Slide 1

Administrivia

- Homework 1 is on the Web (mail sent late Thursday).
- Course “useful links” page updated.
- What to put as subject line of minute essay? really just “minute essay” is enough this semester (T/R I have only this course, and your name and the date are easily findable).

Slide 2

Assembly Language and MIPS Instructions — Recap

- Last time we talked a little in general about what assembly language looks like.
- We also looked at some simple instructions in the MIPS ISA (arithmetic and load/store).

Slide 3

A Little About the Simulator

- Your code goes in a file with extension `.s`. (Sample starter code on "Sample programs" page. Contains many things we haven't talked about yet but could still be useful for trying things out.)
- Start the simulator with command `xspim` (`spim` for command-line version). (Short demo.)

Slide 4

Representing Instructions in Binary

- "It's all ones and zeros" applies not only to data but also to programs — "stored program" idea. (Some very early computers didn't work that way — programming was by rewiring(!).)
- So we need a way to represent instructions in binary . . .

Slide 5

Representing Instructions in Binary, Continued

- First consider what we have to represent:
  - For all instructions, which instruction it is.
  - For `add` and `sub`, three operands (all register numbers).
  - For `lw` and `sw`, three operands (two register numbers and a "displacement").
  - And so forth . . .
- So, each instruction will have "fields" — consistent format for storing pieces of data, a little like a C `struct`.

Slide 6

Representing Instructions in Binary, Continued

- So, can we use the same format for all instructions? Some data ("which instruction") is common to all, but operands may need to be different.
- Can we / should we make all instructions the same length? For MIPS, yes (other architectures differ), and then define different ways of dividing up the length — "formats".

  Basic principle: "Good design involves good compromises."

Slide 7

R Format

- Meant for instructions such as `add`.
- Fields:
  - `op` — op code, 6 bits
  - `rs` — first source operand, 5 bits
  - `rt` — second source operand, 5 bits
  - `rd` — destination operand, 5 bits
  - `shamt` — "shift amount" (not used for `add`), 5 bits
  - `funct` — "function field", 6 bits
- Example — find binary representation of

```
add     $t0, $s1, $s2
```

Slide 8

I Format

- Meant for instructions such as `lw`.
- Fields:
  - `op` — op code, 6 bits
  - `rs` — first source operand, 5 bits
  - `rt` — destination operand, 5 bits
  - `disp` — displacement, 16 bits
- Example — find binary representation of

```
lw      $t0, 1200($t1)
```

- How can we tell which format is being used? determined by value for `op`.

Slide 9

Logical Operations

- Sometimes useful to be able to work with individual bits — e.g., to implement a compact array of boolean values.
- Thus, MIPS instruction set provides "logical operations". Hard to say whether these exist to support C bit-manipulation operations, or C bit-manipulation operations exist because most ISAs provide such instructions!

Slide 10

"Shift" Instructions

- C << and >> (on unsigned numbers) are translated into `sll` ("shift left logical") and `srl` ("shift right logical").
- `sll` and `srl` do what the names imply — bits "fall off" one side, and we add zeros at the other side. These are R-format instructions, and they use that "shift amount" field.
- When shifting left, filling with zeros makes sense. But when shifting right, we might want to extend the sign bit instead. `sra` ("shift right arithmetic") does that.
- Examples?

Bitwise And and Or

- C `&` is translated into `and` or `andi`. C `|` is translated into `or` or `ori`. Format/operands are analogous to `add` and `addi`.

  (Notice/recall that C has two sets of and/or operators — logical and bitwise. These are the bitwise ones.)

Slide 11

- We could use these to test/set particular bits. Examples? Could we use them to, e.g., compute remainder when dividing by power of 2?

Other Logical Operations

- "Exclusive or" implements — what the name suggests (see textbook).
- "Nor" likewise. Can be used to implement "not" (see textbook).

Slide 12

Slide 13

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.)
- We need instructions that allow us to "make a decision" — `beq` ("branch if equal"), `bne` ("branch if not equal").
- Illustrate with an example . . .

Slide 14

Flow of Control Example

- Suppose we have this in C

```
                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 for `f`, `g`, `h`, `i`, `j`.
- (For now, punt on how to represent `L1`.)

Slide 15

Another Flow of Control Example

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

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

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

Slide 16

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.

- Or how about something that looks more like normal C?

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

Slide 17

More Flow of Control

- 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 difficult to implement well, so 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.
- Also define that register 0 (`$zero`) always contains 0.
- Example — compile the following C:

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

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

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 (next time . . .).

Slide 19

Minute Essay

- None — sign in. (Unless you have questions!)