POPi
Final Project Report

Adam Lehenbauer
Alexander Robertson
Javier Coca
Yashket Gupta

Embedded Systems Design

May 09, 2006
# Table of Contents

1. Overview .................................................................................................................. 3

2. Hardware .................................................................................................................. 4

3. Software ..................................................................................................................... 8

4. Lessons Learned ....................................................................................................... 9

5. Appendix .................................................................................................................. 11

<table>
<thead>
<tr>
<th>File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EtherSend.c</td>
<td>11</td>
</tr>
<tr>
<td>main.c</td>
<td>14</td>
</tr>
<tr>
<td>JayCam.h</td>
<td>15</td>
</tr>
<tr>
<td>POP.c</td>
<td>16</td>
</tr>
<tr>
<td>POP.h</td>
<td>18</td>
</tr>
<tr>
<td>Terminal.c</td>
<td>19</td>
</tr>
<tr>
<td>Terminal.h</td>
<td>21</td>
</tr>
<tr>
<td>SRAM_Ethernet.vhd</td>
<td>22</td>
</tr>
<tr>
<td>TestBench.vhd</td>
<td>33</td>
</tr>
<tr>
<td>TCPstack.c</td>
<td>38</td>
</tr>
</tbody>
</table>
1. OVERVIEW

POPi project displays a message (SMS) in the video screen of the XESS board, sent from a cellular telephone to an e-mail address. The project was designed to take advantage of both hardware and software resources. In consequence the design was divided in two equally important parts.

The Hardware components of the project includes a peripheral which acts as a bridge between the SRAM and Ethernet Chip, as shown in figure Num. 1.

The POP project implements the TCP IP, Ethernet and the POP3 technology to read an email message send from a Mobile through a cellular network. The board connects to a remote server using the Ethernet chip, the POP program executes the POP3 commands to check and download messages present on the server.

The Bridge takes care of accesses to the Ethernet chip and the SRAM as the TCP/IP and the POP programs are executed from the SRAM the bridge needs to provide sufficient functionality to handle 8,16 and 32 bit access from the microblaze. The Video hardware is taken directly from lab2 with the difference that the data is read from the data block on the SRAM. The TCP/IP stack extracts the data from the email and the microblaze puts it to the video. All the communication between the FPGA and the peripherals will be hosted by the OPB.

Figure 1. Hardware block diagram.
The Software that the project will require is a very significant part of its functionality. The two major "subsystems," the video controller and the Ethernet controller, and the software is the communication between them. The SRAM will be transparent to the software and appear only as a base address.

![Software block diagram](image)

Figure 2. Software block diagram.

2. HARDWARE.

There are two major parts for the Hardware Part of our Design.

1. BRIDGE – A single peripherals connected to the SRAM and Ethernet forming a BRIDGE that allows the Microblaze to talk to both the Ethernet chip and the SRAM

2. Video – We used the Design of Lab 2 for our video with certain modifications.

The Bridge

Team JAYCam, from last year's class, was able to successfully use the ethernet controller. Our implementation of the Ethernet was based on theirs. The SRAm fuctionality was impledment in the lab 6 but since our code was large and could not be accomodated on the BRAM we had to provide 8,16 and 32 bit access to the SRAM, as C program could call functions which might require such accesses.
Data Paths:

![Data Paths Diagram]

Figure No 3. DataPaths
The two chips SRAM and Ethernet were memory mapped in the peripheral with the upper 11 bits of address bits hard wired and the 12th bit giving the access to Ethernet or SRAM. The Address was incremented whenever there was a 32 bit access using an OR Gate with a Select signal coming in from the FSM.
The SRAM and the Ethernet timings behave totally different. The SRAM is a much faster chip whereas Ethernet is pretty slow to read or write data from the bus considering the correct timings are provided for the Chip Select and Output and Write Enables.

The Jaycam project managed to hack the timings by providing a clock running at twice the frequency of the Microblaze.
3. SOFTWARE

The software for POPi is the glue that holds the different pieces of hardware together and drives them to act in a useful way. The software can be considered as two subsystems that each operate a different piece of hardware, and a top level program that coordinates between these subsystems.

**Video Subsystem**

The video subsystem consists of the SRAM chip and the video driver hardware from Lab 2. These two peripherals were initially compiled together into one project that could address both of them. Later, when the Ethernet and SRAM bridge was prepared, the video subsystem was joined with the bridge hardware instead of only the SRAM hardware.

The video subsystem is a modified terminal program (like that written for lab 2) that displays characters sent through a function call. The `terminal` receives one character and writes it to the appropriate place in the video memory. All cursor placement and scrolling is handled by the terminal software. The software also supports carriage returns and line feeds, so that messages will appear exactly as they were intended by the sender (within the restrictions of the screen). Since the event of a newline type character requires more than echoing a character, the terminal software adjusts the cursor appropriately.

The video is a 80x25 character display using an IBM console font. Each character occupies the same number of pixels.

**Ethernet Subsystem**

The Ethernet subsystem consists of all the software required to check for a message on the remote mail server and deliver it to the top level subsystem bridge. The entry point from the bridge is simply to ask for a new message until the Ethernet subsystem delivers one.

From the bridge the pop program connects to the remote POP3 server to ask for new messages. If there are no messages present on the server, the pop program returns and the bridge
When the pop program is checking for new messages, it must make a TCP connection to the remote server. Due to the needs of our TCP stack, we must first activate the stack with the pop program, then yield execution control to the stack. The stack makes the TCP connection with a three-way handshake, then asks the pop program what to send (assuming successful connection). The pop program responds with the appropriate POP3 commands, the TCP stack transmits these packets and calls the pop program again with the response from the server. The pop program uses a simple state machine to keep track of where in the transaction it is. There are only two possible paths of execution that the pop program can follow--one in which the remote server does have a new message, and one in which it has zero messages. For simplicity, the pop program does not care if there is more than one message on the server, it only retrieves one message at a time. Each message retrieved is deleted during the transaction.

If the server has no new messages for the pop program, the pop program will send the QUIT command immediately after it receives the response from the STAT command (which asks how many messages are on the server). If there are messages (at least one) on the server, the pop program will download message number one then delete it. Following successful deletion, the pop program sends the QUIT command and the stack will terminate the TCP connection. If the pop program has retrieved a message, it will return a pointer to that message to the bridge. If it has found no message, it will respond indicating this.

