UNIVERSITY OF CALIFORNIA AT BERKELEY
COLLEGE OF ENGINEERING
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Lab 5
Network Audio

1.0 Motivation

In order to better acquaint you with a more refined design process, as well as provide an interesting and yet simple lab, this week you will be designing a circuit to decode Ethernet frames containing PCM audio data.

The cornerstones of the EECS150 class are registers and gates, which can be built into incredibly powerful structures like counters, shift-registers and the like. However all too often the temptation is to use a more general Finite State Machine construction where a simpler more regular component like a counter could do a better job with fewer bugs and less code.

2.0 Introduction

As described in section 1.0 Motivation above, the **primary goal of this lab is to parse Ethernet packets**, filtering them based on address and type information and then send the PCM audio data payload on to be played **without using a general FSM**. The idea in this lab is to acquaint you with more powerful design techniques which will not only save you design and coding time, but will avoid many bugs, while simultaneously making your circuits smaller, easier to understand and more efficient in terms of power and space.

Part of the point of this lab is also to expose you to the kind of interesting circuits you can build in this class, and the kind of circuits you will build for your project. To make this interesting, we have set up a computer in the lab which will be broadcasting music. The audio stream is being broadcast directly from a PC running Microsoft® WindowsXP® and Nullsoft's® Winamp5® with a custom made output plugin which takes PCM audio data as decoded from MP3 files and sends it out in raw Ethernet packets.

DURING THIS LAB YOU MAY NOT USE FINITE STATE MACHINES OR BEHAVIORAL VERILOG. YOU MUST USE ASSIGNS STATEMENTS FOR ALL COMBINATIONAL LOGIC AND THE COUNTER.V OR REGISTER.V MODULES FOR ALL SEQUENTIAL ELEMENTS.

2.1 Networking Basics

Across the world most LANs (Local Area Networks) are currently built on the 100BaseTX Ethenet standard, because it uses inexpensive cable, the hardware is cheap (and backward compatible) and it is both fast and reliable over relatively long distances (250m).

Ethernet is a **packet switched network** as opposed to a **circuit switched network** like the telephone system. This means that unlike a phone call, which establishes some kind of connection between you and the person you call, Ethernet simply takes a bunch of short messages, or **packets** and shuffles them around. Normally this has all kinds of implications, like the fact that packets can be **lost**, **corrupted** and can even arrive **out of order**. However for this checkpoint we wont worry about these problems, since they are unlikely to be noticeable on our small controlled network.

2.2 Network Organization

Figure 1 below is a simple diagram of the test network set up in 125 Cory for this lab. Notice that the **green (0) network** is fully connected to all boards and will carry the **broadcast audio stream** which you will be receiving.

Note that part of your debugging process should be to make sure that you are using the correct network and that all the right cables are connected. There is a **green LED** near each **RJ45 network jack** will be appear lit when **the network port is connected**. The **amber LED** will blink or turn on when there is **network traffic** on that port. These LEDs are an invaluable and simple way to debug basic connection problems.

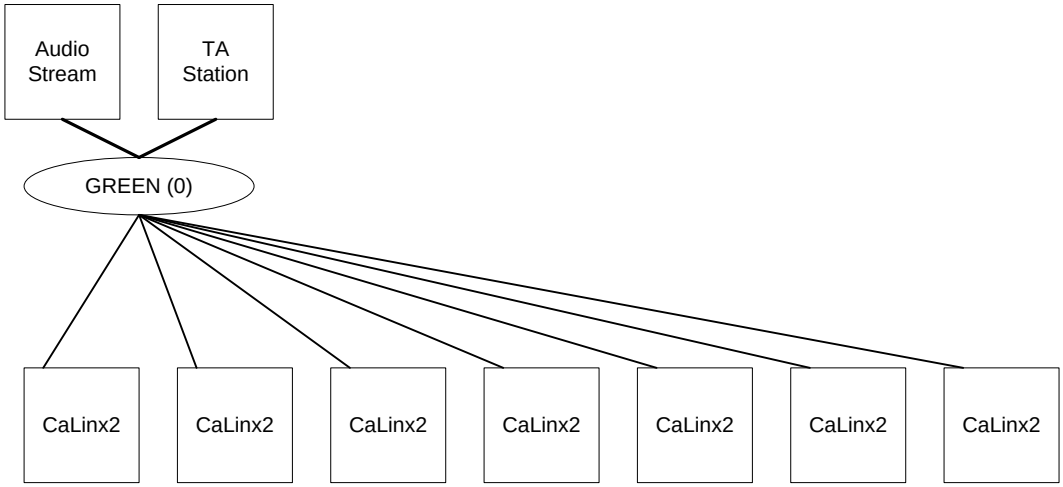

Figure 1: Network Organization for Lab #5

2.3 Network Protocol Layers

Remember in **2.1 Networking Basics** how we said that Ethernet was **packet switched** and that it might do things like reorder and lose packets? Well most applications can't tolerate that. Imagine if your **e-mail** just sometimes got **broken up** or **reordered**.

