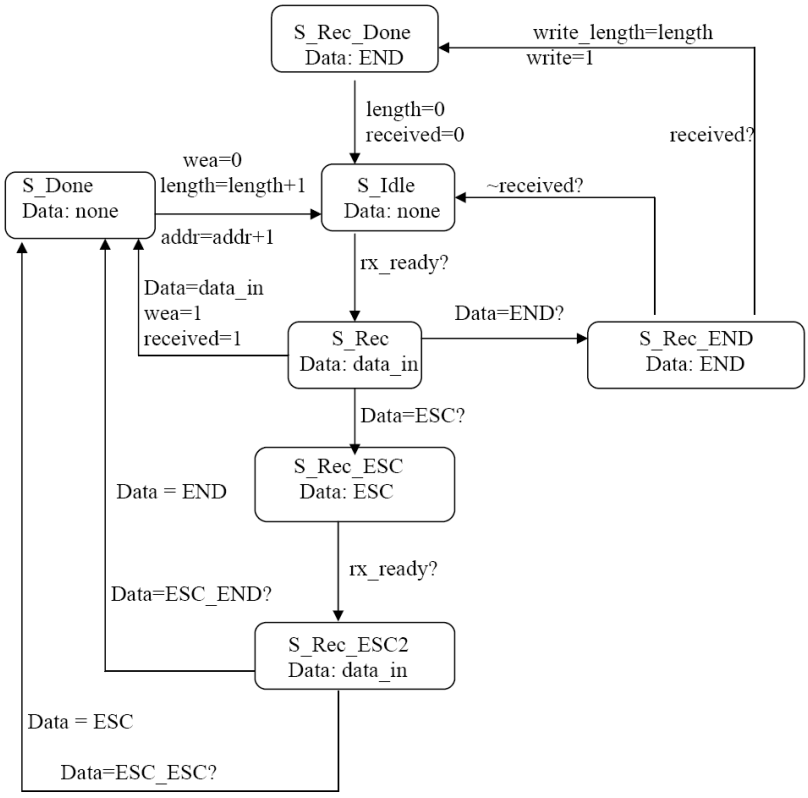

Figure 9: Graphical representation of the SlipRec module FSM

3.2 IP Decoder

Once we verified that the IP Encoder was functioning properly, we only need to see if our IP Decoder module correctly decoded packets received from the IP Decoder module. Because the SLIP send and receive module were working, we sent data from the labkit to itself using a loopback connection and compared the data sent with the data received. Since the data was being sent and received from the same labkit, the received and sent data were synchronized to the same clock, making comparison on the logic analyzer easy. This comparison method helped us work out many initial bugs in the IP Decoder, including a bug that allowed IP header information to pass through the decoder as audio data.

3.3 Tigon Protocol Encoder/Decoder

The Tigon Protocol Encoder and Decoder modules were tested and debugged in a manner similar to that of the IP encoder and decoder modules, respectively. Output from the Tigon encoder was viewed using the logic analyzer, and it was compared to the output of the Tigon decoder. The debugging was complicated slightly by timing issues with the handshaking signals used to communicate with the IP encoder and decoder modules.

3.4 RS232

In the early stages of development, the RS232 module was tested to ensure it was working properly. This was done using the keyboard module supplied on the website. Using the keyboard as an input device, many different values of data were able to be sent across the serial line. To test this data the serial line was attached to the PC in the lab and HyperTerminal was used to both send data to the lab kit and to receive sent data. This was a very easy way to test the validity of the data being both sent and received by the lab kit.

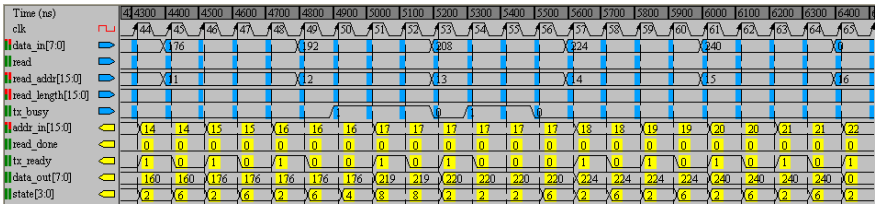

Figure 10: Testbench of SLIP send module

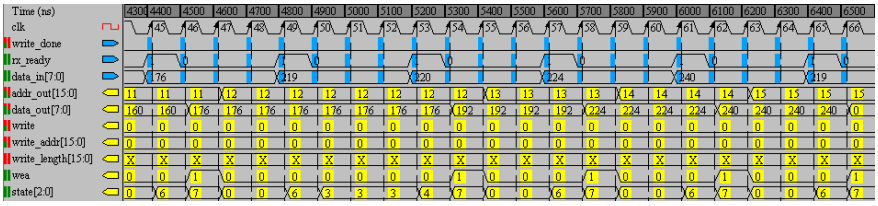

Figure 11: Testbench of SLIP receive module

3.5 SLIP Controller

The SLIP Controller required more means of testing than the RS232 alone. Since packets of data were being sent, these modules were easiest to test in simulation. By using patterned test benches as shown in Figure 10 and Figure 11 it was easy to test the validity of a packet being sent and received. Testing the SLIP Controller on the lab kit was not difficult at all. To test the SlipSend module, a Windows program called Serial Data Logger was used to record data which could then be reviewed later. A test module that supplied the SlipSend module with a counter was used for a packet source. This way, just by validating incrementing values the SlipSend module could be tested. After the SlipSend module was fully tested, it could then be used to test the SlipRec module. This was done first by using a loop back connector. This in effect just returns the data sent on the serial line from the transmit line back to the receive line. Using the loop back connector it was easy to check the validity of the received data just by viewing it on the logic analyzer.

3.6 PhoneControls

The PhoneControls module was implemented as an extra after the tigon Protocol was done. PhoneControls adds the ability to send other types of data besides voice through the SLIP connection. One feature of PhoneControls is the volume control feature. By hooking up directly to the AC97 module the module can change the headphone volume in real time. It takes as input the left and right buttons of the labkit and maps them to increases and decreases in volume. The other feature of the PhoneControls module is its ability to send data over the SLIP connection. It is through the PhoneControls module that the PongGame module is able to send and receive the data it needs on either end. Phone controls has inputs and outputs for the ball position and the paddle position which go to the PongGame. The PongGame inputs and outputs are mapped straight to bits that are communicated back and forth in every packet.

3.7 PongGame

The pong game module was implemented as an extra after the tigon Protocol was done. PongGame adds 28 bits of data to each packet which contains the x-y position of the ball and the position of the user's paddle. The module uses one more bit to chose who is the host of the game. The first player to start the game becomes the host. The host of the game does all of the calculations and sends the correct positions to the guest. Both players display the other players paddle from the tigon data. The guest also displays the ball from the tigon data. The host can also change the speed of the ball and restart the game.

3.8 SimpleAudioCodec

The SimpleAudioCodec module is responsible for supplying the tigonEncoder with down-sampled audio data. SimpleAudioCodec down-samples 20 bit PCM to 8 bit PCM data by

grabbing the high order eight bits. It also down-samples from 48KHz to 6KHz by taking every eighth sample of PCM data from the AC97. These values were chosen so that the data could be sent fast enough to fall under the 112500 Kbit/s baud rate used. It is also responsible for receiving the received audio data from the tigonDecoder and interpolating it. The data is interpolated to reduce the high frequency noise caused by down-sampling.

3.9 Entire System

The entire system was tested mainly using the loop back connector on the serial port. This way, the logic analyzer could be connected to the lab kit and both transmit and receive data could be viewed at the same time. In this way, many of the problems of integrations of the subsystems were debugged. This method of debugging was in fact necessary due to limitations in the number of available labkits, especially labkits located within our cable reach of each other.

Once we had our system working on a single labkit, we placed the code on two separate labkits and began implementing features that required the use of two labkits such as connection handshaking. Debugging our phone is this manner was rather difficult as we had to load each newly compiled bit file onto both systems.

It was also difficult of both lab partners to work on files simultaneously, and we frequently ran into problems with synchronizing verilog files. This issue may have been avoided through the use of a center CVS repository, and we would certainly recommend this to anyone taking on a project of this magnitude in the future.

3.10 Conclusion

In conclusion, this project was a success. Users are able to talk to each other over labkits and play two-player pong at the same time. They can even adjust the volume of their headphones.

Many grueling hours were spent in lab toiling over the many problems encountered during the implementation of this project. In the end it was worth it because the product works and is fun to play with. The original proposal of this project planned for an ethernet implementation of the phone. While the project ended up being implemented over SLIP, we still would have liked to have our phone work over a LAN. If we could do the project again we would probably would have used ethernet instead of SLIP. Using SLIP, however, did allow us to implement some extras. Because using SLIP simplified some of the problems associated with a LAN we were able to implement more advanced phone controls. We were even able to create our own proprietary protocol on top of IP which allowed us to implement two-player pong. In addition, the proprietary protocol implemented on top of IP allowed for the implementation of not only communication of voice, but also of video data and control data. While we view our project as a success, there is one thing we think we missed out on. That is to actually send out IP packets out on a LAN and see if they are routed correctly. While the packets were tested for correctness, it would have been interesting to see how a real network would have affected them. We learned a great deal about digital design, complexity, modularity, and integration from doing this project. It was a great experience and we would recommend the class to everyone.

