

## Minute Essay From Last Lecture (Review. Most people figured out that I format seemed right, but many expressed some confusion. A lot of this *is* new and strange, but I think with exposure it will start to make sense. Or ask if not!) Maybe this is where I can say: I started out in FORTRAN but it wasn't until a second-semester course in IBM mainframe assembly language that I really started to understand how programming worked. "Hm!"?



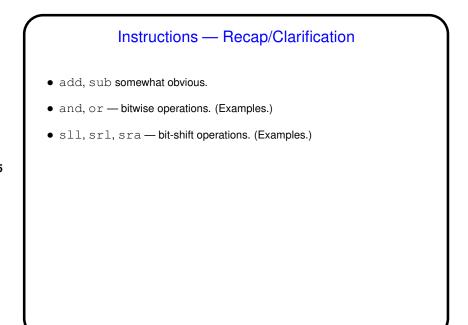
• Basic problem being solved is this: How to represent different kinds of instructions in binary? We've already seen that some instructions have the same kinds of operands (add and sub, e.g.), but not all the same (add and lw, e.g.).

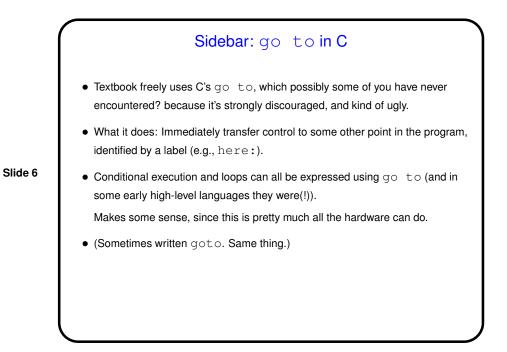
Slide 3

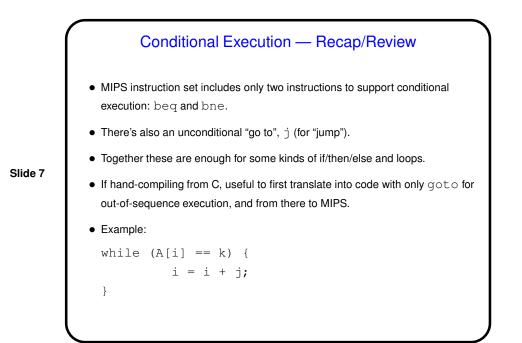
• MIPS solution: Make all machine-language instructions same size (32 bits), and always use the first 6 bits for "opcode" (something identifying instruction), then define different ways of splitting up the remaining bits — different "instruction formats", each with "fields".

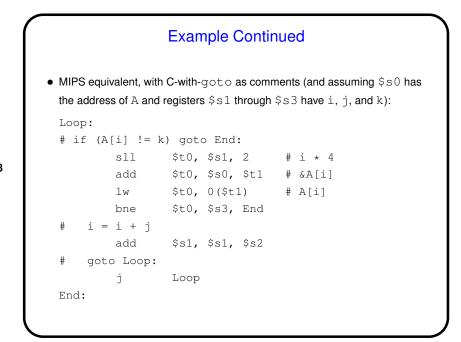


- Recall(?) simple trick for converting between binary (base 2) and hexadecimal (base 16): Based on observation that each hexadecimal digit represents four binary digits.
- (Why this works simple algebra based on writing out numbers as a sequence of multiples of powers of the base.)

