CS341 - Computer Architecture and Organization Bob Wilson
MP4: External World Problems
Due: At start of class 23

The purpose of this assignment is to use the SAPC to study some object in the real world. There are two projects designed to do that: a chip tester and a digital oscilloscope (i.e., a way to record and replay a signal gathered over a very short interval of time). They're described in the following sections of this document: Chip Testing Software and A Digital Oscilloscope

The best way to do this is in teams of two. As usual, each group provides one complete solution for grading.

The provided makefile is Makefile, following the vague convention that more complicated makefiles are capitalized in their name. make looks for either name, and usually complains if both are found, but uses the lower-case-named makefile then. Some people argue that the capitalized name is superior because capital letters sort to before all the lower-case letters, so that Makefile is grouped with README right at the top of the page when the ls command is used.

NOTES:

a. The actual details for each project are given in the project description. Make sure you read those carefully and remember that each person is to provide one complete solution for grading.

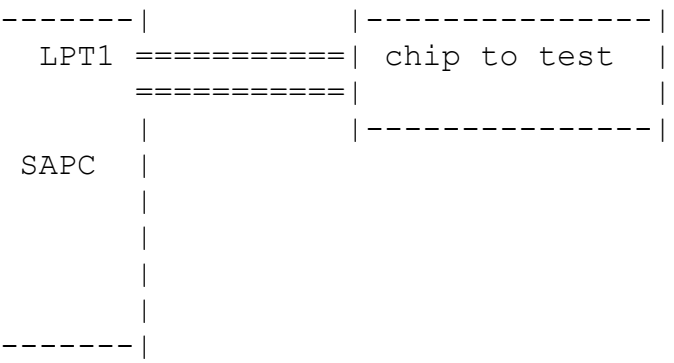
b. In the event that you are unable to correctly complete this assignment by the due date, do not remove the work you were able to accomplish - partial credit is always better than none.

c. You will find files in ~bobw/cs341/mp4 that you should copy to your mp4 directory.

CHIP TESTING SOFTWARE

In this project you will build and exercise a program that turns the SAPC into a machine to test chips.

Sketch of Experimental setup:
SAPCs 5 and 6: chip to test is LS00
SAPCs 7 and 8: chip to test is LS138



The part of the code that's the same for all chips lives in chiptest.c. Chip specific code is in _chip.c files, one for each kind of chip you wish to test. The makefile compiles the chiptest.c and whatever_chip.c file(s) and links them to produce a .lnx file to download to the SAPC. When that program is run it first tells the technician how to connect the chip to be tested to the SAPC parallel port, then loops on all possible combinations of 0's and 1's on the input pins. For each combination it asserts the proper voltages via LPT1's PDR, reads the chip output via its PSR and compares those values with what the _chip.c file says they should be. It reports any errors in a useful way.

I could write formal specifications for the _chip.c file, but haven't. Instead I've provided a commented sample, ls00_chip.c, specifying what chiptest needs to know to test (part of) an LS00 quad nand gate chip. I'm not claiming my setup is the best possible way to do it--you can suggest other ways in your discussion. Note the basic idea that all chip information/functionality necessary for chiptest is in struct chip. Note also that the TT array is not in struct chip, because it is just subordinate to the softchip function.

There's also a stub chiptest.c, and a Makefile. The command

make chiptest.lnx

produces chiptest.lnx for downloading. If you had the real chiptest.c running chiptest.lnx on the SAPC (before you add another chip, see part 3 below) would produce this output:

Chip tester for chip(s):
0-- LS00

Enter chip to test: 0

Testing LS00 chip

Connection instructions:

VCC to chip pins: 14

GND to chip pins: 7

LPT1 pin:	chip pin:
2	1
3	2
4	4
5	5
6	9
7	10
8	12
9	13
15	3
13	6
12	8
10	11

<CR> when ready:

... report any chip errors ...

Note that SAPCs 5 and 6 are connected up exactly as instructed above.

Development hint: Even though you can test a chip only with code running on an SAPC, you can do a lot of your chiptest.c development on UNIX -- all the loop construction and much of the debugging – just about everything but asserting voltages to LPT1 and reading them from there. The Makefile provides for

make chiptest

to produce code that runs on UNIX.

Debugging

This project has the advantage over the scope project that all the SAPCs involved do have remote gdb capability, since this experimental setup does not use COM1. Thus you have the choice

of remote gdb, UNIX gdb, Tutor, and printf's to help you out. Note the INTERACTIVE preprocessor symbol--you can comment out its definition and get rid of user input, making remote gdb easier to use. Don't forget to reinstate it at the end!

