Slide 1

Administrivia

- Reminder: Homework 3 due today (written problems 5:30pm, programming problem by 11:59pm).
- Homework 4 posted; due next Friday.

  One written problem; may look intimidating but doable if you go step by step. One programming problem; should not to be too hard using factorial examples as a model.
- Quiz 3 next Wednesday.

Slide 2

More Administrivia

- Yes, the homeworks are coming at you fairly quickly. This is because we have an exam scheduled in two weeks, and I want to wrap up the discussion of MIPS assembler language before that, preferably in time for you to do homeworks and get feedback on them.

  (I'll probably send out a mail-to-all later today asking about the pace of the class, the scheduling of the exam, etc.)

Slide 3

More Administrivia

- Homework 1 graded. (Finally!) Most people did well.

  In case you wonder about point deductions, my scheme for problems with multiple parts is this:

  Deductions for individual parts to the right, deductions for whole problem on the left. (I *think* this makes it easier for me to add them up.)

Slide 4

Minute Essay From 2/13 Lecture

- Many people seem to be finding this material not-easy, but many also say it becomes clearer when they do the homeworks. That's the goal!
- One person remarked that it seems like every time she starts to make sense of one concept, here comes another, equally "foreign and confusing". Frustrating, I suppose, but as long as things make sense with time and practice?

Slide 5

Minute Essay From 2/15 Lecture

- Most people who've replied (16 out of 42 as of 2:20pm today — ?!) said the review was helpful. Good!

Slide 6

Homework 1 Essays

- Many people found *something* about the problems interesting, though many also found them somewhat tedious, and one person said “busy-work”. “Yeah well”?
- Several people commented on the problem about fallacies. This is one of the things I like about this discussion — performance not as simple as one number.

  One person commented that it's interesting that after discussion of how hard it is to quantify performance we then talk about “best performance”. Good point!
- For the problem about computing parallel execution times, several people didn't realize “sequential time” and “parallel time on 1 processor” were distinct things. But they are, and this is not atypical of real programs.

Slide 7

Homework 1 Essays, Continued

- Some people *did* write code for that problem, though nowhere a majority. Of 40, 12 people wrote code:
  - 4 Scala
  - 2 Python
  - 2 spreadsheet
  - 1 C
  - 1 C++
  - 1 Haskell
  - 1 unknown-to-me language

  (It would never have occurred to me to use a spreadsheet. But getting new-to-me perspectives is a fun part of my job?)

Slide 8

From Source Code to Execution — Recap/Review

- Four main phases, conceptually at least — compile, assemble, link, load.
- Real systems (or simulators) may combine steps, in appearance or even in reality — e.g., a compiler might go directly from high-level source to object code, in appearance or in fact, and the SPIM simulator assembles “on the fly”.

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.)
- But eventually compilers got “smarter” . . .

Slide 10

Compiling, Continued

- One reason compilers are so big and complicated is that more and more they try to “optimize” (generate code that’s more efficient than a naive translation), for example, by keeping values in registers to reduce the number of memory accesses.
- 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). (A bit more about this another time.)

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`.
- Two worth additional mention here — `.text`, `.data`:

  Typically output includes “text (code) segment” consisting of machine-language instructions and “data segment” containing fixed/static data.

  `.text`, `.data` tell assembler which of the these to use for following code.

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.

Slide 17

Linking, Continued

- 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".
- (Note in passing that this is "static linking", as opposed to "dynamic linking". More about the latter soon.)

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 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.

  (More soon.)

Slide 19

Minute Essay Answer

- None — quiz.