Slide 1

Administrivia

- Quiz 1 is scheduled for Wednesday. Quizzes are open book, open notes.
- "Useful links" Web page updated with info about SPIM.
- Tuesday office hours now 12:30pm to 3:30pm.

Slide 2

High-Level Languages Versus Assembly Language

- In a high-level language you work with "variables" — conceptually, names for memory locations. You can do arithmetic on them, copy them, etc.
- In machine/assembly language, what you can do may be more restricted — e.g., in MIPS architecture, you must load data into a register before doing arithmetic).
- The compiler's job is to translate from the somewhat abstract HLL view to machine language. To do this, normally associate variables with registers — load data from memory into registers, calculate, store it back. A "good" compiler tries to minimize loads/stores.

Slide 3

Minute Essay From Last Lecture

- Question: Write MIPS assembly code for the following C program fragment:

```
a = b + c + d + e
```

  Assume we have `b`, `c`, `d`, `e` in `$s1` through `$s4` and want to have `a` in `$s0`

  (Optional: Can you think of more than one way to do it? If you can, does one seem better than the other, and why?)

- Possible answers?

Slide 4

Load/Store Example

- Suppose we have this in C

```
a[12] = h + a[8];
```

- What instructions should compiler produce? Assume we're using `$s3` for starting ("base") address of `a`, `$s2` for `h`.

Slide 5

Another Load/Store Example

- Suppose we have this in C

```
g = h + a[j];
```

- What instructions should compiler produce? Assume we're using `$s3` for starting ("base") address of `a`, `$s1`, `$s2`, `$s4` for `g`, `h`, `j`.

Slide 6

Another Load/Store Example, Continued

- The method we used for finding the address of `a[j]` seems clumsy. Shouldn't we use multiply? (Maybe not!)
- Wouldn't it be convenient to specify the address with two registers? (Yes, and some architectures allow this, but MIPS doesn't.)

Slide 7

Representing Instructions in Binary

- 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 8

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".

  "Design Principle 3: Good design involves good compromises."

Slide 9

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
```

  Look up `op` in table on inside back cover, registers in table on p. A-23.

Slide 10

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)
```

  Look up `op` in table on inside back cover, registers in table on p. A-23.
- How can we tell which format is being used? determined by value for `op`.

Slide 11

Minute Essay

- Write MIPS assembler code to exchange the values of `a[0]` and `a[1]`. Assume register `$s0` contains the address of `a` (start of the array), and `a` is an array of integers.