Slide 1

Administrivia

- Quiz 1 sample solution online.
- Reminder: Homework 2 due Wednesday.
- Next quiz next Monday.

Slide 2

Procedure Calls — Review/Recap

- Calling procedure must:
  - Put parameters in `$a0` through `$a3` (if more than four, on stack).
  - Use `jal` to jump to called procedure (which saves the return address in register `$ra`).
  - Get return value from `$v0` (and `$v1`, if used).
- Called procedure must:
  - Save registers as needed, including return address.
  - Retrieve parameters and do calculation.
  - Put results in `$v0` (and `$v1`, if used).
  - Restore saved registers.
  - Return to caller with `jr $ra`.

Slide 3

Addressing Modes

- We've been unspecific about how to specify addresses of a lot of things.
- So, now look at various "addressing modes" — ways to specify where to find an operand.
- Which is used? Defined by instruction format (R, I, J). (J? yes, format for jump instructions that include a label — `jal` and `j`.)

Slide 4

Addressing Modes, Continued

- Register addressing: Value is in one of the general-purpose registers. Assembler defines symbolic names for them (e.g., `$t0`).
- Immediate addressing: Value is in instruction itself (as in, e.g., `addi`).
- Base-displacement addressing: Value is in memory, with address calculated by adding a displacement to what's in a register. Example is memory-address operand of `lw`, `sw`.
- PC-relative addressing (more shortly).
- Pseudo-direct addressing (more shortly).

Slide 5

PC-Relative Addressing

- Address is formed by adding offset in instruction (16 bits) and contents of the program counter (special register).

  (Actually, address is offset times 4, plus the *updated* program counter. The simulator doesn't quite simulate this, unless run with the flag `-delayed_branches`.

- Example is conditional branches (`beq`, `bne`).
- Does this limit what we can do with `beq` and `bne`? If so, how often will it matter? What could we do to work around it?

Slide 6

Pseudo-Direct Addressing

- Address is formed by combining address in instruction (26 bits) and upper bits of program counter.

  (Actually, address is address in instruction times 4, or'd with upper bits of program counter.)

- Example is unconditional branch (`j`).
- Does this limit what we can do with `j`? If so, will that be a problem? Can we work around it?

Slide 7

Example

- As an example, try translating the following C first into MIPS assembly language and into machine language. (Assume registers `$s0` and `$s1` are being used for `a` and `b`.)

```
if (a < b)
        a = a + 1;
else
        b = b - 1;
```

Slide 8

A Little (More) About Assembly Language and Assemblers

- We've done a few short examples of translating assembly language into machine language.
- Normally this is done programmatically, by an "assembler". Accepts symbolic representations of instructions. Also 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 9

Writing Complete Programs for the Simulator

- The 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 is done by making "system calls".
- Complete programs can be run from the command line with, e.g., `spim -file hello.s`.

Slide 10

System Calls

- System calls are how user programs request service from the 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., the text to write). As with procedure calls, we do this by putting values in particular registers, but then rather than `jal` we use `syscall`.

Slide 11

System Calls, Continued

- An 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 the O/S. The very primitive O/S included in `spim` supports some for simple I/O; details in Appendix A.

Slide 12

Complete Programs — Examples

- We can now write some simple but complete programs for the simulator(!).
- (Examples on "sample programs" page.)

Slide 13

Minute Essay

- What does the following code do? i.e., what is in registers `$s0` and `$s1` after it executes?

```
        add     $s0, $zero, $zero
        addi    $s1, $zero, 1
        addi    $s2, $zero, 4
label:
        addi    $s0, $s0, 1
        add     $s1, $s1, $s1
        bne     $s0, $s2, label
```

Slide 14

Minute Essay Answer

- We could trace through the code, which sets values in three registers and then executes a loop:

  `$s0` is initially set to 0 and then takes on values 1, 2, 3, and 4

  `$s1` is initially set to 1 and then takes on values 2, 4, 8, and 16

  `$s2` is initially set to 4 and doesn't change