Slide 1

Administrivia

- Reminder: Homework 1 due Wednesday, 6pm. *If you turn something in later, please write on it when you turned it in.*

Slide 2

Minute Essay From Last Lecture

- (Review.) Most people got the point, but not everyone.

Slide 3

Parallelism — Hardware (Review/Recap)

- Several ways to achieve "more than one thing at a time" in hardware:
- Multiple independent processing elements sharing memory (multicore processors, multiple processors).
- "Hyperthreading" — hardware to enable very fast context switching. Not true concurrency but helps with "hiding latency".
- Computers connected by a network.
- Multiple processing elements operating in lockstep (e.g., GPU). For GPU, also involves separate memory, with need to move data back and forth between it and main RAM.
- (Pipelining and vector processing? much more about pipelining later.)

Slide 4

Parallelism — Software (Review/Recap)

- Multithreading — for multicore processors, multiple processors: Single "process" (from operating-system perspective) with multiple "threads" (software streams) interacting via shared single memory space.
- Message-passing — for computers connected by network: Multiple "processes", not sharing memory, interacting by sending each other messages.
- SIMD ("single instruction, multiple data") — for graphics processing units: Single software stream, executing in-effect-simultaneously on all elements of an array (or other coolection?). May require explicit data copying.

Slide 5

Parallelism — Performance

- One use of multithreading is to make the code simpler, at least for the programmer. (Example: typical GUI-based program, where it makes sense to think in terms of one thread of control for getting user input and one for drawing.) Doable on a single processor via interleaving. May improve performance by "hiding latency".
- But it *can* also be used to improve performance. Performance often discussed in terms of "speedup".
- Here, "speedup" is defined thus:

  For P processing elements (cores, fully independent processors, etc.), speedup $S(P)$ is execution time using 1 PE to execution time using P PEs.

Slide 6

Parallel Performance, Continued

- Might seem like with P processing elements you could get a speedup of P? *But* in fact most if not all programs have at least a few parts that have to be executed sequentially. This limits $S(P)$, and if we can estimate what fraction of the program is sequential we can calculate an upper bound on $S(P)$.
- Further, typically "parallelizing" programs involves adding some sort of overhead for managing and coordinating more than one stream of control.
- But even ignoring those, as long as any part must remain sequential . . .

Slide 7

One More Thing About Performance — Amdahl's Law

- (Named after Gene Amdahl, a key figure in developing some of IBM's early mainframes who left to start his own company to make hardware "plug-compatible" with IBM's. Aside: Interaction between the two companies was — interesting?)
- His observation ("Amdahl's law") can be more generally stated, but in the context of parallel programming it's this:
  If γ is the "serial fraction", speedup on P PEs is (at best, i.e., ignoring overhead)
  $$S(P) = \frac{1}{\gamma + \frac{1-\gamma}{P}}$$
  and as P increase, this approaches $\frac{1}{\gamma}$ — upper bound on speedup.

Slide 8

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.)
- Implementing this architecture. (This is the "second half".)

Slide 9

"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 10

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.
- 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 11

Design Goals for Instruction Set

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

Slide 12

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.

  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     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 15

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.
- 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 16

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 17

Example

- Suppose we have this in C (and assuming all variables are 32-bit integers):

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

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

Slide 18

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 19

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 20

Example

- Suppose we have this in C (and assuming `g` and `h` are 32-bit integers and `a` is an array of same):

```
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 21

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 22

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 23

A Little About the Simulator

- As mentioned, installed on our machines is a simulator you can use to try your programs. Simulates a MIPS processor running a *very* primitive operating system (just enough to load programs and do some simple console I/O). 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 24

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`.

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

Slide 25

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
```