Slide 1

Administrivia

- Reminder: Homework 3 posted. A few written problems, and one programming problem. *Note* that I want you to submit your solution to the programming problem by e-mail, as you did in Low-Level.

  (Guiding principle: If it's something I'll want to test for myself — such as code — e-mail, otherwise hardcopy.)

Slide 2

Minute Essay From Last Lecture

- Several people asked interesting questions, some of which made me think some review/clarification would be a help. So . . .
- (If I were doing this in person I would figure on there being some questions. So maybe be prepared to pause the video and make a note of questions you want to ask?)

Slide 3

Machine Language – Review/Recap

- Basic problem to be solved here is how to encode instructions in binary. Could possibly define a unique or mostly-unique way for each different instruction, but to simplify process design it makes more sense to define a small number of standard formats.
- Translating assembly language into machine language is fairly straightforward:

  Write down values for all fields in instruction (specifics vary by format).

  How to get from that to 32-bit binary number or 8-digit hexadecimal number? concatenate fields, convert.

Slide 4

R Format (Review)

- Meant for arithmetic instructions (e.g., `add`) and also for shifts (e.g., `sll`).
- Fields:
  - `op` — op code, 6 bits (zero for arithmetic/logical operations, and `funct` below specifies which one)
  - `rs` — first source operand, 5 bits
  - `rt` — second source operand, 5 bits
  - `rd` — destination operand, 5 bits
  - `shamt` — "shift amount" (only used for shift instructions), 5 bits
  - `funct` — "function field", 6 bits (only used for arithmetic/logical operations)

Slide 5

I Format (Review)

- Meant for instructions that involve a 16-bit constant (e.g., `addi`, `lw`, `beq`).
- Fields:
  - `op` — op code, 6 bits
  - `rs` — first source operand, 5 bits
  - `rt` — destination operand, 5 bits
  - `imm`, `offset` — constant/offset, 16 bits
- For `beq` and `bne`, `imm` is offset from next instruction to target, divided by 4.

Slide 6

J Format

- Meant for instructions that involve an "absolute" address (e.g., `j`, `jal`).
- Fields:
  - `op` — op code, 6 bits
  - `target` — address/4, 26 bits
- Note that `target` depends on where in memory program resides. More about this soon.

Slide 7

Decoding Machine Language

- How to go the other way — machine instruction to assembly language?
- If what you have is hexadecimal, first write down binary equivalent.
- Look first at opcode (first six bits). Look that up to find out which instruction and which format. (If you haven't already found this — there *is* a table mapping opcodes to instructions, hidden in Appendix A (figure A.10.2).)
- Then break other 26 bits into fields based on instruction format, and translate as appropriate.
- (Keep in mind that this must be possible to do without too much intelligence, since processors have to do something similar!)

Slide 8

Register Usage — Recap/Review

- To the hardware, those 32 general-purpose registers are all the same, except that:

  0 always has value 0

  31 is used by `jal`
- However, it's useful to adopt conventions for what they're used for.
  Appendix A has a summary of register names/usage, as does the reference summuary ("green card" in paper copy).

Slide 9

Memory Layout — Review/Clarification

- Again the hardware imposes no particular distinctions on how memory is used, but useful to adopt conventions. The one described in the text is typical. From smallest to largest addresses:
  - A reserved block (usually for O/S use).
  - A block for the program's text segment (code).
  - A block for the program's data segment, divided into static data (globals, etc.) and dynamic data ("the heap"). UNIX systems further subdivide this into a segment for fixed data with values assigned at compile time and a segment with space for other static data (not initialized) and dynamic data.
  - Possibly unused space.
  - A block for the stack segment.
- Note that the data segment grows toward larger addresses, the stack segment toward smaller addresses.

Slide 10

Variables — Review/Clarification

- Declaring a variable in a high-level language (e.g., `int x;` in C) reserves space for it in memory (in principle anyway — more shortly) and assigns it a name (for the purposes of compilation).

  Space can be in "data" segment of memory, for static/global variables, or "on stack" for local variables.
- Referencing the variable implies accessing the associated memory location. (Figuring out the instructions to do that is part of the compiler's job. Presumably it has some sort of map from names to locations.)

  In MIPS, that means a load (for read) or store (for write). A very simple compiler would do this for every access. But . . .

Slide 11

Variables, Continued

- Memory access is slow compared to processor speed, so good compilers will streamline things by sometimes keeping values of frequently-used variables in registers, only loading or storing when necessary to preserve semantics.

  This is why the textbook examples talk about associating registers with variables. (Clearer?)

- I said "in principle" because a good compiler might even figure out that it might be possible to just use a register to hold a variable's value and never assign it a memory location. Simple contrived example:

```
int foobar(int x) {
    int y = x+1;
    return y;
}
```

  No need to have `y` in memory at all, right?

Slide 12

System Calls — Review/Revisited

- Idea of system calls: Typically there are things application programs want to do (e.g., get more memory) that in general-purpose system should only be done by a central authority (the operating system). Mechanism for doing that in a safe/secure way — "system calls". System call is a request for the O/S to do something.
- Conceptually much like procedure call, but with an important difference, having to do with what the called code is allowed to do.
- How it works in assembly language varies by architecture.
- What services are provided varies by operating system.

Slide 13

System Calls in MIPS — Review

- Instruction `syscall` (no operands) makes a system call.
- How does O/S know which service is requested . . .
- In SPIM anyway, `$v0` has number indicating which service. (Appendix A has a list.)

  Some also need parameters, which go in `$a0` and `$a1`.

  Return value in `$v0`.

Slide 14

Assembly Language — Program Elements

- Instructions: Self-explanatory? Each represents one machine-language instruction — usually anyway. Some are are "pseudoinstructions", translated into one or more "real" instructions (ones that have machine-language equivalents). Example is `la`, translated into combination of `lui` and `ori`.
- Labels: Identifier (following usual rules for such) followed by `:`.
  Useful/necessary in writing code but not (usually) preserved in object code.
- Directives: Start with `.` and tell the assembler something. (Next slide.)

Slide 15

Assembly Language — Directives

- `.text` indicates that what follows is instructions.
- `.data` indicates that what follows is data.
- `.word`, `.asciiz`, `.space` reserve space for data (and also, for the first two, initialize it).
- `.globl` identifies a label that might be referenced by outside code. (Think "separate compilation" and how one might combine object files. More about this soon.)

Slide 16

Assembly Language, Etc. — More Examples

- Textbook presents extended example (sort). Skim as an example of using MIPS instructions.
- As another example both of writing procedures in MIPS and writing complete programs for the simulator, let's write a program to compute factorial using recursion.

Slide 17

Minute Essay

- Was the review helpful? Questions?