Pin numbering conventions

Note that the program needs to know the LPT1 pin layout, for example, that pin2 is the first output pin, and pin 10 is the first input pin. It also needs to know what inputs to complement. Your challenge in programming is to express this knowledge in a robust and clear way.
For simplicity, implement the case of using up to 8 output pins and 5 input pins via PDR and PSR respectively. Discuss any other possibilities in your written report.

Let us standardize our handling of pins, so we can share experimental setups. The data register has bits 0, 1, 2, ..., 7, and these naturally connect to the chip input pins in their natural order. The status register has only 5 usable bits, bits 3 to 7, and we use them in increasing order as well.

```
 Data
 Reg    DB25    LS00    LS138
 bit     pin     pin    Input
  0       2       1      A-0  <---logical 0th in pin: inpins[0]
  1       3       2      A-1  <---logical 1st in pin: inpins[1]
  2       4       4      A-2     ...
  3       5       5      E-1
  4       6       9      E-2
  5       7      10      E-3
  6       8      12
  7       9      13

Status
 Reg    DB25    LS00    LS138
 bit     pin     pin    Output
  3      15       3      O-0   <---logical 0th out pin: outpins[0]
  4      13       6      O-1   <---logical 1st out pin: outpins[1]
  5      12       8      O-2     ...
  6      10      11      O-3
  7      11              O-4
```

Thus data bit 0 connects via DB25 pin 2 to LS00 pin 1 or LS138 pin A0, and data bit 3 connects via DB25 pin 5 to LS138 pin E1, etc. The DB25 pins are set up in arrays in the provided chiptest.c code.

Mtip systems set up for this project

John Lentz has set up online SAPCs 5 and 6 each with a LS00 chip connected in the prescribed way to LPT1, and SAPCs 7 and 8 each with a LS138 chip. He has also provided diagnostic programs you can use to check the setups: testLS00.lnx and testLS138.lnx. Simply download the appropriate test and run it to check the hardware.

Scripts of such runs are available in testLS00.script and testLS138.script.

Your assignment:

1. Write chiptest.c by filling in the missing code. In general, do *not* use special-case code to do one thing for ls00, another for ls138. Instead use the info available in the Chip object to guide your code so it would work for *any* chip that can be described by a Chip.

2. Test chiptest on SAPCs 5 or 6, which both have good LS00 chips attached. Use SAPCs 7 and 8 as having bad LS00s to make sure your program reports errors. Read and run the shell script chiptest.sh and watch it run mtip and your program.

3. Write ls138_chip.c, specifying the 3-to-8 decoder. You'll want to use C logic rather than a truth table lookup in function softchip(). Use SAPCs 7 and 8 to test as good LS138s and 5 or 6 for bad ones.

4. Write chip_discussion.txt, in which you discuss the design problems you encountered and solved, and how you tested your program. Your chiptest program works only for chips implementing combinatorial logic. Testing sequential logic chips (memory chips, LS175's with flipflops) is more subtle. Discuss why. Suggest extensions to the contents of the _chip.c file and to the algorithms you would need to build such a program.

Please turn in a hard copy of:

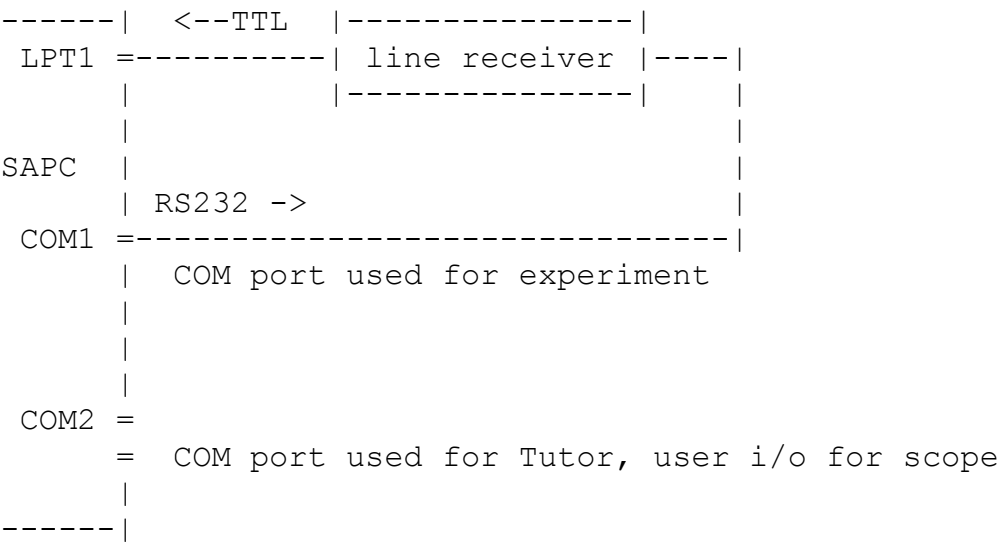
chip_discussion.txt
chiptest.c
ls00_chip.c
ls138_chip.c

Note: make sure chiptest.sh works at delivery time. INTERACTIVE should be defined to make this work.