To prevent problems like that CS majors build **protocol stacks** such as the one shown in **figure 2**. In this model, Ethernet would be the **physical layer** upon which a **data link layer** depends. **TCP/IP** which adds a lot of features (like protection against missing or reordered packets) would be **layers 3 and 4** the **network and transport layers**. In essence each layer depends upon, and expands the capabilities of the layer below it.

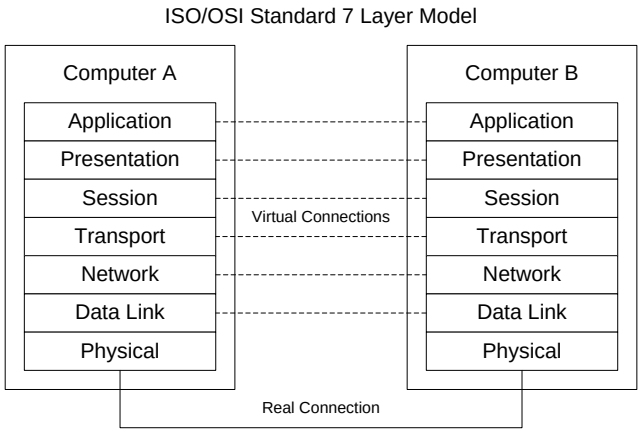

Figure 2: ISO/OSI Standard Protocol Layers

Looking at **figure 2** you can see it might be a little much to have to build for just one lab. So instead we're going to use a **much simpler** model, as shown in **figure 2** below. In this model we have a **Windows PC running Winamp5** connected to an **LXT975 Ethernet Phy** chip **directly** or through a **network switch**.

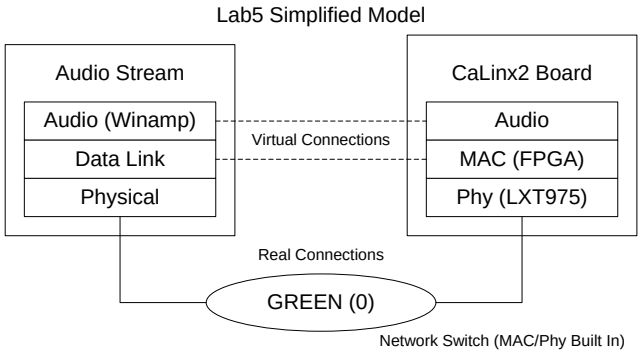

Figure 3: Simplified Lab5 Protocol Layers

You will also build the **Audio** layer, which isn't really a normal networking layer. This is just our version of the **application layer** from **figure 2**. Our application is sending audio over Ethernet, so at the top of our **protocol stack** we have an **audio layer** circuit which will **decode and filter the incoming packets** into a **32bit FIFO**. The audio layer doesn't have to know **or care** about the MAC and Phy layer magic taking place behind the scenes. **That's the whole point of protocol layers, you can build the simple audio layer without knowing anything about the circuits we're providing.**

