Slide 1

Administrivia

- Via e-mail.

Slide 2

Assemblers, Linkers, Loaders — Recap/Review

- Assemblers translate source code into object files containing translated code and data plus some tables to make linking with other object files possible.
- Linkers combine object files to produce executables.
- Loaders load executable files into memory and start the program.

Slide 3

Linking — Example

- To recap/review:
- Textbook presents an example starting on p. 132. Details are frankly confusing, especially for SPIM users — and trying to clarify is what sent me off on my days-long attempt to understand better!
- Part of the problem — discussion doesn't seem to mesh well with how SPIM does things . . .

Slide 4

Referencing Data in Data Segment

- Textbook correctly points out that keeping an address into the data segment in a register (`$gp`) means we can get an address one instruction not two. *However*, note that the range of data that can be addressed this way is limited to what can be accessed using 16-bit offsets.
- What if you have a data segment bigger than that?

Slide 5

Referencing Data in Data Segment, Continued

- Solution used in some (many? most?) systems — separate data segment into distinct pieces:
  - Read-only data, initialized; no size limit.
  - Data, initialized; no size limit.
  - Data, unintialized; no size limit.
  - "Small" initialized data.
  - "Small" uninitialized data.

  Combined size of "small" pieces must fit in memory addressable using `$gp` and a 16-bit offset.

Slide 6

`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 . . .

Slide 7

`lw`, `sw` With Labels — Textbook's Way

- Register `$gp` points into the data segment, at an address that will allow addressing as much of the data segment as is possible using a 16-bit signed value (which is what displacement is in `lw` and `sw`).
- `lw` and `sw` are assembled into code that uses `$gp`, e.g.,

  ```
  lw  $t0, X
  ```

  is assembled into

  ```
  lw  $t0, D($gp)
  ```

  where `D` is the displacement from `$gp` to `X`, calculated during linking.
- (What if the data segment is too big for this? Not sure the textbook talks about that!)

Slide 8

`lw`, `sw` With Labels — SPIM's Way

- SPIM puts its data segment at `0x1000 1000`. It does initialize `$gp` to `0x1000 8000`, like the textbook says.
- But what does this say about referencing code in SPIM's data segment using `$gp`, given that SPIM's data segment starts at `0x1000 1000`? Pause and think about it . . .

Slide 9

`lw`, `sw` With Labels — SPIM's Way, Continued

- With `$gp` set to `0x1000 8000`, 16-bit offsets allows referencing `0x1000 0000` through `0x1000 ffff`. And SPIM's data segment starts where . . .
- (I can't believe I didn't think of it that way earlier! No *wonder* I couldn't find any way to make SPIM accept code that would assemble to what the textbook says!)

Slide 10

`lw`, `sw` With Labels — SPIM's Way, Continued

- SPIM apparently defines pseudoinstructions for `lw` and `sw` with labels. Based on some experiments . . .
- Just referencing a label, e.g.,

  ```
  lw  $t0, A
  ```

  assembles into an `lui` to put the top 16 bits of the address of SPIM's data segment into `$at` (and zero the low-order bits), and then a `lw` that uses `$at` for the register and the offset to `A` as the displacement (calculated using symbol table).

  (Try it!)

Slide 11

`lw`, `sw` With Labels — SPIM's Way, Continued

- Referencing a label and a register, e.g.,

  `lw  $t0, A($t1)`

  assembles similarly, except that the `lui` to set `$at` to the address of the data segment is followed an `addu` (unsigned add) to add the contents of `$t1`. (Note that if `$t1` is an index into an array of "words" this won't do what you might want.)

Slide 12

Linking — Textbook Example Continued

- Computing displacements for `lw` and `sw` — why it works may be unclear.
- Goal is to come with up displacements, call them `DX` and `DY`, that when added to address in `$gp` (`0x10008000`) gives addresses of `X` and `Y`.

  (We know what those are based on positioning data segments one after the other starting at `0x10000000`).
- Some simple algebra says that, e.g.,

  `DX` is `0x10000000 - 0x10008000`

  which when turned into a *16-bit* quantity in two's complement is what the textbook says (try adding it to what's in `$gp` using sign extension).

Slide 13

Sidebar: Making it (Somewhat) Real

- I like to be able to try things and see if they work like the books say they do. Maybe you too (or maybe not!).
- To do this with MIPS assembling and linking, would need a cross-assembler. But I thought it might be interesting to try it on x86 . . .
- I started by writing a simple C program with two files, generated object and executable files, and tried out commands . . .

Slide 14

Sidebar Continued

- `nm pgm.o` shows symbol table. `man nm` for what the more-inscrutable things mean.

  `objump -t` shows roughly the same thing in a different format. `man objdump` explains.
- `objdump -r` shows relocation information.
- `nm main` is — interesting. Way more than one might think. Undefined symbols still — dynamically-linked code. `ldd main` may help some.
- `objdump -r main` not useful, but `obdump -R main` is.
- `objdump -t main` — again, more than one might think.

Slide 15

Homework 4 — Example of Assembling / Linking

- Next homework will ask you to work through (some) details, using as input a couple of representative MIPS source-code files. Useful but time-consuming. I'm working on how to streamline . . .

Slide 16

Minute Essay

- Does this all make (some) sense? In a way I feel like a lot of the details are kind of common sense once you understand the goal (allow for separate compilation, including combining code in different languages). Agreed? or maybe "agreed, but the devil is in the details"?