Slide 1

Administrivia

- Reminder: Homework 2 due today. (Don't forget essay, both parts of "honor code statement" (pledge, collaboration).)
- Reminder: Quiz 2 Monday. Topics from chapter 2, up through addressing modes.
- (I *am* working on grading Homework 1. Soon! If I finish before Monday, I can leave papers in my mailbox for you to pick up?)
- Homework 3 to be posted later today. Due a week from today.
  Some written problems and one programming problem (in MIPS assembler)!
  One more homework before Exam 1, to be due the following Wednesday.

Slide 2

Minute Essay From Last Lecture

- Most people thought the quiz was fine. Good!

Slide 3

Addressing Modes — Recap/Review

- Register addressing (value in general-purpose register, as in, e.g., `add`).
- Immediate addressing (value in instruction, as in, e.g., `addi`).
- Base-displacement addressing (value in memory, addressed using register and fixed displacemenet, as in, e.g., `lw`).
- PC-relative addressing (value computed using current value of PC, assembler-generated constant, as in, e.g., `beq`).
- Pseudo-direct addressing (value mostly in instruction, combined with high-order bits of PC, as in, e.g., `j`).

Slide 4

Pseudo-Direct Addressing — Example

- As an example, trying working out machine code for the previous example with `j There` replacing the `bne`:

```
        j       There
        add     $t2, $zero, $zero
        add     $t3, $zero, $zero
        add     $t4, $zero, $zero
There:
        sub     $t5, $zero, $zero
```

Slide 5

Pseudo-Direct Addressing — Example, Continued

- Look up opcode — `0x2`.
- To get 26-bit value for the address, need not a relative location (as for `bne`) but an absolute one.

  To do that, need to know where in memory the (machine) code resides. Suppose we paste this code into the starter example, right after the "opening linkage" code, and use as starting address of whole progrram location where SPIM puts `main:`. That's `0x0040 0024`. Counting up, get an address of `0x0040 003c` for `There`. Remove top four bits of that and divide by 4 to get

  `0000 0100 0000 0000 0000 0011 11`
- Putting the two fields together and converting to hexadecimal gives `0810000f`, which agrees with SPIM.

Slide 6

A Little (More) About Assembly Language and Assemblers

- We've done short examples of translating assembly language into machine language.
- Normally this is done programmatically, by an "assembler". Accepts symbolic representations of instructions. Also allows defining "labels" (strings ending `:`) and uses some directives (starting with ".", e.g., `.word`) to help keep track of instructions, define character strings, etc.
- Details for MIPS assembler in Appendix A.

Slide 7

Assembly Language — Program Elements

- Instructions: Self-explanatory? Each represents one machine-language instruction — usually anyway. Some are are "pseudoinstructions", translated into one or more "real" instructions (ones that have machine-language equivalents). Example is `la`, translated into combination of `lui` and `ori`.
- Labels: Identifier (following usual rules for such) followed by **:**. Useful/necessary in writing code but not (usually) preserved in object code.
- Directives: Start with **.** and tell the assembler something. (Next slide.)

Slide 8

Assembly Language — Directives

- `.text` indicates that what follows is instructions.
- `.data` indicates that what follows is data.
- `.word`, `.asciiz`, `.space` reserve space for data (and also, for the first two, initialize it).
- `.globl` identifies a label that might be referenced by outside code. (Think "separate compilation" and how one might combine object files. More about this soon.)

Slide 9

System Calls

- System calls are how user programs request service from operating system — not just in MIPS, but in general. In MIPS the instruction is `syscall`; other architectures have something analogous.
- System calls similar to procedure calls in some ways: Need to communicate to O/S which service you want (e.g., write some text to "standard output") and possibly parameters (e.g., text to write). As with procedure calls, do this by putting values in particular registers, but then rather than `jal` we use `syscall`.

  So why not just *use* `jal`?? Well . . .

Slide 10

System Calls, Continued

- Important distinction (discussed more in O/S courses, such as our CSCI 3323): Code for "system call" executes as part of the O/S, which means not subject to same restrictions as user programs (e.g., on memory access).
- Details (e.g., what services are offered) depend on O/S. Very primitive O/S included in `spim` supports some for simple I/O; details in Appendix A.

Slide 11

System Calls in MIPS — Details

- How to specify which service, arguments?

  Put number indicating which service in `$v0`. (Appendix A has a list of services.)

  If parameters needed, put them in `$a0` and `$a1`.
- Return value in `$v0`.

Slide 12

Writing Complete Programs for the Simulator

- Simulator includes what's in essence a very primitive operating system, with facilities to load programs and do simple I/O. As in real operating systems, I/O done by making "system calls".
- Complete programs can be run from command line with, e.g., `spim -file hello.s`.

Slide 13

Complete Programs — Examples

- Can now write some simple but complete programs for the simulator(!).
- (Examples on “sample programs” page.)

Slide 14

Assembly Language, Etc. — More Examples

- Textbook presents extended example (sort). Skim as an example of using MIPS instructions.
- Longer examples coming soon (next class?).

Slide 15

Minute Essay

- Is this all making (some!) sense? Questions? What if anything seems murkiest to you at this point?