Slide 1

Administrivia

- (None.)

Slide 2

Flow of Control — Review/Recap

- With `beq`, `bne`, `j`, and `slt`, we have almost everything we need to translate from a HLL to MIPS assembly.
- Except for calls to functions (what the textbook calls "procedures"). So . . .

Slide 3

Procedure Calls

- How do we call procedures (a.k.a. functions, methods)? Consider an example:

```
        a = a + a;
        x = foo(a);
        b = b + b;
        y = foo(b);
        /* .... */
int foo(int n) { return n+1; }
```

- If we've compiled this code (and function `foo`), what do we have in memory when it's running? What's supposed to happen when we get to a call to `foo`?

Slide 4

Procedure Calls, Continued

- What we have in memory is machine code for the calling program and code for `foo`.
- What should happen when `foo` is called:
  - Jump to start of `foo`, passing it one parameter.
  - Execute `foo`.
  - Jump back to caller, at the point in the code just after the call, with a return value the caller can use.
- Jumping to `foo` we know how to do, but how to get back? And how do we manage parameters and the return value?

Slide 5

Procedure Calls, Continued

- So what we have to do to call a procedure is:
  - Put parameters where procedure can find them.
  - Transfer control to procedure.
  - Acquire storage resources for procedure (recall that every time you call a C function you get a "new copy" of all its local variables).
  - Run procedure.
  - Put results where caller can find them.
  - Return control to caller, to a point just after the call.
- How to do all this?

Slide 6

Procedure Calls, Continued

- Aside(?):

  Every language that compiles (or assembles) to machine language *could* do it differently, but useful to define standard way, so languages can interoperate. (Also allows operating system to load program and start it up without knowing source-code language.)

Slide 7

Sidebar: Register Conventions Revisited

- From hardware point of view, all general-purpose registers are in some sense the same, with the sort-of exception of registers 0 (always has value 0) and 31 (discussed soon).
- From software point of view, it's useful to agree about how to use them — for parameters, return values, etc. Idea is that compilers automatically enforce conventions, human-written assembly code should follow them too.

Slide 8

Register Conventions, Continued

- So far:

  `$s0` through `$s7` for variables.

  `$t0` through `$t9` as "scratch pads".

- Add two more groups:

  `$a0` through `$a3` for parameters (punt for now on what to do if more than four).

  `$v0` and `$v1` for return values. (Why two? to make it easy to return a 64-bit value such as used for floating-point.)

Slide 9

Jumping To/From Procedures

- When we jump to a procedure, must remember where we came from so we can return. Do this with "jump and link"

  ```
  jal     label
  ```

  which puts address of next instruction in register `$ra` (31) and jumps to `label`. (How do we know address of next instruction? "Program counter" (special register) has address of current instruction.)

- We can then get back with "jump to register"

  ```
  jr    r1
  ```

  which jumps to address in register `r1`.

Slide 10

Register Saving and Local Variables

- Actually running the called procedure is straightforward, right?
- Yes, except we need some way to save/restore registers — so we don't mess up caller. (By convention, "temporary" registers might change, but most others don't.)
- We also need a way to make space for local variables.

Slide 11

Register Saving and Local Variables, Continued

- Typical solution: Use part of memory as a stack (familiar ADT, right?), for saving registers and other local storage. Makes recursive procedures easier.
- By convention, stack starts at high address and "grows" to lower addresses. and register `$sp` ("stack pointer") points to top. "Push" and "pop" are then straightforward. (Note: `$sp` just a symbolic name for one of the 32 general-purpose registers.)

  (Recall discussion of "buffer overflows" from CSCI 1120?)
- (Review starter code. Everything in it should now make some sense?)

Slide 12

Example

- How to compile the following?

```
int main(void) {
int a, b, c, x;
        a = 5; b = 6; c = 7;
        x = addproc(a, b, c);
        return 0;
}
int addproc(int a, int b, int c) {
        return a + b + c;
}
```

(Sample program `call-addproc.s`.)

Slide 13

Variables

- Space for local variables typically allocated on the stack. Since `$sp` can change during computation, can use register `$fp` ("frame pointer" — another of the 32 general-purpose registers) to point to start of area ("procedure frame") for saved registers, local variables.
- What about other variables?

  Two basic types: fixed/static (think global variables) and dynamically allocated (think C `malloc()`. (e.g., with `malloc` in C).

  MIPS convention: Put them right after the program code, use register `$gp` ("global pointer", also one of general-purpose ones) to point to them.

  Typically call the memory used for dynamically-allocated variables "the heap".

Slide 14

A Little (More) About Assembly Language and Assemblers

- We've done short examples of translating assembly language into machine language.
- Normally this is done programmatically, by an "assembler". Accepts symbolic representations of instructions. Also allows defining "labels" (strings ending `:`) and uses some directives (starting with ".", e.g., `.word`) to help keep track of instructions, define character strings, etc.
- Details for MIPS assembler in Appendix A.

Slide 15

Assembly Language — Program Elements

- Instructions: Self-explanatory? Each represents one machine-language instruction — usually anyway. Some are are "pseudoinstructions", translated into one or more "real" instructions (ones that have machine-language equivalents). Example is `la`, translated into combination of `lui` and `ori`.
- Labels: Identifier (following usual rules for such) followed by `:`. Useful/necessary in writing code but not (usually) preserved in object code.
- Directives: Start with `.` and tell the assembler something. (Next slide.)

Slide 16

Assembly Language — Directives

- `.text` indicates that what follows is instructions.
- `.data` indicates that what follows is data.
- `.word`, `.asciiz`, `.space` reserve space for data (and also, for the first two, initialize it).
- `.globl` identifies a label that might be referenced by outside code. (Think "separate compilation" and how one might combine object files. More about this soon.)

Slide 17

System Calls

- System calls are how user programs request service from operating system — not just in MIPS, but in general. In MIPS the instruction is `syscall`; other architectures have something analogous.
- System calls similar to procedure calls in some ways: Need to communicate to O/S which service you want (e.g., write some text to "standard output") and possibly parameters (e.g., text to write). As with procedure calls, do this by putting values in particular registers, but then rather than `jal` we use `syscall`.

  So why not just *use* `jal`?? Well . . .

Slide 18

System Calls, Continued

- Important distinction (discussed more in O/S courses, such as our CSCI 3323): Code for "system call" executes as part of the O/S, which means not subject to same restrictions as user programs (e.g., on memory access).
- Details (e.g., what services are offered) depend on O/S. Very primitive O/S included in `spim` supports some for simple I/O; details in Appendix A.

Slide 19

System Calls in MIPS — Details

- How to specify which service, arguments?

  Put number indicating which service in `$v0`. (Appendix A has a list of services.)

  If parameters needed, put them in `$a0` and `$a1`.

- Return value in `$v0`.

Slide 20

Minute Essay

- Questions?