In the even that the pop program retrieves a message, it must then strip the SMTP headers that have been added to the message on its path through the Internet. The body of an email message is always preceded by an blank line, and is always proceeded by a line consisting only of a period ´.´. The pop program calls a strip function that returns only the message, without the headers. This is the message that is returned to the bridge.

Bridge

The top-level bridge program executes as follows. The program begins by initiating the video driver (this blanks the screen). Next it enters a loop that executes the pop program and only exits when the pop program returns a message. In this way, it will simply wait until a message is available. When it has received a message from pop, it stores it locally and sends each character individually to the video terminal. At the end of the string, the bridge starts over and asks pop for a new message again.

4. LESSONS LEARNED

Who did what:

Yash and Javier - Ethernet hardware, SRAM/Ethernet bridge
Adam - Video subsystem, remote POP3 server, integration
Alex - TCP/IP and POP software
All - SRAM hardware
All - Documentation and Presentation.
What we learned:

Alex - Get started earlier because functionalities such as TCP/IP are difficult to work with. Be prepared to change any and everything if you must. Try to use CVS to prevent loss of code you may have to revert to.

Adam - It was good to have tangible milestones like a working video subsystem. The ethernet subsystem was more difficult than we imagined, and should have been a top priority earlier than it was. We probably got more done than we would have if we hadn't finished some pieces early, but if we started on the hardest part first (ethernet and tcp), we may have had a better chance of success. We also probably separated the labor too much, so we were less able to help each other when problems arose.

Javier - It is good to be able to rely on team members to deliver discrete parts of the project. However, actually working together is also important to bring the project together in the end. Learning vhdl and the xilinx toolkit will be useful in future academic endeavors.

Yash - Timing issues are the most complex factors in designing and using hardware. We probably should have started debugging at a lower level when our ethernet peripheral did not succeed after a few iterations. Getting the timing issues completely correct is the top priority, datapath and connection issues come second.

Advice for future projects:

If you plan to do your own implementation of an Ethernet bridge, start on that first, and as early as possible. Getting pieces of hardware from labs to work together is not nearly as difficult as getting a new peripheral to behave.

Breaking the project down into discrete functional units is good for making progress and dividing labor, but can be harmful if done too much. The team as a whole should tackle the most complex issues, members should handle easier tasks independently.

Keep a record of problems that arise and solutions you find for them. The first time you combine two peripherals you will learn a lot about the XPS, and run into many stupid issues (usually syntactical) that will slow you down a lot. If you have to do this again, you don't want to have to reinvent the wheel. A class repository of solutions to xilinx problems would be a very useful tool for future semesters.

Ask Christian (or your TA) about useful command line tools such as vsim and vcom, fpga-editor, etc. There is a lot available on these computers that you probably don't know about.

Learn how to tar files and do it often.

Be nice to Christian and make sure you have scripts to remove cached build files and add the xilinx directory to your path.
EtherSend.c

#include "jaycam.h"

static BYTE incr = 0;

BYTE send(WORD *data, WORD addr, WORD dmalen, WORD sendlen){
    int i, j, h;
    WORD counter;
    WORD word;

    counter = dmalen>>1;
    outnic(RSAR0, (addr&0xff)); // set DMA starting address
    outnic(RSAR1, (addr>>8));
    outnic(ISR, 0xFF); // clear ISR
    outnic(RBCR0, (dmalen&0xff)); // set Remote DMA Byte Count
    outnic(RBCR1, (dmalen>>8));
    outnic(TBCR0, (sendlen&0xff)); // set Transmit Byte Count
    outnic(TBCR1, (sendlen>>8));
    outnic(CMDR, 0x12); // start the DMA write

    // change order of MS/LS since DMA
    // writes LS byte in 15-8, and MS byte in 7-0
    for(i=0; i<counter; i++){
        word = (data[i]<<8)|(data[i]>>8);
        outnic(DATAPORT, word);
    }
    incr++;

    if(!(innic(ISR)&0x40)){
        print("Data - DMA did not finish\r\n");
        return 1;
    }
    outnic(TPSR, TXSTART); // set Transmit Page Start Register
    outnic(CMDR, 0x24); // start transmission
    j=1000;
    while(j-->0 && !(innic(ISR)&0x02));
    if(!j){
        return 1;
    }
    outnic(ISR,0xFF);
    return 0;
}

// diagnostic that tests nic functionality
void diag(){
    int i;
    print("ethernet baseaddr is: ");
    putnum(NIC_BASE);
    print(" Command Register Page Switching Test\r\n");
print("Switching to page 1\n");
outnic(CMDR, 0x61);

print("Writing to and reading from 0x0D (value should be 0x4e): ");
outnic(0x0D*2,0x4e);
putnum(innic(0x0D*2));
print("\n");

print("Switching to page 0\n");
outnic(CMDR, 0x21);

print("Reading from reg offset 0x0D: ");
putnum(innic(0x0D*2));
print("\n");

print("Switching to page 1\n");
outnic(CMDR, 0x61);

print("Reading from reg offset 0x0D (should be 0x4e): ");
putnum(innic(0x0D*2));
print("\n\n");

print("          Default Value Test\n");
print("Switching to page 0\n");
outnic(CMDR,21);

print("Reading from 0x16 (value should be 0x15): ");
putnum(innic(0x16*2));
print("\n");

print("Reading from 0x12 (value should be 0x0c): ");
putnum(innic(0x12*2));
print("\n");

print("Reading from 0x13 (value should be 0x12): ");
putnum(innic(0x13*2));
print("\n");

int init_etherne()
{
    outnic(NE_RESET, innic(NE_RESET));  // Do reset
    delay(2);
    if (((innic(ISR) & 0x80) == 0))       // Report if failed
    {
        print("  Ethernet card failed to reset!\n");
        return 0;
    }
    else
    {
        print("Ethernet card reset successful!\n");
        resetnic(); // Reset Ethernet card,
        print("Ethernet card intialization complete!\n");
    }
    return 1;
}

void resetnic()
{
    int i, count;
    outnic(CMDR, 0x21);  // Abort and DMA and stop the NIC
    delay(10);
    outnic(DCR, DCRVAL);  // Set word-wide access
    outnic(RBCR0, 0x00);  // Clear the count regs
    outnic(RBCR1, 0x00);
}
outnic(IMR, 0x00);   // Mask completion irq
outnic(ISR, 0xFF);   // clear interrupt status register
outnic(RCR, 0x20);   // 0x20 Set to monitor mode
outnic(TCR, 0x02);   // put nic in normal operation

