Slide 1

Administrivia

- Reminder: Homework 2 due Monday.
- Quiz 2 next Wednesday.

Slide 2

Minute Essay From Last Lecture

- Some thought the math in the homework was interesting or even kind of fun, others that it was repetitious. At least a few wrote code to help and found that useful or even fun.
- More than one person mentioned dimensional analysis.
- Several people did say it helped them understand concepts better.
- A few agreed with me that “this number is not the whole story on performance” was an interesting/good point.

Slide 3

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 4

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 5

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 6

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 7

PC-Relative Addressing, Continued

- 16-bit offset obviously does limit how far we can "jump". But it's probably fine for most uses (conditional execution, loops).
- If it's not, we could rework the code so we can either use `j` or `jr`.

Slide 8

PC-Relative Addressing — Example

- As an example, try working out machine code for the `bne` in this line (comments with relative locations included so we can easily compute the offset we need):

```
# location 0
        bne     $t0, $t1, There
# location 4
        add     $t2, $zero, $zero
# location 8
        add     $t3, $zero, $zero
# location 12
        add     $t4, $zero, $zero
# location 16
There:
        sub     $t5, $zero, $zero
```

Slide 9

PC-Relative Addressing — Example, Continued

- Look up opcode — `0x5`.
- Look up register numbers — 8, 9.
- Compute needed offset by subtracting relative location of the instruction *after* the `bne` from the relative location of the "branch target" (`There`),
- Rearranging bits and converting to hexadecimal, we get `0x15090003`.

  Does this agree with what SPIM shows? Not quite . . . For some reason, SPIM by default computes offsets from the current instruction rather than the next. No idea why, but we can force it to compute the "right" offsets with flag `-delayed_branches`.

Slide 10

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 11

Pseudo-Direct Addressing, Continued

- 26-bit address does limit what we can do, but it's probably fine for most uses (conditional execution and loops, procedure calls).
- If it's not enough, we can rework the code so we can use `jr`.

Slide 12

Pseudo-Direct Addressing — Example

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

```
# location 0
        j       There
# location 4
        add     $t2, $zero, $zero
# location 8
        add     $t3, $zero, $zero
# location 12
        add     $t4, $zero, $zero
# location 16
There:
        sub     $t5, $zero, $zero
```

Slide 13

Pseudo-Direct Addressing — Example, Continued

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

  We'll pick addresses that will let us check results with SPIM. It seems to put the first instruction of `main` at `0x0040 0024`, so if we make that the location of the `j`, we get an address of `0x0040 0034` for `There`. Removing the top four bits of that and dividing by 4, we get

  `0000 0100 0000 0000 0000 0011 01`
- Putting the two fields together and converting to hexadecimal gives `0810000d`, which agrees with SPIM.

Slide 14

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 allows defining "labels" (string 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. More next time.

Slide 15

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 16

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 17

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 18

Complete Programs — Examples

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