Slide 1

Administrivia

- Reminder: Homework 1 due Monday. Hard copy please, in class or in one of my mailboxes (outside my office or in the ASO).
- Quiz 1 Wednesday. Topics from chapter 1.

  Quizzes will be about 10 minutes, at the end of class. Open book / notes (meaning you can consult the textbook, anything on the course Web site, or your notes, and you can use whatever tools you need to do that, but no others.) Problems will likely be similar to homeworks and/or minute essays.
- Code shown in simulator last time on “sample programs” page.
- If you wonder about the fact that the e-mail address TMail shows for me isn’t the same as the one I give in my course materials — it’s a long story, but you can use either one.

Slide 2

Minute Essay From Last Lecture

- (Most people who tried it came pretty close.)
- (By the way: All minute-essay answers get the same credit, so don’t worry if you don’t have the right answer, at least from a grade standpoint.)

Slide 3

MIPS Instructions — Recap/Review

- We looked at a few instructions — `add` (and `sub`), `addi`, `lw`, `sw`. Syntax highly constrained, unlike high-level languages.
- Many operands are register numbers. Maybe think of (general-purpose) registers as a fixed-size array of 32-bit values, and register number is index into this array. Assembler also allows using symbolic names (`$t0`, e.g.). (List of values in MIPS reference — green card in front of paper version of textbook, link to online version on "Useful links".) Notice that register 0 (`$zero`) is special — value always zero.

Slide 4

Registers and Variables

- Examples in textbook and in class talk about registers being associated with variables.
- The idea is more or less this: In MIPS, can only do arithmetic on values in registers. So if compiling from a high-level language, to do arithmetic on variables, have to first load values into registers, then do arithmetic, then store the results back.
- Repeated loads/stores can be inefficient, though, so "good" compilers typically try to associate a register with each variable and do loads/stores only when necessary. (If more variables than registers? then use registers for most-frequently-used variables, do more loads/stores.)

Arithmetic Instructions — Review

Slide 5

- `add` and `sub` take three operands, all register numbers.
- `addi` also takes three operands, two register numbers and a constant ("immediate value"). Curiously enough(?), no `subi`. (Why not? What could you use instead?)

Load/Store Instructions — Review

Slide 6

- Load and store instructions take two operands, one a register to load into / store from, and one specifying address in terms of register containing base address and displacement (constant).
- Fixed displacement isn't maybe ideal for all situations (e.g., array element), but simple, and displacement useful for addressing element of, say, a C `struct`.
- (How then to address array element? compute address by computing displacement and adding to base address. Example on next slide.)

Slide 7

Example — Array Element Access

- Suppose register `$s1` contains the address of an array `A` of 32-bit integers, and register `$s2` contains the value of a variable `i`. We could use the following to load the value of `A[i]` into register `$t0` (keeping in mind that addresses are in bytes, and each array element occupies 4 bytes):

```
add     $t0, $s2, $s2 # $t0 <- 2*i
add     $t0, $t0, $t0 # $t0 <- 4*i
add     $t1, $t0, $s1 # $t1 <- &A[i]
lw      $t0, 0($t1)
```

- (Isn't there a multiply instruction we could use instead of this cumbersome double addition?? yes, but it's likely to be quite slow.)

Slide 8

MIPS Assembly Language Program Structure

- Review (updated) example from last time.
- Overall structure mixes instructions and "directives" (things that start with `.`). Programs typically have two sections, one for code (starting with `.text` directive) and one for data (starting with `.data`.
- For now, ignore "opening linkage" and "closing linkage". Most of the rest should seem at least sort of plausible?

Slide 9

Simulator, Revisited

- `xspim` starts graphical version; most-often used buttons are probably "load" and "step".
- `spim` starts command-line version; commands include `load`, `p` to print, `s` to step.
- Most of the code being executed should look pretty much like your code — except
  - Before your code there's a tiny bit of SPIM's rudimentary O/S, which jumps to (your) `main`.
  - Some assembly "instructions" (e.g., `la`) are actually "pseudo-instructions" that assemble to more than one machine instruction.

Slide 10

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 11

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 12

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 13

I Format

- Meant for instructions such as `lw`, `sw`.
- Fields:
  - `op` — opcode, 6 bits
  - `rs` — source operand, 5 bits
  - `rt` — destination operand, 5 bits
  - `disp` — displacement, 16 bits

Slide 14

I Format — Example

- Find binary representation of

```
lw      $t0, 12($t1)
```

- Fields:
  - `op` — look up `lw` in MIPS reference (green card in textbook or online), result `0x23`
  - `rs` — look up `$t1`, result 9
  - `rt` — 8
  - `disp` — convert 12 to 16-bit value (`0x000c`).
- Convert all of the above to binary and concatenate. Use the simulator to check.

Slide 15

R Format

- Meant for instructions such as `add`, `sub`.
- Fields:
  - `op` — opcode, 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 all instructions)
  - `funct` — "function field", 6 bits (not used for all instructions
- Somewhat unusual in that opcode doesn't completely determine which instruction it is; instead, what's unique is the combination of opcode and function field.

Slide 16

R Format — Example

- Find binary representation of

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

- Fields:
  - `op` — 0
  - `rs` — 17 (from reference)
  - `rt` — 18
  - `rd` — 8
  - `shamt` — 0 (not used)
  - `funct` — `0x20` (from reference)
- Convert all of the above to binary and concatenate. Use the simulator to check.

Interpreting Machine-Language Instructions

- So that's how to get machine language from assembly language. How to go the other way?
- At first might seem tricky — which format is being used? but all have 6-bit opcode first, and it determines format for the rest.

Slide 17

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 18

Bitwise And and Or

Slide 19

- 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.

Other Logical Operations

Slide 20

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

Slide 21

"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.
- These instructions are very useful for multiplying and dividing by small powers of 2, important since multiplication and division are likely to be slow (more later in the course).

Slide 22

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". Perhaps surprisingly, MIPS provides only two: `beq` ("branch if equal"), `bne` ("branch if not equal").
- Illustrate with an example . . .

Flow of Control Example

Slide 23

- 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`.)

Flow of Control Example, Continued

Slide 24

- 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 25

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 26

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 (punt for now on how to do unconditional `goto`).

Loops

Slide 27

- 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.)

Loops — Example Continued

Slide 28

- Result

```
Loop:   sll     $t1, $s3, 2      # $t1 <- 4*i
        add     $t1, $t1, $s5    # $t1 <- address 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!=j) goto Loop
```

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 29

Minute Essay

- Does one of the two instruction formats (I and R) seem like it would work for `addi`? If so, which one, and can you say anything about what the values of the various fields might be? If not, what fields would you need in a new format?
- Anything particularly unclear?

Slide 30

Minute Essay Answer

- I format works — the operands of `addi` are two register numbers and a 16-bit constant value, same as `lw` and `sw`. Like those two instructions, it has "source" and "destination" registers, which can go in those two fields, and a 16-bit immediate valuel that can go in the field used for displacement in the load/store instructions.