Appendices

A debounce.v

```
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
```

B display_string.v

```
////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- 16 characer ASCII string display
//
//
// File:   display_string.v
```

```
// Date:   24-Sep-05
// Author: I. Chuang <ichuang@mit.edu>
//
// Based on Nathan Ickes' hex display code
//
// This module drives the labkit hex displays and shows the value of
// 8 ascii bytes as characters on the displays.
//
// Uses the Jae's ascii2dots module
//
// Inputs:
//
//   reset       - active high
//   clock_27mhz - the synchronous clock
//   string_data - 128 bits; each 8 bits gives an ASCII coded character
//
// Outputs:
//
//    disp_*     - display lines used in the 6.111 labkit (rev 003 & 004)
//
//////////////////////////////////////////////////////////////////////////////

module display_string (reset, clock_27mhz, string_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 [16*8-1:0] string_data; // 8 ascii bytes 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
```

```
   wire [39:0] dots; // dots for a single digit
   reg [39:0] rdots; // pipelined dots
   reg [7:0] ascii; // ascii value 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
      state <= state+1;
   end

 8'h04:
   begin
      // Setup the control register
      disp_rs <= 1'b1; // Select 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;
      char_index <= 15; // set this up early for pipeline
   end

 8'h05:
   begin
      // Latch the control register data / dot data
      disp_ce_b <= 1'b1;
      dot_index <= 39; // init for single char
      rdots <= dots; // store dots of char 15
      char_index <= 14; // ready for next char
      state <= state+1;
   end

 8'h06:
   begin
      // Load the user's dot data into the dot reg, char by char
      disp_rs <= 1'b0;   // Select the dot register
      disp_ce_b <= 1'b0;
      disp_data_out <= rdots[dot_index]; // dot data from msb
      if (dot_index == 0)
        if (char_index == 15)
```

```
          state <= 5; // all done, latch data
else
begin
  char_index <= char_index - 1; // goto next char
  dot_index <= 39;
  rdots <= dots; // latch in next char dots
end
      else
dot_index <= dot_index-1; // else loop thru all dots
   end

       endcase

   // combinatorial logic to generate dots for current character
   // this mux, and the ascii table lookup, are slow, so note that
   // this is pipelined by one display clock stage in the always
   // loop above.

   always @(string_data or char_index)
     case (char_index)
       4'h0: ascii = string_data[7:0];
       4'h1: ascii = string_data[7+1*8:1*8];
       4'h2: ascii = string_data[7+2*8:2*8];
       4'h3: ascii = string_data[7+3*8:3*8];
       4'h4: ascii = string_data[7+4*8:4*8];
       4'h5: ascii = string_data[7+5*8:5*8];
       4'h6: ascii = string_data[7+6*8:6*8];
       4'h7: ascii = string_data[7+7*8:7*8];
       4'h8: ascii = string_data[7+8*8:8*8];
       4'h9: ascii = string_data[7+9*8:9*8];
       4'hA: ascii = string_data[7+10*8:10*8];
       4'hB: ascii = string_data[7+11*8:11*8];
       4'hC: ascii = string_data[7+12*8:12*8];
       4'hD: ascii = string_data[7+13*8:13*8];
       4'hE: ascii = string_data[7+14*8:14*8];
       4'hF: ascii = string_data[7+15*8:15*8];
     endcase

   ascii2dots a2d(ascii,dots);

endmodule
```

```
////////////////////////////////////////////////////////////////////////
// Display font dots generation from ASCII code

module ascii2dots(ascii_in,char_dots);

input [7:0] ascii_in;
output [39:0] char_dots;

  ////////////////////////////////////////////////////////////////////
  // ROM: ASCII-->DOTS conversion
  ////////////////////////////////////////////////////////////////////
reg [39:0] char_dots;

always @(ascii_in)
 case(ascii_in)
8'h20: char_dots = 40'b00000000_00000000_00000000_00000000_00000000; //  32 ' '
8'h21: char_dots = 40'b00000000_00000000_00101111_00000000_00000000; //  33  !
8'h22: char_dots = 40'b00000000_00000111_00000000_00000111_00000000; //  34 "
8'h23: char_dots = 40'b11111111_00000000_00000000_00000000_00000000; //  #  1
8'h24: char_dots = 40'b11111111_11111111_00000000_00000000_00000000; //  $  2
8'h25: char_dots = 40'b11111111_11111111_11111111_00000000_00000000; //  %  3
8'h26: char_dots = 40'b11111111_11111111_11111111_11111111_00000000; //  &  4
8'h27: char_dots = 40'b11111111_11111111_11111111_11111111_11111111; //  ' 5
8'h28: char_dots = 40'b00000000_00011110_00100001_00000000_00000000;//  40  (
8'h29: char_dots = 40'b00000000_00100001_00011110_00000000_00000000; //  41  )
8'h2A: char_dots = 40'b00000000_00101010_00011100_00101010_00000000; //  42  *
8'h2B: char_dots = 40'b00001000_00001000_00111110_00001000_00001000; //  43   +
8'h2C: char_dots = 40'b00000000_01000000_00110000_00010000_00000000; //  44 ,
8'h2D: char_dots = 40'b00001000_00001000_00001000_00001000_00000000; //  45  -
8'h2E: char_dots = 40'b00000000_00110000_00110000_00000000_00000000; //  46  .
8'h2F: char_dots = 40'b00010000_00001000_00000100_00000010_00000000; //  47  /
8'h30: char_dots = 40'b00000000_00011110_00100001_00011110_00000000; //  48  0
--> 17
8'h31: char_dots = 40'b00000000_00100010_00111111_00100000_00000000; //  49  1
8'h32: char_dots = 40'b00100010_00110001_00101001_00100110_00000000; //  50  2
8'h33: char_dots = 40'b00010001_00100101_00100101_00011011_00000000; //  51  3
8'h34: char_dots = 40'b00001100_00001010_00111111_00001000_00000000; //  52  4
8'h35: char_dots = 40'b00010111_00100101_00100101_00011001_00000000; //  53  5
8'h36: char_dots = 40'b00011110_00100101_00100101_00011000_00000000; //  54  6
8'h37: char_dots = 40'b00000001_00110001_00001101_00000011_00000000; //  55  7
8'h38: char_dots = 40'b00011010_00100101_00100101_00011010_00000000; //  56  8
8'h39: char_dots = 40'b00000110_00101001_00101001_00011110_00000000; //  57  9
```

```
8'h3A: char_dots = 40'b00000000_00110110_00110110_00000000_00000000; //  58  :
--> 27
8'h3B: char_dots = 40'b01000000_00110110_00010110_00000000_00000000; //  59  ;
8'h3C: char_dots = 40'b00000000_00001000_00010100_00100010_00000000; //  60  <
8'h3D: char_dots = 40'b00010100_00010100_00010100_00010100_00000000; //  61  =
8'h3E: char_dots = 40'b00000000_00100010_00010100_00001000_00000000; //  62  >
8'h3F: char_dots = 40'b00000000_00000010_00101001_00000110_00000000; //  63  ?
8'h40: char_dots = 40'b00011110_00100001_00101101_00001110_00000000; //  64  @
8'h41: char_dots = 40'b00111110_00001001_00001001_00111110_00000000; //  65  A
--> 34
8'h42: char_dots = 40'b00111111_00100101_00100101_00011010_00000000; //  66  B
8'h43: char_dots = 40'b00011110_00100001_00100001_00010010_00000000; //  67  C
8'h44: char_dots = 40'b00111111_00100001_00100001_00011110_00000000; //  68  D
8'h45: char_dots = 40'b00111111_00100101_00100101_00100001_00000000; //  69  E
8'h46: char_dots = 40'b00111111_00000101_00000101_00000001_00000000; //  70  F
8'h47: char_dots = 40'b00011110_00100001_00101001_00111010_00000000; //  71  G
8'h48: char_dots = 40'b00111111_00000100_00000100_00111111_00000000; //  72  H
8'h49: char_dots = 40'b00000000_00100001_00111111_00100001_00000000; //  73  I
8'h4A: char_dots = 40'b00010000_00100000_00100000_00011111_00000000; //  74  J
8'h4B: char_dots = 40'b00111111_00001100_00010010_00100001_00000000; //  75  K
8'h4C: char_dots = 40'b00111111_00100000_00100000_00100000_00000000; //  76  L
8'h4D: char_dots = 40'b00111111_00000110_00000110_00111111_00000000; //  77  M
8'h4E: char_dots = 40'b00111111_00000110_00011000_00111111_00000000; //  78  N
8'h4F: char_dots = 40'b00011110_00100001_00100001_00011110_00000000; //  79  O
8'h50: char_dots = 40'b00111111_00001001_00001001_00000110_00000000; //  80  P
8'h51: char_dots = 40'b00011110_00110001_00100001_01011110_00000000; //  81  Q
8'h52: char_dots = 40'b00111111_00001001_00011001_00100110_00000000; //  82  R
8'h53: char_dots = 40'b00010010_00100101_00101001_00010010_00000000; //  83  S
8'h54: char_dots = 40'b00000000_00000001_00111111_00000001_00000000; //  84  T
8'h55: char_dots = 40'b00011111_00100000_00100000_00011111_00000000; //  85  U
8'h56: char_dots = 40'b00001111_00110000_00110000_00001111_00000000; //  86  V
8'h57: char_dots = 40'b00111111_00011000_00011000_00111111_00000000; //  87  W
8'h58: char_dots = 40'b00110011_00001100_00001100_00110011_00000000; //  88  X
8'h59: char_dots = 40'b00000000_00000111_00111000_00000111_00000000; //  89  Y
8'h5A: char_dots = 40'b00110001_00101001_00100101_00100011_00000000; //  90  Z
--> 59
8'h5B: char_dots = 40'b00000000_00111111_00100001_00100001_00000000; //  91  [
8'h5C: char_dots = 40'b00000010_00000100_00001000_00010000_00000000; //  92  \
8'h5D: char_dots = 40'b00000000_00100001_00100001_00111111_00000000; //  93  ]
8'h5E: char_dots = 40'b00000000_00000010_00000001_00000010_00000000; //  94  ^
8'h5F: char_dots = 40'b00100000_00100000_00100000_00100000_00000000; //  95  _
8'h60: char_dots = 40'b00000000_00000001_00000010_00000000_00000000; //  96  '
```

```
8'h61: char_dots = 40'b00011000_00100100_00010100_00111100_00000000; //  97  a
--> 66
8'h62: char_dots = 40'b00111111_00100100_00100100_00011000_00000000; //  98  b
8'h63: char_dots = 40'b00011000_00100100_00100100_00000000_00000000; //  99  c
8'h64: char_dots = 40'b00011000_00100100_00100100_00111111_00000000; // 100  d
8'h65: char_dots = 40'b00011000_00110100_00101100_00001000_00000000; // 101  e
8'h66: char_dots = 40'b00001000_00111110_00001001_00000010_00000000; // 102  f
8'h67: char_dots = 40'b00101000_01010100_01010100_01001100_00000000; // 103  g
8'h68: char_dots = 40'b00111111_00000100_00000100_00111000_00000000; // 104  h
8'h69: char_dots = 40'b00000000_00100100_00111101_00100000_00000000; // 105  i
8'h6A: char_dots = 40'b00000000_00100000_01000000_00111101_00000000; // 106  j
8'h6B: char_dots = 40'b00111111_00001000_00010100_00100000_00000000; // 107  k
8'h6C: char_dots = 40'b00000000_00100001_00111111_00100000_00000000; // 108  l
8'h6D: char_dots = 40'b00111100_00001000_00001100_00111000_00000000; // 109  m
8'h6E: char_dots = 40'b00111100_00000100_00000100_00111000_00000000; // 110  n
8'h6F: char_dots = 40'b00011000_00100100_00100100_00011000_00000000; // 111  o
8'h70: char_dots = 40'b01111100_00100100_00100100_00011000_00000000; // 112  p
8'h71: char_dots = 40'b00011000_00100100_00100100_01111100_00000000; // 113  q
8'h72: char_dots = 40'b00111100_00000100_00000100_00001000_00000000; // 114  r
8'h73: char_dots = 40'b00101000_00101100_00110100_00010100_00000000; // 115  s
8'h74: char_dots = 40'b00000100_00011111_00100100_00100000_00000000; // 116  t
8'h75: char_dots = 40'b00011100_00100000_00100000_00111100_00000000; // 117  u
8'h76: char_dots = 40'b00000000_00011100_00100000_00011100_00000000; // 118  v
8'h77: char_dots = 40'b00111100_00110000_00110000_00111100_00000000; // 119  w
8'h78: char_dots = 40'b00100100_00011000_00011000_00100100_00000000; // 120  x
8'h79: char_dots = 40'b00001100_01010000_00100000_00011100_00000000; // 121  y
8'h7A: char_dots = 40'b00100100_00110100_00101100_00100100_00000000; // 122  z
--> 91
8'h7B: char_dots = 40'b00000000_00000100_00011110_00100001_00000000; // 123  {
8'h7C: char_dots = 40'b00000000_00000000_00111111_00000000_00000000; // 124  |
8'h7D: char_dots = 40'b00000000_00100001_00011110_00000100_00000000; // 125  }
8'h7E: char_dots = 40'b00000010_00000001_00000010_00000001_00000000; // 126  ~
--> 95
default: char_dots = 40'b01000001_01000001_01000001_01000001_01000001;
    endcase

endmodule
```