// Set Rx start, Rx stop, Boundary and TX start regs
outnic(PSTART, RXSTART);
outnic(PSTOP, RXSTOP);
outnic(BNRY, (BYTE)(RXSTOP-1));
outnic(TPSR, TXSTART);

outnic(ISR, 0xFF);   // clear interrupt status register
outnic(IMR, 0x00);   // Mask completion irq

// put nic in start mode
outnic(CMDR, 0x22); // start nic
outnic(TCR, 0x00);   // put nic in normal operation

void delay(int mult){
    int i;
    int delay = 1000000*mult;
    for(i=0; i<delay; i++);
}
#include "xparameters.h"
#include "xio.h"
#include "xbasic_types.h"

#ifndef BYTE
#define BYTE unsigned char
#endif

#ifndef WORD
#define WORD unsigned short
#endif

#define PACKET_SIZE 204

#define NIC_BASE ( XPAR_SRAM_ETHERNET_0_BASEADDR + 0x00080000 ) // Base I/O address of the our peripheral, plus offset for sram
#define DATAPORT (0x10*2)
#define NE_RESET (0x1f*2)

#define CMDR (0x00*2) // command register for read & write
#define PSTART (0x01*2) // page start register for write
#define PSTOP (0x02*2) // page stop register for write
#define BRY (0x03*2) // boundary reg for rd and wr
#define TPSR (0x04*2) // tx start page start reg for wr
#define TBCR0 (0x05*2) // tx byte count 0 reg for wr
#define TBCR1 (0x06*2) // tx byte count 1 reg for wr
#define ISR (0x07*2) // interrupt status reg for rd and wr
#define RSAR0 (0x08*2) // low byte of remote start addr
#define RSAR1 (0x09*2) // hi byte of remote start addr
#define RBCR0 (0x0A*2) // remote byte count reg 0 for wr
#define RBCR1 (0x0B*2) // remote byte count reg 1 for wr
#define RCR (0x0C*2) // rx configuration reg for wr
#define TCR (0x0D*2) // tx configuration reg for wr
#define DCR (0x0E*2) // data configuration reg for wr
#define IMR (0x0F*2) // interrupt mask reg for wr

#define PAR0 (0x01*2) // physical addr reg 0 for rd and wr
#define CURR (0x07*2) // current page reg for rd and wr
#define MAR0 (0x08*2) // multicast addr reg 0 for rd and WR

#define TXSTART 0x41 // Tx buffer start page
#define TXPAGES 8 // Pages for Tx buffer
#define RXSTART (TXSTART+TXPAGES) // Rx buffer start page
#define RXSTOP 0x7e // Rx buffer end page for word mode
#define DCRVAL 0x01 // DCR values for word mode

// macros for reading and writting registers
#define outnic(addr, data) XIo_Out16(NIC_BASE+addr, data)
#define innic(addr) XIo_In16(NIC_BASE+addr)

void diag();
int init_etherne();
void resetnic();
BYTE send(WORD *data, WORD addr, WORD dmalen, WORD sendlen);

void write_video(int start, int end, int line, Xuint32 *data);
void delay(int mult);
main.c

#include "xparameters.h"
#include "xbasic_types.h"
#include "xio.h"
#include "terminal.h"
#include "pop.h"

#define BUF_SIZE 64
char buf[BUF_SIZE];
int addr;

int main()
{
    int i;
    char *k;
    init_video();
    k = read_pop();       //wait until we have a message
    write_to_sram(k);  //write message to sram
    i = read_from_sram(); //read message from sram
    write_to_video(); //send message to terminal
}

write_to_sram(char *string){
    int i, j = 0;
    int addr;
    do{
        addr = XPAR_SRAM_ETHERNET_0_BASEADDR + (i << 2);
        XIo_Out16(addr, *(string+j));
        i++;
    } while(*(string+(j++)) != '\0');   //read until the end of the
    string (\0)
}

int read_from_sram(){
    int i=0; int j=0;
    do{ //read the first character regardless
        addr = XPAR_SRAM_ETHERNET_0_BASEADDR + (i << 2);
        buf[j] = XIo_In16(addr);
        i++;
    } while(buf[j++] != '\0'); //read string, \0 will be in buf
    return i;
}

void write_to_video(){
    int j = 0;
    while(buf[j] != '\0'){
        char_to_video(buf[j]);
        j++;
    }
}

char *read_pop(){
    char *p;
    *p = '\0';
    while("p == '\0')
        *p = pop();
    return p
}
#include "pop.h"

int pop(){
    // start tcp stack
    popi_init();

