Slide 1

Administrivia

- Reminder: Homework 3 due Wednesday. (I just updated/clarified the instructions for the assemble-and-link problem; I hope they're clear(er) now.)
- Quiz 3 next Monday. I'll say something Wednesday about likely topics.
- Sample solutions to quizzes online, but now linked from the bottom of the "lecture topics" page.

Slide 2

Quiz 2

- Generally people didn't do very well with the second question.
- One common mistake was to use `add` or `addi` to assign a value to an array element. (You need `sw` for that.)
- Even more common was leaving out a `j` so that after the "if" part executes the "then" part is skipped. Remember that the processor executes instruction in the order in which they appear in the code, *unless* there's an explicit branch or jump instruction.

Slide 3

This and That

- If you haven't already found this — there *is* a table mapping opcodes to instructions, hidden in Appendix A (figure A.10.2).
- Also in Appendix A is a summary of register names/usage. Worth noting that with the exception of registers 0 and 31, they're all the same to the hardware; designating some of them for use as temporaries, another as a stack pointer, etc., is purely a matter of convention, but so useful . . .
- Also in Appendix A is a complete list of instructions and pseudoinstructions. I prefer that you not use the pseudoinstructions, with a few exceptions that are hard to avoid, such as `la`.
- MIPS assembly language also provides for defining "macros"; more in section A.2. (Some other assembly languages use this a lot.)

Slide 4

Memory Layout

- Again the hardware imposes no particular distinctions on how memory is used, but useful to adopt conventions. The one described in the text is typical. From smallest to largest addresses:
  - A reserved block (usually for O/S use).
  - A block for the program's text segment (code).
  - A block for the program's data segment, divided into static data (globals, etc.) and dynamic data ("the heap"). UNIX systems further subdivide this into a segment for fixed data with values assigned at compile time and a segment with space for other static data (not initialized) and dynamic data.
  - Possibly unused space.
  - A block for the stack segment.
- Notice that the data segment grows toward larger addresses, the stack segment toward smaller addresses.

Slide 5

From Source to Execution — Linking

- As mentioned, object and executable files contain machine language and other information.
- Details vary, but if you're curious, a Web search on "ELF file format" should find information on a format used in many UNIX-like systems.

  Commands `readelf`, `nm`, and `ldd` are interesting to try with object and executable files.

Slide 6

Textbook's Example of Linking

- I think I misled you last time about "relocation information" — it should be information about any instructions that might need to change during linking/loading.
- Some details of this example are quite unclear to me, such as how they got the reference to `$gp` from a `lw` or `sw`. Apparently SPIM at least will let you use a label as the operand of a load/store, but it's apparently treated as a pseudoinstruction and uses `$at` to hold the address.

Slide 7

From Source to Execution — Loading

- Nice summary in Appendix A of what happens in loading. Operating system must:
  - Read executable file to determine sizes of text and data segments.
  - "Create address space" big enough for text, data, and stack segments. (Details vary by O/S.)
  - Initialize text and data segments from executable file.
  - Set up registers — stack pointer, global pointer, etc.
  - Push any arguments to program onto stack.
  - Jump to start-up code that copies arguments to registers and calls program's `main()`. On return, makes a system call to terminate program.
- Note in passing that code invoked by "system calls" is not part of the program; the `syscall` instruction jumps to code in the O/S's part of memory.

Slide 8

Stack Usage Revisited

- We talked about how each call to a procedure pushes things (the return address if nothing else) onto the stack.
- Might be useful to watch this happen in `xspim`, using the factorial example?

Slide 9

Sidebar(?): Parallel Execution and Synchronization

- A lot of commodity hardware these days features multiple processing units ("cores") sharing access to memory. One reason for this is that in theory we can make individual applications faster by splitting computation up among processing elements.

  (Shameless self(?)-promotion: We plan to try again to offer the parallel-programming elective in the fall.)

- Having processing elements share memory makes parallel programming easier in some ways but has risks ("race conditions"). Avoiding the risks requires some way to control access to shared variables (e.g., to implement notion of "lock").

Slide 10

Parallel Execution and Synchronization, Continued

- Most texts on operating systems discuss synchronization issues and present several solutions ("synchronization mechanisms"), some rather high-level and others not.

  (Why is this in O/S textbooks? because O/Ss typically have to manage "processes" executing concurrently, either truly at the same time or interleaved.)

- The most primitive can (with some simplifying assumptions) be implemented with no hardware support. But hardware support is very useful.

Slide 11

Sidebar: Why is Implementing a Lock Hard?

- It might seem like it would be straightforward to implement a lock — just have an integer variable, with value 0 meaning "unlocked" and anything else meaning "locked". And then you "lock" by looping until the value is 0, then setting to nonzero, and "unlock" by setting back to 0.
- But this doesn't work! (Why not?)

Slide 12

Instructions for Synchronization

- Key goal in designing hardware support for synchronization is to provide "atomic" (indivisible) load-and-store. This allows writing a low-level implementation of "lock" idea.
- Many architectures do this with a single instruction (e.g., "test and set" or "compare and swap"). Requires two accesses to memory so may be difficult to implement efficiently.
- MIPS approach — same idea, but using a pair of instructions, `ll` ("load linked") and `sc` ("store conditional"). Example of use in textbook (p. 122). `sc` "succeeds" only if value at target location has not changed since previous `ll` — i.e., if one can regard the pair of instructions as forming a single atomic load/store.

Slide 13

Preview — Data and Arithmetic

- Next chapter discusses how data is represented and arithmetic is done.
- Some material should be review — how integers and floating-point values are represented in binary, integer addition and subtraction.
- Other material is new — some details of how multiplication and division can be done, what floating-point arithmetic involves.

Slide 14

Minute Essay

- If you think about formats for object and executable files, would you think they'd be the same for all operating systems running on the same architecture? if so, why, and if not, what parts would be the same? what parts might be different? (You may not feel like you can fully answer this, so — speculate?)
- This wraps up what I plan to say about Chapter 2. Any questions before we move on?

Slide 15

Minute Essay Answer

- A few things would likely be the same, or almost the same — the sizes of the text and data segments, the actual machine instructions, and the data for the data segment. But some things in the machine-code parts may be dependent on what the linker does to resolve unresolved references, which might vary depending on the O/S.
- But other things might not be, if for no other reason than that it's not clear (to me anyway) that there would be incentive to standardize across operating systems. And anything related to how the O/S manages memory or dynamically-linked library code would likely need to be different.