C ipDecoder.v

```
module ipDecoder(clk, to_ac97, ac97_ready,
addr_in, data_in,
write, write_addr, write_length, write_done, data_available,
bytes_read,start_of_packet);

// This IP decoder reads IP packets from an external memory
// using a supplied address and recovers the data as well as
// other information supplied in the IP packet header.

// for testing
output [15:0] bytes_read;

input clk;
input ac97_ready;
input [7:0] data_in;
input write;
input [15:0] write_addr;
input [15:0] write_length;
input write_done;

output [15:0] addr_in;
output [7:0] to_ac97;
output data_available;
output start_of_packet;

reg [15:0] addr_in = 0;
reg [7:0] to_ac97 = 0;
reg data_available = 0;

parameter WAIT = 0;
parameter READ_HEADER = 1;
parameter READ_DATA = 2;

reg [3:0] headerLength = 0;
reg [15:0] packetLength = 0;
reg [3:0] state = 1;
reg [1:0] packetState = WAIT;
reg [15:0] current_addr = 0;
reg [15:0] bytes_read = 0;
reg address_received = 0;
```

```
reg start_of_packet = 0; // indicates when new new packet begins (for output)

reg data_ready_check=0;

always @ (posedge clk) begin
case (packetState)
WAIT: begin
         if (address_received) begin
            packetState <= READ_HEADER;
         end
else if (write) begin
packetState <= READ_HEADER;
            current_addr <= write_addr;
packetLength <= write_length;
//packetLength <= 16'd300;
end
end
READ_HEADER: begin
case (state)
1: begin
state <= state + 1;
addr_in <= current_addr;
end
2: begin
    headerLength <= data_in[3:0]; // in 32-bit words
state <= state + 1;
end
3: begin
state <= state + 1;
addr_in <= current_addr + 3;
end
4: begin
state <= state + 1;
//packetLength[15:8] <= data_in;
end
5: begin
state <= state + 1;
addr_in <= current_addr + 4;
end
6: begin
state <= state + 1;
//packetLength[7:0] <= data_in;
```

```
end
7: begin
state <= state + 1;
addr_in <= current_addr + 4 * headerLength;
state <= 1;
packetState <= READ_DATA;
data_available <= 1;
bytes_read <= 4 * headerLength ;
               //bytes_read <= 20;
               start_of_packet <= 1;
end
endcase
end
READ_DATA: begin
      start_of_packet <= 0;
if (bytes_read >= packetLength) begin
packetState <= WAIT;
data_available <= 0;
end
      else if (write) begin
         current_addr <= write_addr;
         packetLength <= write_length;
         address_received <= 1;
      end
else if (ac97_ready) begin
if (~data_ready_check) begin
to_ac97 <= data_in;
addr_in <= addr_in + 1;
data_ready_check <= 1;
bytes_read <= bytes_read + 1;
end
end
else begin
data_ready_check <= 0;
end
end
endcase
end

endmodule
```

D ipEncoder.v

```
module ipEncoder(clk, data, data_ready,
addr_out, data_out, we,
read, read_addr, read_length, read_done,
               start_packet, ready_for_data);

// This IP encoder takes in data from some input, writing
// data to an external shared memory.  When the specified
// packet length is reached, the module indicates the
// address and length of packet data to be read by
// external controller.

input clk; // system clock
input [7:0] data; // data from audio codec
input data_ready; // ready signal from audio codec
input read_done; // signal from controller that read is complete
input start_packet;     // signal used to begin new IP packet

output we; // write-enable to shared memory
output [15:0] read_length; // length of data to read from read_addr
output [15:0] read_addr; // address of data for controller to read
output read; // pulse indicating packet is ready to be read
//  from specified address
output [15:0] addr_out; // address to output shared memory
output [7:0] data_out; // data to output shared memory
output ready_for_data;     // indicates when IP encoder is ready for data

parameter ipVersion = 4'd4; // 4
parameter IHL = 4'd5; //  5 ; Internet Header Length (min length for correct header is
parameter typeOfService = {3'b000, 3'b100, 2'b00}; // {routine precedence, low delay,
parameter identification = 16'b0;
parameter flags = 3'b010; // don't fragment (see RFC 791 page 13)
parameter timeToLive = 8'd255; // should be reduced after testing
parameter protocol = 8'd17; // Corresponds to user datagram.  See RFC 790
parameter ipSrc = {8'd18, 8'd19, 8'd20, 8'd21}; // source ip address
parameter ipDst = {8'd18, 8'd17, 8'd16, 8'd15}; // destination ip address

reg [15:0] headerChecksum = 0; // must be calculated after other header data is known
// options and padding are optional, and may be implemented later
reg [15:0] maxPacketLength = 16'd2000; // in bytes
//reg options[22:0];
```

```
//reg padding[7:0] = 0;

reg [15:0] packetStartAddress = 0;
reg [15:0] currentAddress = 0;
reg [15:0] totalLengthAddress = 0;
reg [15:0] packetLength = 0;

// output registers
reg we = 0;
reg [15:0] read_length = 0;
reg [15:0] read_addr = 0;
reg read = 0;
reg [15:0] addr_out = 0;
reg [7:0] data_out = 0;
reg old_data_ready=0;
reg ready_for_data = 0;

reg new_data; // used to sychronize ready signal from audio codec

parameter WRITE_HEADER = 0;
parameter WRITE_PACKET_LENGTH_1 = 1;
parameter WRITE_PACKET_LENGTH_2 = 2;
parameter WRITE_DATA = 3;

reg [2:0] state = WRITE_HEADER; // state of FSM
reg [4:0] header_byte = 1; // indicated which header word to write
reg start_packet_sync = 0;
reg old_start_packet = 0;

always @ (posedge clk) begin
   old_data_ready <= data_ready;
   new_data = data_ready & ~old_data_ready;
   old_start_packet <= start_packet;
   start_packet_sync <= start_packet & ~old_start_packet;
end

always @ (posedge clk)
begin
case (state)
WRITE_HEADER: begin
currentAddress <= currentAddress + 1;
   packetLength <= packetLength + 1;
```

```
we <= 1;
read <= 0;
case (header_byte)
1: begin
addr_out <= currentAddress;
data_out <= {ipVersion, IHL};
header_byte <= header_byte + 1;
            packetStartAddress <= currentAddress;
end
2: begin
addr_out <= currentAddress;
data_out <= typeOfService;
header_byte <= header_byte + 1;
end
3: begin
            // Packet Length
            //addr_out <= currentAddress;
            //data_out <= maxPacketLength[15:8];
            totalLengthAddress <= currentAddress;
header_byte <= header_byte + 1;
end
4: begin
            //addr_out <= currentAddress;
            //data_out <= maxPacketLength[7:0];
header_byte <= header_byte + 1;
end
5: begin
addr_out <= currentAddress;
data_out <= identification[15:8];
header_byte <= header_byte + 1;
end
6: begin
addr_out <= currentAddress;
data_out <= identification[7:0];
header_byte <= header_byte + 1;
end
7: begin
addr_out <= currentAddress;
data_out <= {flags, 5'b0}; // fragment offset is 0
header_byte <= header_byte + 1;
end
8: begin
```

```
addr_out <= currentAddress;
data_out <= 8'b0; // fragment offset is 0
header_byte <= header_byte + 1;
end
9: begin
addr_out <= currentAddress;
data_out <= timeToLive;
header_byte <= header_byte + 1;
end
10: begin
addr_out <= currentAddress;
data_out <= protocol;
header_byte <= header_byte + 1;
end
11: begin
addr_out <= currentAddress;
data_out <= headerChecksum[15:8];
header_byte <= header_byte + 1;
end
12: begin
addr_out <= currentAddress;
data_out <= headerChecksum[7:0];
header_byte <= header_byte + 1;
end
13: begin
addr_out <= currentAddress;
data_out <= ipSrc[31:24];
header_byte <= header_byte + 1;
end
14: begin
addr_out <= currentAddress;
data_out <= ipSrc[23:16];
header_byte <= header_byte + 1;
end
15: begin
addr_out <= currentAddress;
data_out <= ipSrc[15:8];
header_byte <= header_byte + 1;
end
16: begin
addr_out <= currentAddress;
data_out <= ipSrc[7:0];
```

```
header_byte <= header_byte + 1;
end
17: begin
addr_out <= currentAddress;
data_out <= ipDst[31:24];
header_byte <= header_byte + 1;
end
18: begin
addr_out <= currentAddress;
data_out <= ipDst[23:16];
header_byte <= header_byte + 1;
end
19: begin
addr_out <= currentAddress;
data_out <= ipDst[15:8];
header_byte <= header_byte + 1;
end
20: begin
addr_out <= currentAddress;
data_out <= ipDst[7:0];
header_byte <= 1;
state <= WRITE_DATA;
            ready_for_data <= 1;
end
endcase
end

WRITE_PACKET_LENGTH_1: begin
   ready_for_data <= 0;
   addr_out <= totalLengthAddress;
   data_out <= packetLength[15:8];
   state <= WRITE_PACKET_LENGTH_2;
end

WRITE_PACKET_LENGTH_2: begin
   addr_out <= totalLengthAddress + 1;
   data_out <= packetLength[7:0];

   read_length <= packetLength;
   //read_length <= maxPacketLength;
read_addr <= packetStartAddress;
read <= 1;
```

```
// reset regs and enter into header fsm on next clock cycle
packetLength <= 0;
   state <= WRITE_HEADER;
end

WRITE_DATA: begin
   if ((packetLength >= maxPacketLength) | start_packet_sync) begin
      state <= WRITE_PACKET_LENGTH_1;
      ready_for_data <= 0;
   end
else if (new_data) begin
      //ready_for_data <= 0;
we <= 1;
addr_out <= currentAddress;
data_out <= data;
currentAddress <= currentAddress + 1;
packetLength <= packetLength + 1;
read <= 0;
end
else we <= 0;
end
endcase
end
endmodule
```

E lab3.v

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

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

   input clock_27mhz;
   input reset;
   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;

input [4:0] volume;
wire source;
//assign volume = 4'd22;  //a reasonable volume value
assign source = 1;    //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

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);

ac97commands cmds(clock_27mhz, ready, command_address, command_data,
                  command_valid, volume, source);
```

```
   assign left_out_data = {audio_out_data, 12'b000000000000};
   assign right_out_data = left_out_data;

   //arbitrarily choose left input, get highest-order bits
   assign audio_in_data = left_in_data[19:12];

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 source;

   reg [23:0] command;
   reg command_valid;

   reg old_ready;
   reg done;
   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";
      done <= 1'b0;
      // synthesis attribute init of done is "0";
      old_ready <= 1'b0;
      // synthesis attribute init of old_ready 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;

   always @(posedge clock) begin
      if (ready && (!old_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 <= 24'h1A_0000; // microphone
        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)

      old_ready <= ready;

   end // always @ (posedge clock)
```

```
endmodule // ac97commands

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

module tone750hz (clock, ready, pcm_data);

   input clock;
   input ready;
   output [19:0] pcm_data;

   reg rdy, old_ready;
   reg [8:0] index;
   reg [19:0] pcm_data;

   initial begin
      old_ready <= 1'b0;
      // synthesis attribute init of old_ready is "0";
      index <= 8'h00;
      // synthesis attribute init of index is "00";
      pcm_data <= 20'h00000;
      // synthesis attribute init of pcm_data is "00000";
   end

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

   // one cycle of a sinewave in 64 20-bit samples
   always @(index) begin
      case (index[5:0])
        6'h00: pcm_data <= 20'h00000;
        6'h01: pcm_data <= 20'h0C8BD;
        6'h02: pcm_data <= 20'h18F8B;
        6'h03: pcm_data <= 20'h25280;
        6'h04: pcm_data <= 20'h30FBC;
```

```
6'h05: pcm_data <= 20'h3C56B;
6'h06: pcm_data <= 20'h471CE;
6'h07: pcm_data <= 20'h5133C;
6'h08: pcm_data <= 20'h5A827;
6'h09: pcm_data <= 20'h62F20;
6'h0A: pcm_data <= 20'h6A6D9;
6'h0B: pcm_data <= 20'h70E2C;
6'h0C: pcm_data <= 20'h7641A;
6'h0D: pcm_data <= 20'h7A7D0;
6'h0E: pcm_data <= 20'h7D8A5;
6'h0F: pcm_data <= 20'h7F623;
6'h10: pcm_data <= 20'h7FFFF;
6'h11: pcm_data <= 20'h7F623;
6'h12: pcm_data <= 20'h7D8A5;
6'h13: pcm_data <= 20'h7A7D0;
6'h14: pcm_data <= 20'h7641A;
6'h15: pcm_data <= 20'h70E2C;
6'h16: pcm_data <= 20'h6A6D9;
6'h17: pcm_data <= 20'h62F20;
6'h18: pcm_data <= 20'h5A827;
6'h19: pcm_data <= 20'h5133C;
6'h1A: pcm_data <= 20'h471CE;
6'h1B: pcm_data <= 20'h3C56B;
6'h1C: pcm_data <= 20'h30FBC;
6'h1D: pcm_data <= 20'h25280;
6'h1E: pcm_data <= 20'h18F8B;
6'h1F: pcm_data <= 20'h0C8BD;
6'h20: pcm_data <= 20'h00000;
6'h21: pcm_data <= 20'hF3743;
6'h22: pcm_data <= 20'hE7075;
6'h23: pcm_data <= 20'hDAD80;
6'h24: pcm_data <= 20'hCF044;
6'h25: pcm_data <= 20'hC3A95;
6'h26: pcm_data <= 20'hB8E32;
6'h27: pcm_data <= 20'hAECC4;
6'h28: pcm_data <= 20'hA57D9;
6'h29: pcm_data <= 20'h9D0E0;
6'h2A: pcm_data <= 20'h95927;
6'h2B: pcm_data <= 20'h8F1D4;
6'h2C: pcm_data <= 20'h89BE6;
6'h2D: pcm_data <= 20'h85830;
6'h2E: pcm_data <= 20'h8275B;
```

