Slide 1

Administrivia

- Exam 2 graded. Overall scores pretty good!
- Grade summaries to be mailed soon.
- Extra-credit assignment updated.

Slide 2

Quote of the Day/Week/?

- "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

  — C.A.R. Hoare
- Seems very relevant in the context of circuit design, at least as done in this course!

Slide 3

Homework 7 Essays

- Several said easier than they thought; several said examples and/or videos helped. (Good to hear!)
- Several said indeed tedious but instructive. One mentioned that it helps to know what answers should be. Indeed!
- One said "interesting how simple mechanisms can create complex behavior". Agreed!
- A few found assignment entertaining in a way.
- One referenced a set of videos https://eater.net/8bit. Looks interesting!

Slide 4

Homework 8 Essays

- Several commented that problems helped with understanding. One said assignment was his favorite because it required understanding. Nice to hear!
- One said he thinks he finally understands how computers work top to bottom, aside from some gaps about O/S; proud of that (should be!).
- One person said he had one of the complicated figures onscreen, roommate came by and "gawked". Indeed!
- Several people who worked together said problems were hard to do as a group because everyone had different ideas. (Sadly, the consensus they came up with was wrong.)

Slide 5

Memory Hierarchy — Recap/Review/Revisited

- In a perfect world, 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: "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 6

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, common to design systems in terms of hierarchy, with each level larger but slower than one above it. Idea is then to store (a copy of) most-frequently-used data in upper levels, 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 7

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 8

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 a bit more complicated: Similar idea applies, but must decide whether to write to lower levels immediately or wait. Writing immediately 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 more . . .

Slide 9

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 10

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 11

Caching — Looking Up Data

- For “direct map” cache, simple: Only one cache element to check, so just compare tags. So, 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 12

Caching — Mapping Addresses to Cache Elements, Revisited

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

Slide 13

Caching — Replacing Cache Elements

- On “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 soon. Often approaches 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; again various solutions exist (“page replacement algorithms”).

Slide 14

Caching — How to Manage Writes

- One complication: If cache elements are more than one word, need to read old element, then change 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 15

Virtual Machines — Executive-Level Summary

- Increasing interest lately in "virtual machines" / "virtualization". Some purely software (e.g., Java Virtual Machine); others involve or at least rely on hardware.
- Idea actually goes back a long time: IBM's VM/370 (1970s), a sort of 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. Successors still in use!
- Textbook has other examples; one I recognize is VMware ESX.

Slide 16

Virtual Machines — Semi-Executive-Level Summary

- What the real hardware runs: "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.
- All works better with hardware support for dual-mode operation: Guest O/S's run in regular mode; when they execute privileged instructions (as they more or less have to), 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 17

Virtual Machines — Semi-Executive-Level Summary, Continued

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

Slide 18

Parallel Computing — Overview

- Support for "things happening at the same time" goes back to early mainframe days, in the sense of having more than one program loaded into memory and available to be worked on. If only one processor, "at the same time" actually means "interleaved in some way that's a good fake". (Why? To "hide latency".)
- Support for actual parallelism goes back almost as far, though mostly of interest to those needing maximum performance for large problems. Somewhat controversial, and for many years "wait for Moore's law to provide a faster processor" worked well enough. Now, however . . .

Slide 19

Parallel Computing — Overview, Continued

- Improvements in "processing elements" (processors, cores, etc.) seem to have stalled some years ago. Instead hardware designers are coming up with ways to provide more processing elements.
- One result is that multiple applications can execute really at the same time.
- Another result is that individual applications *could* run faster by using multiple processing elements.

  Non-technical analogy: If the job is too big for one person, you hire a team. But making this effective involves some challenges (how to split up the work, how to coordinate).
- In a perfect world, maybe compilers could be made smart enough to convert programs written for a single processing element to ones that can take advantage of multiple PEs. Some progress has been made, but goal is elusive.

Slide 20

Parallel Computing — Hardware Platforms (Overview)

- Clusters: Multiple processor/memory systems connected by some sort of interconnection (could be ordinary network or fast special-purpose hardware). Examples go back many years.
- Multiprocessor systems: Single system with multiple processors sharing access to a single memory. Examples also go back many years.
- Multicore processors: Single "processor" with multiple independent PEs sharing access to a single memory. Relatively new, but conceptually quite similar to multiprocessors.
- "SIMD" platforms: Hardware that executes a single stream of instructions but operates on multiple pieces of data at the same time. Popular early on (vector processors, early Connection Machines) and now being revived (GPUs used for general-purpose computing).

Slide 21

Parallel Programming — Software (Overview)

- Key idea is to split up application's work among multiple "units of execution" (processes or threads) and coordinate their actions as needed. Non-trivial in general, but not too difficult for some special cases ("embarrassingly parallel") that turn out to cover a lot of ground.
- Two basic models, shared-memory and distributed-memory. Shared-memory has two variants, SIMD ("single instruction, multiple data" and MIMD ("multiple instruction, multiple data"). SPMD ("single program, multiple data") can be used with either one, and often is, since it simplifies things.

Slide 22

Shared-Memory Model (MIMD)

- "Units of execution" are (typically) threads, all with access to common memory space, potentially executing different code.
- Convenient in a lot of ways, but sharing variables makes "race conditions" possible. (Now that you know more about how hardware works you may understand the issues better! A single line of HLL code may translate to multiple instructions . . .)
- Typical programming environments include ways to start threads, split up work, synchronize. OpenMP extensions (C/C++/Fortran) somewhat low-level standard.

Slide 23

Distributed-Memory Model

- "Units of execution" are processes, each with its own memory space, communicating using message passing, potentially executing different code.
- Less convenient, and performance may suffer if too much communication relative to amount of computation, but race conditions much less likely.
- Typical programming environments include ways to start processes, pass messages among them. MPI library (C/C++/Fortran) somewhat low-level standard.

Slide 24

SIMD Model

- "Units of execution" term may not make sense. Parallelism comes from all processing elements executing the same program in lockstep, but with different processing elements operating on different data elements.
- Excellent fit for some problems ("data-parallel"), not for others. Very convenient when it fits, pretty inconvenient when not.
- Typical programming environments feature ways to express data parallelism. OpenCL (C/C++) may emerge as somewhat low-level standard, especially suited for GPGPU.
- Parallel collections (as in Scala) probably fit here. Performance may not be great at this point but may well improve.

Slide 25

Distributed Programming

- All approaches mentioned so far rely to some extent on multiple UEs executing more or less synchronously. Works well for classic high-performance computing, where problems involve relatively frequent need for multiple threads of execution to exchange information. (Think simulation of large-scale physical system.)
- However, with some problems there's less need for thread of execution to communicate (think anything involving exploring multiple more or less independent possibilities).
- Various frameworks exist for this. Sadly, not something I know enough about.
- "Actors" model as used in Scala seems to fit best here.

Slide 26

Shameless Self-Promotion

- It's not (quite) too late to add CSCI 3366 for next fall! Likely will not be offered again for two years.
- What's the course about? I sent a description to CSMajors some time ago, and if you read that you know. (And if you didn't and were curious, um, . . .)
- Short version:

  When I've taught it previously, the focus has been on a somewhat low-level view and traditional HPC applications. Programs in C with OpenMP, C with MPI, C with OpenCL, Java. No exams; programming assignments and a project more or less of your choice.

  Likely I'll still do most of that but may include more GPGPU programming, more about distributed programming.

Slide 27

Minute Essay

- How did Exam 2 compare to your expectations? with regard to length, difficulty, topics?