    return 1;
}
/*All commands are terminated by a
*CRLF pair.*/

int state_machine(char *s){
    if(s[0] == '+'){ // we can tell pop accepts our commands
        // by looking at the first character of the response
        switch(state){
            case 0: // connected, send user
                state++;
                return user;
            case 1: // user accepted, send pass
                state++;
                return pass;
            case 2: // pass accepted, get stat
                state++;
                return stat;
            case 3:
                // figure out what stat said
                if(*s+4 > '0') { // there is at least one message, get it
                    state++;
                    return retr;
                } else { // there are 0 messages, quit
                    state = 6;
                    return quit;
                }
            case 4: // got the message
                state++;
                strip_message(s); // remove smtp headers and others
                return dele;
            case 5: // message has been deleted
                state++;
                return quit;
            case 6: // connection closed
                print("pop closed the connection successfully\r\n");
                default:
                    ;
                    }
    } else if(s[0] == '-')
        print("definite bad response, pop fails for unknown reason.
        check the server and pop.c\r\n");
    else
        print("unkown response from server. pop crashes hard\r\n");
}
char * strip_message(char *m){
    // ignore until blank line, which should appear as crlfcrlf
    while(*m+0 != '\r' && *m+1 != '\n' && *m+2 != '\r' && *m+3 != '\n') { // when we have crlfcrlf, break, OR continue on all
        m++;
    // add everything
    char *p;
}
p = m;
while(*p+0 != '\r' && *p+1 != '\n' && *p+2 != '.' && *p+3 != '\r'
     && *p+4 != '\n')
    p++;
    //stop at . on a line
    //remember to look for crlf
    *(p+2) = '\0';
return m;
}
```
#ifndef POP
#define POP

int pop();
int state_machine(char *s);
char *strip_message(char *m);

static int state=0;
static char user[] = "USER embedded\r\n";
static char pass[] = "PASS systems\r\n";
static char stat[] = "STAT\r\n";
static char retr[] = "RETR 1\r\n";
static char dele[] = "DELE 1\r\n";
static char quit[] = "QUIT\r\n";

#endif
```
#include "xbasic_types.h"
#include "xio.h"
#include "xintc_l.h"
#include "xuartlite_l.h"
#include "xparameters.h"
#include "terminal.h"
/* Defined in mymicroblaze/xparameters.h */
/* #define XPAR_VGA_BASEADDR 0xFEFF1000 */
/* #define XPAR_VGA_HIGHADDR 0xFEFF1fff */
#define FONT_OFFSET 0xA00
/* Address of a particular character on the screen (rows are 80) */
#define CHAR(r,c) (((unsigned char *)(XPAR_VGA_BASEADDR))[c + ((r) << 6) + ((r) << 4)])
/* Start of font memory */
#define FONT ((unsigned char *)XPAR_VGA_BASEADDR + FONT_OFFSET)

typedef font_t font_t;

//*** these all moved to terminal.h
// * buf variables booted because we do not use buff
// * same for lp, fp
static int col = -1;
static int row = 0;

/* Write a text string on the display */
void put_string(char string)
{
    //these move the cursor
    if(string == 0x0d) //if control-m (carriage return)
    {
        string = 0x20; //use blank
        col = -1;
    }
    else
    {
        string = 0x20; //use blank
        row++;
        col--;
    }

    // This part writes the character to video
    //
    //get the screen position and write the character to it
    //print("setting cursor\r\n");
    char *cursor = &((unsigned char *)XPAR_VGA_BASEADDR)]]; + ((r) << 6) + ((r) << 4));
    *cursor = string;
    //print("wrote string: ");
    //print(&string);
    //print("\r\n");
}

void init_video(){
    volatile unsigned char *p;
print("initing video\r\n");
for ( p = &(CHAR(0,0)) ; p < &(CHAR(29,80)) ; ++p)
    "p = ' ';
}

char_to_video(char c)
{
    // volatile unsigned char *p;
    unsigned char v, vv;
    int i, j, k;
    int cnt;
    /* Clear the screen */
    // print("received character ");
    print(&c);
    //print("\r\n");
    if (col>=79) // narrowing to the 80 columns of the Display
    {
        col = 0;
        row++;
    }
    else
    
    col++;
    if(row >= 30)
    {
        for(i=0;i<30;i++)       //row
        {
            for(j=0;j<80;j++)   //column
                CHAR(i,j)=CHAR(i+1,j);
        }
        row = 29;
    }
   //This actually puts the character on the screen
    put_string(c);

    if (col>77)
    {
        col=1;
        row++;
    }
    //else
    //col++;
Terminal.h

#ifndef TERMINAL
#define TERMINAL
void put_string(char string);
void init_video();
char_to_video(char c);
#endif
library ieee;
use ieee.std_logic_1164.all;

library UNISIM;
use UNISIM.Vcomponents.all;

entity sram_ethernet is

  generic  (
    C_OPB_AWIDTH : integer               := 32;  
    C_OPB_DWIDTH : integer               := 32; 
    C_BASEADDR  : std_logic_vector(0 to 31) := X"00000000"; 
    C_HIGHADDR  : std_logic_vector(0 to 31) := X"FFFFFFFE" 
  );

  port  (
    -- ports between peripheral and OPB
    OPB_Clk     : in  std_logic;
    OPB_Rst     : in  std_logic;
    OPB_ABus    : in  std_logic_vector(0 to C_OPB_AWIDTH-1);
    OPB_BE      : in  std_logic_vector(0 to C_OPB_DWIDTH/8-1);  -- not using
    OPB_DBus    : in  std_logic_vector(0 to C_OPB_DWIDTH-1);
    OPB_RNW     : in  std_logic;
    OPB_select  : in  std_logic;
    OPB_seqAddr : in  std_logic;        -- Sequential Address
    Sln_DBus    : out std_logic_vector(0 to C_OPB_DWIDTH-1);
    Sln_errAck  : out std_logic;        -- (unused)
    Sln_retry   : out std_logic;        -- (unused)
    Sln_toutSup : out std_logic;        -- Timeout suppress
    Sln_xferAck : out std_logic;       -- Transfer acknowledge

    -- ports between peripheral and Ethernet
    ETH_CS       : out std_logic;
    ETH_IOCS     : in  std_logic;
    ETH_RDY_DTAK : in  std_logic;
    ETH_IREQ     : in  std_logic;

    -- ETH_data : inout std_logic_vector(15 downto 0)

    -- ports between peripheral and SRAM
    -- IOR IOW AEN_PSEN BHE address and data
    RAM_CE       : out std_logic;
    PB_OE_IOR    : out std_logic;
    PB_WE_IOW    : out std_logic;
    PB_LB_BHE    : out std_logic;
    PB_LSB_AEN   : out std_logic;
    PB_D        : inout std_logic_vector(0 to 15);
    PB_A        : out std_logic_vector(0 to 17);

    FLASH_CE : out std_logic;
    SDRAM_CE :out std_logic;
    ADC_OE :out std_logic;
    AU_CSN :out std_logic;
    USB_CS :out std_logic;
    NV_CS0 :out std_logic;
    NV_CS1 :out std_logic
  );
end sram_ethernet;

architecture Behavioral of sram_ethernet is

--deleted old signals from bram

type opb_state is (Idle,
Select_Ram,
Select_Ethernet,
read_ram_16_a,
read_ram_16_b,
read_ram_32_a,
read_ram_32_b,
read_ram_32_c,
write_ram_32,
write_ETH_a,
write_ETH_b,
write_ETH_c,
write_ETH_d,
-- write_ETH_e,
Read_ETH_a,
Read_ETH_b,
Read_ETH_c,
Read_ETH_d,
-- Read_ETH_e,
Xfer);

signal present_state, next_state : opb_state;

