Slide 1

Administrivia

- Homework 3 (covering chapter 3) coming soon, probably Friday.
- If you're curious about full details of MIPS assembly language — read appendix A.

Slide 2

Minute Essay From Last Lecture

- Question: What does the following code do? i.e., what is in registers `$s0` and `$s1` after it executes?

```
        add     $s0, $zero, $zero
        addi    $s1, $zero, 1
        addi    $s2, $zero, 4
l1:
        addi    $s0, $s0, 1
        add     $s1, $s1, $s1
        bne     $s0, $s2, l1
```

- Answer?

Slide 3

Stack Usage — Recap/Review

- MIPS convention is to define a "stack" in memory to hold parameters, saved registers, local variables. `$sp` register points to top of stack, which "grows" toward lower addresses.

  (How would this look during a recursive procedure?)
- An aside — how this relates to "buffer overruns".
- This works well and is pretty common, but other approaches are possible.

Slide 4

Data Formats, Revisited

- Recall, inside the computer "it's all ones and zeros" — so we must encode anything we want to represent:
  - Integers — binary numbers, often 32 bits for MIPS, but could be other sizes too. How to represent negative integers? Later.
  - Text — ASCII (8 bits per character) or Unicode (16 bits).
  - Real numbers — floating-point format, again later.
  - Many, many more complex formats (`.doc`, `MP3`, `GIF`, etc.).
- MIPS architecture defines `lw` and `sw` for loading/storing data in 32-bit chunks; also defines `lb` ("load byte") and `sb` ("store byte") for loading/storing data in 8-bit chunks, plus instructions to load/store data in 16-bit chunks.

  All must align on appropriate boundaries.

Slide 5

Working with Constants, Revisited

- Recall `addi` instruction. Exists because often we need to use a small constant in a program.

  "Design Principle 4: Make the common case fast."
- Uses same format ("I format") as `lw` and `sw`, which allows 16 bits for constant.
- What if we need more than 16 bits? "Load upper immediate" instruction:

  `lui register, constant`

  Puts (16-bit) constant in "upper" 16 bits of register. Follow with `addi` (or, better, `ori`) to load a full 32-bit constant.

  Example?

Slide 6

Addressing Modes

- We've been unspecific about how to specify addresses of a lot of things.
- So, now look at various "addressing modes" — ways to specify where to find an operand:
  - Register addressing: Value is in one of the general-purpose registers. Examples?
  - Immediate addressing: Value is in instruction itself. Examples?
  - Base-displacement addressing: Value is in memory, with address calculated by adding a displacement to what's in a register. Examples?
  - PC-relative addressing.
  - Pseudo-direct addressing.
- Which is used? Defined by instruction format (R, I, J).

Slide 7

PC-Relative Addressing

- Address is formed by adding offset in instruction (16 bits) and contents of the program counter (special register).

  (Actually, address is offset times 4, plus the updated program counter.)
- Examples? conditional branches (`beq`, `bne`).
- Does this limit what we can do with `beq` and `bne`? If so, how often will it matter? What could we do to work around it?

Slide 8

Pseudo-Direct Addressing

- Address is formed by combining address in instruction (26 bits) and upper bits of program counter.

  (Actually, address is address in instruction times 4, or'd with upper bits of program counter.)
- Example? unconditional branch (`j`).
- Does this limit what we can do with `j`? If so, will that be a problem? Can we work around it?

Addressing Modes and Machine Language

- Nice summary of addressing modes in textbook figure 3.17.
- Now let's look at an example — machine language for this C:

```
while (a[i] == k) {
        i = i + j;
}
```

  Assume we're using `$s3` through `$s6` for `i`, `j`, `k`, address of `a`, and that code is in memory at (decimal) location 80000.

- What does the machine code look like?

Minute Essay

- Write MIPS assembler code for the following procedure, saving/restoring the return address at least:

```
int foo(int a, int b) {
        return a + b;
}
```