```
        6'h2F: pcm_data <= 20'h809DD;
        6'h30: pcm_data <= 20'h80000;
        6'h31: pcm_data <= 20'h809DD;
        6'h32: pcm_data <= 20'h8275B;
        6'h33: pcm_data <= 20'h85830;
        6'h34: pcm_data <= 20'h89BE6;
        6'h35: pcm_data <= 20'h8F1D4;
        6'h36: pcm_data <= 20'h95927;
        6'h37: pcm_data <= 20'h9D0E0;
        6'h38: pcm_data <= 20'hA57D9;
        6'h39: pcm_data <= 20'hAECC4;
        6'h3A: pcm_data <= 20'hB8E32;
        6'h3B: pcm_data <= 20'hC3A95;
        6'h3C: pcm_data <= 20'hCF044;
        6'h3D: pcm_data <= 20'hDAD80;
        6'h3E: pcm_data <= 20'hE7075;
        6'h3F: pcm_data <= 20'hF3743;
      endcase // case(index[5:0])
   end // always @ (index)
endmodule


////////////////////////////////////////////////////////////////////////
//
// Record/playback
//
////////////////////////////////////////////////////////////////////////

module recorder(clock_27mhz, switch, reset, playback, ready, to_ac97_data,address,we,
   input clock_27mhz;              // 27mhz system clock
   input switch; // if high, use interpolation
   input reset;                    // 1 to reset to initial state
   input playback;                 // 1 for playback, 0 for record
   input ready;                    // 1 when AC97 data is available
   input [7:0] to_ac;     // 8-bit PCM data from mic, from ram
   output [7:0] to_ac97_data;      // 8-bit PCM data to headphone

   //input [7:0] from_ac97_data;
   output [15:0]address;
   //output [15:0] read_addr,read_length;
   //output[7:0] s1,s2;
   //output[2:0] eight;
```

```
   output we;//read;

   reg [15:0] address = 0;
   reg we = 0, read=0;
   reg old_playback = 0;
   reg [7:0] to_ac97_data = 0;
   reg [2:0] eight = 0;
   reg [10:0] s1,s2,s3;
   reg [15:0] highaddress = 0;
   reg [9:0] count = 0;

   reg [15:0] read_addr,read_length;


   wire [7:0] to_ac;


// ram64x8 foo(address,clock_27mhz,from_ac97_data,to_ac,we);

   // detect clock cycle when READY goes 0 -> 1
   // f(READY) = 48khz
   wire new_frame;
   reg old_ready;
   always @ (posedge clock_27mhz) old_ready <= reset ? 0 : ready;
   assign new_frame = ready & ~old_ready;

   // test: playback 750hz tone, or loopback using incoming data
   wire [19:0] tone;
   tone750hz xxx(clock_27mhz, ready, tone);

   always @ (posedge clock_27mhz) begin
      if (new_frame) begin
  we <=0;
eight <= eight + 1;
if (old_playback & playback)
begin
if (switch)
begin
case (eight)
0: s3 <= (s2+s2+s2+s2+s2+s2+s2+s2);
1: s3 <= (s2+s2+s2+s2+s2+s2+s2+s1);
2: s3 <= (s2+s2+s2+s2+s2+s2+s1+s1);
```

```
3: s3 <= (s2+s2+s2+s2+s2+s1+s1+s1);
4: s3 <= (s2+s2+s2+s2+s1+s1+s1+s1);
5: s3 <= (s2+s2+s2+s1+s1+s1+s1+s1);
6: s3 <= (s2+s2+s1+s1+s1+s1+s1+s1);
7: s3 <= (s2+s1+s1+s1+s1+s1+s1+s1);
default: s3 <= 0;
endcase
to_ac97_data <= s3[10:3];
  if (eight==7)
begin
s1[7:0]<=to_ac;
s1[10:8]<={3{to_ac[7]}};
s2<=s1;
if (address==highaddress) address <= 0;
else address <= address + 1;
end

end


else
begin
       if (eight==0)
begin
if (address==highaddress) address <= 0;
else address <= address + 1;
to_ac97_data <= to_ac;
end
end

end
else if (old_playback & !playback)
begin
address <= 0;
eight <= 7;
end
else if (!old_playback & playback)
begin
highaddress <= address;
address <= 0;
eight <= 7;
end
```

```
else if (!old_playback & !playback)
begin
if (read==1) read <= 0;
if (eight==0)
begin
count <= count+1;
if (address==65534) address <= 0;
else address <= address + 1;
we <= 1;
end
end

  old_playback <= playback;
 // just received new data from the AC97
 //to_ac97_data <= playback ? tone[19:12] : from_ac97_data_from_mem;
      end
   end

endmodule
```

F lab4.v

```
//////////////////////////////////////////////////////////////////////////////////
//
// Pushbutton Debounce Module (video version)
//
//////////////////////////////////////////////////////////////////////////////////

module debounce (reset, clock_65mhz, noisy, clean);
   input reset, clock_65mhz, noisy;
   output clean;

   reg [19:0] count;
   reg new, clean;

   always @(posedge clock_65mhz)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 650000) clean <= new;
     else count <= count+1;
```

```
endmodule


module debounce2 (reset, clock, noisy, clean);

   input reset, clock, noisy;
   output clean;

   reg [30:0] 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 == 10000000)
       clean <= new;
     else
       count <= count+1;

endmodule

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

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

```
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
////////////////////////////////////////////////////////////////////////////////

module lab4   (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

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

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

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
```

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

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

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

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

// LED Displays
assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
```

```
assign disp_data_out = 1'b0;
// disp_data_in is an input

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

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

// 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;

//////////////////////////////////////////////////////////////////////
//
// lab4 : a simple pong game
//
//////////////////////////////////////////////////////////////////////

// use FPGA's digital clock manager to produce a
```

```
   // 65MHz clock (actually 64.8MHz)
   wire clock_65mhz_unbuf,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
   .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

   // ENTER button is user reset
   wire reset,user_reset;
   debounce db1(power_on_reset, clock_65mhz, ~button_enter, user_reset);
   assign reset = user_reset | power_on_reset;

   // UP and DOWN buttons for pong paddle
   wire up,down;
   debounce db2(reset, clock_65mhz, ~button_up, up);
   debounce db3(reset, clock_65mhz, ~button_down, down);

   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);

   // feed XVGA signals to user's pong game
   wire [2:0] pixel;
   wire phsync,pvsync,pblank;
   pong_game pg(clock_65mhz,reset,up,down,switch[7:4],
hcount,vcount,hsync,vsync,blank,
phsync,pvsync,pblank,pixel);

   // switch[1:0] selects which video generator to use:
   //  00: user's pong game
   //  01: 1 pixel outline of active video area (adjust screen controls)
   //  10: color bars
```

```
   reg [2:0] rgb;
   reg b,hs,vs;
   always @(posedge clock_65mhz) begin
      if (switch[1:0] == 2'b01) begin
 // 1 pixel outline of visible area (white)
 hs <= hsync;
 vs <= vsync;
 b <= blank;
 rgb <= (hcount==0 | hcount==1023 | vcount==0 | vcount==767) ? 7 : 0;
      end else if (switch[1:0] == 2'b10) begin
 // color bars
 hs <= hsync;
 vs <= vsync;
 b <= blank;
 rgb <= hcount[8:6];
      end else begin
         // default: pong
 hs <= phsync;
 vs <= pvsync;
 b <= pblank;
 rgb <= pixel;
      end
   end

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

   assign led = ~{3'b000,up,down,reset,switch[1:0]};

endmodule

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

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

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

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

   // vertical: 806 lines total
   // display 768 lines
   wire      vsyncon,vsyncoff,vreset,vblankon;
   assign    vblankon = hreset & (vcount == 767);
   assign    vsyncon = hreset & (vcount == 776);
   assign    vsyncoff = hreset & (vcount == 782);
   assign    vreset = hreset & (vcount == 805);

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

      vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
      vblank <= next_vblank;
```

```
      vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low

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

//////////////////////////////////////////////////////////////////////////////////
//
// pong_game: the game itself!
//
//////////////////////////////////////////////////////////////////////////////////

module pong_game (vclock,reset,up,down,pspeed,
  hcount,vcount,hsync,vsync,blank,
  phsync,pvsync,pblank,pixel,paddle_in,
  paddle_out,x_in,y_in,x_out,y_out,host_in,host_out,play);
   input vclock; // 65MHz clock
   input reset; // 1 to initialize module
   input up; // 1 when paddle should move up
   input down; // 1 when paddle should move down
   input [3:0] pspeed;  // puck speed in pixels/tick
   input [10:0] hcount; // horizontal index of current pixel (0..1023)
   input [9:0]  vcount; // vertical index of current pixel (0..767)
   input hsync; // XVGA horizontal sync signal (active low)
   input vsync; // XVGA vertical sync signal (active low)
   input blank; // XVGA blanking (1 means output black pixel)

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

input [9:0] paddle_out,y_out;
input [10:0] x_out;
input host_out,play;
output host_in;
output [9:0] paddle_in,y_in;
output [10:0] x_in;

reg [9:0] paddle_in=0;
reg [9:0] y_in=0;
reg [10:0] x_in=0;
```

```
reg host_in=0;
reg state = 0;



reg [9:0] paddle_y = 0;
reg [9:0] paddle_y2 = 0;
wire [2:0] paddle_pixel;
wire [2:0] paddle_pixel2;
wire [2:0] ball_pixel;
reg [2:0] pixel=0;
reg [10:0] ball_x = 480;
reg [9:0] ball_y = 352;
reg xdir = 1;
reg ydir = 1;
reg lose = 0;
reg start = 0;

   // REPLACE ME! The code below just generates a color checkerboard
   // using 64 pixel by 64 pixel squares.
   assign phsync = hsync;
   assign pvsync = vsync;
   assign pblank = blank;
   //assign pixel = hcount[8:6] + vcount[8:6];

   blob ball(ball_x,ball_y,hcount,vcount,ball_pixel);

blob paddle(11'd0,paddle_y,hcount,vcount,paddle_pixel);
defparam paddle.WIDTH = 16;
defparam paddle.HEIGHT = 128;
defparam paddle.COLOR = 3'b110; // yellow!

blob paddle2(11'd1007,paddle_y2,hcount,vcount,paddle_pixel2);
defparam paddle2.WIDTH = 16;
defparam paddle2.HEIGHT = 128;
defparam paddle2.COLOR = 3'b110; // yellow!

always @ (posedge vclock)
```

```
begin
if (paddle_pixel>0) pixel <= paddle_pixel;
else if (paddle_pixel2>0) pixel <= paddle_pixel2;
else pixel <= ball_pixel;
end

always @ (negedge vsync) begin

 case (state)
 0: begin
host_in <= 0;

if (up & down) paddle_y2 <= paddle_y2;
else if (up)
begin
if (paddle_y2 > 3) paddle_y2 <= paddle_y2 - 4;
else paddle_y2 <= 0;
end
else if (down)
begin
if (paddle_y2 < 636) paddle_y2 <= paddle_y2 + 4;
else paddle_y2 <=639;
end

paddle_in <= paddle_y2;
paddle_y <= paddle_out;
ball_x <= x_out;
ball_y <= y_out;

if (play) state <=1;
end

1: begin
if (host_out==1) state <=0;
else begin
host_in <= 1;

       paddle_in <= paddle_y;
x_in <= ball_x;
y_in <= ball_y;
paddle_y2 <= paddle_out;
```