2.4 Ethernet Packets

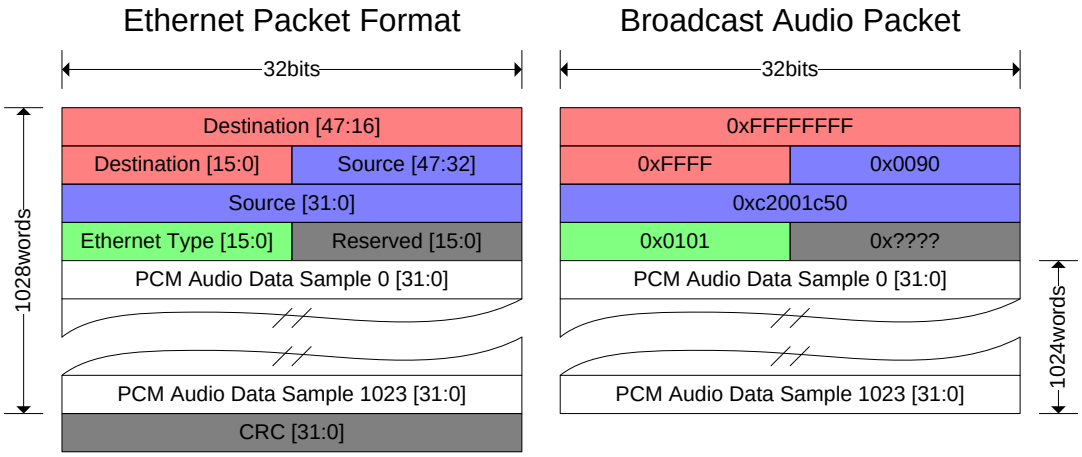

Figure 4: Ethernet Packet Formats

Figure 4 above shows the basic format of the Ethernet packets you will be working with. Aside from **1024x 32bit words of PCM Audio Data** each packet **must have** a **header** and, if it is to go through the network switches a CRC, which you will need to remove from the packet.

In order to get the packets to their proper destination **and so that the destination knows the packet is meant for it** each packet starts with a **destination address** field. (Think carefully about why the destination should be first) After that is a similar **source address** field identifying the sender of the packet. The final 16bits of the actual Ethernet header consists of an **Ethernet Type** field indicating what the data in the body of the packet is, so that different upper layer protocols can share a piece of Ethernet without interfering with each other. Because our PCM Data is **32bits wide** we insert a **16bit padding field** marked **reserved**. After that are the **1024x 32bit words of PCM Audio Data**, bringing the total packet length to **1028x 32bit words**.

The **right hand side of figure 4** above shows the specific values you should **filter packets with**. Basically **any packet which matches the values (minus the source field)** shown in that diagram should have its **1024x 32bit words of audio** put into your **asynchronous FIFO**. **Do not worry what the source bits are for this lab**.

3.0 Prelab

Please make sure to complete the prelab before you attend your lab section. **You will not be able to finish this lab in 3hrs otherwise!**

1. **Read this handout thoroughly**. Pay particular attention to section **4.0 Lab Procedure** as it describes what you will be doing in detail.
    a. Make sure you **understand how the fifo_async32 module works**
2. **Examine the Verilog** provided for this weeks lab.
    a. You should become intimately familiar with the **Lab5Testbench.v** file as **you will need to debug with it.**
    b. Watch the ChipScope tutorial, examine the example verilog
        i. http://www-inst.eecs.berkeley.edu/~cs150/fa04/Documents.htm#Tutorials

   c. You should understand **how to use the fifo_async32.v** module, though you do not need to understand how it works.
3. **Write your Verilog ahead of time**.
   a. **Eth2Audio.v** will be a relatively simple module, you can build it from **fifo_async32.v**, **Counter.v** and **Register.v** but you should build it ahead of time.
4. You will need the **entire 3hr lab**!
   a. You will need to test and debug both your verilog and ours.
   b. **You will be asked to use the ChipScope for the first time.**
      i. Its easy to use, but you will need to play with it to learn it

4.0 Lab Procedure

Remember to **manage your Verilog, projects and folders well**. Doing a poor job of managing your files can cost you **hours of rewriting code**, if you accidentally delete your files.

DURING THIS LAB YOU MAY NOT USE FINITE STATE MACHINES OR BEHAVIORAL VERILOG. YOU MUST USE ASSIGNS STATEMENTS FOR ALL COMBINATIONAL LOGIC AND THE COUNTER.V OR REGISTER.V MODULES FOR ALL SEQUENTIAL ELEMENTS.

Below are sections describing the various modules you may work with for this lab. Note that you will need at least one instance of each of these modules.

4.1 ChipScope

For this part of the lab, you will need to use Chipscope to ensure the functionality of your Eth2Audio.v works properly. Chipscope is an invaluable tool in helping to debug problems that occur on the board. You will need to look at the wires going into and out of your fifo to ensure that the values are being sent at the right time to the asynchronous fifo. It will be helpful to trigger on specific wires such as when you are reading or writing data to your fifo in order to determine what values are being written to them in chipscope.

4.1.1 Working with ChipScope

ChipScope is an embedded, software based logic analyzer. By inserting an "integrated controller core" (**icon**) and an "integrated logic analyzer" (**ila**) into your design and connecting them properly, you can monitor any or all of the signals in your design. ChipScope provides you with a convenient software based interface for controlling the "integrated logic analyzer," including setting the triggering options and viewing the waveforms.

There are six main steps to using ChipScope, as detailed below.

