EECS150: Spring 2010 Project Checkpoint 2, Ethernet & Serial I/O Interfaces

UC Berkeley College of Engineering
Department of Electrical Engineering and Computer Science

Revision D

1 Time Table

ASSIGNED	Monday, March 29^{th}
DUE	Week 13: April $13^{th} - 15^{th}$, demo during your lab section

2 Motivation

In this checkpoint, you will implement adapters between UART and Ethernet devices so that your MIPS CPU can communicate with a front-end PC.

The purpose of the UART interface is to send small amounts of information (single bytes) from PC to FPGA and back. You already implemented the UART adapter in lab 5. At the end of this checkpoint, you will be provided "boot monitor" MIPS assembly code that will act as a basic operating system for your processor. Thus, if you successfully completed lab 5, the work that you have to do for the UART part of this checkpoint has already been completed. The boot monitor coupled with the UART adapter will allow you to control your processor through a command-line interface.

Ethernet is designed to move large amounts of data between your processor and a PC. In the context of this project, you will be using Ethernet to transfer data and instructions (user programs). All Ethernet communication will be directly between your FPGA and a PC. The FPGA will never be connected to a larger network. Furthermore, all transfers will be over the TFTP protocol. The boot monitor that we provide you will come with a compliant TFTP client that will be able to communicate with a TFTP server running on your PC. Your task is to implement the "Ethernet adapter" - the Ethernet counterpart to the UART adapter.

3 Adapter Interfaces

The Ethernet and UART adapters have the same interface and both expose the same flavor of `Control` and `Data` registers. You should already be familiar with these from lab 5, and the addresses used for both UART and Ethernet are shown in Table 1. The two addresses reserved for the PC MAC will be discussed further in Section 5.

The boot monitor will communicate to the UART and Ethernet using the same protocol as was described in lab 5. For more information, please revisit the lab 5 specification.

4 Ethernet/CPU Adapter

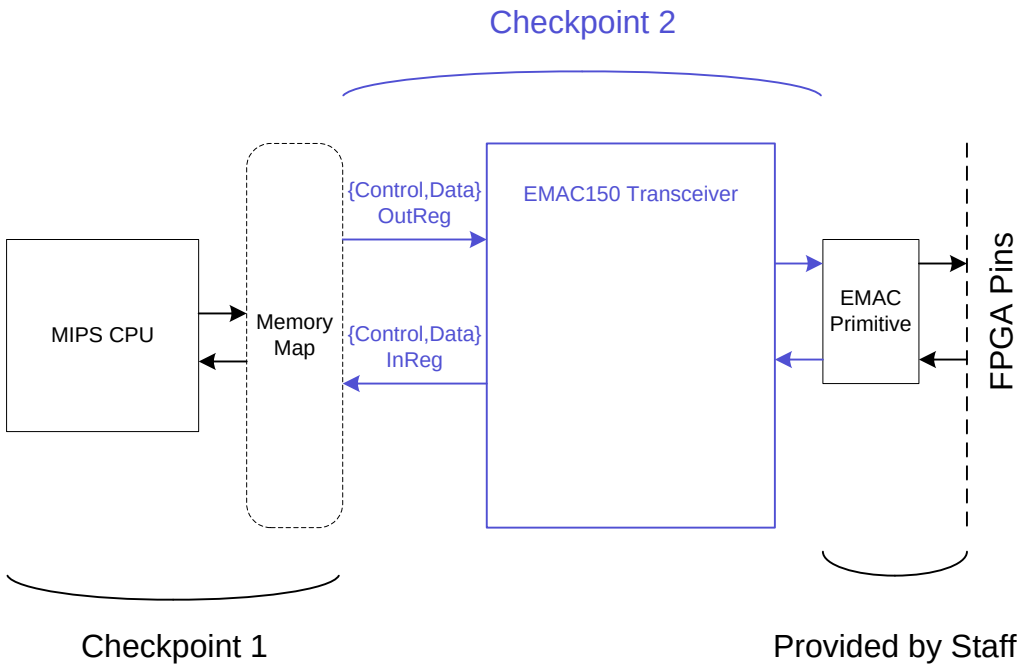
The path between Ethernet's physical interface and your CPU is shown in Figure 1. The link layer will be handled by the Xilinx Ethernet primitive `EMAC`. `EMAC` is a hard block built straight into the FPGA

Table 1 Ethernet & UART adapter memory map.

`0xffff0000`	R	Serial Interface (ControlInReg)
`0xffff0004`	R	Serial Interface (DataInReg)
`0xffff0008`	R	Serial Interface (ControlOutReg)
`0xffff000c`	W	Serial Interface (DataOutReg)
`0xffff0010`	R	Ethernet Interface (ControlInReg)
`0xffff0014`	R	Ethernet Interface (DataInReg)
`0xffff0018`	R	Ethernet Interface (ControlOutReg)
`0xffff001c`	W	Ethernet Interface (DataOutReg)

fabric.[1]

Figure 1 Ethernet datapath (you will implement everything colored blue).



