

Minute Essay From Last Lecture
(Review.) Key point here is that which "format" to use depends on the syntax(?) of the instruction (how many and what kinds of operands) rather than on semantics(?) (is it arithmetic or — whatever).



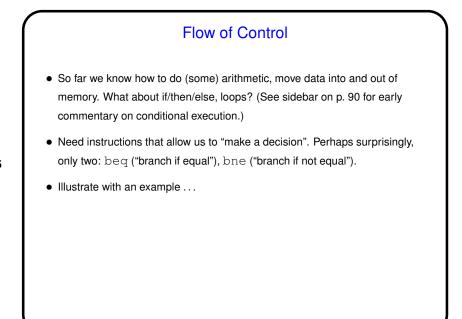
• Basic problem being solved is this: How to represent different kinds of instructions in binary? We've already seen that some instructions have the same kinds of operands (add and sub, e.g.), but not all the same (add and lw, e.g.).

Slide 3

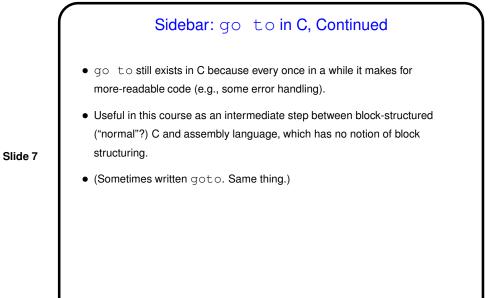
 MIPS solution: Make all machine-language instructions same size (32 bits), and always use the first 6 bits for "opcode" (something identifying instruction), then define different ways of splitting up the remaining bits — different "instruction formats", each with "fields".

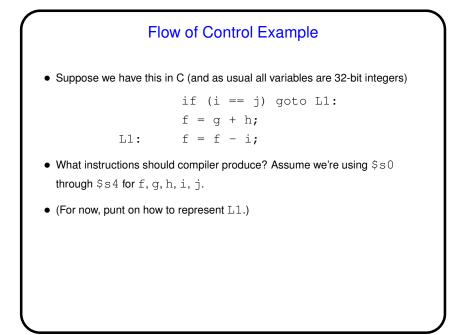
## Sidebar: Converting between Binary and Hexadecimal

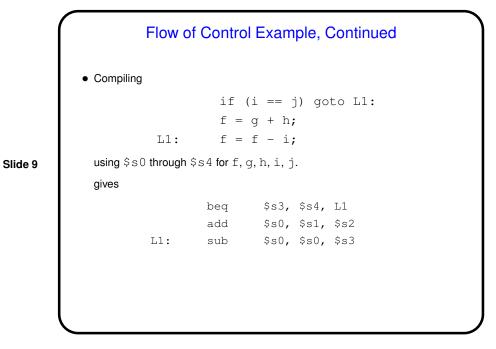
- Recall(?) simple trick for converting between binary (base 2) and hexadecimal (base 16): Based on observation that each hexadecimal digit represents four binary digits.
- (Why this works simple algebra based on writing out numbers as a sequence of multiples of powers of the base.)
- (Review if you don't remember how to do this.)



## Sidebar: go to Some very early HLLs implemented conditional execution using goto. What it does: Immediately transfer control to some other point in the program, identified by a label (e.g., here:). Conditional execution and loops can all be expressed using go to. Makes some sense, since this is pretty much all the hardware can do. Very quickly became apparent that this made for code that was hard to reason about. So later languages have been "block structured".

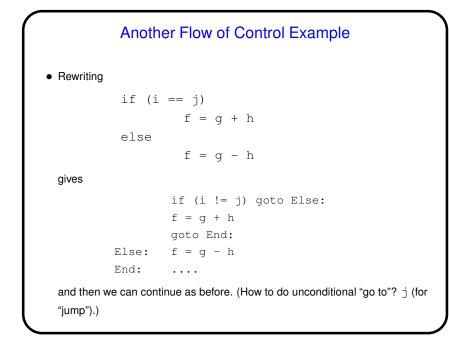






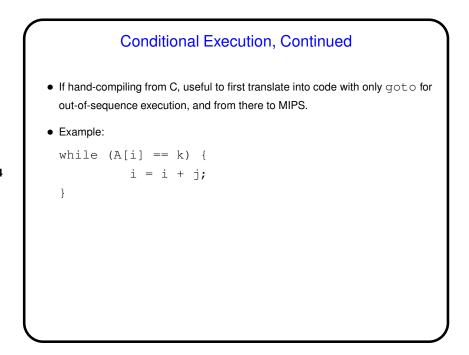
Of course, we don't usually have goto in C. More likely is this:
if (i == j) f = g + h else f = g - h
What to do with this? Rewrite using goto ...

