COSC-251, Computer Systems Fundamentals
Spring, 2006

Syllabus

Instructor:

Prof. Richard K. Squier
St. Mary's Hall, Room 339
Computer Science Dept.
Georgetown University
202-687-6027
squier@cs.georgetown.edu

Office hours: Tue-Thu, 3:00-4:00, or by appointment, or drop in anytime.

Description:

The course takes two different approaches to understanding computer systems. One concentrates on the low-level hardware/software systems interactions by incrementally building an extremely primitive software system (a booter/loader/OS), that boots a standard x86 in real mode and provides simple services (loading and execution, and keyboard and video I/O). The other approach looks at a pipelined processor implemented in verilog. The higher-level system interactions are explored by modifying a basic pipelined MIPS processor implemented in verilog simulation code. The modifications will include a series of memory hierarchy components: main memory, a Memory Management Unit (MMU), and instruction and data caches (direct and associative). Test programs and simulation code are developed to confirm correctness and evaluate performance. Weekly assignments, two semester projects, and a final exam. Course notes are posted after lectures on the course web site. Prerequisite: COSC-250.

Texts:

Required:
Computer Organization and Design, 3/e, Hennessey and Patterson (Morgan-Kauffman)

Recommended:
Introduction to Computing Systems, 2/e, Patt and Patel (McGraw-Hill)
The material on the C programming language and the IA32 (x86) architecture will be helpful. However, other sources for this material are easy to come by, for instance, online sources.

Assignments:

Assignments consist of a series of exercises culminating in two semester projects: picoOS, a low-level interface to system resoures, and MIPs simulation, modifying a verilog model of the MIPS processor. Specific exercises will be posted on the course web site.

Readings:

COD, chapters 5.1-.6, 5.8, 6.1-.8, 7.1-.4, 8.1-.5, and Appendix B. This reading will be coordinated with the development of the MIPS simulation project. Also, online material will be posted on the course web site (IA-32 manuals, debugger documentation, boot information, BIOS information, assemblers and compilers and their documentation, boot floppy images, links to tutorials.

Exercises:

Reading and homework exercises will be posted on the course web site. They include a series of warm ups for the picoOS project, and x86-based primitive operating system:

1. The 80186 standard architecture, using a low-level debugger, assembly and unassembly, x86 addressing, interpreting instruction bytes, modifying boot sectors, writing to VRAM, writing modified boot blocks to disk.
2. Create an executable and a boot block, load them to disk using BIOS-13, boot and run, use the FAT structure to locate the executable.
3. Have your booter load the executable to memory, initialize its environment, and start execution; have the executable write directly to VRAM.
4. Redo the above, but use C for higher-level routines, linking your C modules to your assembly modules.
5. Create an OS service for video output, accessed via interrupt (system call) routines. Have your picoOS booter set it up a boot.
6. Add a keyboard interrupt handler to your OS.
7. Create your OS main loop with command input.

picoOS Project

Expand your OS to have the following capabilities.

(1) issue a prompt to screen

(2) read keyboard input to detect two commands: "L" and "R".

"L" causes the system to display a list of programs that can be loaded and run by the OS. Program file names can be restricted to single characters.

"R n" loads and runs the n-th program (or use whatever naming system you care to implement to refer to the user-level programs).

(3) Load to your bootable disk at least three user-level programs that can be listed using "L", and run using "R". These programs need not do anything but produce some output so that one can tell they actually ran. They can loop infinitely after that, or you can provide a means for user-level programs to return to the OS's main loop. (N.B. For that last bit, a new OS service routine to jump back to the OS would probably be the best bet.)

Any inhancements beyond the bare bones capabilities receives extra credit, of course; for

instance, video modules that keep track of screen position, keyboard service code that does scancode-to-ascii conversions, or OS function calls for user-level programs that provide keyboard services.