-- new signals

signal CS, ETHcs, RAMcs, rnw, ub, lb, FSM_tristate, ub_tmp, lb_tmp : std_logic;
signal FSM_CS_RAM, FSM_CS_ETH, fsm_oe_ior, fsm_we_iow, sel32, sl_ld_hi,
sl_ld_lo : std_logic;
signal pb_ub_i, pb_lb_i, ram_cs_i, eth_cs_i, pb_oe_i, pb_we_i : std_logic;
signal pb_d_i, data_fix, data_out, pb_d_o, data_low, tristate_i, data_high :
std_logic_vector(0 to 15);
signal addr, pb_a_i, addr_s32 : std_logic_vector(0 to 17);
signal be : std_logic_vector(0 to 3);

component OBUF_F_24
port ( O : out std_ulogic;
       I : in std_ulogic);
end component;

compoment IOBUF_F_24
port ( O : out std_ulogic;
       IO : inout std_ulogic;
       I : in std_ulogic;
       T : in std_ulogic);
end component;

compoment FDCE
port (C : in std_logic;
      CLR : in std_logic;
      CE : in std_logic;
      D : in std_logic;
      Q : out std_logic);
end component;
component FDPE

    port (C : in std_logic;
           PRE : in std_logic;
           CE : in std_logic;
           D : in std_logic;
           Q : out std_logic);

end component;

attribute iob : string;
attribute iob of FDCE : component is "true";
attribute iob of FDPE : component is "true";

begin

FLASH_CE <= '1';
SDRAM_CE <= '1';
ADC_OE <= '1';
AU_CSN <= '1';
USB_CS <= '1';
NV_CS0 <= '1';
NV_CS1 <= '1';

CS <= OPB_Select when OPB_ABus(0 to 11) = C_BASEADDR(0 to 11) else '0';

ETHcs <= CS and OPB_ABus(12);
RAMcs <= CS and (not OPB_ABus(12));

process (OPB_Rst, OPB_Clk) -- address
begin -- process
if OPB_Clk'event and OPB_Clk='1' then
    addr <= OPB_ABus(13 to 30);
    be <= OPB_BE(0 to 3);
    rnw <= OPB_rnw;
end if;
if OPB_Rst='1' then
    addr <= (others => '0');
    be <= (others => '0');
    rnw <= '0';
end if;
end process;

addr_s32 <= addr(0 to 16) & (addr(17) or sel32);

lb_tmp <= NOT (be(3) or be(1));
ub_tmp <= NOT (be(2) or be(0));

a_pad: for i in 0 to 17 generate
    a_pad: FDCE port map(
                        C => OPB_Clk,
                        CLR => OPB_Rst,
                        CE => '1',
                        D => addr_s32(i),
                        Q => pb_a_i(i)
                    );
end generate a_pad;

ub_pad: FDPE port map (  
                        C => OPB_Clk,
                        PRE => OPB_Rst,
                        CE => '1',
                        D => ub,
Q => pb_ub_i
);
ul_pad : FDPE port map (  
C => OPB_Clk,
PRE => OPB_Rst,
CE => '1',
D => lb,
Q => pb_lb_i
);

CS_RAM_pad : FDPE port map (  
C => OPB_Clk,
PRE => OPB_Rst,
CE => '1',
D => FSM_CS_RAM,
Q => ram_cs_i
);

CS_ETH_pad: FDPE port map (  
C => OPB_Clk,
PRE => OPB_Rst,
CE => '1',
D => FSM_CS_ETH,
Q => eth_cs_i
);

oe_pad: FDPE port map (  
C => OPB_Clk,
PRE => OPB_Rst,
CE => '1',
D => fsm_oe_ior,
Q => pb_oe_i
);

we_pad : FDPE port map (  
C => OPB_Clk,
PRE => OPB_Rst,
CE => '1',
D => fsm_we_iow,
Q => pb_we_i
);

d_pad: for i in 0 to 15 generate

di_pad : FDPE port map (  
C => OPB_Clk,
PRE => OPB_Rst,
CE => '1',
D => data_fix(i),
Q => pb_d_i(i)
);

dt_pad : FDPE port map (  
C => OPB_Clk,
PRE => OPB_Rst,
CE => '1',
D => FSM_tristate,
Q => tristate_i(i)
);

do_pad : FDPE port map (  
C => OPB_Clk,
PRE => OPB_Rst,
CE => '1',
D => pb_d_o(i),
Q => data_out(i)
);
end generate;

-- OPB DataPath Write

process (OPB_Rst, OPB_Clk) -- write
begin -- process
if OPB_Clk'event and OPB_Clk = '1' then
  data_low <= OPB_DBus(0 to 15);
  data_high <= OPB_DBus(16 to 31);
end if;
if OPB_Rst = '1' then
  data_low  <= (others => '0');
  data_high <= (others => '0');
end if;
end process;

process (OPB_Rst, OPB_Clk) -- read
begin -- process
if OPB_Clk'event and OPB_Clk = '1' then
  if sl_ld_hi = '0' and sl_ld_lo = '0' then
    Sln_Dbus <= (others => '0') ;
  else
    if sl ld_hi = '1' then
      Sln_Dbus(16 to 31) <= data_out;
    end if;
    if sl ld_lo = '1' then
      Sln_Dbus(0 to 15) <= data_out;
    end if;
  end if;
if OPB_Rst = '1' then
  Sln_DBus <= (others => '0');
end if;
end process;

with sel32 select
  data_fix <=
    data_low when '0',
    data_high when others;

ADDR_PADS : for m in 0 to 17 generate -- C_SRAM_AWIDTH-1 generate
begin
  addr_pad : component OBUF_F_24 port map ( 
    I => pb_a_i(m),
    O => PB_A(m));
end generate ADDR_PADS;

PB_UB_PAD :OBUF_F_24 port map( 
  O => PB_UB_BHE,
  I => pb_ub_i);

