Slide 1

Administrivia

- All homeworks graded. If you missed one, I'm willing to accept it for partial credit (provided of course that you haven't looked at a sample solution!) through next Wednesday.
- I will grade Exam 2 soon and send out grade summaries.
- Extra-credit problems posted. Totally optional and can only help your grade (I add any points to the "your points" number without changing the "total points" number.) Due May 8 at 5:30pm (firm deadline).
- About office hours, I may be on campus a few days next week (details by e-mail early next week) and either way will plan to do some virtual office hours (times TBA). And I do respond to e-mail pretty well!

Slide 2

Minute Essay From 4/11

- (Not a lot of responses, but most were fairly sensible. Review question / my answer.)

Slide 3

Virtual Machines — Executive-Level Summary

- 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 4

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 5

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 6

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 7

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 8

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 9

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 10

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 11

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 12

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 13

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.

Slide 14

Shared-Memory Hardware, Revisited

- Figure 6.7 sketches basic idea — multiple processing elements (call them processors, cores, whatever) connected to a single memory.
- Synchronization (locking) *can* be done with no hardware support, but it's tricky. Simple approach is something such as:

```
while (lock != 0) {};
lock = 1;
```

  which doesn't work because test and set are separate instructions. (What goes wrong?)
- Somewhat-tricky algorithms exist for solving this problem in software, but . . .

Slide 15

Shared-Memory Hardware — Locking

- Locking is much easier if ISA provides some support, in the form of an instruction that allows . . . Well, essentially allows both read and write access to a location *as a single atomic operation*.
- Some architectures implement this directly, via a "compare and swap" or "test and set" instruction. But for MIPS that might be challenging (why?).
- So MIPS defines two instructions, "load linked" (`ll`) and "store conditional" (`sc`). Tricky, but an example may help some.

Slide 16

Shared-Memory Hardware — Locking, Continued

- Example from text, p. 122, to exchange a memory location with register contents, slightly modified (first line is wrong?!):

```
again:  add     $t0, $zero, $s2 # $t0 <- $s2
        ll      $t1, 0($s1) # $t1 <- 0($s1)
        sc      $t0, 0($s1) # $t0 -> 0($s1) IF unchanged
        beq     $t0, $zero, again  # try again if changed
        add     $s2, $zero, $t1 # $s2 <- old value of 0($s2)
```

- How this works: The `ll` "remembers" the load-from address. The following `sc` "succeeds" only if the value at that address hasn't been changed (e.g., by another processor). Tricky!

Slide 17

Shared-Memory Hardware — Memory

- Access to RAM can be reasonably straightforward — only one processor at a time. Caches complicate things (next slide).
- "Single memory" may actually be multiple memories, with each processing element having access to all memory, but faster access to one section ("NUMA" (Non-Uniform Memory Access)). Making good use of this can affect performance — and may be non-trivial to accomplish, especially if programming environment doesn't give you appropriate tools.

Slide 18

Shared-Memory Hardware — Caches

- As noted, even if access to RAM is one-processor-at-a-time, if each processing element has its own cache, things may get tricky. Typically hardware provides some way to keep them all in synch (the "cache coherency" problem discussed in Chapter 5).
- Further, application programs may have to deal with "false sharing" — multiple threads access distinct data in the same "cache line". Cache coherency guarantees correctness of result, but performance may well be affected. (Example — multithreaded program where each thread computes a partial sum. Having the partial sums as "thread-local" variables can be much faster than having a shared array of partial sums.)

Distributed-Memory Hardware, Revisited

- Figure 6.13 sketches basic idea — multiple systems (processor(s) plus memory) communicating over a network.
- No special hardware required, though really high-end systems may provide a fast special-purpose network.

Slide 19

SIMD Hardware

- Various ways to implement this idea in hardware.
- One approach: multiple processing elements sharing access to memory and all executing the same instruction stream,

  This is more or less how GPUs work. A complication — they often have a separate memory, so data must be copied to/from RAM. Potential performance problem, may be cumbersome for programmers.
- Another approach: “vector processing units” that stream/pipeline operation on data elements to get the data-parallelism effect.

Slide 20

Slide 21

Other Hardware Support for Parallelism

- Instruction-level parallelism (discussed in not-assigned section(s) of Chapter 4) allows executing instructions from a single instruction stream at the same time, if it's safe to do so. Requires hardware and compiler to cooperate, and (sometimes?) involves duplicating parts of hardware (functional units).
- Hardware multithreading (discussed in Chapter 6) includes several strategies for speeding up execution of multiple threads by duplicating parts of processing element (as opposed to duplicating full PE, as happens with "cores").

Slide 22

Course Recap — Topics

- A little about performance. (It's not simple!)
- MIPS assembler language; translating C to MIPS assembler language.
- Compiling, assembling, and linking.
- Binary representation of instructions.
- Binary representation of data (integers, ASCII, floating-point numbers); basics of computer arithmetic.
- Gate-level logic design.
- Design of a processor — ALU, datapath, control; a little about pipelining.
- A little about caches.
- Other schools spread this material over two or even three courses (though they presumably cover more in all). So, we have done a lot?

Slide 23

Course Recap, Continued

- Some topics — representation of data, computer arithmetic, maybe finite state machines — are review, or small extension of what you know.
- Others, though — assembler language, gate-level logic, designing a processor in terms of AND/OR/NO and how it works — are not familiar to most, and involve a new perspective, or mindset, or "mental model". My observation — some students take to it, others struggle.
- I do hope, however, that all of you have come away with more understanding of how things work "under the hood" than you had!

Slide 24

Minute Essay

- How did you like having "class" via video lecture? my thinking is that the videos can be replayed, which could be a plus, but they don't offer an easy way to ask questions.

  If you've taken another course in which lectures were all on video and class time was used for something else, how did that work for you? If you haven't, does it sound like something you'd like? (Now that I know how to make video lectures I'm speculating about how to use them as a supplement rather than substitute.)
- And best wishes for a good summer break!