Slide 1

Administrivia

- Almost everyone did well on Quiz 1 (yay!). If you didn't — six of these, and I drop the lowest score. Solutions will show up online, linked from the bottom of "Lecture topics and assignments", shortly after the quiz.
- Reminder: Homework 2 due today.
- Next homework on the Web; due in two weeks. This one may take you some time, so try to start soon.

  There are a couple of programming problems on this one. It's up to you what tool to use to write the programs, but this might be an opportunity to get better with `vim`. I recently wrote up some notes for my 1120 class and put a link on this course's "Useful links".
- Quiz 2 rescheduled for next Wednesday (2/14).

Slide 2

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

Slide 3

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 for shifts) instructions), 5 bits
  - `funct` — "function field", 6 bits (only for arithmetic/logical operations)

Slide 4

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

Slide 5

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

Slide 6

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.
- Then break other 26 bits into fields based on instruction format, and translate as appropriate.
- (Examples — use simulator to assemble one instruction of each format and show hexadecimal, then work back the other way.)

Slide 7

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, try program to compute factorial. (Next time.)

Slide 8

From Source Code to Execution, Revisited

- Conceptually, four steps: compile, assemble, link, load.
- Real systems may merge/modify steps (e.g., might combine compile and assemble steps).

Slide 9

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 10

Compiling, Continued

- Conventional wisdom now is that compilers can generate better assembly-language code than humans, at least most of the time.
- Further, many architectures ("RISC", short for Reduced Instruction Set Computing) designed with the idea that most programs will be written in a high-level language, so ease of use for assembly-language programmers not a goal.
- Some compilers will show you the assembly-language result (e.g., `gcc` with the `-S` flag).

Slide 11

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. (Compilers usually non-trivial to implement; assemblers much easier.)
- Input to assembler is program consisting of instructions, labels, "directives".

Slide 12

Assembling — Instructions

- Instructions generally are symbolic representations of machine-language instructions.
- However, assemblers can also support "pseudoinstructions" — shorthand for commonly-occurring uses/combinations of real instructions, readily translated to real instructions. (Examples in MIPS include `li`, `la`; simulator shows what they're translated into.)

Slide 13

Assembling — Labels

- Labels in program define symbols that can be referenced as branch and jump targets and by `la`. How does that work?
- Assembler decides where to put code and variables (at two fixed addresses in simulator). Assembler then builds a "symbol table" mapping names to addresses and uses it to fill in operands of `la`, branch and jump instructions.

Slide 14

Assembling — Directives

- Assembler directives (starting `.` in MIPS) tell the assembler — something. Examples include `.word` to define a 4-byte constant, `.end`.
- (Note in passing that some assemblers also support defining and using macros, similar to C preprocessor. The one built into SPIM, alas, does not.)

Slide 15

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.

Slide 16

Linking, Continued

- How? define format for "object file" — 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.

  So, output of assembler is one of these, including information about symbols defined in this code fragment and about unresolved (external) references.
- Linker's job is then to combine object files, 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".

Slide 17

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.
- "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.

Slide 18

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 . . .

Slide 19

Minute Essay

- 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
label:
        addi    $s0, $s0, 1
        add     $s1, $s1, $s1
        bne     $s0, $s2, label
```

- Anything noteworthy about Homework 2?

Slide 20

Minute Essay Answer

- We could trace through the code, which sets values in three registers and then executes a loop:

  `$s0` is initially set to 0 and then takes on values 1, 2, 3, and 4

  `$s1` is initially set to 1 and then takes on values 2, 4, 8, and 16

  `$s2` is initially set to 4 and doesn't change