Slide 1

Administrivia

- As mentioned in e-mail, there will be a second lecture for this week, as a video lecture, available Friday I hope. There will also be one for Monday's class since we have another faculty candidate interviewing.
- Homework 3 will be on the Web 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

Administrivia

- Quiz 1 graded and sample solution posted (bottom of "lecture topics" etc. page).
- Quiz 2 next Wednesday. Topics from chapter 2, up through addressing modes.
- (I *am* working on grading Homework 1. Soon?)

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

Variables

- Space for local variables typically allocated on the stack. Since `$sp` can change during computation, can use register `$fp` ("frame pointer") to point to start of area ("procedure frame") for saved registers, local variables.
- What about other variables? Two basic types: fixed/static (think global variables) and dynamically allocated (think C `malloc()`. (e.g., with `malloc` in C).

  By convention, we put them right after the program code and use register `$gp` ("global pointer") to point to them. Typically call the memory used for dynamically-allocated variables "the heap".

Slide 5

More Load/Store Instructions

- MIPS architecture defines `lw` and `sw` for loading/storing data in 32-bit chunks; also defines `lb` (“load byte”) and `sb` (“store byte”) for loading/storing data in 8-bit chunks, plus instructions to load/store data in 16-bit chunks.

  All must align on appropriate boundaries.

Slide 6

Working with Constants, Revisited

- Recall `addi` instruction. Exists because often we need to use a small constant in a program.
- Uses same format (“I format”) as `lw` and `sw`, which allows 16 bits for constant.
- What if we need more than 16 bits? “Load upper immediate” instruction:

  ```
  lui register, constant
  ```

  Puts (16-bit) constant in “upper” 16 bits of register. Follow with `addi` (or, better, `ori`) to load a full 32-bit constant.
- An example is the two instructions the assembler generates for a `la` pseudoinstruction (example in simulator).

Slide 7

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 8

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 9

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 10

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 11

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):

```
        bne     $t0, $t1, There
        add     $t2, $zero, $zero
        add     $t3, $zero, $zero
        add     $t4, $zero, $zero
There:
        sub     $t5, $zero, $zero
```

Slide 12

PC-Relative Addressing — Example, Continued

- Look up opcode — `0x5`.
- Look up register numbers — 8, 9.
- Compute needed offset by . . . Strictly speaking, should be offset from relative location of instruction *after* the `bne` to "branch target" (`There`), divided by 4. But just counting instructions gives the same effect (and here's it 3).
- Rearranging bits and converting to hexadecimal, we get `0x15090003`. Does this agree with what SPIM shows? Not quite . . .

Slide 13

PC-Relative Addressing — Example, Continued

- 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 14

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 15

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 16

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 17

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.

  To do that we 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 the starting address of the whole progrram the location where SPIM puts `main:`. That's `0x0040 0024`. Counting up, we get an address of `0x0040 003c` for `There`. Removing the top four bits of that and dividing by 4, we get

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

Slide 18

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 19

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, do this by putting values in particular registers, but then rather than `jal` we use `syscall`.

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

Slide 20

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 21

Complete Programs — Examples

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

Slide 22

Minute Essay

- Any questions? Is this all starting to make sense to you?