Slide 1

Administrivia

- Reminder: Homework 3 due Monday. Turn in written problems in hardcopy, programming problems by e-mail.

Slide 2

Minute Essay From Last Lecture

- Many people had at least looked at Homework 3, but most had not gotten very far with it (yet!).

Slide 3

Sidebar(?): Parallel Execution and Synchronization

- A lot of commodity hardware these days features multiple processing units ("cores") sharing access to memory. One reason for this is that in theory we can make individual applications faster by splitting computation up among processing elements.
- Having processing elements share memory makes parallel programming easier in some ways but has risks ("race conditions"). Avoiding the risks requires some way to control access to shared variables (e.g., to implement notion of "lock").

Slide 4

Parallel Execution and Synchronization, Continued

- Most texts on operating systems discuss synchronization issues and present several solutions ("synchronization mechanisms"), some rather high-level and others not.

  (Why is this in O/S textbooks? because O/Ss typically have to manage "processes" executing concurrently, either truly at the same time or interleaved.)
- The most primitive can (with some simplifying assumptions) be implemented with no hardware support. But hardware support is very useful.

Slide 5

Sidebar: Why is Implementing a Lock Hard?

- It might seem like it would be straightforward to implement a lock — just have an integer variable, with value 0 meaning “unlocked” and anything else meaning “locked”. And then you “lock” by looping until the value is 0, then setting to nonzero, and “unlock” by setting back to 0.
- But this doesn’t work! (Why not?)

Slide 6

Instructions for Synchronization

- Key goal in designing hardware support for synchronization is to provide “atomic” (indivisible) load-and-store. This allows writing a low-level implementation of “lock” idea.
- Many architectures do this with a single instruction (e.g., “test and set” or “compare and swap”). Requires two accesses to memory so may be difficult to implement efficiently.
- MIPS approach — same idea, but using a pair of instructions, `ll` (“load linked”) and `sc` (“store conditional”). Example of use in textbook (p. 122). `sc` “succeeds” only if value at target location has not changed since previous `ll` — i.e., if one can regard the pair of instructions as forming a single atomic load/store.

Numbers and Arithmetic — Overview?

Slide 7

- Most architectures these days represent integers as fixed-length two's complement binary quantities.
- Most architectures these days represent real numbers using one or more of the formats laid out by the IEEE 754 standard. Based on a base-2 version of scientific notation, plus special values for zero, plus/minus "infinity", and "not a number" (NaN).

  (Worth noting, though, that historically there have been architectures that could represent fractional quantities using base-10 "fixed-point" notation, and this may be coming back.)

Binary Versus Decimal (Review?)

Slide 8

- 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.

Slide 9

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 10

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*).

Binary Versus Decimal, Continued

Slide 11

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

Representing Integers (Review?)

Slide 12

- 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 13

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 . . .

Slide 14

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 15

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 16

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?

Slide 17

Implementing Arithmetic — Preview

- In the next chapter we start talking about hardware design (though still at a somewhat abstract level).
- For now it may be useful to know that the low-level building blocks are entities that can evaluate Boolean expressions — very simple ones at the lowest level, and slightly more complex ones one level up.
- So for example we can implement addition by first making a “one-bit adder” that maps three inputs (two operands and carry-in) to two outputs (result and carry-out), and then chaining together 32 of them.
- Multiplication and division, however, may need to be more complex, involving multiple steps and control-flow logic. (Historical(?) aside: Early implementations may have just done the simple dumb thing — repeated additions or subtractions. (!))

Slide 18

Integer Addition, Subtraction, and Negative Values

- Recall(?) how addition works — right to left with carry. Carry-in to rightmost bit is (of course?) 0.
- Recall(?) also how finding the negative of a number works — “flip all the bits” and add 1.
- Notice then how if we can build an adder, we can more or less get subtraction “for free” — compute $a - b$ by adding a and bitwise negation of b with a carry-in to the rightmost bit of 1. (This is one reason two’s complement notation is attractive!)

  Further, textbook comments that `slt` could also be implemented using the same logic — to check for `a < b`, compute `a-b` and check for negative result (high-order bit on). Clever!(?)

Slide 19

Multiplication

- As with addition, first think through how we do this “by hand” in base 10. (Review terminology: In $a \times b$, call a the “multiplicand” and b the “multiplier”.) Example?
- We can do the same thing in base 2, but it’s simpler, no? computing the partial results is easier. This gives the textbook’s first algorithm, shown in figures 3.3 through 3.6. (Work through example.)

  Notice also that overflow could be a lot worse here — so normally we’ll compute a result twice as big as the inputs.
- What about signs? Algorithm works, if we extend the sign bit when we shift right.

Slide 20

Multiplication, Continued

- In MIPS architecture, 64-bit product / work area is kept two special-purpose registers (`lo` and `hi`). Two instructions needed to do a multiplication and get the result:

  ```
  mult rs1, rs2
  mflo rdest
  ```

  Assembler provides a “pseudoinstruction”:

  ```
  mul rdest, rs1, rs2
  ```
- Notice, however, that a “smart” compiler might turn some multiplications into shifts. (Which ones?)

Slide 21

Minute Essay

- None — quiz.