A DIGITAL OSCILLOSCOPE

It would be nice to look at a serial byte in flight with an oscilloscope or logic analyzer. These fairly expensive devices track signals over time, slicing it down to microseconds or below. For slow enough signals, we can do the same thing with our microprocessor system. We just connect the transmit line from a UART (not the console COM port) to a line of LPT1 capable of receiving input. We need to make sure that the input to LPT1 is TTL compatible, not RS232, by using a line receiver, a chip that does the RS232 to TTL voltage conversion.

Sketch of Experimental setup: SAPCs 9 and 10



We can write a program that generates a serial byte by writing to the COM1 UART register and then monitoring the resulting signal through LPT1, storing successive values in an array in memory for playback after the character transmission has completed.

We've seen that "small" instructions take about 2 ns, so that one pass through a data-collection loop (say 20 instructions) should take about 40 ns, plus possibly as much as 1 usec to read the LPT1 port. One bit at 9600 baud (bits/sec, approximately) takes about 100 usecs, so it should be possible to collect about 100 data points in every bit-time.

The data collection loop itself is a candidate for an assembly language routine called from C, since we want it to go as fast as possible. We can pass (to the as function) the starting address of the array to hold the data and the number of spots in the array to fill. When this loop is done, the C program can go on at leisure to display the collected results.

To display the results semi-graphically, we can use "_" for a low point and "|" for a high point. For more than 4 data consecutive data points all low, we need to summarize (or we'll fill the whole screen with confusing data). Use __63__ to denote 63 consecutive lows and ||63|| for 63 consecutive highs, for example. Thus a valid data display would be:

||88||__67__|__||___||82||

Here there were 88 highs, the 67 lows, then 1 high, then 2 lows, then 2 highs, then 3 lows, then 82 highs. With this notation, we should be able to display a serial byte on one line, or two at the most. The code to do this display is provided in scope.c, along with the code for the user interaction to obtain the needed character and baudrate. Also see scope.script for a script of a working scope program.

The user gets to choose what byte to display -- echo it normally (as a character and in binary) and then as the signal trace captured by the 'scope. The user interface code is provided in scope.c.

Mtip systems set up for this project

We have set up online SAPCs 9 and 10 to take COM1's TXD line through a line receiver and back to LP_BUSY on LPT1. He has also provided a diagnostic program test-scope.lnx. Just download and run it and follow its directions to test the hardware. See test-scope.script in this directory for a script of a run of test-scope.lnx.

Debugging

Note that because we are using COM1 for the experiment, we don't have it connected for remote gdb, on SAPCs 9 and 10. You can run the program on a system that can do remote gdb, but then beware that any output (or baudrate settings) you do on COM1 could affect your remote gdb session. Probably it's best to use a combination of gdb on the UNIX build plus Tutor plus good old printf's.

1. Write scope.c and ccollect.c, the C version of code for the basic collection loop:

```
void collect(char * data, int maxdata);
```

Once collect is called, it just loops, reading the LPT1's status register and putting the values in the data array. To start with, leave set_baudrate as a no-op function (as it is provided). Tutor initializes COM1 to be 9600 baud, so that will hold.

Test scope.lnx on SAPCs 9 or 10. Read and run the shell script scope.sh and watch it run mtip and your program.

2. Reimplement ccollect.c in assembly language ascollect.s.

3. Use scope.lnx to display the serial signals generated under varying circumstances -- different characters, different transmission speeds. For the latter, implement the function

```
void set_baudrate(int dev, int baudrate)
```

that sets the specified baudrate for the specified COM line dev.

4. Write scope_discussion.txt, in which you discuss the design problems you encountered and solved, how you tested your program, and the cases tested and what happened. Did assembly language speed up the collection loop? Did you try optimization (-O or -O2, for example) on the C loop?

What happened with different baudrates? Some standard baud rates are: 2400, 4800, 9600, 19200, 38400, and 115200. You don't have to try all of them but try a few at each end of the list.

The formula for the divisor … is 1843200/(16 x Baud Rate), because PC UARTs use a 1.8432-MHz clock frequency and the UARTs need a 16 x baud rate input to sample the input in the middle of the baud period. So ... 1 corresponds to the baud rate 115200, and 12 corresponds to 9600.

Note that the provided collect files are empty. They were created by the "touch" command to check that the Makefile works OK. Try out "make scope.lnx" "make fscope.lnx" and see that the build works.

Please turn in a hardcopy of:

scope_discussion.txt
scope.c # main program
ccollect.c # C collect loop
ascollect.s # assembler collect loop

Note: make sure scope.sh works at delivery time. INTERACTIVE should be defined to make this work.