

Slide 1

Administrivia

- Reminder: Homework 1 due today, by 5pm. Turn in at end of class or in my mailbox in department office.
- Homework 2 on Web, due next Friday.

Slide 2

Minute Essay From Last Lecture

- Question:
 - Suppose for a given program you have
- | | <i>Instructions</i> | <i>Avg cycles/instr</i> | <i>Cycle time</i> |
|-----------|---------------------|-------------------------|-------------------|
| 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".)
- Answer?

Slide 3

Recap — Layers of Abstraction

- Software:
 - Applications.
 - High-level language.
 - Machine language.
- Hardware:
 - Five-part design (control, datapath, memory, input, output).
 - ...
 - Silicon, metal, etc.
- Interface between bottom layer of software and top layer of hardware — “architecture”.

Slide 4

“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.

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.

(Intermezzo on binary numbers?)

Slide 5

- 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).

Terminology Recap/Review

- 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”.

Slide 6

Slide 7

Design Goals for Instruction Set

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

Slide 8

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`). Can also try things on “real hardware” — `puck.cs.trinity.edu` (SGI machine).

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.)

Slide 9

- Is this expressive enough?
- Should we have more instructions (with different numbers of operands, e.g.)?
“Design Principle 1: 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.

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?
“Design Principle 2: 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`.
Complete list in table A-23 in textbook.

Slide 10

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`.

Slide 11

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 12

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”.

Slide 13

```
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.

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 14

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?

Slide 15