Slide 1

Administrivia

- Reminder: Quiz 2 Wednesday. Likely topics those covered in Homework 2 (so, probably C to assembly or the reverse, assembly to machine language or the reverse, "what does this code do?").
- Minor updates to Homework 3 posted:

  For the first problem, I want you to show the machine instruction in hexadecimal.

  For the assemble/link problem I want you to use the addresses SPIM uses for the text and data segments.

  For the first programming problem, you need a solution to a problem on Homework 2. I've put a sample solution on Google Drive and shared it with all of you (I hope).

Slide 2

This and That

- If you haven't already found this — there *is* a table mapping opcodes to instructions, hidden in Appendix A (figure A.10.2).
- Also in Appendix A is a summary of register names/usage. Worth noting that with the exception of registers 0 and 31, they're all the same to the hardware; designating some of them for use as temporaries, another as a stack pointer, etc., is purely a matter of convention, but so useful . . .
- Also in Appendix A is a complete list of instructions and pseudoinstructions. I prefer that you *not* use the pseudoinstructions, with a few exceptions that are hard to avoid, such as `la`.
- MIPS assembly language also provides for defining "macros"; more in section A.2. Alas, not supported by SPIM. (Some other assembly languages use this a lot.)

Slide 3

Memory Layout

- Again the hardware imposes no particular distinctions on how memory is used, but useful to adopt conventions. The one described in the text is typical. From smallest to largest addresses:
  - A reserved block (usually for O/S use).
  - A block for the program's text segment (code).
  - A block for the program's data segment, divided into static data (globals, etc.) and dynamic data ("the heap"). UNIX systems further subdivide this into a segment for fixed data with values assigned at compile time and a segment with space for other static data (not initialized) and dynamic data.
  - Possibly unused space.
  - A block for the stack segment.
- Notice that the data segment grows toward larger addresses, the stack segment toward smaller addresses.

Slide 4

From Source to Execution — Linking

- As mentioned, object and executable files contain machine language and other information.
- Details vary, but if you're curious, a Web search on "ELF file format" should find information on a format used in many UNIX-like systems.

  Commands `readelf`, `nm`, and `ldd` are interesting to try with object and executable files.

Slide 5

Linking — Review

- Job of linker is combine one or more object files into "executable file". Details vary among platforms, but must include anything the operating system needs to load the program into memory and start it up — sizes of code and data segments, location of starting address, anything that needs to be resolved/fixed at runtime.
- So at a minimum, linker must:
  - Merge tables of "global" symbols into combined symbol table.
  - Use it to resolve unresolved references.
  - Merge code segments, data segments.
  - Modify any absolute addresses.
  - Output executable file.

Slide 6

Linking — Example

- Textbook presents an example starting on p. 127. Some details seem a bit murky, so let's work through it . . .
- One source of possible confusion is the handling of `lw` and `sw` instructions, which apparently . . .

Slide 7

`lw`, `sw` Revisited

- Strictly speaking, these instructions specify a memory address using a register and a fixed displacement.
- However, seems useful to be able to be able to load and store from address specified via label. Assembler could support that . . .
- SPIM apparently does so by turning a load/store referencing a label into two instructions, a `lui` to in effect put the address of its data segment in `$at` (register used as "assembler temporary" — in expanding pseudoinstructions), and then a load/store using this register and a displacement calculated during combined assemble/link/load.
- The textbook's example presupposes a different scheme: Register `$gp` points into the middle of the data segment, and load/store instructions are assembled into code that references this register and a displacement calculated during link stage.

Slide 8

Linking — Example, Continued

- The one more thing we need to know to do the link is the location of the text (code) segment and possibly the data segment.
- The textbook's example uses `0x00400000` for the location of the text segment and `0x10000000` for the location of the data segment, and also that register `$gp` contains `0x10008000`.

  In the homework, I ask you instead to assume a fixed location for the data segment, as SPIM does.

Linking — Example, Continued

- A real linker would need to "patch" machine language for the `lw`, `sw`, and `jal` instructions that are incomplete in the object code.
- For the `jal`, it would need to look up the label in the symbol table and then divide by 4.
- For the `lw` and `sw`, it would need to look up the label in the symbol table (say it's at address P), and compute a 16-bit displacement D that when sign-extended and added to the value in `$gp` gives P.

From Source to Execution — Loading

- Nice summary in Appendix A of what happens in loading. Operating system must:
  - Read executable file to determine sizes of text and data segments.
  - "Create address space" big enough for text, data, and stack segments. (Details vary by O/S.)
  - Initialize text and data segments from executable file.
  - Set up registers — stack pointer, global pointer, etc.
  - Push any arguments to program onto stack.
  - Jump to start-up code that copies arguments to registers and calls program's `main()`. On return, makes a system call to terminate program.
- Note in passing that code invoked by "system calls" is not part of the program; the `syscall` instruction jumps to code in the O/S's part of memory.

Minute Essay

- Questions about today's material (or anything else)?
- How far have you gotten with Homework 3?