1. Generate an "integrated controller core" or **icon**
2. Generate one or maybe more "integrated logic analyzers" or **ila**s
3. Connect the **ila**s to the **icon** and make all of these modules part of your design.

4. Synthesize, and implement your design (including the **icon** and **ila**) as normal.
5. Program the CaLinx board
6. Run the ChipScope software to access and use the **ila**s (the ChipScope software requires the **icon** to gain access to the **ila**s)

For a more detailed ChipScope tutorial, please refer to the documents page of the website (http://www-inst.eecs.berkeley.edu/~cs150/sp06/Documents.php#Tutorials) where you will find a ChipScope tutorial document.

4.2 Register.v

This is most likely the simplest module you will ever work with. The primary reason to make this a separate module, is to save some time and allow us to write cleaner verilog that is easier to read.

Signal	Width	Dir	Description
`Clock`	1	I	The `Clock` signal
`Reset`	1	I	`Reset` the register to all 1'b0s
`Set`	1	I	`Set` the register to all 1'b1s
`Enable`	1	I	Enable the register to load a new value from `In`
`In`	`width`	I	New value to load when `Enable` is high
`Out`	`width`	O	`Output` value held in the register

Table 1: Port Specification for Register.v

In order to make this module more useful, it can **vary in width using a Verilog parameter**: a constant which is known during synthesis, much like a preprocessor `define` in C/C++ or a `static` member in Java. Below is an example instantiation of an 32-bit register.

```
1.Register     WordReg(  .Clock(   Clock),
2.                       .Reset(   Reset),
3.                       .Set(     1'b0),
4.                       .Enable(  NextWordValid),
5.                       .In(      NextWord),
6.                       .Out(     Word));
7.defparam     WordReg.width =     11;
```

Notice on line 7 where we set the `width` of the `register`. This statement, called a "`defparam`" says that the simulation and synthesis tools should set the parameter called "`width`" on the module instantiation called "`WordReg`" to "`32`". Notice that 32 could be any expression which evaluates to a synthesis time constant.

4.3 Counter.v

This is the second simplest module you will ever work with, its only slightly more complicated than Register.v, it is a counter. This module represents a complete up-counter with synchronous load which will roll over when it reaches its upper limit.

Signal	Width	Dir	Description
Clock	1	I	The Clock signal
Reset	1	I	Reset the register to all 1'b0s
Set	1	I	Set the register to all 1'b1s
Load	1	I	Load the value on the In input into the counter
Enable	1	I	Enable the counter to increment by one
In	width	I	New value to load when Load is high
Count	width	O	Output value held in the counter

Table 2: Port Specification for Counter.v

Like the register.v module in **section 4.2 Register.v** the counter module **has a width parameter** which can be used to create a counter with any bit-width desired.

4.4 fifo_async32.v

In order to cross from the EthernetClock domain to the AudioClock domain, you will need a **special circuit designed to work with the two clocks** not running at the same rate, and are therefore not related in any way.

The module you will have is called an **asynchronous FIFO**. It is a simple, **two interface module**, you write into one and read from the other. The way this module assists is that **the read and write interfaces can have separate clock signals**. Thus in this assignment you will **write to the FIFO using the** EthernetClock and re**ad from it using the** AudioClock.

MAKE SURE YOU DO NOT USE SIGNALS FROM THE WRONG CLOCK DOMAIN, TRYING TO USE AN RD_* SIGNAL IN THE ETHERETCLOCK DOMAIN OR VICE VERSA WILL CAUSE YOUR CIRCUIT TO FAIL RANDOMLY ON THE CALINX2 BOARD.

Signal	Width	Dir	Clock	Description
din	32	I	wr	Data input bus
wr_en	1	I	wr	Write the value on din into the FIFO on the clock
wr_clk	1	I	wr	The write clock signal (EthernetClock)
rd_en	1	I	rd	Read enable, dout will be valid after the clock
rd_clk	1	I	rd	The read clock signal (AudioClock)
ainit	1	I	none	Asynchronous reset, will empty the FIFO
dout	32	O	rd	Data output bus, valid next cycle after rd_en
full	1	O	wr	Indicates that the FIFO is currently full
empty	1	O	rd	Indicates that the FIFO is currently empty
wr_count	10	O	wr	Indicates how many words are in the FIFO
rd_count	10	O	rd	Indicates how many words are in the FIFO
rd_ack	1	O	rd	Asserted for one the cycle when dout is valid, right after rd_en has been signalled

