Slide 1

Administrivia

- Reading quiz 2 posted; due 03/03. More reading quizzes soon — more than one for Chapter 2 since it's long.

Slide 2

E-Mail and Me

- Some of you are curious about why for e-mail *to* me I ask you to use my TMail address, but e-mail *from* me comes from a `@cs` address, and you ask which to use, or worry that you get it wrong, or send things to both addresses?

  Partly for historical reasons I prefer to deal with mail using the `@cs` address. But my TMail address forwards there, so whichever one you use should reach me.

  Sorry about the confusion.

Slide 3

What's Next — Overview

- Defining a representative architecture (MIPS): what "architecture" means in context, assembly language programming, machine language. (This is the "first half" of the course.)
- Designing a simplified implementation of this architecture. (This is the "second half".)

Slide 4

"Architecture" as Interface Definition

- "Architecture" here means "instruction set architecture" (ISA), a key abstraction.
- 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 5

Architecture — Key Abstractions

- Memory: Long long list of binary "numbers", encoding all data (including programs!), each with "address" and "contents".

  When running a program, program itself is in memory; so is its data.

  (Very powerful concept! Major innovation during early days of digital computers.)
- Instructions: Primitive operations processor can perform.
- Fetch/execute cycle: What the processor does to execute a program; repeatedly get next instruction (from memory, location defined by "program counter"), increment program counter, execute instruction.
- Registers: Fast-access work space for processor, typically divided into "special-purpose" (e.g., program counter), "general-purpose" (integer and floating-point). Unlike memory, these are part of the processor.

Slide 6

Design Goals for Instruction Set

- From software perspective — expressivity.
- From hardware perspective — good performance, low cost.
- (Yes, these can sometimes be opposing forces!)

Slide 7

Why Study MIPS Architecture?

- Goal is not to become good assembly-language programmers, but to understand how things work at this level. Once you understand basic principles, learning another assembly language is easier.
- MIPS architecture is simple but representative. Not currently used much in desktop/laptop world but (supposedly) similar architectures popular in embedded-systems world.

Slide 8

SPIM Simulator

- It's of course useful as you learn assembly language to be able to try programs. Various simulators that let you do that. The one I like is SPIM. Old and a bit clunky but has some features I really like, so it's the one I'll use in this course.
- Simulates assembly and execution of assembly program; incorporates a very primitive operating system that makes it possible to do text input/output.
- Installed on department's Linux machine, so easy to use from Linux virtual desktop. If that doesn't work for you, can install on your own machine: Now hosted on `Sourceforge.org`. Web-search on SPIM and `Sourceforge.org` for link or follow the one under "Links" on course Web site (soon).
- Commands `spim` and `xspim` (graphical). Sample programs under "Sample programs". More about these, and demo, soon.

Slide 9

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 10

Arithmetic Instructions — Addition

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

  ```
      add     r1, r2, r3
  ```

  Adds `r2` and `r3` giving `r1`.

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

Slide 11

Registers

- Access to main memory slow compared to processor speed, so useful to have a within-the-chip work space — "registers".
- MIPS architecture defines 32 "general-purpose" registers, each 32 bits. Essentially interchangeable except for $0 (always zero) and $31 (used by hardware to support procedure calls).
- 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, define symbolic names indicating usage.

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

Slide 12

High-Level Languages Versus Assembly Language

- In a high-level language you work with "variables" — conceptually, names for memory locations. 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, must load data into a register before doing arithmetic.
- 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 13

Example

- Textbook gives detailed example of arithmetic on registers on p.73 (section 2.3).
- (Where do values come from? Next topic . . .)

Slide 14

Memory, Revisited

- Usually think of memory as big 1D array of 8-bit "bytes", each with address (index into array) and contents (value of array element).
- Often operate on elements in larger units. For MIPS, natural unit is 32-bit "word". (Other architectures also often operate on words. 32 bits was common until recently; 64 bits more so now.)
- MIPS is a "load/store" architecture — access to memory limited to copying data between memory and registers. Alternatives include arithmetic instructions to operate on memory directly.

Slide 15

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 similar.

Slide 16

Example

- Textbook gives detailed example of loading with fixed displacement on p.75 (section 2.3).
- Fine for accessing elements of `struct`. What about array elements? Compute address by computing displacement and adding to base address. Example on next slide.

Slide 17

Array Element Access — Example

- 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)
```

Slide 18

Array Element Access, Continued

- Isn't there a multiply instruction we could use instead of double addition?? yes, but it's likely to be quite slow. Bit-shifting is better — to be discussed soon.
- And Yes, for a programmer it would be great if it were possible to load from an address given via a base address in one register and an index in another, but it's not Not sure why; maybe too much for single instruction.

Slide 19

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 20

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.
- Note that how bytes are stored in memory (least-significant first or last) not same in all ISAs: "Big-endian" (MIPS) versus "little-endian". Names come from *Gulliver's Travels*.

Slide 21

MIPS Assembly Language Program Structure

- (Time permitting.)
- 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”. Most of the rest should seem at least sort of plausible? (More soon.)

Slide 22

Minute Essay

- Anything today that was particularly unclear?