## Administrivia

- Reminder: Quiz 2 Thursday. Topics from chapter 2, the sections we've covered in class.

- Appendix A has some additional information about MIPS assembler language. in particular a section with short descriptions of all instructions and table that maps opcode to instruction name.

  Also includes a kind-of-discussion of why offsets in branch instructions are not what I think they should be.

**Slide 1**

## Machine Language, Revisited

- We worked through at least one example of getting machine language (binary) from assembly language, using "reference card" in front of (paper version of) textbook.

- Additional examples might be useful? using different "instruction formats"?

**Slide 2**

## R Format (Review)

**Slide 3**

- Meant for instructions such as `add`, `sll`.

- Fields:
  - `op` — op code, 6 bits
  - `rs` — first source operand, 5 bits
  - `rt` — second source operand, 5 bits
  - `rd` — destination operand, 5 bits
  - `shamt` — "shift amount" (not used for all instructions), 5 bits
  - `funct` — "function field", 6 bits (not used for all instructions), 6 bits

- (Examples.)

## I Format (Review)

**Slide 4**

- 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

- (Examples.)

# J Format

- Meant for instructions that involve an "absolute" address (e.g., `j`).

- Fields:

  - `op` — op code, 6 bits

  - `target` — address/4, 26 bits

**Slide 5**

- (Example.)

# Decoding Machine Language

- As a check on whether what we have is sensible — try going from machine language back to assembly language, using same examples?

- (How to know which format? depends on opcode, always first 6 bits.)

**Slide 6**

## From Source Code to Execution, Revisited

**Slide 7**

- Conceptually, four steps: compile, assemble, link, load.

- Real systems may merge/modify steps (e.g., might combine compile and assemble steps).

## Compiling

**Slide 8**

- 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, e.g. Conventional wisdom now is that compilers can generate better assembly-language code than humans, at least most of the time.

- Some compilers will show you the assembly-language result (e.g., $gcc$ with the $-S$ flag).

## Assembling

- Assembler's job is (mostly!) to translate assembly language into ones and zeros (machine language). Goal is for this process to be simple and mechanical, unlike compiling (usually?).

**Slide 9**

- As part of this, assemblers typically allow programmer to use symbolic labels to refer to addresses (targets of jumps and conditional branches, variables). To make this work, assembler must keep "symbol table" mapping names to addresses.

- Assemblers also sometimes support "pseudoinstructions" — shorthand for commonly-occurring uses/combinations of real instructions, readily translated to real instructions. (Examples in MIPS include `li`, `la`.)

- (Some assemblers also support defining and using macros, similar to C preprocessor.)

## 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).

**Slide 10**

- 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.

## Linking, Continued

**Slide 11**

- How? define format for "object code" — machine language, plus additional information about size of code, size of statically-allocated variables, symbols, and instructions that need to be "patched" to correct addresses. Format is part of complete "ABI" (Application Binary Interface), specific to combination of architecture and operating system.

- Linker's job is then to combine pieces of object code, merging code and static-variable sections, resolving references, and patching addresses. Result should be something operating system can load into memory and execute — "executable file".

## Sidebar: Dynamic Linking

**Slide 12**

- In earlier times linkers behaved as just described, linking in all needed library code. But this may not be efficient: 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.

- "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 . . .

**Slide 13**

- 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 . . .

## This and That

- Textbook presents extended example (sort). Skim as an example of using MIPS instructions.

- Textbook goes into some detail about compiling C code to loop through an array, showing a version that uses indices and one that uses pointers. Skim it

**Slide 14**

  as another example, but the take-home message is that whether C programmers need to consider such things — probably not with a sufficiently "good" (optimizing) compiler.

# Minute Essay

- One advantage of dynamic linking is that it allows for replacing/updating library procedures (with no need to recompile/relink applications that use them). Is there a disadvantage to this?

**Slide 15**

# Minute Essay Answer

- Yes — if the replacement library code has new bugs, applications that worked may fail. Also, applications that rely on undocumented behavior may stop working.

**Slide 16**