Slide 1

Administrivia

- Reminder: Homework 4 due today. If you haven't watched the video lecture from last week (and at this point only about two-thirds of those enrolled have sent me a minute-essay response indicating they had), I strongly encourage you to do so before finishing this assignment. Remember: You have the option of turning in a preliminary version of the homework today and a better version soon.
- Exam 1 postponed until after break. I plan to use part of the class period just before it for review, but there's a review sheet on the course Web site if you want to start preparing.
- Reminder: Quiz 3 Wednesday. Likely topics are more MIPS programming (including procedure calls) and linking.
- I should be able to return graded Homework 2 later today. Sorry about the delay!

Slide 2

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 3

Compiling — Review(?)

- 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 4

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).

Slide 5

Compiling, Continued

- Compilers are big and complicated partly because they try to generate efficient code (while, one hopes, preserving the program's meaning!).
- As an example: 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. A "good" compiler will likely generate the same code for both.

  Can test this with `gcc` — write it both ways, compile with `-S`, and compare. Last time I checked, identical if compiled with `-O`, but not so with current version. "Hm!"?
- Note in passing that compiler optimizations can play havoc with attempts to time things: C compilers are allowed to just skip any code that doesn't have an observable effect (i.e., result isn't printed or otherwise used). (In practice they may or may not.)

Slide 6

Assembling — Review(?)

- 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 7

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.)

  (Aside: I prefer to mostly avoid these; I think you understand the primitive operations better if you stick to "real" instructions with a few exceptions such as `la`.)

Slide 8

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 9

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 10

Linking — Review(?)

- 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 11

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 12

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 later.)

Slide 13

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 (more about that later). 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.

Slide 14

Arithmetic Overflow

- You might notice that the factorial example quietly gives wrong results for larger inputs (and they don't even have to be very large!).
- Compare to what happens if you write an equivalent program in a high-level language . . .
- When result-in-process gets too big to fit into available space (32-bit register here), two options: Hardware can signal exception, or it can just drop high-order bits. Result can look negative, or it can just be wrong.

Slide 15

Arithmetic Overflow, Continued

- "Signal exception"? Yes. We'll talk more about this later, but possible to build hardware that detects overflow and does something. (Apparently SPIM doesn't do this.)
- But since many programming languages ignore overflow, often instructions have signed form that checks and unsigned form that doesn't (e.g., `addu` versus `add`).
- Really careful programmers put in their own checks for overflow. May actually be *easier* in assembly language: `mult` instruction generates 64-bit result in special-purpose registers . . .

Slide 16

Multiplication and Division

- In the factorial example I use what appears to be an instruction `mul`. Really a pseudoinstruction (though SPIM doesn't seem to think so).
- "Real" multiply has only two operands

  ```
  mult src_reg1, src_reg2
  ```

  and it puts a 64-bit result in special-purpose registers `lo`, `hi`. Can access them with `mflo`, `mfhi`, e.g.,

  ```
  mflo dest_reg
  ```

- Divide (`div`) similar; quotient goes in `lo` and remainder in `hi`.

Slide 17

More Examples

- Now we could add code to the factorial example to check for overflow. (I'll write this, maybe with your help.)
- Another possibly interesting example would be a a procedure like the simple one I show for CSCI 1120 to divide, with two pointer parameters to allow "returning" both quotient and remainder. (Next time.)

Slide 18

Minute Essay

- In the programming problem for Homework 4, I say you won't get full credit if you don't follow conventions for calling procedures. Why does this matter? Couldn't you pass arguments to a procedure in whatever registers you want, as long as caller and called agree?
- And why save/restore registers?
- (P.S. Vote!)

Slide 19

Minute Essay Answer

- If you write both the calling program and its called procedures, it might seem like it hardly matters how you communicate between caller and called. But think about how well (or not well) this would work for a larger project! and even for small projects, isn't it easier to always follow convention rather than inventing one for each procedure?
- Saving/restoring registers . . . You can skip this if the procedure doesn't modify any of the registers normally saved/restored. I say probably good style to do it anyway; better to just copy boilerplate than try to think through exactly what each case needs?