Slide 1

Administrivia

- Homework 1 to be on the Web later today. I will send mail.

Slide 2

Minute Essay From Last Lecture

- What kinds of products probably use embedded processors?
- Some answers that seem likely: car, microwave, washer, dryer, radio, calculator(?), digital watch, TV, router, Google glass(?), insulin pump, thermostat, GPS system, printer,
- Some answers I'm skeptical about: iPad, tablet, PDA.
- Possibly not very clear where to draw the line . . .

Slide 3

Defining Performance — Recap/Review

- What does it mean to say that computer A "has better performance than" computer B?
- Really — "it depends". Some answers:
  - Computer A has better response time / smaller execution time.
  - Computer A has higher throughput.
- We'll use execution time, and say

$$\frac{\text{Performance}_A}{\text{Performance}_B} = n$$

exactly when

$$\frac{\text{Execution time}_B}{\text{Execution time}_A} = n$$

Slide 4

Calculating (Approximating?) Execution Time

- CPU execution time for program X is given by

$$\text{CPU cycles} \times \text{clock cycle}$$

- How would you write that using clock rate instead of clock cycle?
- How would you write it if you know number of instructions and (average) number of cycles per instruction?
- What if you can define different classes of instructions, each with a different number of cycles per instruction?

Slide 5

Calculating Performance — Example

- Suppose for a given program you have

	Instructions	*Avg cycles/instr*	*Cycle time*
Machine X	1 million	1.5	1 ns
Machine Y	1 million	2	0.5 ns

(1 second = 10^9 ns)

Which machine is faster? by how much? (e.g., “X is twice as fast as Y”.)

Slide 6

Calculating Performance — Example Continued

- time for X = $10^6 \times 1.5 \times 10^{-9} = 1.5 \times 10^{-3}$

  time for Y = $10^6 \times 2 \times 0.5 \times 10^{-9} = 10^{-3}$

  so Y is 1.5 times as fast as X

Slide 7

One More Thing About Performance — Amdahl's Law

- Parallel-computing version: Can define "speedup" gained by using P processors as ratio of execution time using 1 processor to execution time using P processors. (So, in a perfect world it would be P).
- But most "real programs" have some parts that have to be done sequentially. Gene Amdahl (principal architect of early IBM mainframe(s)) argued that this limits speedup — "Amdahl's Law":

  If γ is the "serial fraction", speedup on P processors is (at best — this ignores overhead)

  $$S(P) = \frac{1}{\gamma + \frac{1-\gamma}{P}}$$

  and as P increase, this approaches $\frac{1}{\gamma}$ — upper bound on speedup.
- Textbook points out that this is more broadly applicable!

Slide 8

"Architecture" as Interface Definition

- 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.

Terminology Recap/Review

Slide 9

- Repertoire of primitive operations processor can carry out — “instruction set”.
- Sequence of instructions encoded as binary — “object code” or “machine language”.
- Encoded in symbolic form — “assembly language”.

Architecture — Key Abstractions

Slide 10

- 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.
- 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).

Slide 11

Design Goals for Instruction Set

- From software perspective — expressivity.
- From hardware perspective — good performance, low cost.

Slide 12

Why Study MIPS Architecture?

- Goal is not to become 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.

  Aside: SPIM simulator will let you experiment (commands `spim` and `xspim`).

Slide 13

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 14

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.

Slide 15

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., refer to registers 8 through 15 as `$t0` through `$t7`.

Slide 16

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`.

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 17

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 18

Example

Slide 19

- 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`.

High-Level Languages Versus Assembly Language

Slide 20

- 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 21

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 22

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 23

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 24

Minute Essay

- Was anything today particularly unclear? (What?)
- Do you have an exposure to assembly language (for any processor)?