PB_LB_PAD :OBUF_F_24 port map( 
  O => PB_LB_AEN,
  I => pb_lb_i);
RAM_CE_PAD :OBUF_F_24 port map (  
  O => RAM_CE,  
  I => ram_cs_i);  

ETH_CE_PAD :OBUF_F_24 port map (  
  O => ETH_CS,  
  I => eth_cs_i);  

PB_OE_PAD :OBUF_F_24 port map (  
  O => PB_OE_IOR,  
  I => pb_oe_i);  

PB_WE_PAD :OBUF_F_24 port map (  
  O => PB_WE_IOW,  
  I => pb_we_i);  

DATA_PADS : for m in 0 to 15 generate -- C_SRAM_DWIDTH-1 generate  
begin  
  addr_pad : component IOBUF_F_24 port map (  
    I => pb_d_i(m),  
    T => tristate_i(m),  
    IO => PB_D(m),  
    O => pb_d_o(m));  
end generate DATA_PADS; 

-- unused outputs  
Sln_errAck <= '0';  
Sln_retry <= '0';  
Sln_toutSup <= '0';  

--changed RAM_DWIDTH TO SRAM_DWIDTH  

--in the below we changed RAM_WIDTHS TO SRAM_WIDTHS  
-- we changed it Awidth to ETH_Awidth 

-- Sequential part of the FSM  
fsm_seq : process(OPB_Clk, OPB_Rst) begin  
  if OPB_Rst = '1' then  
    present_state <= Idle;  
  elsif OPB_Clk'event and OPB_Clk = '1' then  
    present_state <= next_state;  
  end if;  
end process fsm_seq;  

-- Combinational part of the FSM  
fsm_comb : process(present_state, ETHcs, RAMcs, OPB_Select, rnw, be) begin  
  -- Default values  
  FSM_tristate <= '1';  
  fsm_we_IOW <= '1';  
  fsm_oe_ior <= '1';  
  FSM_CS_RAM <= '1';  
  FSM_CS_ETH <= '1';  
  Sln_xferAck <= '0';  
  sel32 <= '0';  
  sl_id_hi <= '0';  
  sl_id_lo <= '0';  
  ub <= '1';
lb <= '1';

case present_state is

when Idle =>
  if RAMcs = '1' then
    next_state <= Select_Ram;
  elsif ETHcs = '1' then
    next_state <= Select_Ethernet;
  else
    next_state <= Idle;
  end if;

when Select_Ram =>
  if OPB_Select = '1' then
    if rnw = '1' then --READING
      fsm_we_iow <= '1';
      FSM_CS_RAM <= '0';
      fsm_oe_ior <= '0';
      ub <= ub_tmp;
      lb <= lb_tmp;
    end if;
    if be = "1111" then
      next_state <= read_ram_32_a;
    else
      next_state <= read_ram_16_a;
    end if;
  else --WRITING
    FSM_CS_RAM <= '0';
    fsm_we_iow <= '0';
    FSM_tristate <= '0';
    ub <= ub_tmp;
    lb <= lb_tmp;
    if be = "1111" then
      next_state <= write_ram_32;
    else
      next_state <= Xfer;
    end if;
  end if;
else
  next_state <= Idle;
end if;

when read_ram_16_a =>
  if OPB_Select = '1' then
    FSM_CS_RAM <= '0';
    fsm_we_iow <= '0';
    ub <= ub_tmp;
    lb <= lb_tmp;
    next_state <= read_ram_16_b;
  else
    next_state <= Idle;
  end if;
when read_ram_16_b =>
  if OPB_Select = '1' then
    FSM_CS_RAM <= '0';
    fsm_oe_ior <= '0';
    sl_ld_hi <= '1';
    sl_ld_lo <= '1';
    ub <= ub_tmp;
    lb <= lb_tmp;
    next_state <= Xfer;
  else
    next_state <= Xfer;
  end if;
when Read_RAM =>
  if OPB_Select = '1' then
    FSM_CS_RAM <= '0';
    fsm_we_iow <= '0';
    ub <= ub_tmp;
    lb <= lb_tmp;
    if be = "1111" then
      next_state <= read_ram_32_a;
    else
      next_state <= read_ram_16_a;
    end if;
  else
    next_state <= Idle;
end if;
end if;

when read_ram_32_a =>
    if OPB_Select = '1' then
        FSM_CS_RAM <= '0';
        fsm_oe_ior <= '0';
        sel32 <= '1';
        ub <= ub_tmp;
        lb <= lb_tmp;
        next_state <= read_ram_32_b;
    else
        next_state <= Idle;
    end if;

delimiter when read_ram_32_b =>
    if OPB_Select = '1' then
        FSM_CS_RAM <= '0';
        fsm_we_iow <= '0';
        sl_ld_lo <= '1';
        ub <= ub_tmp;
        lb <= lb_tmp;
        next_state <= read_ram_32_c;
    else
        next_state <= Idle;
    end if;

delimiter when read_ram_32_c =>
    if OPB_Select = '1' then
        FSM_CS_RAM <= '0';
        fsm_we_iow <= '0';
        sl_ld_hi <= '1';
        ub <= ub_tmp;
        lb <= lb_tmp;
        next_state <= Xfer;
    else
        next_state <= Idle;
    end if;

delimiter when write_ram_32 =>
    if OPB_Select = '1' then
        FSM_CS_RAM <= '0';
        fsm_we_iow <= '0';
        sel32 <= '1';
        FSM_tristate <= '0';
        ub <= ub_tmp;
        lb <= lb_tmp;
        next_state <= Xfer;
    else
        next_state <= Idle;
    end if;

