Slide 1

Administrivia

- Homework 2 posted. Due next Monday.
- Quiz 2 next Wednesday. Topics from chapter 2, up through addressing modes.
- Quiz 1 scores good! Sample solution on course Web site.

Slide 2

Procedure Calls — Recap/Review

- Calling procedures (a.k.a. functions or methods) more complicated than it maybe looks from a HLL. Several requirements (review next slide).
- Every language that compiles (or assembles) to machine language *could* do it differently, but useful to define standard way, so languages can interoperate. (Also allows operating system to load program and start it up without knowing source-code language.)

  Most of this is software; main role of hardware is to provide instruction to jump while "remembering" where we came from.

Slide 3

Procedure Calls — Requirements

- Put parameters where procedure can find them.
- Transfer control to procedure.
- Acquire storage resources for procedure (for local variables, etc.).
- Run procedure.
- Put results where caller can find them.
- Return control to caller.

Slide 4

Register Saving and Local Variables

- Actually running called procedure straightforward, except:

  Called procedure may want to use registers in some way not compatible with caller. (If nothing else, consider what happens with `$ra` if the called procedure in turn calls another.)

  MIPS convention: `$sN` registers retain value across procedure call; others (especially `$tN` registers) might not.
- To make this work, need some way to save/restore registers.
- Also need a way to make space for local variables.

Slide 5

Register Saving and Local Variables, Continued

- Typical solution: Use part of memory as a stack (familiar ADT, right?), for saving registers and other local storage. Makes recursive procedures easier.
- By convention, stack starts at high address and “grows” to lower addresses. and register `$sp` (“stack pointer”) points to top. “Push” and “pop” are then straightforward. (Note: `$sp` just a symbolic name for one of the 32 general-purpose registers.)

  (Recall discussion of “buffer overflows” from CSCI 1120?)
- (Review starter code. Everything in it should now make some sense?)

Slide 6

Example

- How to compile the following?

```
int main(void) {
int a, b, c, x;
        a = 5; b = 6; c = 7;
        x = addproc(a, b, c);
        return 0;
}
int addproc(int a, int b, int c) {
        return a + b + c;
}
```

(Sample program `call-addproc.s`.)

Slide 7

Variables

- Space for local variables typically allocated on the stack. Since `$sp` can change during computation, can use register `$fp` ("frame pointer" — another of the 32 general-purpose registers) 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).

  MIPS convention: Put them right after the program code, use register `$gp` ("global pointer", also one of general-purpose ones) to point to them.

  Typically call the memory used for dynamically-allocated variables "the heap".

Slide 8

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 9

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.
- Example: two instructions assembler generates for `la` pseudoinstruction (example in simulator).

Slide 10

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 11

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 12

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. 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 13

PC-Relative Addressing, Continued

- 16-bit offset obviously limits how far we can “jump”. But probably fine for most uses (conditional execution, loops).
- If not, rework code to use `j` or `jr`.

Slide 14

PC-Relative Addressing — Example

- As an example, try working out machine code for the `bne` in this line. (May be helpful to annotate with relative locations so we easily compute 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 15

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*. (Why divided by 4? always a multiple of 4! so last two digits always 0 ...) 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 16

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 can force it to compute the “right” offsets with flag `-delayed_branches`.

Slide 17

Pseudo-Direct Addressing

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

  As with PC-relative addressing, no real need to store last 2 digits, since always 0.

  Actual address is address field in instruction, times 4, OR'd with upper bits of program counter to give 32 bits in all.
- Example of use 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 18

Pseudo-Direct Addressing, Continued

- 26-bit address does limit what we can do, but probably fine for most uses (conditional execution and loops, procedure calls).
- If not enough, can rework code to use `jr`.
- (To be continued.)

Slide 19

Minute Essay

- How did the quiz compare to your expectations?
- Any questions? Is this all starting to make sense to you?