We have instantiated an `EMAC` for you, and also connected it to the FPGA pins. You are to design everything from the `EMAC` to your processor. This is broken into two parts (for the input and output paths): header filtering/parsing and buffering. As with the UART/CPU Adapter, the CPU will receive and send one byte of Ethernet data at a time.

4.1 Header Filtering/Parsing

As discussed in class, Ethernet packets are constructed as layers of protocols, each with a header and payload. The first header is always the Ethernet (or hardware) header. The other headers used in TFTP are IP, UDP and TFTP. The TFTP client that we provide you knows how to parse IP, UDP and TFTP headers. You must process the hardware header in the following ways:

1. Remove the hardware header from incoming packets.
2. Add the hardware header to outgoing packets.

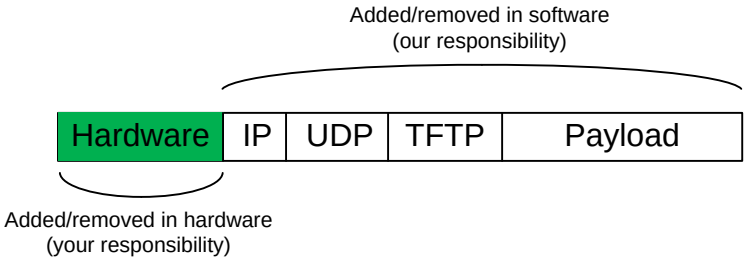
[1] As with block RAM and other elements on the FPGA, this is done because of how often Ethernet is used in user designs.

3. Ensure that packets not destined for the FPGA's MAC are not read by the CPU.

4. Ensure that non-IP packets are not read by the CPU.

The composition of headers is shown in Figure 2.

Figure 2 Headers in a TFTP packet.



4.1.1 Packet Filtering (`EMAC` to CPU)

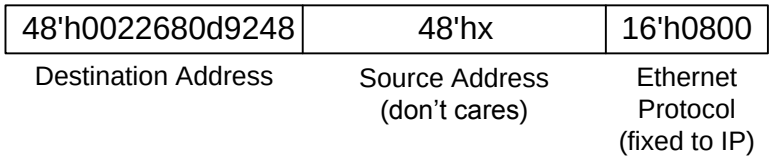
We are providing you with a MAC address for your FPGA[2]. Thus, 3 can be done by comparing this MAC with the hardware header's destination address for incoming packets. If the field doesn't match, you are to discard the entire packet before the CPU can read any word of it. *Hint:* You can either build logic for this part yourself, or use a parameter on the `EMAC`.

You are not required to filter the source MAC address as the PC has been configured to not forward packets from arbitrary sources to the FPGA.

Filtering non-IP packets (4) is done in much the same way as filtering the destination address, except through matching the **Ethernet Protocol** field of the hardware header.

Concretely speaking, the hardware header that you should pass to the CPU is shown in Figure 3.

Figure 3 The "correct" hardware header. Any packet without this header should be dropped.



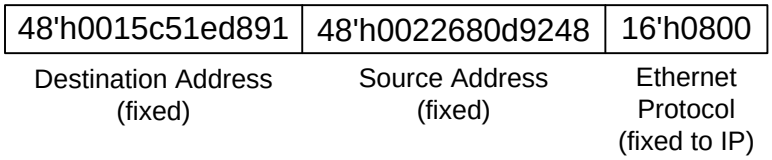
4.1.2 Header Construction (CPU to `EMAC`)

You don't have to worry about filtering packets from the CPU to the `EMAC` but do have to add the hardware header to each packet. Each header's source address will be the MAC assigned to your FPGA. The destination address will match the MAC of the Network Interface Card (NiC) on the PC that your FPGA is connected to. Under normal circumstances, this would be a unique 48 bit address for each NiC (and therefore for each PC). We have spoofed the MAC address on each PC's NiC, however. Consequently, the destination address that you have to add to outgoing packets is static just like the source address.

Concretely speaking, the hardware header that you should add to all outgoing packets is shown in Figure 4.

[2]All FPGAs will share the same MAC. For those who are interested: in truth, each FPGA is registered with a unique MAC address (check for a sticker under the XUPV5 board). We are setting every FPGA's MAC to the same address for simplicity. In general, this MAC address faking is called *spoofing*.

Figure 4 The outgoing hardware header.



4.1.3 Packet Buffering

The `EMAC` primitive is *feed-forward* by design and uses a clock specific to 100 $\frac{Mb}{s}$ Ethernet (a MII standard 25 Mhz clock). Feed-forward means without buffering. As soon as the `EMAC` receives a packet from off-chip, it passes the packet onto the user design immediately. Since the Control and Data register interface allows the CPU to take its time in performing a read, we need to buffer the packets that come in ourselves.

