Slide 1

Administrivia

- Reminder: Homework 4 due Wednesday.
- Reminder: Quiz 3 Wednesday. Likeliest questions involve data representation — base 10 to two's complement and vice versa, limitations of formats (for integers and floating point).
- Reminder: First exam a week from Wednesday. If I don't get homework graded by then I will plan to distribute sample solutions. I'll post a short review sheet by this Wednesday. Be advised, though, that rules are similar to those for quizzes — open book, open notes.

Slide 2

Minute Essay From Last Lecture

- Not a lot of requests to review, but a few, so we'll do that first.

Slide 3

Accessing Array Elements in MIPS Assembly Language — Review

- To load to or store from array element `A[i]`, need to compute its address.
- Conceptually simplest way is to first compute offset from start of array, in bytes — `i` times 4 (use `sll`!), then add to base address.
- This gives address for load or store.
- Note however that if looping over all elements of an array it may be faster to just increment the address directly — similar to accessing array in C using a pointer rather than with an index.

Slide 4

MIPS Assembly Language to Machine Language — Review

- First look up instruction in reference summary ("green card") to get its opcode and format (R, I, or J). Note that for R-format instructions list not only opcode but function field.
- Write down values in binary for all fields. (What fields are needed depends on format.) Somewhat tricky bits are "immediate" value for branches (offset from next instruction to target, divided by 4), jumps (address of target, divided by 4).
- Merge into single 32-bit value and write down in hexadecimal.
- Can check with SPIM (remember to use `-delayed_branches`).

Slide 5

MIPS Machine Language to Assembly Language — Review

- First convert hexadecimal to binary.
- Next get opcode — first 6 bits — and look up. This tells you the instruction and the format (R, I, J).
- Next get remaining fields based on format (R, etc.). Remember to multiply branch-target offsets and jump address by 4.
- Finally, write down in assembly form.

Slide 6

Base-10 to Floating-Point Representation — Review

- Convert base 10 number to binary. (If not a power-of-2 fraction, this might get messy. I won't ask you to do that.) Put in scientific notation.

  (If you're having trouble with the fractional part, you could try multiplying repeatedly by 2 until you get an integer, keeping track of how many multiplications.)
- Convert exponent to binary and add bias factor (defined by format).
- Write down pieces — sign, significand (without leading 1, padding with zeros to the right), biased exponent — in binary, then merge to give hexadecimal.

Slide 7

Floating-Point Representation to Base-10

- Convert hexadecimal to binary and split into sign, significand, and biased exponent.
- Compute real exponent by subtracting bias factor.
- Write as base-2 scientific notation, adding leading 1 to significand.
- Convert to base-10.

Slide 8

One More Thing — Data Alignment

- Like many (but not all) architectures, MIPS load/store instructions require that the address be appropriately "aligned" — words on word boundaries, etc.
- Assembly directives such as `.word` do the right alignment (skipping space if necessary — e.g, a `.word` after a string). `.space`, however, does not. Can use `.align` to force alignment.

  (So using `.space` to declare an array right after a string might lead to problems, because alignment might not be right.)
- How does this work when linking object files? Good question! (I think object file would have to include something about alignment of data segment — either that or the convention would have to be to always use the most restrictive alignment. SPIM apparently doesn't!)

Slide 9

One More Example — Working With Text

- In most of our examples we've worked with integers. Is there a way to work with text? Yes . . .
- `lb` and `sb` load and store bytes (characters, in C anyway).
- (Example program(s).)

Slide 10

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?)

Slide 11

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.