-- ethernet FSM

delimiter when Select_Ethernet =>
    if OPB_Select = '1' then
        FSM_CS_ETH <= '0';
        ub <= '0';
        lb <= '0';
        if rnw = '1' then               --READING from ETHERNET
            fsm_oe_ior <= '0';
            next_state <= Read_ETH_a;
        else                            --WRITING to the ETHERNET
            FSM_tristate <= '0';
            -- fsm_we_iow <= '0';
        end if;
    end if;
next_state <= Write_ETH_a;
end if;
else
next_state <= Idle;
end if;

when write_ETH_a=>
  if OPB_Select = '1' then
    FSM_CS_ETH <= '0';
    ub <= '0';
    lb <= '0';
    fsm_we_iow <= '0';
    FSM_tristate <= '0';
    next_state <= Write_ETH_b;
  else
    next_state <= Idle;
  end if;

when write_ETH_b =>
  if OPB_Select = '1' then
    FSM_CS_ETH <= '0';
    fsm_we_iow <= '0';
    FSM_tristate <= '0';
    ub <= '0';
    lb <= '0';
    next_state <= Write_ETH_c;
  else
    next_state <= Idle;
  end if;

when write_ETH_c =>
  if OPB_Select = '1' then
    FSM_CS_ETH <= '0';
    FSM_tristate <= '0';
    fsm_we_iow <= '0';
    ub <= '0';
    lb <= '0';
    next_state <= Write_ETH_d;
  else
    next_state <= Idle;
  end if;

when write_ETH_d=>
  if OPB_Select = '1' then
    FSM_CS_ETH <= '0';
    -- fsm_we_iow <= '0';
    FSM_tristate <= '0';
    ub <= '0';
    lb <= '0';
    next_state <= Xfer;
  else
    next_state <= Idle;
  end if;

when Read_ETH_a=>
  if OPB_Select = '1' then
    FSM_CS_ETH <= '0';
    fsm_oe_iior <= '0';
    ub <= '0';
    lb <= '0';
    next_state <= Read_ETH_b;
  else

next_state <= Idle;
end if;

when Read_ETH_b=>
  if OPB_Select = '1' then
    FSM_CS_ETH <= '0';
    fsm_oe_ior <= '0';
    ub <= '0';
    lb <= '0';
    next_state <= Read_Eth_c;
  else
    next_state <= Idle;
  end if;

when Read_ETH_c=>
  if OPB_Select = '1' then
    FSM_CS_ETH <= '0';
    --                   fsm_oe_ior <= '0';
    ub <= '0';
    lb <= '0';
    next_state <= Xfer;
  else
    next_state <= Idle;
  end if;

when Read_ETH_d=>
  if OPB_Select = '1' then
    FSM_CS_ETH <= '0';
    --            fsm_oe_ior <= '0';
    ub <= '0';
    lb <= '0';
    next_state <= Xfer;
  else
    next_state <= Idle;
  end if;

-- State encoding is critical here: xfer must only be true here
when Xfer =>            -- Actual Xfer
  Sln_xferAck <= '1';
  next_state <= Idle;
end case;
end process fsm_comb;

end Behavioral;

TestBench.vhdl

library ieee;
use ieee.std_logic_1164.all;

entity tb is
end tb;

architecture tb of tb is

signal OPB_Clk     : std_logic;
signal  OPB_Rst     : std_logic;
signal   OPB_ABus    : std_logic_vector(0 to 31);
signal    OPB_BE     : std_logic_vector(0 to 3);
-- not usg this
signal    OPB_DBus   : std_logic_vector(0 to 31);
signal    OPB_RNW    : std_logic;
signal    OPB_select : std_logic;
signal  OPB_seqAddr : std_logic;
-- Sequential Address
signal Sln_DBus     : std_logic_vector(0 to 31);
signal Sln_errAck   : std_logic;
-- (unused)
signal Sln_retry    : std_logic;
-- (unused)
signal Sln_toutSup  : std_logic;
-- Timeout suppress
signal Sln_xferAck  : std_logic;
-- Transfer acknowledge

--ports between peripheral and Ethernet
signal ETH_CS       : std_logic;
signal ETH_IOCS     : std_logic;
signal ETH_RDY_DTAK : std_logic;
signal ETH_IREQ     : std_logic;

-- ETH_data : out std_logic_vector(15 downto 0)

--ports between peripheral and SRAM
-- IOR IOW AEN_PSEN BHE address and data
signal RAM_CE        : std_logic;
signal PB_OE_IOR     : std_logic;
signal PB_WE_IOW     : std_logic;
signal PB_UB_BHE     : std_logic;
signal PB_LB_AEN     : std_logic;
signal PB_D          : std_logic_vector(0 to 15);
signal PB_A          : std_logic_vector(0 to 17);

begin

ut: entity work.sram_ethernet port map (
    OPB_Clk => OPB_Clk,
    OPB_Rst => OPB_Rst,
    OPB_ABus => OPB_ABus,
    OPB_BE => OPB_BE,
    OPB_DBus => OPB_DBus,
    OPB_RNW => OPB_RNW,
    OPB_select => OPB_select,
    OPB_seqAddr => OPB_seqAddr,
    Sln_DBus => Sln_DBus,
    Sln_errAck => Sln_errAck,
    Sln_retry => Sln_retry,
    Sln_toutSup => Sln_toutSup,
    Sln_xferAck => Sln_xferAck,

    --ports between peripheral and Ethernet
    ETH_CS => ETH_CS,
    ETH_IOCS => ETH_IOCS,
    ETH_RDY_DTAK => ETH_RDY_DTAK,
    ETH_IREQ => ETH_IREQ,

    RAM_CE => RAM_CE,
    PB_OE_IOR => PB_OE_IOR,

)
PB_WE_IOW => PB_WE_IOW,
PB_UB_BHE => PB_UB_BHE,
PB_LB_AEN => PB_LB_AEN,
PB_D => PB_D,
PB_A => PB_A
);

process
begin
  OPB_Rst <= '1';
  wait for 30 ns;
  OPB_Rst <= '0';
  wait;
end process;

process
begin
  OPB_Clk <= '1';
  wait for 10 ns;
  OPB_Clk <= '0';
  wait for 10 ns;
end process;