Table 3: Port Specification for fifo_async32.v

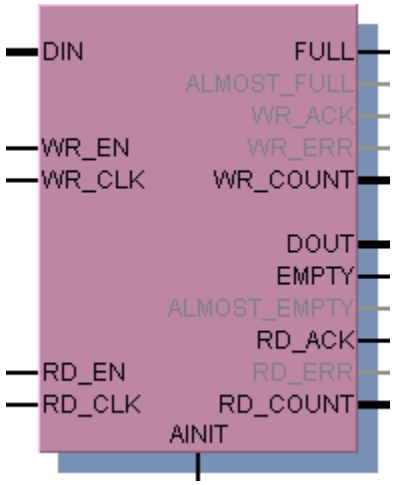

Figure 5: fifo_async32 Block Symbol

Your job with `Eth2Audio` is to put valid audio data into the asynchronous fifo whenever possible and to read it out when needed. **The biggest challenge will be to generate the handshaking signals needed to support the I/O spec for** `Eth2Audio`, not to mention gating `wr_en` based on where in the packet we currently are and whether it is a good packet.

4.5 Eth2Audio.v

This module is the one you will have to write for this lab. This module, like the fifo_async32 module, has two separate interfaces: one for incoming packets and one for outgoing audio data. Both of these interfaces use simple 32-bit data busses and a request/valid signaling protocol. Table 4 below shows the I/O specification and figure 6 shows the block symbol for this module.

Signal	Width	Dir	Description
`DIn`	32	I	Packet data input bus
`InValid`	1	I	Indicates that `DIn` is valid on this cycle
`InPacketValid`	1	I	Indicates the end of a packet with a valid CRC
`InPacketInvalid`	1	I	Indicates the end of a packet with an invalid CRC
`EthernetClock`	1	I	25MHz Ethernet Clock
`EthernetReset`	1	I	`EthernetClock` synchronous reset
`AudioClock`	1	I	12.288MHz Audio Clock
`AudioReset`	1	I	`AudioClock` synchronous reset
`DOut`	32	O	Audio data output bus
`OutRequest`	1	I	Request for a new audio word output
`OutValid`	1	O	Indicates that DOut is currently valid

Table 4: Port Specification for Eth2Audio.v

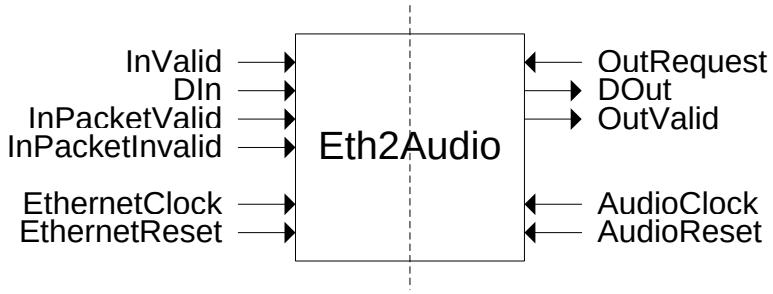

Figure 6: Eth2Audio Block Symbol

This module should accept a new 32-bit word **at most one per cycle** when `InValid` is `1'b1`. If it is a word in the header of a packet, judging by the **word counter**, then you should set or clear a r**egister indicating whether the packet is valid or not**. Once the header is finished the next **0 to 1024 words** should be **written into the async_fifo32, but only if the header was valid**. **After 1024 words of audio data**, you should **throw out the remainder of the packet**. When the packet is **finished**, the filtering circuit should **reset to its initial state**, but the **FIFO should not**.

On the output side, you will need to **design a simple circuit to create the** `OutValid` signal based on the `OutRequest` signal and the state of the `rd_*` interface to the FIFO. Note that within the Request/Valid signaling protocol, `DOut` and `OutValid` **cannot ever change except on a rising edge when** `OutRequest` is `1'b1`, which **may happen on every cycle or may happen very rarely**. `OutValid` should be `1'b1` when `DOut` is valid. Note that the `dout` output from the fifo_async32 module **will not change ever except on the rising edge of a clock cycle when** `rd_en` is `1'b1`.

5.0 Lab5 Checkoff

Name: _____ Name: _____
Section: _____

I	Audio on Board	_____ (35%)
II	Clean, FSM free Verilog	_____ (35%)
III	ChipScope view of Eth2Audio and ability to Trigger	_____ (30%)

IV	Total:	_____
V	TA:	_____
VI	Hours Spent:	_____

RevA – 9/13/2004	Greg Gibeling	Created a new lab Based on the old Spring 2004 Checkpoint3