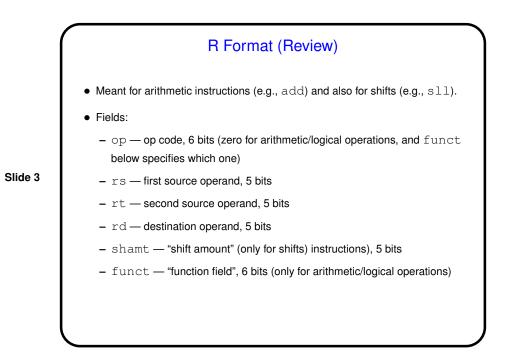


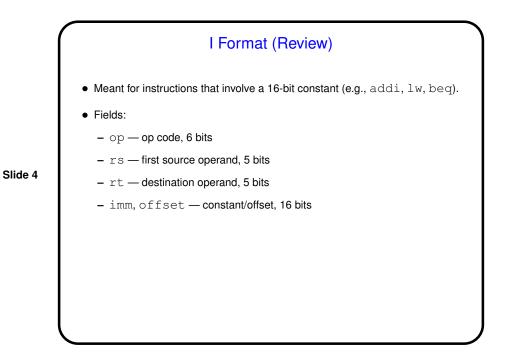
Machine Language – Review/Recap

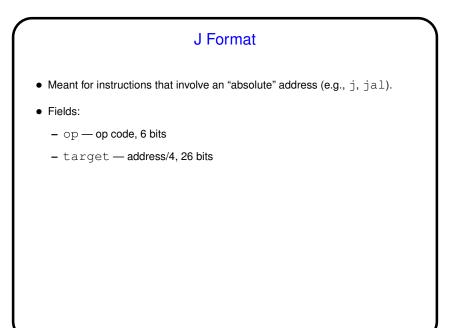
• We've worked through some examples of getting machine language (binary) from assembly language, using "reference card" in front of (paper version of) textbook. First step is to write down values for all fields in instruction (specifics different for different types of instructions). How to get from that to 32-bit binary number or 8-digit hexadecimal number? concatenate fields, convert.

Slide 2

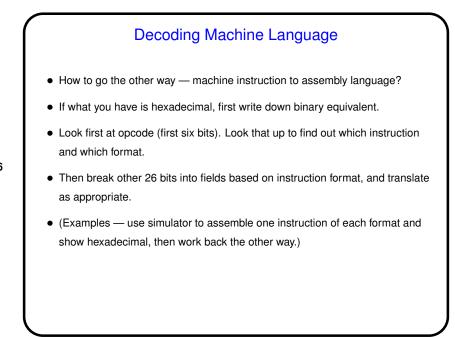
• (Could do more examples, but not needed?) Quickly reviewing formats to remind you about what they're used for ...

