Slide 1

Administrivia

- Reminder: Homework 1 due today at 5pm. Hardcopy preferred, but e-mail is acceptable in some circumstances.
- Sample programs on Web (follow "sample programs" link).
- Wikipedia article on "MIPS architecture" is (mildly) interesting reading. Still in use!

Slide 2

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);
```

- 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 3

Procedure Calls, Continued

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

Slide 4

Register Conventions

- From hardware point of view, all registers are equal (except 0).
- 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.
- So far — `$s0` through `$s7` used for variables, `$t0` through `$t9` used as "scratch pads". (See "green card" for numeric equivalents.)
- 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.

Slide 5

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` 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 6

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 7

Register Saving and Local Variables, Continued

- Common 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 `$sp` ("stack pointer") points to top.
- How to push / pop?
- Since `$sp` can change during computation, can use register `$fp` ("frame pointer") to point to start of area ("procedure frame") for saved registers, local variables.

Slide 8

Other Variables

- Last but not least, we (may?) need someplace to store variables that can be preallocated (static/global) and variables that are dynamically allocated (e.g., with `malloc` in C).
- By convention, we put them right after the program code and use register `$gp` ("global pointer") to point to them. Typically call the memory used for dynamically-allocated variables "the heap".

Slide 9

Procedure Calls, Revisited

- Calling procedure must:
  - Put parameters in `$a0` through `$a3` (if more than four, on stack).
  - Determine address of called procedure and jump there, saving address of next instruction.
  - Get return value from `$v0` (and `$v1`, if used).
- Called procedure must:
  - Save registers as needed, including return address.
  - Retrieve parameters and do calculation.
  - Put results in `$v0` and `$v1`.
  - Restore saved registers.
  - Return to caller.
- Example next time . . .

Slide 10

Minute Essay

- None — quiz.