Administrivia

Slide 1

- Reminder: Homework 1 due today (but accepted without penalty through tomorrow). Hardcopy please. Usually I say 5pm for written work but really anytime before 11:59pm is okay if you put it in the mailbox outside my office.
- For minute essays with “right” answers there will be a sample solution in the final version of the online notes.
- Sample solutions for quizzes will be linked from the “lecture topics and assignments” page after (both sections of) class.

Minute Essay From Last Lecture

Slide 2

- Many people came up with something pretty much right, but by no means all.
- (Review answers?)

Slide 3

MIPS Instructions — Recap/Review

- MIPS instructions include some for arithmetic (which operate on registers and small constants) and some for transfer between memory and registers.
- Registers include some special-purpose ones (e.g., program counter) and 32 general-purpose ones. Each holds a 32-bit value. Can reference the latter by number (0 through 31) or using symbolic names (shown in "MIPS reference" in textbook).

Slide 4

SPIM Simulator

- Simulator (command `spim` or `xspim`) emulates a real MIPS processor and can be used to assemble (on the fly) and execute assembly-language programs.
- At startup it contains in memory what amounts to a very primitive operating system, including code to do some simple setup and call a `main` procedure and code for some "system calls" for very simple console I/O.
- `main` procedures include some boilerplate "linkage" at start and end, as in `starter.s` on sample programs page on course Web site. No I/O yet but you can watch values in registers change.
- (Continue demo from last time.)

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

Slide 7

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 8

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 9

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 10

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 11

“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?

Slide 12

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.)
- 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 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

Slide 15

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 16

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 17

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 18

More Flow of Control (Preview)

- We can do if/then/else and loops, but only if condition being tested is equals / not equals.
- So, we need instructions that will allow less-than comparisons.
- (We also need something that allows an unconditional branch, but we may punt on that for a while too.)

Slide 19

Minute Essay

- None —- quiz.
- Quiz is “open book, open notes”, which means you can look at:
  - Textbook (paper or electronic).
  - Course Web site (my “notes”. sample programs).
  - Your notes (paper or electronic).

  but nothing else.