```
if (play)
begin
ball_x <= 480;
ball_y <= 352;
xdir <= 1;
ydir <= 1;
paddle_y <= 400;
lose <= 0;
end
else if (lose==0)
begin
if (xdir)
begin
ball_x <= ball_x + pspeed;
if ((ball_x + pspeed) > 959)
begin
lose <= 1;
end


if ((ball_x+pspeed)>943)
begin
if (((ball_y+64)>paddle_y2)&(ball_y<(paddle_y2+128)))
begin
ball_x <= 943;
xdir <= !xdir;
end
end
end
else
begin
ball_x <= ball_x - pspeed;
if ((ball_x-pspeed)<=16)
begin
if (((ball_y+64)>paddle_y)&(ball_y<(paddle_y+128)))
begin
ball_x <= 16;
xdir <= !xdir;
end
end
else if (pspeed > ball_x)
begin
```

```
ball_x <= 0;
lose <=1;
end
end

if (ydir)
begin
ball_y <= ball_y + pspeed;
if ((ball_y + pspeed) > 703)
begin
ball_y <= 703;
ydir <= !ydir;
end
end
else
begin
ball_y <= ball_y - pspeed;
if (pspeed > ball_y)
begin
ball_y <= 0;
ydir <= !ydir;
end
end

if (up & down) paddle_y <= paddle_y;
else if (up)
begin
if (paddle_y > 3) paddle_y <= paddle_y - 4;
else paddle_y <= 0;
end
else if (down)
begin
if (paddle_y < 636) paddle_y <= paddle_y + 4;
else paddle_y <=639;
end


if (ball_x==16)
begin
if ((ball_y+64)<paddle_y)
lose <= 1;
if (ball_y>(paddle_y+128))
```

```
lose <=1;
end

if (ball_x==943)
begin
if ((ball_y+64)<paddle_y2)
lose <= 1;
if (ball_y>(paddle_y2+128))
lose <=1;
end

      end
     end
end

endcase


 end

endmodule


module blob(x,y,hcount,vcount,pixel);
parameter WIDTH = 64; // default width: 64 pixels
parameter HEIGHT = 64; // default height: 64 pixels
parameter COLOR = 3'b111; // default color: white
input [10:0] x,hcount;
input [9:0] y,vcount;
output [2:0] pixel;
reg [2:0] pixel;
always @ (x or y or hcount or vcount) begin
if ((hcount >= x && hcount < (x+WIDTH)) &&
(vcount >= y && vcount < (y+HEIGHT)))
pixel = COLOR;
else pixel = 0;
end
endmodule
```

G labkit.v

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

```
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
//////////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Sep-09: Added missing default assignments to "ac97_sdata_out",
//              "disp_data_out", "analyzer[2-3]_clock" and
//              "analyzer[2-3]_data".
//
```

```
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-Apr-29: Change history started
//
////////////////////////////////////////////////////////////////////////////////

module 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
   //
   ////////////////////////////////////////////////////////////////////////

//lab4 video stuff
   wire clock_65mhz_unbuf,clock_65mhz;
   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
   // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
   // synthesis attribute CLKIN_PERIOD of vclk1 is 37
   BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));

   // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(clock_65mhz), .Q(power_on_reset),
   .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;

// ENTER button is user reset
   wire user_reset;
   debounce db1(power_on_reset, clock_65mhz, ~button_enter, user_reset);
   assign reset = user_reset | power_on_reset;

   // UP and DOWN buttons for pong paddle
   wire up,down,left,right,but0,but1;
   debounce db2(reset, clock_65mhz, ~button_up, up);
```

```
   debounce db3(reset, clock_65mhz, ~button_down, down);
   debounce db4(reset, clock_27mhz, ~button_left, left);
   debounce db5(reset, clock_27mhz, ~button_right, right);
   debounce db6(reset, clock_27mhz, ~button0, but0);
   debounce db7(reset, clock_27mhz, ~button1, but1);

   // generate basic XVGA video signals
   wire [10:0] hcount;
   wire [9:0]  vcount;
   wire hsync,vsync,blank;
   xvga xvga1(clock_65mhz,hcount,vcount,hsync,vsync,blank);

   wire [9:0]paddle_in,paddle_out,y_in,y_out;
   wire host_in,host_out,start_in,start_out;
   wire [10:0] x_in,x_out;


   // feed XVGA signals to user's pong game
   wire [2:0] pixel;
   wire phsync,pvsync,pblank;
   pong_game pg(clock_65mhz,reset,up,down,switch[7:4],
hcount,vcount,hsync,vsync,blank,
phsync,pvsync,pblank,pixel,paddle_in,
  paddle_out,x_in,y_in,x_out,y_out,host_in,host_out,but1);



   // switch[1:0] selects which video generator to use:
   //  00: user's pong game
   //  01: 1 pixel outline of active video area (adjust screen controls)
   //  10: color bars
   reg [2:0] rgb;
   reg b,hs,vs;
   always @(posedge clock_65mhz) begin
      if (switch[1:0] == 2'b01) begin
 // 1 pixel outline of visible area (white)
 hs <= hsync;
 vs <= vsync;
 b <= blank;
 rgb <= (hcount==0 | hcount==1023 | vcount==0 | vcount==767) ? 7 : 0;
      end else if (switch[1:0] == 2'b10) begin
 // color bars
```

```
hs <= hsync;
vs <= vsync;
b <= blank;
rgb <= hcount[8:6];
     end else begin
        // default: pong
hs <= phsync;
vs <= pvsync;
b <= pblank;
rgb <= pixel;
     end
  end

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

  //assign led = ~{3'b000,up,down,reset,switch[1:0]};




  // Audio Input and Output
  assign beep= 1'b0;


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


  // Video Output
  assign tv_out_ycrcb = 10'h0;
```

```
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

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

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

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

   // 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 analyzer3_data = 16'h0;
   assign analyzer3_clock = 1'b1;


//*******************************

//lab3 audio stuff

   wire [7:0] from_ac97_data, to_ac97_data, to_ac97_data1, to_ac;
   wire ready;
   wire [2:0] no4;
   wire wea;
   wire [15:0] address;
```

```
wire [15:0] addr_in_matt, addr_in_brian, addr_out_brian, addr_out_matt,
read_length, write_length, read_addr, write_addr;
wire [7:0] data_in_matt, data_out_matt, data_in_brian, data_out_brian;
wire read, write, write_done, read_done, we_matt, we_brian;


wire [63:0] dots;
wire [7:0] keyboard_ascii;
wire char_ready, tx_ready, tx_busy, rx_ready;


wire [7:0] rx_data;
wire [7:0] tx_data, tx_data1;


wire button0clean, button1clean;

   wire playback;

wire matt_ready;
wire [7:0] from_matt, to_matt;
wire data_available_from_decoder;
wire ready_for_data, start_packet;
wire [4:0] volume;


   // AC97 driver
   audio a(clock_27mhz, reset, from_ac97_data, to_ac97_data, ready,
   audio_reset_b, ac97_sdata_out, ac97_sdata_in,
   ac97_synch, ac97_bit_clock,volume);


   assign playback = 0;


assign tx_data = tx_data1;

wire [15:0] length2;
wire volume_display;
```

```
wire [16*8-1:0] string_data;
wire [16*8-1:0] string_data1;
 assign string_data = switch[2] ? "  HELLO WORLD!  " : string_data1;

assign dots = {40'h0000_00,from_matt,to_matt, tx_data, rx_data, to_ac97_data};

assign led = 8'h00;

rs232 serial_comm(.reset(reset), .clock_27mhz(clock_27mhz),
.txd(rs232_txd), .rxd(rs232_rxd),
.rts(rs232_rts), .cts(rs232_cts),
.rx_data(rx_data), .tx_data(tx_data), .tx_ready(tx_ready), .tx_busy(tx_busy), .rx_rea

display_string disp(.reset(0), .clock_27mhz(clock_27mhz),.string_data(string_data),
.disp_blank(disp_blank), .disp_clock(disp_clock), .disp_rs(disp_rs),
.disp_ce_b(disp_ce_b), .disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));

ps2_ascii_input ps_in(.clock_27mhz(clock_27mhz), .reset(!button_enter),
  .clock(keyboard_clock), .data(keyboard_data),
.ascii(keyboard_ascii), .ascii_ready(char_ready) );

slipSend sender(.addr_in(addr_in_brian),.data_in(data_in_brian),.read(read),.read_addr
.read_length(read_length),.read_done(read_done),.clk(clock_27mhz),
.tx_ready(tx_ready),.tx_busy(tx_busy),.data_out(tx_data1), .length(length2));


slipRec receiver(.addr_out(addr_out_brian),.data_out(data_out_brian),.write(write),
.write_addr(write_addr),.write_length(write_length),.write_done(write_done),
.wea(we_brian),.clk(clock_27mhz),.rx_ready(rx_ready),.data_in(rx_data));


wire [15:0] bytes_read;
wire [39:0] tigon_data_in;
wire [39:0] tigon_data_out;
wire send;
wire play;
wire ip_new_packet;
wire ip_encoder_ready;
wire ip_decoder_ready;
wire codec_ready;
wire [7:0] codec_output;
wire [7:0] codec_input;
```

```
simpleAudioCodec codec(.clk_27mhz(clock_27mhz), .input_data(codec_input), .output_data
                       .ready(codec_ready), .ac97_ready(ready),
                       .from_ac97_data(from_ac97_data), .to_ac97_data(to_ac97_data));

ipDecoder decoder(.clk(clock_27mhz), .to_ac97(from_matt), .ac97_ready(ip_decoder_ready
.addr_in(addr_in_matt), .data_in(data_in_matt),
.write(write), .write_addr(write_addr), .write_length(write_length),
 .write_done(write_done),
 .data_available(data_available_from_decoder),
 .bytes_read(bytes_read),
                .start_of_packet(ip_new_packet));

ipEncoder encoder(.clk(clock_27mhz), .data(to_matt), .data_ready(ip_encoder_ready),
.addr_out(addr_out_matt), .data_out(data_out_matt), .we(we_matt),
.read(read), .read_addr(read_addr), .read_length(read_length), .read_done(read_done),
               .start_packet(start_packet), .ready_for_data(ready_for_data));

tigonProtocolEncoder tigonEncode(.clk(clock_27mhz), .tigon_data_in(tigon_data_in),
                       .ip_data_out(to_matt), .ip_ready(ip_encoder_ready), .ip_start_pa
                       .ip_ready_for_data(ready_for_data),
                       .send(send),
                       .from_audio_codec(codec_output),
                       .codec_ready(codec_ready));

tigonProtocolDecoder tigonDecoder(.clk(clock_27mhz),.tigon_data_out(tigon_data_out),
                       .ip_data_in(from_matt), .ip_ready(ip_decoder_ready),
                       .ip_data_available(data_available_from_decoder),
                       .play(play),
                       .to_audio_codec(codec_input),
                       .codec_ready(codec_ready),
                       .ip_new_packet(ip_new_packet));


phoneControls phone(clock_27mhz,tigon_data_out,tigon_data_in,send,play,right,left,but
string_data1,paddle_in,paddle_out,x_in,y_in,x_out,y_out,host_in,host_out);




portram ram1(
```

```
addr_out_brian,
addr_in_matt,
clock_27mhz,
clock_27mhz,
data_out_brian,
nothing,
data_in_matt,
we_brian);


portram ram2(
addr_out_matt,
addr_in_brian,
clock_27mhz,
clock_27mhz,
data_out_matt,
nothing,
data_in_brian,
we_matt);