We want to read the packet's contents in order, so we will buffer data using a FIFO. Furthermore, as we need to cross from the MII clock domain to the CPU clock domain, we will use an asynchronous FIFO. Fortunately, Xilinx CoreGren allows you to specify parameters to and generate asynchronous FIFOs. Feel free to use this tool for this purpose for the entire checkpoint.

5 Ethernet Restrictions

Aside from packet filtering and buffering, we must perform additional operations to correctly follow the Ethernet protocol.

5.1 `EMAC` to CPU

When the `EMAC` passes a packet to the CPU, you must alert the CPU if that packet has a bad Ethernet CRC or if the receive side packet FIFO overflows.

A **bad CRC** means that some data in the packet was corrupted. If this happens, we will throw the packet out. Since we are only using TFTP, the server running on the PC will automatically resend the packet after some period of time (so we don't have to worry about sending a "retry" packet).

To simplify the hardware implementation, we will mark packets with a bad CRC using tags. The tag encoding is given in Table 2.

Table 2 Receive side tags.

`9'h1X1`	Start frame
`9'h1X2`	End frame (Good packet)
`9'h1X4`	End frame (Bad packet)
`9'h0XX`	Data

Each word read by the CPU on `DataInReg` has a tag and is 9 bits wide. The most significant bit indicates whether the low-order byte should be read by the CPU as data or as the start or end of a frame.

When a new packet comes in, your implementation must ensure that the first data it sees is the **Start frame**. All data in the payload will have a most significant (tag) bit of 0.

After the packet has been fully passed through the `EMAC`, the `EMAC` will assert one of two signals. The first indicates that a "good frame" has been received. The second indicates a "bad frame" (or bad CRC). In both cases, you are to add the appropriate tag word as shown in the above table. The TFTP client provided by the staff will throw away packets marked with the "bad" end tag.

For more information on when the "good"/"bad" frame signals are asserted by the `EMAC`, please read the `EMAC` documentation in Section 7.

Indicating when each frame starts is important if the FPGA CPU receives packets too quickly, can't process them fast enough, and suffers packet FIFO overflow. Try to convince yourself why this is necessary! To help you along, when a FIFO overflow occurs, the end of the last packet will not make it into the FIFO. If this happens, and then *another* packet arrives when the FIFO has space again, what will the data stream look like?

5.2 CPU to `EMAC`

Going from the CPU to the `EMAC` is simpler in that we don't have to worry about bad CRCs or packet overflow (why might this be the case?). Ethernet requires, however, that once a packet begins transmission over the physical medium, the datastream be continuous until the packet has been fully transmitted. Because of the `Control/Data OutReg` interface, there is no guarantee that the CPU will be able to satisfy this requirement. So, we will buffer an entire to-be-transmitted packet in the outbound FIFO and only start sending its contents when the CPU asserts an "end frame" tag.

Table 3 Transmit side tags.

`9'h1XX`	End frame
`9'h0XX`	Data

Transmit side tags are shown in Table 3. Like on the receive side, the special "end frame" word does not contain packet data. Unlike the receive side, however, you will only need to check the 9^{th} bit in the word to determine if the word has data or is an "end frame" word. This is because there is only one special kind of data word on the transmit side.[3]

6 Restrictions

You must abide by the following restrictions (concisely stated) for this checkpoint:

1. Implement the memory map interface (Section 3).
2. Filter incoming packets (Section 4.1.1).
3. Remove and add hardware headers (Sections 4.1 and 4.1.2).
4. Add and handle the receive and transmit side tags (Section 5.1 and 5.2).
5. Send continuous packets (Section 5.2).

Beyond these requirements, you are free to design this checkpoint as you see fit. The order in which you buffer data and process the hardware header is up to you. What will make for the most efficient implementation?

Keep in mind: the only type of communication we will be doing over Ethernet is TFTP. You can optimize your implementation given this assumption.

7 Additional Information

The following sources might be helpful throughout this checkpoint:

1. `EMAC` documentation (read the "Ethernet MAC Overview" section)
2. TFTP Protocol
3. Lab 5

[3] Some of you might be unconvinced as to why we don't just add 3 special bits to the top of each word on the receive side to encode start/end good/end bad words. The reason is that FPGAs efficiently implement 9 bit wide FIFOs and memories (8 bits of data and 1 bit of parity). (See Section 7 for more information.) Thus, we try as best as possible to fit into 9 bit wide datapaths.

Rev.	Name	Date	Description
D	Chris Fletcher	4/6/2010	Moved the "in SVN" deadline to the demo deadline.
C	Chris Fletcher	4/4/2010	Removed some misleading English in Section 4.1 that implied that the RX side needs to filter based on the source address.
B	Chris Fletcher	3/30/2010	Spoofed the PC MAC address and removed the PC MAC address registers in the memory map. Changed receive side tags so that the "end frame" word no longer contains valid data.
A	Chris Fletcher & John Wawrzynek	3/29/2010	Designed new checkpoint.