Slide 1

Administrivia

- Homework 2 to be posted later today. I'll send e-mail. Due in a week.

Slide 2

Minute Essay From Last Lecture

- (Review.) Key point here is that which "format" to use depends on the syntax(?) of the instruction (how many and what kinds of operands) rather than on semantics(?) (is it arithmetic or — whatever).

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.)
- (Review if you don't remember how to do this.)

Slide 5

Flow of Control

- So far we know how to do (some) arithmetic, move data into and out of memory. What about if/then/else, loops? (See sidebar on p. 90 for early commentary on conditional execution.)
- Need instructions that allow us to “make a decision”. Perhaps surprisingly, only two: `beq` (“branch if equal”), `bne` (“branch if not equal”).
- Illustrate with an example . . .

Slide 6

Sidebar: `go to`

- Some very early HLLs implemented conditional execution using `goto`. 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`. Makes some sense, since this is pretty much all the hardware can do.
- Very quickly became apparent that this made for code that was hard to reason about. So later languages have been “block structured”.

Slide 7

Sidebar: `go to` in C, Continued

- `go to` still exists in C because every once in a while it makes for more-readable code (e.g., some error handling).
- Useful in this course as an intermediate step between block-structured ("normal"?) C and assembly language, which has no notion of block structuring.
- (Sometimes written `goto`. Same thing.)

Slide 8

Flow of Control Example

- Suppose we have this in C (and as usual all variables are 32-bit integers)

```
          if (i == j) goto L1:
          f = g + h;
L1:       f = f - i;
```

- What instructions should compiler produce? Assume we're using `$s0` through `$s4` for `f`, `g`, `h`, `i`, `j`.
- (For now, punt on how to represent `L1`.)

Slide 9

Flow of Control Example, Continued

- Compiling

```
            if (i == j) goto L1:
            f = g + h;
L1:         f = f - i;
```

using `$s0` through `$s4` for `f`, `g`, `h`, `i`, `j`.

gives

```
          beq     $s3, $s4, L1
          add     $s0, $s1, $s2
L1:       sub     $s0, $s0, $s3
```

Slide 10

Another Flow of Control Example

- Of course, we don't usually have `goto` in C. More likely is this:

```
if (i == j)
          f = g + h
else
          f = g - h
```

- What to do with this? Rewrite using `goto` ...

Slide 11

Another Flow of Control Example

- Rewriting

```
if (i == j)
        f = g + h
else
        f = g - h
```

gives

```
        if (i != j) goto Else:
        f = g + h
        goto End:
Else:   f = g - h
End:    ....
```

and then we can continue as before. (How to do unconditional "go to"? `j` (for "jump").)

Slide 12

Loops

- Do we have enough to do (some kinds of) loops? Yes — example:

```
Loop:   g = g + A[i];
        i = i + j;
        if (i != h) goto Loop:
```

assuming we're using `$s1` through `$s4` for `g`, `h`, `i`, `j`, and `$s5` for the address of `A`.

(This time we'll use `sll` rather than two `add`s to multiply `i` by 4.)

Slide 13

Loops — Example Continued

- Result

```
Loop:   sll     $t1, $s3, 2      # $t1 <- 4*i
        add     $t1, $t1, $s5    # $t1 <- & of A[i]
        lw      $t0, 0($t1)      # $t0 <- A[i]
        add     $s1, $s1, $t0    # g = h + A[i]
        add     $s3, $s3, $s4    # i = i + j
        bne     $s3, $s2, Loop   # if (i!=h) goto Loop
```

Slide 14

Conditional Execution, Continued

- 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 15

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 16

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 17

Example Continued

- Equivalent MIPS:

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

Slide 18

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 19

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 20

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 21

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 22

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 23

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 24

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 25

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.
- (To be continued.)

Slide 26

Minute Essay

- None — quiz.