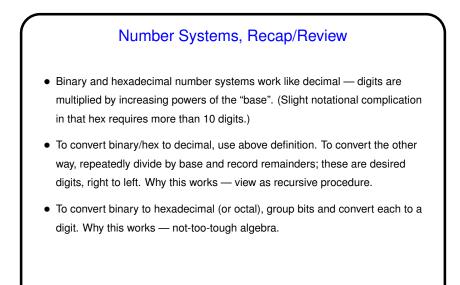


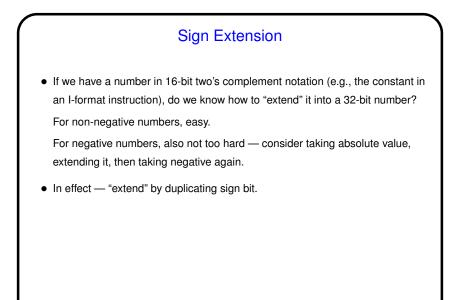
Slide 2



Representing Integers, Review/Recap
Representing non-negative integers is easy — convert to binary and pad on the left with zeros.
What about negative integers? "Two's complement" notation — makes arithmetic simpler, as we'll see. Idea loosely based on car-odometer analogy — if we have *n* bits, number "after" all ones is all zeros. We then decide to use half the possible values (the ones starting with one) to represent negative numbers.
How to get two's complement representation of -*x*? Notice that if we have *n* bits, adding 2ⁿ to *x* gives us *x* again. This leads to an easy way to compute -*x*: Compute 2ⁿ - *x*, and notice that 2ⁿ - *x* = (2ⁿ - 1) - *x* + 1 which is very easy to compute. (Try some examples.)

Slide 3

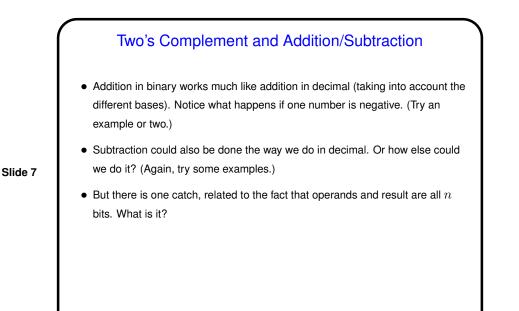
Slide 4



Slide 5

Slide 6

Signed Versus Unsigned
If we have n bits, we can use them to represent signed values in — what range?
Or we can use them to represent non-negative values only ("unsigned values") — then what range?
Many MIPS instructions have "unsigned" counterparts — addu, addiu, sltu, etc.
Example: Suppose we have
0000 0000 in \$t0
ffff fff2 in \$t1
What happens if we execute slt \$t2, \$t0, \$t1?
What happens if we execute sltu \$t2, \$t0, \$t1?
(Same bits, different interpretations!)



Addition/Subtraction and Overflow
If we're adding A and B, there are four cases to think about — both non-negative, etc. Two of them can give a wrong result because there aren't enough bits. Which ones? How can we tell the result is wrong?
MIPS signed arithmetic instructions detect overflow and "generate an exception" (more later).
MIPS unsigned arithmetic instructions ignore overflow. In a HLL, you may or may not want an exception on overflow. The compiler can choose signed if yes or unsigned if no.

Slide 8

