ECE/CS 250 – Prof. Board
Recitation #09
Pipelining and Free Exploration

Objective: In this recitation, you will explore real-world practicalities of computer architecture.

DIRECTIONS: This recitation collects previous "time allowing" activities and introduces a few new ones. Pick **one or two** and do them during recitation. After that, you can use the time to work on the current homework, continue to explore these tasks, develop your own experiments to run or free-form questions to ask, or use the time for final exam Q&A with the TAs.

1. Pipelining: Types of Dependences and the Hazards They Create

There are multiple kinds of data dependences in programs. For the pipeline we learned in lecture, the one that can lead to hazards is RAW (read-after-write). You can also have WAR (write-after-read) and WAW (write-after-write) dependencies, which would affect designs that do out-of-order execution.

It's useful to be able to identify RAW, WAR, and WAW dependences through registers and through memory. There's no need to worry about RAR (read-after-read), since these "dependences" are never problematic.

Some dependences lead to hazards and some don't. Whether a dependence leads to a hazard is a function of the pipeline design.

(a) Construct a 2-line MIPS code snippet that has a RAW dependence through a register.
(b) Construct a 3-line MIPS code snippet in which the 2nd and 3rd instructions have RAW dependences on the register written by the 1st instruction.
(c) Construct a 2-line MIPS code snippet that has a RAW dependence through memory.
(d) Construct a 2-line MIPS code snippet that *may or may not* have a RAW dependence through memory. (Think about how you might not be able to tell if there's a RAW dependence!)
(e) Construct a 2-line MIPS code snippet that has a WAR dependence through a register.
(f) Construct a 2-line MIPS code snippet that has a WAW dependence through a register.
(g) Construct a 3 or 4 line MIPS code snippet that has at least one RAW, WAR, and WAW dependence.

2. Pipelining in the Real World

Time permitting, break up into groups of 3 or 4 people each, and in each group, pick one of these real-world processors. UTAs will help to make sure that every group has a different processor.

1. Intel's Core i7, code name: Broadwell

2. Intel's Atom, code name: Cedarville
3. AMD's Bobcat, code name: Llano
4. ARM's Cortex-M7
5. IBM's Power7
6. Qualcomm's Scorpion CPU
7. Alpha 21364 (also called EV7)
8. Atmel ATMEGA328 microcontroller[1]

Now learn as much as you can about your chosen processor:

(a) What is the ISA?
(b) How many registers does it have?
(c) How many stages are in its pipeline?
(d) What are those stages? Don't be worried if many stage names don't make sense yet, e.g., "register renaming", "wakeup", etc.
(e) What is the clock frequency? (There is likely to be a range of frequencies.)
(f) What are the sizes and associativities of the caches (L1I, L1D, L2, L3)? What are the cache block sizes?
(g) What are the sizes and associativities of the TLBs?

3. Running out of Virtual Memory

On a 64-bit machine, it might appear you could never possibly run out of virtual memory. But what happens if you write a recursive program that (due to a bug) never reaches its base case? Try this. What happens?

4. Running out of Physical Memory

You can never actually run out of physical memory, but you can try to use more memory than you have, in which case the computer spends a lot of its time paging (servicing page faults). Open up the Task Manager to see how much memory you're currently using. Now start launching programs that use a lot of memory. What do you see in the Task Manager? (or using "top" in a console window on a Mac)

5. Caches and Memory in the Real World

1) How much cache is in your laptop? There's two ways to tell – you can find your CPU model and google it to find the stats OR install a special program that shows cache stats directly:

Platform	To find CPU model manually	CPU info program
Windows 10/11	Open start menu, type "processor", pick "View processor info".	CPU-Z

[1] This is a microcontroller rather than a CPU, meaning it has many general-purpose IO pins for interfacing with custom circuits. You'll find that it's different from the CPUs in this exercise in a few other ways, too.

Mac*	Go to "Apple Logo" > System Report	MacCPUID (if you have a newer non-Intel Mac, just look up to chip stats)
Ubuntu Linux	`cat /proc/cpuinfo`	CPU-X: `sudo apt install cpu-x`

2) How much physical memory do you have? This can be found in most of the places/apps listed above, and on Linux you can also run "top" or look at "proc/meminfo".
3) How much disk space do you have? See Windows Explorer, Mac Finder, or Linux `df` command.
4) How much is being used for "swap" (i.e., to hold pages that don't currently fit in physical memory)? See Windows Task Manager, Mac Activity Monitor, or Linux `top` command.

6. Benchmarking memory and cache

1) Download the program "memdance.c" from the course site and compile it with:
   ```
   g++ -O3 -o memdance memdance.c
   ```
   This program is a simple benchmark to measure the rate of random memory access to a block of memory of a given size for a given duration of a time. Run it without arguments for help.
2) Use this program on the machine of your choice to measure the memory throughput for different buffer sizes. Just run it as "`./memdance default`", and it will cycle through buffer sizes of 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, and 128MB for 3 seconds per test.
3) Compare the results to the cache size found in "Caches and Memory in the Real World" above, especially the lowest level cache (L3 on most modern systems). What do you observe? What's the percentage difference between the fastest and slowest rate? What does this tell you about the importance of cache to software performance?

ALL DONE?

You did *every* task? Wow. Well, don't head out yet – work on the current homework and talk to the TAs for help. You can leave if your homework tester shows all passing and you've turned in all the written & code materials.