process
begin
  OPB_Select <= '0';
  wait for 100 ns;
-- wr 32
  wait until OPB_Clk'event and OPB_Clk='1';
  OPB_Select <= '1';
  OPB_ABus <= X"0000_0000";
  OPB_BE <= "1111";
  OPB_DBus <= X"1234ABCD";
  OPB_RNW <= '0';
  wait until Sln_xferack = '1';
  wait until OPB_Clk'event and OPB_Clk='1';
  OPB_Select <= '0';
  wait until OPB_Clk'event and OPB_Clk='1';
  OPB_Select <= '1';
  OPB_ABus <= X"0000_0000";
  OPB_BE <= "1111";
  OPB_DBus <= X"11111111";
  OPB_RNW <= '1';
  wait until Sln_xferack = '1';
  wait until OPB_Clk'event and OPB_Clk='1';
  OPB_Select <= '0';
-- wr16
  wait until OPB_Clk'event and OPB_Clk='1';
  wait until OPB_Clk'event and OPB_Clk='1';
  OPB_Select <= '1';
  OPB_ABus <= X"0000_0000";
  OPB_BE <= "1100";
  OPB_DBus <= X"12341234";
  OPB_RNW <= '0';
wait until Sln_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';
wait until OPB_Clk'event and OPB_Clk='1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '1';
OPB_ABus <= X"0000_0002";
OPB_BE <= "0011";
OPB_DBus <= X"12341234";
OPB_RNW <= '0';
wait until Sln_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';
wait until OPB_Clk'event and OPB_Clk='1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '1';
OPB_ABus <= X"0000_0000";
OPB_BE <= "1000";
OPB_DBus <= X"CDCDCDCD";
OPB_RNW <= '0';
wait until Sln_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';
wait until OPB_Clk'event and OPB_Clk='1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '1';
OPB_ABus <= X"0000_0001";
OPB_BE <= "0100";
OPB_DBus <= X"EFEFEFEF";
OPB_RNW <= '0';
wait until Sln_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';
wait until OPB_Clk'event and OPB_Clk='1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '1';
OPB_ABus <= X"0000_0002";
OPB_BE <= "0010";
OPB_DBus <= X"ABABABAB";
OPB_RNW <= '0';
wait until Sln_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';
wait until OPB_Clk'event and OPB_Clk='1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '1';
OPB_ABus <= X"0000_0003";
OPB_BE <= "0001";
OPB_DBus <= X"34343434";
OPB_RNW <= '0';
wait until Sln_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';

-- Ethernet write 16 bit

wait until OPB_Clk'event and OPB_Clk='1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '1';
OPB_ABus <= X"0008_0000";
OPB_BE <= "1100";
OPB_DBus <= X"1A2A1A2A";
OPB_RNW <= '0';

wait until Slx_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '1';
OPB_ABus <= X"0009_0002";
OPB_BE <= "0011";
OPB_DBus <= X"A1A1A1A1";
OPB_RNW <= '0';

wait until Slx_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '1';
OPB_ABus <= X"0000_0000";
OPB_BE <= "1111";
OPB_DBus <= X"11111111";
OPB_RNW <= '1';

wait until Slx_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '1';
OPB_ABus <= X"0000_0000";
OPB_BE <= "0001";
OPB_DBus <= X"11111111";
OPB_RNW <= '1';

wait until Slx_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '1';
OPB_ABus <= X"0000_2220";
OPB_BE <= "0010";
OPB_DBus <= X"11111111";
OPB_RNW <= '1';
wait until Sln_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';

wait until OPB_Clk'event and OPB_Clk='1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_ABus <= X"0000_3330";
OPB_BE <= "0100";
OPB_DBus <= X"11111111";
OPB_RNW <= '1';

wait until Sln_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';

wait until OPB_Clk'event and OPB_Clk='1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_ABus <= X"0000_4440";
OPB_BE <= "1000";
OPB_DBus <= X"11111111";
OPB_RNW <= '1';

wait until Sln_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';

wait until OPB_Clk'event and OPB_Clk='1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_ABus <= X"0008_2220";
OPB_BE <= "0011";
OPB_DBus <= X"11111111";
OPB_RNW <= '1';

wait until Sln_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';

wait until OPB_Clk'event and OPB_Clk='1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_ABus <= X"0009_4440";
OPB_BE <= "1100";
OPB_DBus <= X"11111111";
OPB_RNW <= '1';

wait until Sln_xferack = '1';
wait until OPB_Clk'event and OPB_Clk='1';
OPB_Select <= '0';

wait;

end process;

pb_d <= X"E" & pb_a(6 to 17) when pb_oe_iom = '0' and eth_cs = '0' else X"5" & pb_a(6 to 17) when pb_oe_iom = '0' and ram_ce = '0' else (others => 'Z');
end tb;

TCPstack.C

// Standard includes (may not be required):
#include "stdio.h"
// lwip include (required):
#include "tcp.h"

// Response to server request
static char packetData[] = "";

static void popi_start(void *arg, struct tcp_pcb *pcb, struct pbuf *p) {
    char *data;

    // If we got a NULL pbuf in p, the remote end has closed the connection.
    if(p != NULL) {
        // The payload pointer in the pbuf contains the data in the TCP segment.
        data = p->payload;

        // packetData is later written to the stack
        packetData = state_machine(data);

        // If enter was not received (newline escape char)
        if(strlen(packetData) != 0) {
            // send packetData response to pop server
            tcp_write(pcb, packetData, sizeof(packetData), 0);
        } else {
            // Close the connection on our end.
            tcp_close(pcb);
        }
    } else {
        // Connection closed on other end, so close ours also.
        tcp_close(pcb);
    }

    // Free the pbuf.
    pbuf_free(p);
} else {

// This is the callback function that is called when a connection
// has been accepted.  
static void popi_accept(void *arg, struct tcp_pcb *pcb) {

    // Set up the function popi_start so that it is called when a packet is received on the tcp connection
    tcp_recv(pcb, popi_start, NULL);
}

// This is the callback function that is called when a connection
// has been accepted.
static void popi_connect(void *arg, struct tcp_pcb *pcb) {
    // Set up the function popi_start to be called when a packet is
    // received on the tcp connection
    // after the handshake this method is called to handle data passed
    // from server to client
    // Handshaking is handle internally by lwip
    tcp_recv(pcb, popi_start, NULL);
}

// The initialization function.
void popi_init(void) {
    struct tcp_pcb *pcb;
    struct ip_addr ipaddr;
    u16_t port = 110;

    //Create a new TCP PCB.
    pcb = tcp_pcb_new();

    //Bind the PCB to TCP port 110
    //TODO: Change this to operating port of pop3 server
    tcp_bind(pcb, NULL, 110);

    //Configure the ip of the outgoing message
    IP4_ADDR(&ipaddr, 160,39,190,152);

    //Set up popi_connect function to be called when a new connection
    //arrives
    tcp_connect(pcb, NULL, 110, popi_connect);
}