Slide 1

Administrivia

- Reminder: Homework 1 due Monday. For written assignments (such as this one), hardcopy please, in class or in my mailbox by 6pm.

Slide 2

Minute Essay From Last Lecture

- Many people got the intended answer but not all, and some were pretty far off. I replied to all who were far off; check slides for intended answer.

Slide 3

“Architecture” as Interface Definition — Review

- From software perspective, “architecture” defines lowest-level building blocks — what operations are possible, what kinds of operands, binary data formats, etc.
- From hardware perspective, “architecture” is a specification — designers must build something that behaves the way the specification says.

Slide 4

A Bit About Assembly Language Syntax

- Syntax for high-level languages can be complex. Allows for good expressivity, but translation into processor instructions is complicated.
- Syntax for assembly language, in contrast, is very simple. Less expressivity but much easier to translate into (binary form of) instructions.

Slide 5

Arithmetic Instructions — Addition

- Instruction for integer addition (in assembly-language form):

```
add     a, b, c
```

  Adds `b` and `c` giving `a`.

  (Notice the format — symbolic name, operands.)

- Is this expressive enough?
- Should we have more instructions (with different numbers of operands, e.g.)?

  Basic principle: "Simplicity favors regularity."

  Easier to build simple hardware if ISA is "regular" — e.g., all arithmetic instructions have exactly three operands.

- `sub` (subtraction) is similar. Multiplication and division are more complicated, so punt for now.
- What are the operands? Registers. What are those? Well . . .

Slide 6

Registers

- Access to main memory is slow compared to processor speed, so it's useful to have a within-the-chip memory — "registers".
- MIPS architecture defines 32 "general-purpose" registers, each 32 bits.
- Would more be better?

  Basic principle: "Smaller is faster."

- In machine language, reference by number.
- In assembly language, useful to adopt conventions for which registers to use for what, use symbolic names indicating usage.

  E.g., use registers 8 through 15 for "temporary" values (short-term), refer to as `$t0` through `$t7`.

Slide 7

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 8

Example

- Suppose we have this in C

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

- What instructions should compiler produce? Assume we're using `$s0` for `f`, `$s1` for `g`, `$s2` for `h`, `$s3` for `i`, `$s4` for `j`.

  (Symbolic register names starting `$s` are used for for slightly longer-term storage than the ones starting `$t`.)

  (Where do values come from? Next topic . . .)

Slide 9

Memory, Revisited

- Usually we think of memory as big 1D array of 8-bit “bytes”, each with address (index into array) and contents (value of array element).
- Often we operate on elements in groups of 4 — 32-bit “word”.
- MIPS is a “load/store” architecture, meaning access to memory is limited to copying data between memory and registers. Alternatives include arithmetic instructions to operate on memory directly.

  (How would that be better? worse?)

Slide 10

Memory-Access Instructions — Load

- Goal is to get one 32-bit word from memory and put in a register.
- How to specify location in memory? Seems most useful to have address in a register. For a little more flexibility, specify address in terms of “base” and “displacement”.

  ```
  lw      r, d(b)
  ```

  Address specified by contents of register `b` plus (constant) `d`. Loads word into register `r`.
- `sw` (“store word”) instruction is similar.

Slide 11

Example

- Suppose we have this in C

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

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

Slide 12

Addition Using Constant

- "Add immediate"

  ```
  addi r1, r2, c
  ```

  adds constant `c` (16-bit signed integer, can be negative) to contents of `r2`, puts result in `r1`.

- Exists because often we need to use a small constant in a program.

  Basic principle: "Make the common case fast."

Slide 13

Representing (Integer) Data in Binary

- Remember that to the hardware "it's all ones and zero" — any data you're working with.
- As an example — representation of signed integers using two's complement notation. Should have been covered in CSCI 1320, but read/skim 2.4 if you don't remember.

Slide 14

A Little About the Simulator

- As mentioned, installed on our machines is a simulator you can use to try your programs. It simulates a MIPS processor running a *very* primitive operating system (just enough to load programs and do some simple console I/O). It assembles programs on the fly.
- 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 it with command `xspim` (`spim` for command-line version).

  (Short demo.)

Slide 15

Minute Essay

- 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?

  OR

- 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.

Slide 16

Minute Essay Answer

- One way:

  ```
  add    $s0, $s1, $s2
  add    $s0, $s0, $s3
  add    $s0, $s0, $s4
  ```

  Another way (not as good since uses more registers?):

  ```
  add    $t0, $s1, $s2
  add    $t1, $s3, $s4
  add    $s0, $t0, $t1
  ```

- One way:

  ```
  lw    $t0, 0($s0)
  lw    $t1, 4($s0)
  sw    $t0, 4($s0)
  sw    $t1, 0($s0)
  ```