   assign analyzer1_clock = clock_27mhz;
   assign analyzer1_data[0] = audio_reset_b;
   assign analyzer1_data[1] = ac97_sdata_out;
   assign analyzer1_data[2] = ac97_sdata_in;
   assign analyzer1_data[3] = ac97_synch;
   assign analyzer1_data[15:4] = 0;

   assign analyzer2_clock = ready;
   //assign analyzer2_data = {to_ac97_data,data_out_brian};
   //assign analyzer2_data = {to_ac97_data,7'b0,matt_ready};
   assign analyzer2_data = {ip_decoder_ready,data_available_from_decoder,ip_new_packe
   //assign analyzer2_data = {to_ac97_data,bytes_read[7:0]};

   assign analyzer4_clock = clock_27mhz;
   assign analyzer4_data = {data_in_matt, to_ac97_data};


endmodule
```

H phoneControls.v

```
module phoneControls(clk,tigon_data_in,tigon_data_out,send,play,volume_up,volume_down
enter,volume,string_data,paddle_in,paddle_out,x_in,y_in,x_out,y_out,host_in,host_out)

input clk,volume_up,volume_down,enter,host_in;
input [39:0] tigon_data_in;
input [9:0] paddle_in,y_in;
input [10:0] x_in;


output [16*8-1:0] string_data;
output send,play,host_out;
output [4:0] volume;
output [39:0] tigon_data_out;
output [9:0] paddle_out,y_out;
output [10:0] x_out;



reg [4:0] volume = 5'd22;
reg [16*8-1:0] string_data = "  HELLO WORLD!  ";
reg [16*8-1:0] volume_string = "  HELLO WORLD!  ";
reg [23:0] count = 1;
reg [39:0] tigon_data_out = 0;
reg disp_switch = 1;
reg send = 0,play = 0;
reg [9:0] paddle_out=0;
reg [9:0] y_out=0;
reg [10:0] x_out=0;
reg volume_display = 0,host_out = 0;
reg statechange = 0,volumechange = 0;
reg storedstate=0;
reg oldenter=0;
reg new_enter=0;

parameter S_Idle = 0;
parameter S_Rec_Call = 1;
parameter S_Volume = 2;
parameter S_Send_Call = 3;
parameter S_Send_Hang = 4;
parameter S_Connected = 5;
```

```
reg [2:0] state = S_Idle;

always @ (posedge clk) begin
oldenter <= enter;
new_enter <= enter & ~oldenter;
end

always @ (posedge clk) begin

x_out <= tigon_data_in[31:21];
y_out <= tigon_data_in[20:11];
paddle_out <= tigon_data_in[10:1];
host_out <= tigon_data_in[0];

tigon_data_out[39:32] <= 0;
tigon_data_out[31:21] <= x_in;
tigon_data_out[20:11] <= y_in;
tigon_data_out[10:1] <= paddle_in;
tigon_data_out[0] <= host_in;
tigon_data_out[6:0] <= 0;




case (state)

S_Idle: begin
play <= 1;
send <= 1;
     if (new_enter) begin
state <= S_Connected;
end
       string_data <= "   Connected.   ";
if (volume_up) begin
if (volume==31) volume <= 31;
else volume <= volume +1;
string_data <= volume_string;
state <= S_Volume;
storedstate <= state;
end
else if (volume_down) begin
```

```
if (volume==0) volume<=0;
else volume <= volume - 1;
string_data <= volume_string;
string_data <= volume_string;
state <= S_Volume;
storedstate <= state;
end

end

S_Volume: begin
count <= count + 1;
if (count==0) begin
count <= 1;
state <= storedstate;
end
end


S_Connected: begin
play <= 1;
send <= 0;
if (new_enter) begin
statechange <= 0;
state <= S_Idle;
end
       string_data <= " Not Connected. ";
if (volume_up) begin
if (volume==31) volume <= 31;
else volume <= volume +1;
string_data <= volume_string;
state <= S_Volume;
storedstate <= state;
end
else if (volume_down) begin
if (volume==0) volume<=0;
else volume <= volume - 1;
string_data <= volume_string;
string_data <= volume_string;
state <= S_Volume;
storedstate <= state;
end
```

```
end

endcase



case (volume)
5'd0: volume_string <= {"0             31"};
5'd1: volume_string <= {"0$            31"};
5'd2: volume_string <= {"0&            31"};
5'd3: volume_string <= {"0'#           31"};
5'd4: volume_string <= {"0'%           31"};
5'd5: volume_string <= {"0''           31"};
5'd6: volume_string <= {"0''$          31"};
5'd7: volume_string <= {"0''&          31"};
5'd8: volume_string <= {"0'''#         31"};
5'd9: volume_string <= {"0'''%         31"};
5'd10: volume_string <= {"0''''         31"};
5'd11: volume_string <= {"0''''$        31"};
5'd12: volume_string <= {"0''''&        31"};
5'd13: volume_string <= {"0'''''#       31"};
5'd14: volume_string <= {"0'''''%       31"};
5'd15: volume_string <= {"0''''''       31"};
5'd16: volume_string <= {"0''''''$      31"};
5'd17: volume_string <= {"0''''''&      31"};
5'd18: volume_string <= {"0'''''''#     31"};
5'd19: volume_string <= {"0'''''''%     31"};
5'd20: volume_string <= {"0''''''''     31"};
5'd21: volume_string <= {"0''''''''$    31"};
5'd22: volume_string <= {"0''''''''&    31"};
5'd23: volume_string <= {"0'''''''''#   31"};
5'd24: volume_string <= {"0'''''''''%   31"};
5'd25: volume_string <= {"0''''''''''   31"};
5'd26: volume_string <= {"0''''''''''$  31"};
5'd27: volume_string <= {"0''''''''''&  31"};
5'd28: volume_string <= {"0'''''''''''# 31"};
5'd29: volume_string <= {"0'''''''''''% 31"};
5'd30: volume_string <= {"0''''''''''' 31"};
5'd31: volume_string <= {"0''''''''''''31"};
default: volume_string <= {"0             31"};
 endcase
```

```
end


endmodule
```

I ps2new.v

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

module ps2_ascii_input(clock_27mhz, reset, clock, data, ascii, ascii_ready);

   // module to generate ascii code for keyboard input
   // this is module works synchronously with the system clock

   input clock_27mhz;
   input reset;   // Active high asynchronous reset
   input clock;   // PS/2 clock
   input data;    // PS/2 data
   output [7:0] ascii;   // ascii code (1 character)
   output ascii_ready;  // ascii ready (one clock_27mhz cycle active high)

   reg [7:0]   ascii_val; // internal combinatorial ascii decoded value
   reg [7:0]   lastkey; // last keycode
   reg [7:0]   curkey; // current keycode
   reg [7:0]   ascii; // ascii output (latched & synchronous)
   reg         ascii_ready; // synchronous one-cycle ready flag

   // get keycodes

   wire        fifo_rd; // keyboard read request
   wire [7:0]  fifo_data; // keyboard data
   wire        fifo_empty; // flag: no keyboard data
   wire        fifo_overflow; // keyboard data overflow

   ps2 myps2(reset, clock_27mhz, clock, data, fifo_rd, fifo_data,
     fifo_empty,fifo_overflow);

   assign      fifo_rd = ~fifo_empty; // continous read
   reg         key_ready;

   always @(posedge clock_27mhz)
```

```
     begin

// get key if ready

curkey <= ~fifo_empty ? fifo_data : curkey;
lastkey <= ~fifo_empty ? curkey : lastkey;
key_ready  <= ~fifo_empty;

// raise ascii_ready for last key which was read

ascii_ready <= key_ready & ~(curkey[7]|lastkey[7]);
ascii <=  (key_ready & ~(curkey[7]|lastkey[7])) ? ascii_val : ascii;

     end

   always @(curkey) begin //convert PS/2 keyboard make code ==> ascii code
     case (curkey)
       8'h1C: ascii_val = 8'h41; //A
       8'h32: ascii_val = 8'h42; //B
       8'h21: ascii_val = 8'h43; //C
       8'h23: ascii_val = 8'h44; //D
       8'h24: ascii_val = 8'h45; //E
       8'h2B: ascii_val = 8'h46; //F
       8'h34: ascii_val = 8'h47; //G
       8'h33: ascii_val = 8'h48; //H
       8'h43: ascii_val = 8'h49; //I
       8'h3B: ascii_val = 8'h4A; //J
       8'h42: ascii_val = 8'h4B; //K
       8'h4B: ascii_val = 8'h4C; //L
       8'h3A: ascii_val = 8'h4D; //M
       8'h31: ascii_val = 8'h4E; //N
       8'h44: ascii_val = 8'h4F; //O
       8'h4D: ascii_val = 8'h50; //P
       8'h15: ascii_val = 8'h51; //Q
       8'h2D: ascii_val = 8'h52; //R
       8'h1B: ascii_val = 8'h53; //S
       8'h2C: ascii_val = 8'h54; //T
       8'h3C: ascii_val = 8'h55; //U
       8'h2A: ascii_val = 8'h56; //V
       8'h1D: ascii_val = 8'h57; //W
       8'h22: ascii_val = 8'h58; //X
       8'h35: ascii_val = 8'h59; //Y
```

```
       8'h1A: ascii_val = 8'h5A; //Z

       8'h45: ascii_val = 8'h30; //0
       8'h16: ascii_val = 8'h31; //1
       8'h1E: ascii_val = 8'h32; //2
       8'h26: ascii_val = 8'h33; //3
       8'h25: ascii_val = 8'h34; //4
       8'h2E: ascii_val = 8'h35; //5
       8'h36: ascii_val = 8'h36; //6
       8'h3D: ascii_val = 8'h37; //7
       8'h3E: ascii_val = 8'h38; //8
       8'h46: ascii_val = 8'h39; //9

       8'h0E: ascii_val = 8'h60; // `
       8'h4E: ascii_val = 8'h2D; // -
       8'h55: ascii_val = 8'h3D; // =
       8'h5C: ascii_val = 8'h5C; // \
       8'h29: ascii_val = 8'h20; // (space)
       8'h54: ascii_val = 8'h5B; // [
       8'h5B: ascii_val = 8'h5D; // ]
       8'h4C: ascii_val = 8'h3B; // ;
       8'h52: ascii_val = 8'h27; // '
       8'h41: ascii_val = 8'h2C; // ,
       8'h49: ascii_val = 8'h2E; // .
       8'h4A: ascii_val = 8'h2F; // /

       8'h5A: ascii_val = 8'h0D; // enter (CR)
       8'h66: ascii_val = 8'h08; // backspace

       //  8'hF0: ascii_val = 8'hF0; // BREAK CODE

       default: ascii_val = 8'h23; // #
     endcase
   end
endmodule // ps2toascii

////////////////////////////////////////////////////////////////////////
// new synchronous ps2 keyboard driver, with built-in fifo, from Chris Terman

module ps2(reset, clock_27mhz, ps2c, ps2d, fifo_rd, fifo_data,
   fifo_empty,fifo_overflow);
```

```
  input clock_27mhz,reset;
  input ps2c; // ps2 clock
  input ps2d; // ps2 data
  input fifo_rd; // fifo read request (active high)
  output [7:0] fifo_data; // fifo data output
  output  fifo_empty; // fifo empty (active high)
  output  fifo_overflow; // fifo overflow - too much kbd input

 reg [3:0] count;      // count incoming data bits
 reg [9:0] shift;      // accumulate incoming data bits

 reg [7:0] fifo[7:0];   // 8 element data fifo
 reg fifo_overflow;
 reg [2:0] wptr,rptr;   // fifo write and read pointers

 wire [2:0] wptr_inc = wptr + 1;

 assign fifo_empty = (wptr == rptr);
 assign fifo_data = fifo[rptr];

 // synchronize PS2 clock to local clock and look for falling edge
 reg [2:0] ps2c_sync;
 always @ (posedge clock_27mhz) ps2c_sync <= {ps2c_sync[1:0],ps2c};
 wire sample = ps2c_sync[2] & ~ps2c_sync[1];

 always @ (posedge clock_27mhz) begin
   if (reset) begin
     count <= 0;
     wptr <= 0;
     rptr <= 0;
     fifo_overflow <= 0;
   end
   else if (sample) begin
          // order of arrival: 0,8 bits of data (LSB first),odd parity,1
          if (count==10) begin
             // just received what should be the stop bit
             if (shift[0]==0 && ps2d==1 && (^shift[9:1])==1) begin
fifo[wptr] <= shift[8:1];
wptr <= wptr_inc;
fifo_overflow <= fifo_overflow | (wptr_inc == rptr);
             end
             count <= 0;
```

```
   end else begin
              shift <= {ps2d,shift[9:1]};
              count <= count + 1;
   end
         end
    // bump read pointer if we're done with current value.
    // Read also resets the overflow indicator
    if (fifo_rd && !fifo_empty) begin
      rptr <= rptr + 1;
      fifo_overflow <= 0;
    end
  end

endmodule
```

