Slide 1

Administrivia

- Reminder: Homework 2 due Monday.
- Monday we have a candidate for our open faculty job speaking at 2:30pm. I should be there, so no class. I'm planning to do a video lecture and make it available via `echo360.org`.

Slide 2

Minute Essay From Last Lecture

- (Review. Most people figured out that I format seemed right, but many expressed some confusion. A lot of this *is* new and strange, but I think with exposure it will start to make sense. Or ask if not!)
- Maybe this is where I can say:

  I started out in FORTRAN but it wasn't until a second-semester course in IBM mainframe assembly language that I really started to understand how programming worked. "Hm!"?

Slide 3

Instruction Formats — Review/Clarification

- Basic problem being solved is this: How to represent different kinds of instructions in binary? We've already seen that some instructions have the same kinds of operands (`add` and `sub`, e.g.), but not all the same (`add` and `lw`, e.g.).
- MIPS solution: Make all machine-language instructions same size (32 bits), and always use the first 6 bits for "opcode" (something identifying instruction), then define different ways of splitting up the remaining bits — different "instruction formats", each with "fields".

Slide 4

Sidebar: Converting between Binary and Hexadecimal

- Recall(?) simple trick for converting between binary (base 2) and hexadecimal (base 16): Based on observation that each hexadecimal digit represents four binary digits.
- (Why this works — simple algebra based on writing out numbers as a sequence of multiples of powers of the base.)

Slide 5

Instructions — Recap/Clarification

- `add`, `sub` somewhat obvious.
- `and`, `or` — bitwise operations. (Examples.)
- `sll`, `srl`, `sra` — bit-shift operations. (Examples.)

Slide 6

Sidebar: `go to` in C

- Textbook freely uses C's `go to`, which possibly some of you have never encountered? because it's strongly discouraged, and kind of ugly.
- What it does: Immediately transfer control to some other point in the program, identified by a label (e.g., `here:`).
- Conditional execution and loops can all be expressed using `go to` (and in some early high-level languages they were(!)).

  Makes some sense, since this is pretty much all the hardware can do.
- (Sometimes written `goto`. Same thing.)

Slide 7

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 8

Example Continued

- MIPS equivalent, with C-with-`goto` as comments (and assuming `$s0` has the address of `A` and registers `$s1` through `$s3` have `i`, `j`, and `k`):

```
Loop:
# if (A[i] != k) goto End:
        sll     $t0, $s1, 2     # i * 4
        add     $t0, $s0, $t1   # &A[i]
        lw      $t0, 0($t1)     # A[i]
        bne     $t0, $s3, End
#   i = i + j
        add     $s1, $s1, $s2
#   goto Loop:
        j       Loop
End:
```

Slide 9

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 10

Example Continued

- Equivalent MIPS:

  ```
  slt     $t0, $s0, $s1
  bne     $t0, $zero, Less
  ```

Slide 11

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 12

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 13

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 14

Sidebar: Register Conventions Revisited

- 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.

Slide 15

Register Conventions, Continued

- So far:

  `$s0` through `$s7` for variables.

  `$t0` through `$t9` as "scratch pads".

- 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 16

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 17

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 18

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.

  (Recall discussion of "buffer overflows" from CSCI 1120?)
- (Now everything in the starter-code program should make sense?)

Slide 19

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 20

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 21

Minute Essay

- None — quiz.