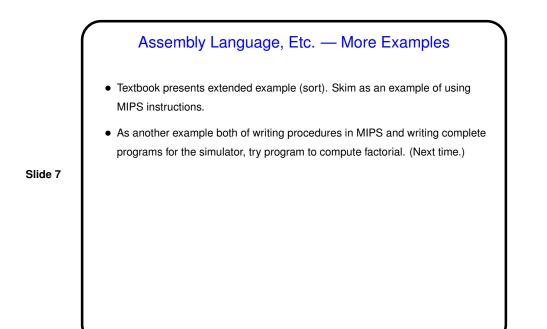


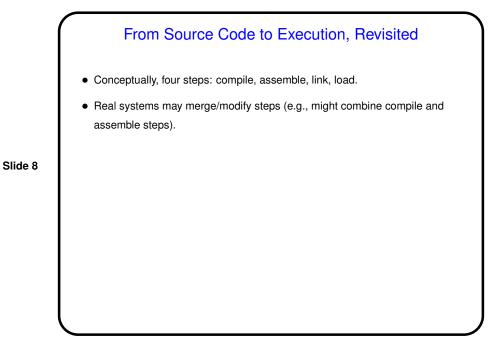




Slide 5

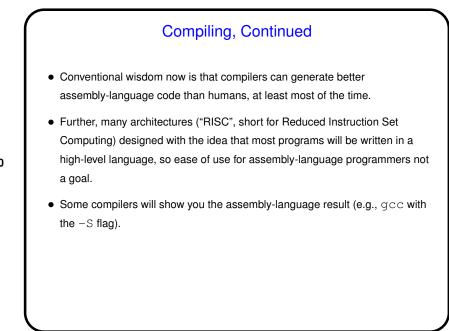


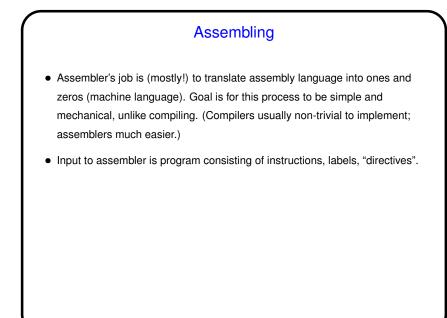


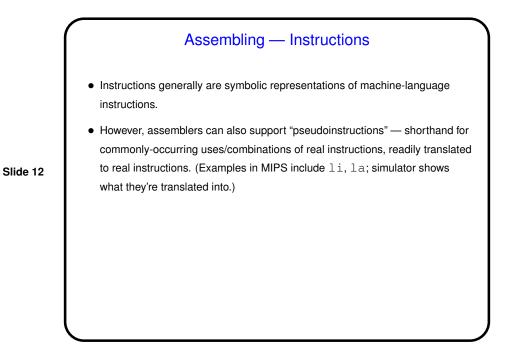


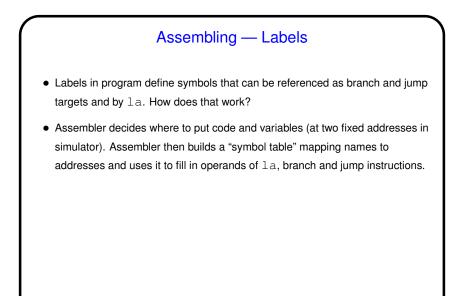
Compiling
Compiler translates high-level language source code into assembly language. A single line of HLL code could generate many lines of assembly language.
Just generating assembly language equivalent to HLL is not trivial. Result, however, can be much less efficient than what a good assembly-language programmer can produce. (When HLLs were first introduced, this was an argument against their use.)
So compilers typically try to optimize — keep values in registers rather than in memory, for example.

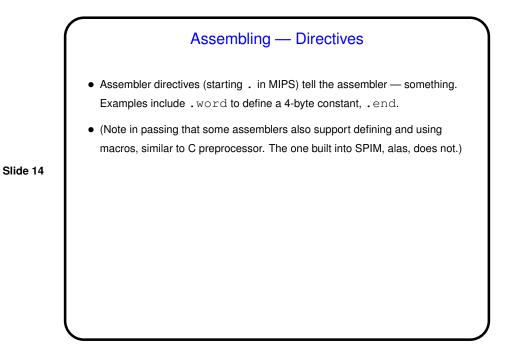
Slide 9







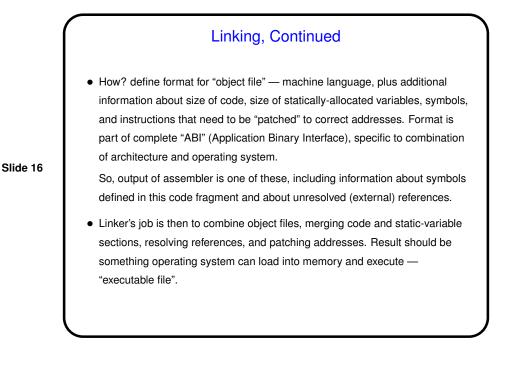




## Linking

 For small programs assembling the whole program works well enough. But if the program is large, or if it uses library functions, seems wasteful to recompile sections that haven't changed, or to compile library functions every time (not to mention that that requires having their source code).

- So we need a way to compile parts of programs separately and then somehow put the pieces back together i.e., a "linker" (a.k.a. "linkage editor").
- To do this, have to define a mechanism whereby programs/procedures can reference addresses outside themselves and can use absolute addresses even though those might change.



## Sidebar: Dynamic Linking

 In earlier times linkers behaved as just described, linking in all needed library code. But this may not be efficient: It may result in pulling in code for unused procedures. Also, if the system supports concurrent execution of multiple threads/applications/etc., might be nice to allow them to share a single copy in memory of library code.

Slide 17

Slide <sup>-</sup>

- "Dynamic linking" supports this, and has the side benefit(?) of allowing updates to library code without relinking all applications that use it. (Is this always a benefit?)
- Implementations have different names "DLL" in Windows, "shared library" in UNIX. How it works is similar — at link time, link in "stub" routine that at runtime locates the desired code, loads it into memory (if necessary!) and patches references.

Loaders
• So what's left
• "Executable file" contains all machine language for program, except for any dynamically-linked library procedures. What does the operating system have to do to run the program? Well
• Obviously it needs to copy the static parts (code, variables) into memory. (How big are they?) Also it needs to set up to transfer control to the main program, including passing any parameters. And it may need to perform dynamic linking. Finally, what about those absolute addresses?
• So as with object code, executable files contain more than just machine language. File format, like that of object code, is part of ABI.
• Textbook has an example of linking. To be reviewed next time