J rs232_v2.v

```
////////////////////////////////////////////////////////////////////////
// TOP-LEVEL  RS232 INTERFACE MODULE
//
module rs232(reset, clock_27mhz, rxd, cts, txd, rts, rx_data, tx_data, tx_ready, tx_bu
  input reset, clock_27mhz;
  input rxd, cts, tx_ready;
  output txd, rts, tx_busy, rx_ready;
  output [7:0] rx_data;
  input [7:0] tx_data;

  //wire rx_ready;
  //wire tx_busy;
  wire [7:0] buff_rx;
//  reg [7:0] buff_tx;
  reg [7:0] rx_data;

  assign rts = ~tx_busy;

  async_receiver rx(.clock_27mhz(clock_27mhz), .rxd(rxd),
   .data_ready(rx_ready),  .rx_data(buff_rx) );

  async_transmitter tx(.clock_27mhz(clock_27mhz), .txd(txd),
   .tx_start(tx_ready & ~tx_busy), .tx_data(tx_data), .tx_busy(tx_busy));

  always @(posedge clock_27mhz) begin
```

```
   if(rx_ready)
rx_data <= buff_rx;
  end

endmodule
////////////////////////////////////////////////////////////////////////////////




//////////////////////////////////////////////////////////////
// RECEIVER MODULE
//
// modified source from
// RS-232 RX module
//  (c) fpga4fun.com KNJN LLC - 2003, 2004, 2005

module async_receiver(clock_27mhz, rxd, data_ready, rx_data);
  input clock_27mhz, rxd;
  output data_ready;   // one clock pulse when rx_data is valid
  output [7:0] rx_data;

  ////////////////////////////////////////////////////////////
  //connect baud generator
  wire serial_clk;
  baud_gen rx_clk(.clock_27mhz(clock_27mhz), .serial_clk(serial_clk),
      .enable(1'b0), .rx_mode(1'b1) );

  ////////////////////////////////////////////////////////////
  //synchronize (& invert) rxd input
  reg sync, rxd_inv;
  always @(posedge clock_27mhz) begin
if(serial_clk)
{rxd_inv, sync} <= {sync, ~rxd};
  end

  ////////////////////////////////////////////////////////////
  reg [1:0] rxd_cnt_inv;
  reg rxd_bit_inv;

  always @(posedge clock_27mhz) begin
if(serial_clk)
```

```
begin
   if( rxd_inv && rxd_cnt_inv!=2'b11)
rxd_cnt_inv <= rxd_cnt_inv + 1;
   else if(~rxd_inv && rxd_cnt_inv!=2'b00)
rxd_cnt_inv <= rxd_cnt_inv - 1;

if(rxd_cnt_inv==2'b00)
rxd_bit_inv <= 0;
   else if(rxd_cnt_inv==2'b11)
rxd_bit_inv <= 1;
end
  end


  ////////////////////////////////////////////////////////////////
  reg [7:0] rx_data;
  reg data_ready;
  reg [3:0] state;
  reg [3:0] bit_spacing;

  // "next_bit" controls when the data sampling occurs
  // depending on how noisy the rxd is, different values might work better
  // with a clean connection, values from 8 to 11 work
  wire next_bit = (bit_spacing==10);
  always @(posedge clock_27mhz) begin
   if(state==0)
     bit_spacing <= 0;
else if(serial_clk)
     bit_spacing <= {bit_spacing[2:0] + 1} | {bit_spacing[3], 3'b000};
  end

  always @(posedge clock_27mhz) begin
if(serial_clk)
  case(state)
   4'b0000: if(rxd_bit_inv) state <= 4'b1000;  // start bit found?
   4'b1000: if(next_bit) state <= 4'b1001;  // bit 0
   4'b1001: if(next_bit) state <= 4'b1010;  // bit 1
   4'b1010: if(next_bit) state <= 4'b1011;  // bit 2
   4'b1011: if(next_bit) state <= 4'b1100;  // bit 3
   4'b1100: if(next_bit) state <= 4'b1101;  // bit 4
   4'b1101: if(next_bit) state <= 4'b1110;  // bit 5
   4'b1110: if(next_bit) state <= 4'b1111;  // bit 6
```

```
   4'b1111: if(next_bit) state <= 4'b0001;  // bit 7
   4'b0001: if(next_bit) state <= 4'b0000;  // stop bit
   default: state <= 4'b0000;
  endcase

data_ready <= (serial_clk && next_bit && state==4'b0001 && ~rxd_bit_inv);  // ready o

if(serial_clk && next_bit && state[3])
rx_data <= {~rxd_bit_inv, rx_data[7:1]};
  end

endmodule



//////////////////////////////////////////////////////////////////////
// TRANSMITTER MODULE
//
// modified source from
//  RS-232 TX module
//  (c) fpga4fun.com KNJN LLC - 2003, 2004, 2005

module async_transmitter(clock_27mhz, tx_start, tx_data, txd, tx_busy);
  input clock_27mhz, tx_start;
  input [7:0] tx_data;
  output txd, tx_busy;

  wire tx_busy;

  ////////////////////////////////////////////////////////////
  //connect baud generator
  wire serial_clk;
  baud_gen tx_clk(.clock_27mhz(clock_27mhz), .serial_clk(serial_clk),
      .enable(tx_busy), .rx_mode(1'b0) );


  ////////////////////////////////////////////////////////////
  reg txd;
  reg [3:0] state;
  wire [2:0] index;
  wire muxbit;
```

```
  assign index = state[2:0];
  assign muxbit = tx_data[index];
  assign tx_busy = (state!=0);

  always @(posedge clock_27mhz) begin
   txd <= (state<4) | (state[3] & muxbit);    // register the output (start, data, st

   case(state)
4'b0000: if(tx_start) state <= 4'b0001;
4'b0001: if(serial_clk) state <= 4'b0100;
4'b0100: if(serial_clk) state <= 4'b1000;  // start
4'b1000: if(serial_clk) state <= 4'b1001;  // bit 0
4'b1001: if(serial_clk) state <= 4'b1010;  // bit 1
4'b1010: if(serial_clk) state <= 4'b1011;  // bit 2
4'b1011: if(serial_clk) state <= 4'b1100;  // bit 3
4'b1100: if(serial_clk) state <= 4'b1101;  // bit 4
4'b1101: if(serial_clk) state <= 4'b1110;  // bit 5
4'b1110: if(serial_clk) state <= 4'b1111;  // bit 6
4'b1111: if(serial_clk) state <= 4'b0010;  // bit 7
4'b0010: if(serial_clk) state <= 4'b0011;  // stop1
4'b0011: if(serial_clk) state <= 4'b0000;  // stop2
default: if(serial_clk) state <= 4'b0000;
endcase
  end

endmodule


//////////////////////////////////////////////////////////////////
// BAUD GENERATOR
// modified source from
//  RS-232 RX module
//  (c) fpga4fun.com KNJN LLC - 2003, 2004, 2005

module baud_gen(clock_27mhz, serial_clk, enable, rx_mode);
  input clock_27mhz, enable, rx_mode;
  output serial_clk;

  parameter clkfreq = 27000000;  // 27 MHz
  parameter baudrate = 115200; //
  parameter baudrate_8 = 115200*8; // 8 times over sampling (when receiving)
```

```
  // Baud generator
  parameter acc_width = 16;
  reg [acc_width:0] accumulator;
  parameter inc = ((baudrate<<(acc_width-4))+(clkfreq>>5))/(clkfreq>>4);
  parameter inc8 = ((baudrate_8<<(acc_width-7))+(clkfreq>>8))/(clkfreq>>7);

  assign serial_clk = accumulator[acc_width];

  always @(posedge clock_27mhz) begin
   if(enable)
accumulator <= accumulator[acc_width-1:0] + inc;
else if (rx_mode)
accumulator <= accumulator[acc_width-1:0] + inc8;
  end
endmodule
```

K simpleAudioCodec.v

```
module simpleAudioCodec(clk_27mhz, input_data, output_data, ready,
                      ac97_ready, from_ac97_data, to_ac97_data);
    input clk_27mhz;
    input [7:0] input_data;
    input ac97_ready;
    input [7:0] from_ac97_data;

    output ready;
    output [7:0] output_data;
    output [7:0] to_ac97_data;

    //wire clk_27mhz;
    //wire [7:0] output_data;
    //wire ready1;  // TODO: figure out ready signals
    //wire ready2;
    //wire ac97_ready;
    //wire [7:0] from_ac97_data;
    //wire [7:0] to_ac97_data;

    //simpleAudioEncoder(clk_27mhz, output_data, ready1, ac97_ready, from_ac97_data);
    //simpleAudioDecoder(clk_27mhz, input_data, ready2, ac97_ready, to_ac97_data);

    reg [7:0] output_data = 0;
    reg [10:0] s1,s2,s3;
```

```
    reg ready = 0;
    reg [7:0] to_ac97_data = 0;
    reg [2:0] count = 0;
    reg [8:0] count2 = 0;
    reg ready_old,new_frame = 0;


    always @ (posedge clk_27mhz) begin
new_frame <= ac97_ready & ~ready_old;
ready_old <= ac97_ready;


    if (new_frame) begin
  count <= count + 1;
case (count)
0: s3 <= (s2+s2+s2+s2+s2+s2+s2+s2);
1: s3 <= (s2+s2+s2+s2+s2+s2+s2+s1);
2: s3 <= (s2+s2+s2+s2+s2+s2+s1+s1);
3: s3 <= (s2+s2+s2+s2+s2+s1+s1+s1);
4: s3 <= (s2+s2+s2+s2+s1+s1+s1+s1);
5: s3 <= (s2+s2+s2+s1+s1+s1+s1+s1);
6: s3 <= (s2+s2+s1+s1+s1+s1+s1+s1);
7: s3 <= (s2+s1+s1+s1+s1+s1+s1+s1);
default: s3 <= 0;
endcase
to_ac97_data <= s3[10:3];
  if (count==7)
begin
     output_data <= from_ac97_data;
s1[7:0]<=input_data;
s1[10:8]<={3{input_data[7]}};
s2<=s1;
end
    if (count == 1) ready <= 1;
         else begin
            ready <= 0;
         end
      end
 end
endmodule
```