Another Flow of Control Example

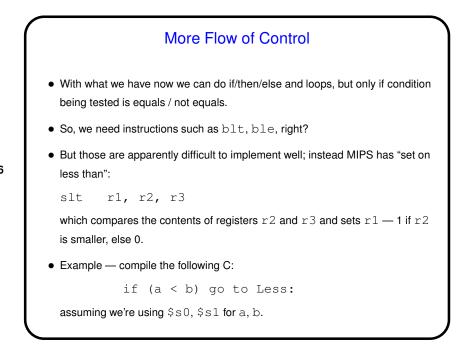


Do we have	e enough to do (some kinds of) loops? Yes — example:
	<pre>g = g + A[i]; i = i + j; if (i != h) goto Loop:</pre>
assuming address of	we're using \$s1 through \$s4 for <code>g, h, i, j, and \$s5</code> for th <sup>:</sup> A.
(This time	we'll use sll rather than two adds to multiply i by 4.)

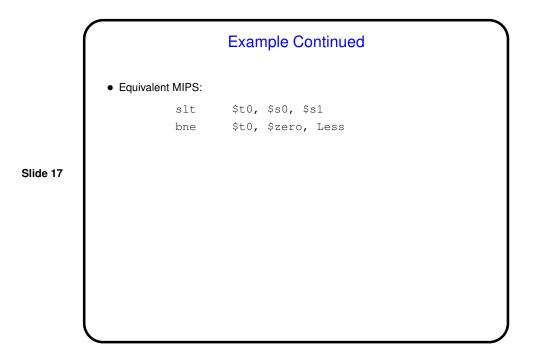
		Loops — Example Continued							
	<ul> <li>Result</li> </ul>								
	Loop:	sll	\$t1,	\$s3,	2	#	\$t1 <- 4*i		
		add	\$t1,	\$t1,	\$s5	#	\$t1 <- & of A[i]		
		lw	\$t0,	0(\$t	1)	#	\$t0 <- A[i]		
		add	\$s1,	\$s1,	\$t0	#	g = h + A[i]		
Slide 13		add	\$s3,	\$s3,	\$s4	#	i = i + j		
		bne	\$s3,	\$s2,	Loop	#	if (i!=h) goto Loop		

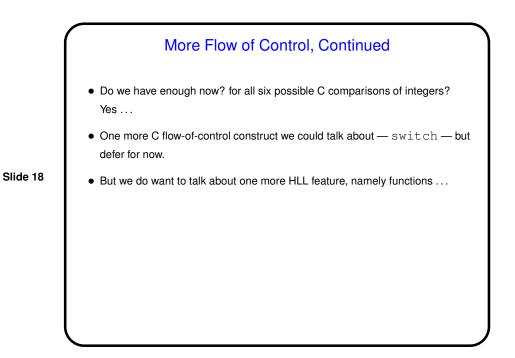


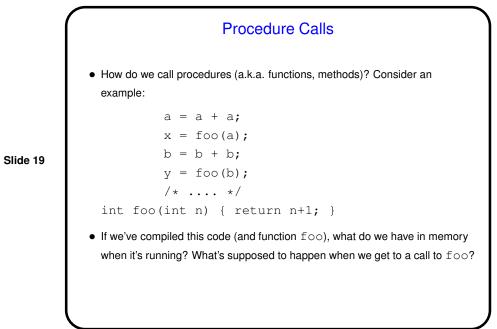
	•		2			`	d assuming \$s0 ha e i, j, and k):
Loop:							
# if	(A[i] != }	k) got	o End	:			
	sll	\$t0,	\$s1,	2	#	i *	4
	add	\$t0,	\$s0,	\$t1	#	&A [	i]
	lw	\$t0,	0(\$t	1)	#	A[i	]
	bne	\$t0,	\$s3,	End			
# i	= i + j						
	add	\$s1,	\$s1,	\$s2			
# g	oto Loop:						
	j	Loop					



Slide 16







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.

How to do all this?

