Administrivia

- Reminder: Quiz 2 Thursday. Topics from chapter 2. Likely kinds of questions are “what does this code do?” and “translate this C into MIPS assembler”.
- Reminder: Homework 2 due Thursday. Late penalty waived if you can turn it in by class time Tuesday.

Slide 1

Linking — Example

- (Work through example starting on p. 127. Notice that we also need information about locations of “text segment” and “data segment”.)

Slide 2

Slide 3

Representing Data, Revisited

- To the hardware "it's all ones and zeros". But those ones and zeros can encode numbers (various forms), text, etc.
- Numbers in particular are interesting because we want to implement arithmetic operations.
- In theory you learned about integer representation and arithmetic in CSCI 1320. Review . . .

Slide 4

Binary Versus Decimal (Review?)

- In decimal (base 10) notation, each digit is multiplied by a power of 10. Same idea for binary (base 2), but using powers of 2.
- So, converting from binary to decimal is easy (if tedious), working from definition. Example?

Slide 5

Binary Versus Decimal, Continued

- Converting from decimal to binary? Repeatedly divide by 2 and record remainders . . .

  We could describe this as a recursive algorithm for computing $bits(n)$:

  - Base case is $n < 2$; trivial.
  - For recursive step, divide n by 2 to get quotient q and remainder r. Then $n = 2q + r$, and:

    The last bit of $bits(n)$ should be r.

    The remaining bits are $bits(q)$, left-shifted by 1.

Slide 6

Binary Versus Decimal, Continued

- Terminology: “Least significant” and “most significant” bits.
- Seems like there would be one obvious way to store the multiple bytes of one of these in memory, but no — “big endian” versus “little endian” (names based on *Gulliver’s Travels*).

Slide 7

Binary Versus Decimal, Continued

- Binary is useful for showing real internal state but not very compact. Decimal is compact but not so easy to convert to/from binary.
- We might notice — easy to convert to/from a base that's a power of 2. Hence the use of "octal" (base 8) and "hexadecimal" (base 16). For the latter, we need more than 10 digits, so we use "A" through "F".

  Examples?
- Notice that we can also convert directly to/from decimal, much as we did for binary.

Slide 8

Representing Integers (Review?)

- Representing non-negative integers is easy — convert to binary and pad on the left with zeros.
- What about negative integers?
- Could try using one bit for sign, but then you have +0 and -0, and there are other complications.
- Or . . . consider a car odometer — in effect, representable numbers form a circle, since adding 1 to largest number yields 0.

Slide 9

Representing Integers, Continued

- We could implement the car-odometer idea in binary, and then choose where to "cut the circle" (between smallest and largest):
  - Between 0 and all ones — unsigned integers.
  - Between largest number with 0 as the MSB and smallest number with 1 as MSB — "two's complement" signed integers.
- Notice that with the two's complement scheme, +1/-1 moves us "around the circle" — nothing special needed for negative numbers.
- 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 ...

  Examples?

Slide 10

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

  0x00000000 in `$t0`

  0xfffffff2 in `$t1`

  What happens if we execute `slt $t2, $t0, $t1`?

  What happens if we execute `sltu $t2, $t0, $t1`?

  (Same bits, different interpretations!)

Slide 11

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.
- (Notice that not all instructions that include a 16-bit constant do this.)

Slide 12

Two's Complement and Addition/Subtraction

- 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 13

- If adding two n-bit numbers, result can be too big to fit in n bits — "overflow".
- For unsigned numbers, how could we tell this had happened?
- How about for signed numbers?

Addition/Subtraction and Overflow, Continued

Slide 14

- Notice that we can't get overflow unless input operands have the same sign.
- If we add two positive numbers and get overflow, how can we tell this has happened? Does this always work?
- If we add two negative numbers and get overflow, how can we tell this has happened? Does this always work?

Slide 15

Addition/Subtraction and Overflow, Continued

- When we detect overflow, what do we do about it?
- From a HLL standpoint, we could ignore it, crash the program, set a flag, etc.
- To support various HLL choices, MIPS architecture includes two kinds of addition instructions:
  - Unsigned addition just ignores overflow.
  - Signed addition detects overflow and "generates an exception" (interrupt) — hardware branches to a fixed address ("exception handler"), usually containing operating system code to take appropriate action.

  This is why, if you look at MIPS assembler for C programs, the arithmetic is unsigned — C ignores overflow, so why bother to look for it.

Slide 16

Minute Essay

- Was everything today (about numbers/arithmetic) reasonably clear / covered in CS1?