L SlipRec.v

```
module slipRec(addr_out,data_out,write,write_addr,write_length,write_done,wea,clk,rx_

output write;
output wea;  //write goes to ip handler, wea to the ram
output [7:0] data_out; //data to ram
output [16:0] write_addr;
output [16:0] write_length; //info for the ip handler
output [16:0] addr_out; //address to the ram
input write_done; //response from ip handler
input clk;
input rx_ready;
input [7:0] data_in;
//output [2:0] state;

  parameter [7:0] END = 192;
  parameter [7:0] ESC = 219;
  parameter [7:0] ESC_END = 220;
  parameter [7:0] ESC_ESC = 221;

reg write;
reg wea;  //write goes to ip handler, wea to the ram
reg [7:0] data_out; //data to ram
reg [16:0] write_addr=0;
reg [16:0] write_length=0;
reg [16:0] length = 0; //info for the ip handler
reg [16:0] addr_out = 0; //address to the ram
reg received = 0;
//reg [16:0] cur_addr = 0;



parameter S_Idle = 0;
parameter S_Rec_END = 1;
parameter S_Rec_Done = 2;
parameter S_Rec_ESC = 3;
parameter S_Rec_ESC2 = 4;
parameter S_Recing = 5;
parameter S_Rec = 6;
parameter S_Done = 7;
```

```
reg [2:0] state = S_Idle;

  always @ (posedge clk)
   begin


case (state)
S_Idle: begin
write <= 0;
wea <= 0;
if (rx_ready) state <= S_Rec;

end

S_Done: begin
  length <= length + 1;
  addr_out <= addr_out + 1;
wea <= 0;
state <= S_Idle;
end

S_Rec: begin
if (data_in==END) state <= S_Rec_END;
else if (data_in==ESC) state <= S_Rec_ESC;
else begin
received <= 1;
  data_out <= data_in;
wea <= 1;
  state <= S_Done;
  if (~received) write_addr <= addr_out;
  end
end


S_Rec_END: begin
if (~received) state <= S_Idle;
else begin
write_length <= length;
write <= 1;
state <= S_Rec_Done;
end
```

```
end
S_Rec_Done: begin
length <= 0;
received <= 0;
state <= S_Idle;
end


S_Rec_ESC: begin
    write <= 0;
wea <= 0;
if (rx_ready) state <= S_Rec_ESC2;
end

S_Rec_ESC2: begin
if (data_in==ESC_ESC) data_out <= ESC;
if (data_in==ESC_END) data_out <= END;
wea <= 1;
state <= S_Done;

end

default:   state <= S_Idle;
endcase
end



endmodule
```

M SlipSend.v

```
module slipSend(addr_in,data_in,read,read_addr,read_length,read_done,clk,tx_ready,tx_

input read; //signal from ip handler
input [7:0] data_in; //data from ram
input [15:0] read_addr,read_length;  //info from handler
output [15:0] addr_in, length; //address to ram
output read_done; //signal handler when done

//output [3:0] state;
```

```
input clk;
output tx_ready;
output [7:0]data_out;
input tx_busy;

reg [15:0] addr_in = 0; //address to ram
reg read_done = 0; //signal handler when done
reg tx_ready = 0;
reg [15:0] length = 2000;
reg [2:0] count = 0;

reg [7:0] data_out = 0;



     parameter [7:0] END = 192;
     parameter [7:0] ESC = 219;
     parameter [7:0] ESC_END = 220;
     parameter [7:0] ESC_ESC = 221;

parameter S_Idle = 0;
parameter S_Send_END = 1;
parameter S_Sending = 2;
parameter S_Send_ESC_ESC = 3;
parameter S_Send_ESC_END = 4;
parameter S_Done = 5;
parameter S_Send = 6;
parameter S_Send_ESC_ESC2 = 7;
parameter S_Send_ESC_END2 = 8;
parameter S_Send_END2 = 9;
parameter S_Sending2 = 10;

reg [3:0] state = S_Idle;

  always @ (posedge clk)
   begin
case (state)
S_Idle: begin
tx_ready <= 0;
read_done <= 0;
if (read)
begin
```

```
addr_in <= read_addr;
length <= read_length;
state <= S_Send_END;
data_out <= END;
end
end
S_Send_END: begin
 tx_ready <= 1;
 state <= S_Sending;
end
S_Send: begin
if (data_in==END) state <= S_Send_ESC_END;
else if (data_in==ESC) state <= S_Send_ESC_ESC;
else begin
 data_out <= data_in;
 tx_ready <= 1;
 length <= length - 1;
 addr_in <= addr_in + 1;
 state <= S_Sending;
 end
end


S_Send_ESC_END: begin
 data_out <= ESC;
 tx_ready <= 1;
 length <= length - 1;
 addr_in <= addr_in + 1;
 state <= S_Send_ESC_END2;

end
S_Send_ESC_END2: begin
tx_ready <= 0;
if (~tx_busy) begin
  data_out <= ESC_END;
  tx_ready <= 1;
  state <= S_Sending;
 end
end


S_Send_ESC_ESC: begin
```

```
 data_out <= ESC;
 tx_ready <= 1;
 length <= length - 1;
 addr_in <= addr_in + 1;
 state <= S_Send_ESC_ESC2;

end
S_Send_ESC_ESC2: begin
tx_ready <= 0;
if (~tx_busy) begin
  data_out <= ESC_ESC;
  tx_ready <= 1;
  state <= S_Sending;
 end
end


S_Sending: begin
count <= count + 1;
if (count==7) state <= S_Sending2;
end

S_Sending2: begin
tx_ready <= 0;
if (~tx_busy) begin
if (length==0) state <= S_Send_END2;
else state <= S_Send;
end
end
S_Send_END2: begin
 data_out <= END;
 tx_ready <= 1;
 state <= S_Done;
end
S_Done: begin
tx_ready <= 0;
read_done <= 1;
state <= S_Idle;
end
default:   state <= S_Idle;
endcase
end
```

```
endmodule
```

N synchronize.v

```
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
```

O tigonProtocolDecoder.v

```
module tigonProtocolDecoder(clk,tigon_data_out,
                      ip_data_in, ip_ready,
                      ip_data_available,
                      play,
                      to_audio_codec,
                      codec_ready,
                      ip_new_packet);

    input clk;
    output [39:0] tigon_data_out;
    input [7:0] ip_data_in;
    input ip_data_available;
    input play;
    input codec_ready;
    input ip_new_packet;

    output [7:0] to_audio_codec;
    output ip_ready;
```

```
reg [39:0] tigon_data_out = 0;
reg [39:0] tigon_data_out_temp = 0;
reg [7:0] to_audio_codec = 0;
reg ip_ready = 0;

parameter WAIT = 0;
parameter READ_LIGON_BYTES = 1;
parameter READ_AUDIO_DATA = 2;

reg [1:0] state = READ_LIGON_BYTES;
reg [3:0] header_byte = 1;

always @ (posedge clk) begin
  case (state)
     WAIT: begin
        if (ip_new_packet) begin
           state <= READ_LIGON_BYTES;
        end
     end
     READ_LIGON_BYTES: begin
        if (ip_data_available) begin
         case (header_byte)
        1: begin
     header_byte <= header_byte + 1;
                 ip_ready <= 1;
     end
        2: begin
     header_byte <= header_byte + 1;
                 tigon_data_out_temp[39:32] <= ip_data_in;
                 ip_ready <= 0;
     end
        3: begin
     header_byte <= header_byte + 1;
                 ip_ready <= 1;
     end
        4: begin
     header_byte <= header_byte + 1;
                 tigon_data_out_temp[31:24] <= ip_data_in;
                 ip_ready <= 0;
     end
        5: begin
```

```
header_byte <= header_byte + 1;
            ip_ready <= 1;
end
   6: begin
header_byte <= header_byte + 1;
            tigon_data_out_temp[23:16] <= ip_data_in;
            ip_ready <= 0;
end
   7: begin
header_byte <= header_byte + 1;
            ip_ready <= 1;
end
   8: begin
header_byte <= header_byte + 1;
            tigon_data_out_temp[15:8] <= ip_data_in;
            ip_ready <= 0;
end
   9: begin
header_byte <= header_byte + 1;
            ip_ready <= 1;
end
   10: begin
header_byte <= header_byte + 1;
            tigon_data_out_temp[7:0] <= ip_data_in;
            ip_ready <= 0;
end
         11:   begin
            header_byte <= 1;
            tigon_data_out <= tigon_data_out_temp;
            state <= READ_AUDIO_DATA;
         end
         default: header_byte <= 1;
      endcase
      end
   end

READ_AUDIO_DATA: begin
   if (play) begin
      if (ip_new_packet) begin
         state <= READ_LIGON_BYTES;
         ip_ready <= 0;
      end
```

```
               else if (codec_ready & ip_data_available) begin
                  to_audio_codec <= ip_data_in;
                  ip_ready <= 1;
               end
               else begin
                  ip_ready <= 0;
               end
            end
            else begin
               state <= WAIT;
            end
         end
      endcase
   end
endmodule
```

P tigonProtocolEncoder.v

```
module tigonProtocolEncoder(clk,tigon_data_in,
                     ip_data_out, ip_ready, ip_start_packet,
                     ip_ready_for_data,
                     send,
                     from_audio_codec,
                     codec_ready);

   input clk;
   input [39:0] tigon_data_in;
   input ip_ready_for_data;
   input send;
   input [7:0] from_audio_codec;
   input codec_ready;

   output [7:0] ip_data_out;
   output ip_start_packet;
   output ip_ready;

   reg [7:0] ip_data_out = 0;
   reg ip_start_packet = 0;
   reg ip_ready = 0;

   parameter AUDIO_BYTES_PER_PACKET = 300;
```

```
reg [15:0] bytes_left_in_packet = AUDIO_BYTES_PER_PACKET;

parameter WAIT = 0;
parameter WRITE_LIGON_BYTES = 1;
parameter WRITE_AUDIO_DATA = 2;

reg [1:0] state = WAIT;
reg [3:0] header_byte = 0;

reg old_codec_ready = 0;
reg codec_ready_sync = 0;
always @ (posedge clk) begin
  old_codec_ready <= codec_ready;
  codec_ready_sync <= codec_ready & ~old_codec_ready;
end

always @ (posedge clk) begin
  case (state)
     WAIT: begin
        if (send) begin
           state <= WRITE_LIGON_BYTES;
        end
     end

     WRITE_LIGON_BYTES: begin
        if (ip_ready_for_data) begin
         case (header_byte)
              0: begin
                    ip_start_packet <= 1;
                    header_byte <= header_byte + 1;
                 end
              1: begin
                    header_byte <= header_byte + 1;
                 end
        2: begin
                 ip_start_packet <= 0;
     header_byte <= header_byte + 1;
                 ip_data_out <= tigon_data_in[39:32];
                 ip_ready <= 0;
     end
        3: begin
     header_byte <= header_byte + 1;
```

```
            ip_ready <= 1;
end
   4: begin
header_byte <= header_byte + 1;
            ip_data_out <= tigon_data_in[31:24];
            ip_ready <= 0;
end
   5: begin
header_byte <= header_byte + 1;
            ip_ready <= 1;
end
   6: begin
header_byte <= header_byte + 1;
            ip_data_out <= tigon_data_in[23:16];
            ip_ready <= 0;
end
   7: begin
header_byte <= header_byte + 1;
            ip_ready <= 1;
end
   8: begin
header_byte <= header_byte + 1;
            ip_data_out <= tigon_data_in[15:8];
            ip_ready <= 0;
end
   9: begin
header_byte <= header_byte + 1;
            ip_ready <= 1;
end
   10: begin
header_byte <= header_byte + 1;
            ip_data_out <= tigon_data_in[7:0];
            ip_ready <= 0;
end
   11: begin
header_byte <= header_byte + 1;
            ip_ready <= 1;
end
         12:   begin
            header_byte <= 0;
            ip_ready <= 0;
            state <= WRITE_AUDIO_DATA;
```

```
                end
                default: header_byte <= 0;
             endcase
             end
          end

       WRITE_AUDIO_DATA: begin
          if (send) begin
             if (bytes_left_in_packet == 0) begin
                bytes_left_in_packet <= AUDIO_BYTES_PER_PACKET;
                state <= WRITE_LIGON_BYTES;
                ip_ready <= 0;
             end
             else begin
                if (codec_ready_sync) begin
                   ip_ready <= 1;
                   ip_data_out <= from_audio_codec;
                   bytes_left_in_packet <= bytes_left_in_packet - 1;
                end
                else begin
                   ip_ready <= 0;
                end
             end
          end
       end
     endcase
   end
endmodule
```