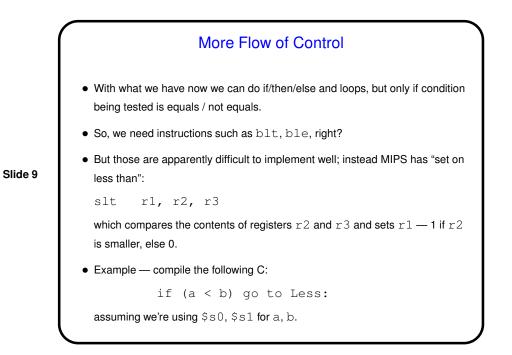






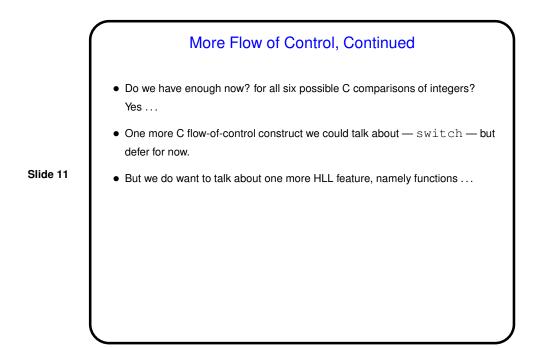


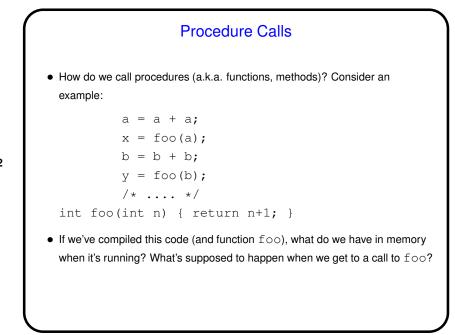
4

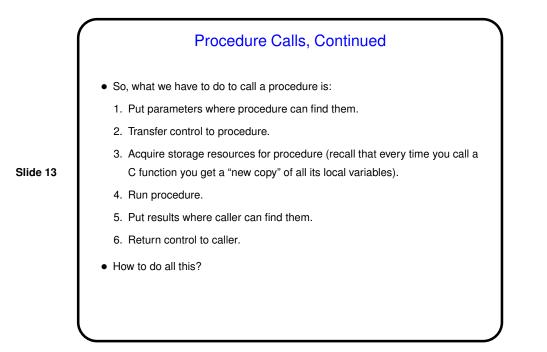


	Example Continued
	Equivalent MIPS:
	slt \$t0, \$s0, \$s1
	bne \$t0, \$zero, Less
Slide 10	

5

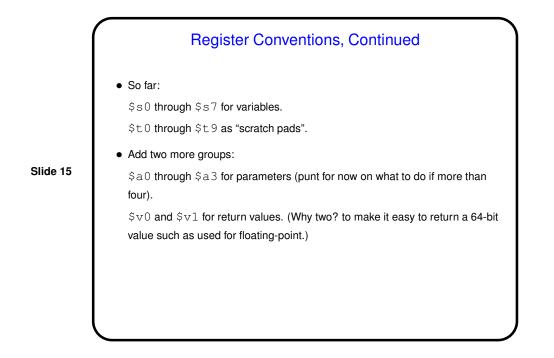


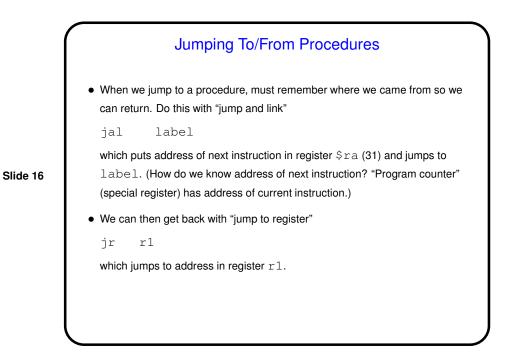




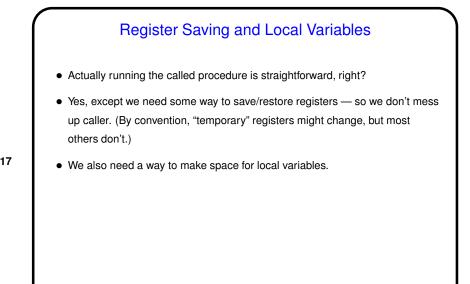
## Sidebar: Register Conventions Revisited

- 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).
- Slide 14
- 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.





8

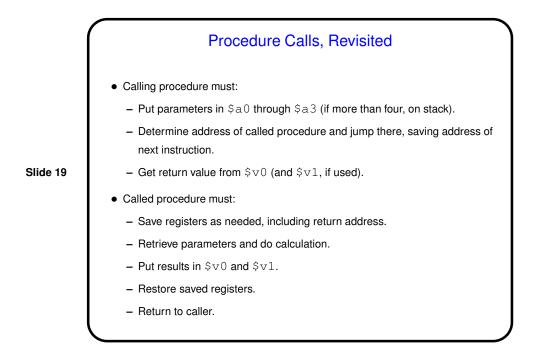


## Typical solution: Use part of memory as a stack (familiar ADT, right?), for saving registers and other local storage. Makes recursive procedures easier.

• 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.

(Recall discussion of "buffer overflows" from CSCI 1120?)

• (Now everything in the starter-code program should make sense?)



	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;
	}
	<pre>int addproc(int a, int b, int c) {</pre>
	return $a + b + c;$
	}
	(Sample program call-addproc.s.)

Slid

