Slide 1

Administrivia

- XSPIM demo in last lecture went wrong because I wasn't sharing all windows. Oops! Try again today.

Slide 2

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 3

Sidebar: Hexadecimal

- Binary is well and good but long strings of it difficult for humans.
- More compact representation is hexadecimal (base 16). Why that one? Any base that's a power of 2 is trivially easy to convert from and to binary: Each digit in base 2^n represents n bits. (Not hard to convince yourself that this works — to binary, just write down definition for other-base number and expand, while from binary, can write each group of n bits and observe that each can be written as 2^m times an n-bit number in the range 0 through $2^n - 1$.
- Base 16 attractive because 4 divides typical word sizes. Base 8 (octal) once popular, and easier for most humans, and made sense when 36-bit words were more common, but not so much now.
- I'll use C convention of prefixing hexadecimal values with `0x`.
- (Short example.)

Slide 4

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 5

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".

  (Another way to say it, maybe; In MIPS all machine-language instructions are 32 bits. Of those, 6 are always something identifying which instruction; the remaining bits are split up differently for different kinds of instructions.)

Slide 6

MIPS Assembly Language Program Structure

- Look at `starter.s` under "sample programs" on course Web site.
- 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" (which will make sense after we talk about procedure calls soon).

Slide 7

SPIM 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. Also can run program directly (not in debug mode); useful for testing programs you think are complete and working.
- Note that there's no machine code for `la`. Why? . . .

Slide 8

Sidebar: Pseudoinstructions

- Assemblers can also allow so-called pseudoinstructions: No matching machine instruction, but easily expanded at assembly time into one or a few "real" instructions.
- I say avoid when you can, but some (such as `la`) are just too useful.
- (Why do I say avoid? because our goal here is not to be ace assembly-language programmers able to write compact programs, but to understand the instructions that the hardware must provide directly.)

Slide 9

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 10

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)
  - `rs` — look up `$t1`
  - `rt` — 8
  - `disp` — convert 12 to 16-bit value
- Convert all of the above to binary and concatenate. Use simulator to check.

Slide 11

Sidebar: Sign Extension

- In computing the actual address for `lw`, the hardware must add a 32-bit value and a 16-bit value. Does that mean we will need to build something that adds two 32-bit values and also a 32-bit value and a 16-bit value?
- Not if we have a way to extend the 16-bit value to a 32-bit value — "sign extension".
- Simple enough for non-negative values (pad on the left with 0s). For negative values, works to pad to the left with 1s). (Textbook explains in section 2.4 why that works.)

Slide 12

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 13

R Format — Example

- Find binary representation of

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

- Fields:
  - `op` — 0
  - `rs` — look up in reference
  - `rt` — look up
  - `rd` — look up
  - `shamt` — 0 (not used)
  - `funct` — look up
- Convert all of the above to binary and concatenate. Use the simulator to check.

Slide 14

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.
- (Example.)

Slide 15

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 16

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 value that can go in the field used for displacement in the load/store instructions.