Slide 1

Administrivia

- Reminder: Homework 1 due today. Hardcopy please. Now or in my mailbox by 5pm.
- Reminder: Quiz 1 Wednesday. Topics from chapter 1.
- Next homework on the Web; due in a week.
- For minute essays, you can use them as a way to ask me pretty much anything — questions about the course, random-curiosity questions about something related to CS — and I'll do my best to answer.

Slide 2

Minute Essay From Last Lecture

- (Review — but most people got it right.)

Slide 3

Conditional Execution — Recap/Review

- MIPS instruction set includes only two instructions to support conditional execution: `beq` and `bne`.
- There's also an unconditional "go to", `j` (for "jump").
- Together these are enough for some kinds of if/then/else and loops.
- If hand-compiling from C, useful to first translate into code with only `goto` for out-of-sequence execution, and from there to MIPS.
- Example:

```
while (A[i] == k) {
          i = i + j;
}
```

Slide 4

More Flow of Control

- With what we have now we can do if/then/else and loops, but only if condition being tested is equals / not equals.
- So, we need instructions such as `blt`, `ble`, right?
- But those are apparently difficult to implement well; instead MIPS has "set on less than":

```
slt   r1, r2, r3
```

  which compares the contents of registers `r2` and `r3` and sets `r1` — 1 if `r2` is smaller, else 0.
- Example — compile the following C:

```
          if (a < b) go to Less:
```

  assuming we're using `$s0`, `$s1` for `a`, `b`.

Slide 5

More Flow of Control, Continued

- Do we have enough now? for all six possible C comparisons of integers? Yes . . .
- One more C flow-of-control construct we could talk about — `switch` — but defer for now.
- But we do want to talk about one more HLL feature, namely functions . . .

Slide 6

Procedure Calls

- How do we call procedures (a.k.a. functions, methods)? Consider an example:

```
          a = a + a;
          x = foo(a);
          b = b + b;
          y = foo(b);
          /* .... */
int foo(int n) { return n+1; }
```

- If we've compiled this code (and function `foo`), what do we have in memory when it's running? What's supposed to happen when we get to a call to `foo`?

Slide 7

Procedure Calls, Continued

- So, what we have to do to call a procedure is:
  1. Put parameters where procedure can find them.
  2. Transfer control to procedure.
  3. Acquire storage resources for procedure (recall that every time you call a C function you get a “new copy” of all its local variables).
  4. Run procedure.
  5. Put results where caller can find them.
  6. Return control to caller.
- How to do all this?

Slide 8

Sidebar: Register Conventions

- From hardware point of view, all general-purpose registers are in some sense the same, with the sort-of exception of registers 0 (always has value 0) and 31 (discussed soon).
- From software point of view, it’s useful to agree about how to use them — for parameters, return values, etc. Idea is that compilers automatically enforce conventions, human-written assembly code should follow them too.
- So far — `$s0` through `$s7` used for variables, `$t0` through `$t9` used as “scratch pads”. (See reference card for numeric equivalents.)
- Add two more groups — `$a0` through `$a3` for parameters (punt for now on what to do if more than four), `$v0` and `$v1` for return values. (Why two? to make it easy to return a 64-bit value such as used for floating-point.)

Slide 9

Jumping To/From Procedures

- When we jump to a procedure, must remember where we came from so we can return. Do this with “jump and link”

  ```
  jal    label
  ```

  which puts address of next instruction in register `$ra` (31) and jumps to `label`. (How do we know address of next instruction? “Program counter” (special register) has address of current instruction.)

- We can then get back with “jump to register”

  ```
  jr   r1
  ```

  which jumps to address in register `r1`.

Slide 10

Register Saving and Local Variables

- Actually running the called procedure is straightforward, right?
- Yes, except we need some way to save/restore registers — so we don’t mess up caller (by convention, “temporary” registers might change, but most others don’t).
- We also need a way to make space for local variables.

Slide 11

Register Saving and Local Variables, Continued

- Common 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 `$sp` (“stack pointer”) points to top. “Push” and “pop” are then straightforward.
- (Now everything in the starter-code program should make sense?)
- (Semi-aside: Since `$sp` can change during computation, can use register `$fp` (“frame pointer”) to point to start of area (“procedure frame”) for saved registers, local variables.)

Slide 12

Other Variables

- Last but not least, we (may?) need someplace to store variables that can be preallocated (static/global) and variables that are dynamically allocated (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 13

Procedure Calls, Revisited

- Calling procedure must:
  - Put parameters in `$a0` through `$a3` (if more than four, on stack).
  - Determine address of called procedure and jump there, saving address of next instruction.
  - 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`.
  - Restore saved registers.
  - Return to caller.

Slide 14

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`.)

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.

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).

Minute Essay

- What if anything was noteworthy (interesting, difficult, etc.) about Homework 1?