Slide 1

Administrivia

- Reminder: Homework 3 due Friday. (Remember that for two problems you should also e-mail me your code.)

Slide 2

Minute Essay From Last Lecture

- Convert 30_{10} to binary and then to hexadecimal.
- Convert -30_{10} to 16-bit two's complement notation; show in binary and hexadecimal.
- Convert $2a_{16}$ to decimal.

Slide 3

Number Systems, Recap/Review

- Binary and hexadecimal number systems work like decimal — digits are multiplied by increasing powers of the "base". (Slight notational complication in that hex requires more than 10 digits.)
- To convert binary/hex to decimal, use above definition. To convert the other way, repeatedly divide by base and record remainders; these are desired digits, right to left. Why this works — view as recursive procedure.
- To convert binary to hexadecimal (or octal), group bits and convert each to a digit. Why this works — not-too-tough algebra.

Slide 4

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 2^n to x gives us x again. This leads to an easy way to compute $-x$: Compute $2^n - x$, and notice that

  $$2^n - x \;=\; (2^n - 1) - x + 1$$

  which is very easy to compute. (Try some examples.)

Slide 5

Sign Extension

- If we have a number in 16-bit two's complement notation (e.g., the constant in an I-format instruction), do we know how to "extend" it into a 32-bit number?

  For non-negative numbers, easy.

  For negative numbers, also not too hard — consider taking absolute value, extending it, then taking negative again.
- In effect — "extend" by duplicating sign bit.

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!)

Two's Complement and Addition/Subtraction

Slide 7

- Addition in binary works much like addition in decimal (taking into account the different bases). Notice what happens if one number is negative. (Try an example or two.)
- Subtraction could also be done the way we do in decimal. Or how else could we do it? (Again, try some examples.)
- But there is one catch, related to the fact that operands and result are all n bits. What is it?

Addition/Subtraction and Overflow

Slide 8

- 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.

Minute Essay

Slide 9

- None — quiz.