Slide 1

## Administrivia

- Sample programs on Web.

Slide 2

## Minute Essay From Last Lecture

- Question: Write MIPS assembler for the following C code fragment:

```
while (i < h) {
        A[i] = i;
        i = i + j;
}
```

  assuming we're using `$s1` through `$s3` for `h`, `i`, `j`, and `$s4` for the address of `A`.

- Answer?

Slide 3

## Procedure Calls, Review

- 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 table on p. A-23 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). Other reason(s)?
- 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

## 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.

Slide 9

## One More Useful Instruction

- "Add immediate"

  `addi r1, r2, c`

  adds constant `c` (16-bit signed integer, can be negative) to contents of `r2`, puts result in `r1`.

Slide 10

## Example

- How to compile the following?

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

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

**Slide 11**

## Minute Essay

- What does the following code do? i.e., what is in registers `$s0` and `$s1` after it executes?

```
        add     $s0, $zero, $zero
        addi    $s1, $zero, 1
        addi    $s2, $zero, 4
l1:
        addi    $s0, $s0, 1
        add     $s1, $s1, $s1
        bne     $s0, $s2, l1
```