MIPS simulation project

BEQ test

The end goal of the MIPS-simulation project is to write a verilog simulation to show the performance of pipelining and caching. We will modify the basic MIPS cpu pipeline code (~squier/verilog/h2.vl), and write test benches to demonstrate performance. The project will be broken up into incremental pieces. The final project is due on the date and time of the final exam. Turn in each part as soon as you have it ready. I will not announce deadlines for the individual project parts. Do not delay moving ahead with this project as there is not much time left in the semester.

For MIPS-simulation-BEQ-test, we will

(1.) Write a test bench to test the correctness of the BEQ instruction execution. You may want to start by modifying the test bench, prog3.vl. You should write up a short description of how your test bench tests the BEQ execution and what conclusions you draw from the simulation output. If necessary, modify the cpu module in h2.vl to correct any problems. Turn in hardcopy of any verilog code used and simulation output.

(2.) Modify h2.vl to change the register file to be negative-edge triggered. Write a test bench to verify the correctness of the cpu's.

You may want to do parts (1.) and (2.) in reverse order. For both parts, include in your writeup either a disk w/ your source code, or indicate the location of your source code on karma.georgetown.edu.

Memory units

In this part, we will move memory accesses to separate modules. The pipeline clock will be stalled (disabled) until a ready signal is returned from the memory unit. For the first version we will have two modules, one for IMemory, and one for DMemory. Accesses to the two memory units will be in parallel, so both memory units must be ready before the pipeline clock is enabled. (It may be the case that only the FETCH stage is doing memory access, in which case only the Imemory's ready signal is required to enable the pipeline clock.)

Main Memory

Next, add a main memory unit that services both the IMemory and DMemory units. Since this memory unit services both, the IMemory and DMemory units must access main memory sequentially if both units are needed at the same pipeline cycle.

Direct Cache

Now add direct caches to both the IMemory and DMemory. On cache hits, the read or write executes without stalling the pipeline. Misses cause the pipeline to stall while a new cacheline is fetched.

Associative Cache
Modify the caches to be associative. (Can be *n*-way or fully associative.)

Grading:

Grades are determined by one-on-one interview inquiries into your projects, and a final exam. Keep all your work together to bring to these interviews. Weights will be determined as the semester develops and the general level of progress on projects can be assessed. A final interview will detail the considerations and information and method used in determining the final grade. Extra credit is given liberally for projects or other work that go beyond the stated requirements.

Academic Integrity Policy:

This course is, of necessity, filled with a great many details that are merely peripheral to what I hold is the intellectual content of the course. In assigning grades, my job as instructor is to ascertain your growth in understanding that intellectual content during the course of our studies together. Course assignments and projects require you to delve into the sea of minutiae that goes along with understanding the intellectual content. Many times, this minutiae may actually seriously hamper your efforts to understand. For that reason, I encourage you to freely exchange information.

The default policy for the Computer Science Department is amended as follows. You are free to discuss problems and solutions of any assignment or project with your classmates or others. You need not cite these conversations nor indicate which parts of your submitted material was garnered from such conversations. You are free to collect information from any source, electronic or otherwise, and you need not indicate the original source nor that the material did not originate with you.

For most courses, this policy probably would make little or no sense, but you will soon see that here it makes perfect sense: there simply is too much detail to be attended to, and borrowing is so central to the enterprise, that annotating this would encumber the effort beyond all practical utility. In addition, in this context, I consider it a fault to withhold useful information from others. You will discover that my grading system does not depend on evaluating your progress based on material of unknown origin. However, I will use your submitted material as a guide in exploring your progress and effort. Simply put, we will get to know each other.

You will be given every opportunity to demonstrate your knowledge and ability through written and oral means. If you feel you are not being evaluated thoroughly enough, it is encumbent on you to bring this to my attention while there is still time to address your concerns before grades are submitted. You are welcome to discuss these issues with me at any time. You might find the recently adopted Association for Computing Machinery (ACM) guidelines on plagiarism interesting:

http://www.acm.org/pubs/plagiarism%20policy.html