Slide 1

Administrivia

- Reminder: Homework 6 due today. Last homework — I had thought there should be one more but no.
- Quiz 6 to be next Wednesday (moved from Monday).

Slide 2

Quiz 5 — Review

- Very few people answered the question I intended — whether longer pipelines offered advantages over shorter ones — but I graded leniently.
- Might be worth noting that descriptions of current processors do mention longer pipelines as a way to improve performance, so there must be *some* potential advantage? see posted solution.

Slide 3

Memory Hierarchy — Recap/Review/Revisited

- In a perfect world, there would be a way to store bits that's very fast and can be had in almost arbitrarily large amounts for a reasonable cost. In this world — a maxim from engineering (or so I hear): "Good, fast, cheap — pick any two."
- Textbook talks about four basic technologies for storing (lots of) bits:
  - SRAM — pretty fast, but costly, so not feasible on a large scale.
  - DRAM — significantly less expensive but also significantly slower.
  - "Flash memory" — slower but cheaper still, but does have the problem of "wearing out".
  - Magnetic disks — cheap enough to be about as big as is needed for most general-purpose computing, but far, far slower.

Slide 4

Memory Hierarchy — Recap/Review/Revisited

- So where does "hierarchy" come in? Well . . .
- Programs' use of memory mainly exhibits "locality" (in both time and space).
- So it's common to design systems in terms of a hierarchy, with each level larger but slower than the one above it, with the hope that we can store (a copy of) most-frequently-used data in an upper level of the hierarchy, where it's fast to get at, and access lower levels less frequently.
- Idea is that data moves up and down in this hierarchy as needed, all in a way that's invisible to application programs, *except* for effects on performance.

Slide 5

Caching — A Bit More Detail

- In order for this to work, each "cache" (hardware or virtual memory) must have space for some data from the next level down, plus some way of (correctly!) reading from / writing to next level down, which means having some way to map from lower-level addresses to elements.
- Idea is that for reads, processor just reads using address as we've discussed, and either:
  - Data is found in the cache — "cache hit" — and given back to processor.
  - Data is not found — "cache miss" — and hardware/software does whatever is necessary to get it there and then continues as for hit.

  Obviously the fewer caches misses the better(?).

Slide 6

Caching — A Bit More Detail, Continued

- But wait — if cache is smaller than what it's caching, how can this work? each cache element could potentially contain one of many pieces of data? So include in cache element a "tag" that says which one it contains, plus a "valid" bit.
- For writes, things are a bit more complicated — similar idea applies, but must decide whether to write to lower levels immediately or wait. Writing immediately is easier but slower, probably enough so that it's worth the trouble to do something more complicated. More details in textbook.
- Overall, textbook (section 5.8) presents four questions that pretty much sum it up; adding one of more . . .

Slide 7

Caching — Size of Elements

- Processor caches *can* store single words, but might store larger units (2 words, or 4, or . . .) — "cache lines". Idea is to exploit spatial locality.
- Virtual memory typically uses much bigger units (often "pages" of 2K or 4K).

Slide 8

Caching — Mapping Addresses to Cache Elements

- "Direct map" cache is simple — each memory address maps to exactly one cache element.
- "Fully associative" cache is opposite extreme — any memory address can map to any cache element.
- "Set associative" cache is in between — each memory element maps to a set of entries. Reasonable compromise between extremes?

Slide 9

Caching — Looking Up Data

- For “direct map” cache, simple — only one cache element to check, so just compare tags. So this method is fast but not very flexible.
- For “fully associative” cache, more complicated — potentially have to search whole cache for matching address. Very flexible but costly to implement with good performance.
- For “set associative” cache, in between — still have to check multiple elements, but fewer of them. Reasonable compromise between extremes?

Slide 10

Caching — Mapping Addresses to Cache Elements, Revisited

- Which is used? for virtual memory, likely fully-associative; for processor caches, one of the others.

Slide 11

Caching — Replacing Cache Elements

- On a "cache miss", if appropriate cache elements are all in use, must pick one to replace. For direct mapping, trivial (only one choice); for the other two not so trivial.
- How to choose? goal should be to replace something that won't be needed again, and often approaches are based on temporal locality (if not used recently maybe won't be used again soon).
- For processor caches, hardware problem, various solutions exist; for virtual memory, software (O/S) problem, and again various solutions exist ("page replacement algorithms").

Slide 12

Caching — How to Manage Writes

- One complication here is that if cache elements are more than one word, need to read old element, then change the word being written.
- And then — write back immediately ("write-through"), or wait (write buffer or "write-back")? former is easier but could be quite slow; latter is more complicated but probably needed for acceptable performance.

Slide 13

Virtual Machines

- There's been increasing interest lately in "virtual machines" / "virtualization". Some are purely software (e.g., Java Virtual Machine) but others involve or at least rely on hardware.
- Idea actually goes back a long time — supported in 1970s by IBM's VM/370, which was (or "is"?) in some sense a stripped-down O/S that allowed running multiple "guest O/S"es side by side. Very useful in its time — physical machines often needed to be shared among people with very different needs w.r.t. O/S. Current versions include the VMware ESX server (other examples in textbook, but this name I recognize).

Slide 14

Sidebar: Dual-Mode Operation

- For simple single-user machines it's more or less reasonable to allow any application to do anything the hardware can do (access all memory, do I/O, etc.) — though even there it means whatever O/S there is is vulnerable to malicious or buggy programs.
- For anything less simple, useful to have a notion of two modes, regular and privileged, where in regular mode some instructions are not permitted (attempts to execute them cause hardware exceptions) (e.g., instructions to do I/O). Normally at least some O/S code runs in privileged mode, and applications in regular mode. Makes it possible for the O/S to defend itself from malicious or buggy applications, and also avoids applications interfering with each other.

Slide 15

Virtual Machines — Semi-Executive-Level Summary

- What the real hardware is running is a "Virtual Machine Monitor", a.k.a. "hypervisor" (term analogous to "supervisor", a term for O/S). Interrupts and exceptions transfer control to this hypervisor, which then decides which guest O/S they're meant for and does the right thing.
- This all works better with hardware support for dual-mode operation: Guest O/S's run in regular mode, and when they execute privileged instructions (as they more or less have to), the hypervisor gets control and then can simulate . . .
- Other than than, programs run as they do without this extra layer of abstraction — they're just executing instructions, after all?

Slide 16

Virtual Machines — Semi-Executive-Level Summary, Continued

- Some architectures make this easier than others — they're "virtualizable".
- Interestingly enough(?), IBM's rather old 370 does, but for many newer architectures the needed support has had to be added on, not always neatly. "Hm!"?
- (Textbook has a few more details, in section 5.8.)

Slide 17

Minute Essay

- Anything noteworthy to report about Homework 6? do you feel like you understand better how the simplified processor works?