Slide 1

Administrivia

- For the adventurous: We have a "real" MIPS machine, `Puck`. The "Useful links" page has a link to some notes on logging in and using it.
- Notice change in reading for today — do read (or at least skim) all sections of chapter 3.
- Homework 3 on Web, due next Friday. Notice that I also want you to turn in source code by e-mail. *Please read the instructions.*

Slide 2

Minute Essay From Last Lecture

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

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

- Answer?

Slide 3

"Reverse Engineering" Machine Language

- Once upon a time, when your program crashed, you got a "core dump" showing contents of memory, including your program's machine language, plus contents of registers (including program counter). How did people cope?
- Even now, the real "spec" for hardware designers is machine language. How to figure out what the ones and zeros mean?
- Let's do an example — figure out assembly-language equivalent of `0x00af8020`.

Slide 4

From HLL Source to Program in Memory

- Compilers turn HLL source into (conceptually) assembly language or (more frequently) object code.
- Assemblers turn assembly-language source into object code.
- Linkers turn object code (plus libraries) into executables.
- Loaders load executables into memory and start them up.

Slide 5

Compilers

- Compiler's job is to turn HLL source into something lower-level. Can produce assembly-language source, but usually goes straight to object code.
- Some aspects are relatively straightforward — e.g., generating assembly/object code that's semantically equivalent.
- Some aspects are not so straightforward — "parsing" source code, and "optimizing" generated code. May be covered in course on programming languages. Or read the "dragon book" (*Compilers: Principles, Techniques and Tools*, by Aho, Sethi, and Ullman).

Slide 6

Assemblers

- Assembler's job is to turn "assembly language source code" into object code. Such code includes:
  - Instructions (from ISA) in symbolic form.
  - "Pseudoinstructions" that are somewhat higher-level but still very easy to convert to real instructions. MIPS examples include `move`, `li`, `la`.
  - Declarations for data (constants, static variables, etc.).
  - Other directives.
- Most aspects of this are straightforward. Usually set up "symbol table" to translatea symbolic address (labels) to addresses. Notice, though:
  - Some addresses might be impossible to compute at this point — e.g., calls to library routines.
  - Some addresses are relative to PC and so don't depend on where in memory the program resides. Others are "absolute".

Slide 7

Linkers

- Linker's job (sometimes looks like part of compiler) is to turn "object code" generated by compiler or assembler into "executable".
- Format of "object code" file can depend on operating system. E.g., on Unix systems typically includes header, info for debugger, and:
  - Text segment — object code, a.k.a. machine language.
  - Data segment — constants, static variables.
  - Relocation information – whatever is needed to "fix up" absolute addresses when program is loaded.
  - Symbol table — locations of externally-visible symbols (e.g., procedure names), unresolved references (e.g., to library procedures).
- Linker must resolve unresolved references, pulling in library code as needed, and also "fix up" absolute addresses if necessary (for modern systems, usually not).

Slide 8

Loaders

- Loader's job is to copy "executable" into memory and get it ready to run. Format of executable depends on operating system, but usually similar to object code.
- Details depend on operating system, but in general, loader must look at executable to determine how much memory is needed, allocate it (ask o/s), copy instructions into memory, set up parameters and registers, and "call" it (passing in parameters as needed and providing a return address — in loader or o/s, usually).

Slide 9

Minute Essay

- Anything we should talk about more before moving on to chapter 4?
- "Pretest" for chapter 4:
  - Convert 34 (base 10) to binary (base 2).
  - Convert 34 (base 10) to hexadecimal (base 16).
  - Convert 19 (base 16) to decimal (base 10).
  - Convert -34 (base 10) to binary, using 16-bit two's complement notation.