Slide 1

Administrivia

- (What happened to the planned video lecture from Monday? well, getting midterm grades in . . . I think what will make sense is to do one for Friday, and use it for detailed examples of some of today's topics.)
- (Next homework coming soon. I'll send e-mail.)
- Everything graded; I also e-mailed each of you a grade summary.

  If you didn't turn in one or more assignments, I'm willing to accept them late, for part credit, *as long as you haven't looked at a sample solution*.
- One student asked about the code I wrote in class to check for overflow in computing factorial. I've posted that to the course Web site under "sample programs".

Slide 2

Homework 4 Essays

- About working through some details of assembling and linking:

  Several people commented that it was tedious; several said it was somehow satisfying; some said both (!).

  Several people said the problem seemed daunting at first but video lecture helped (good to hear!).

Homework 3 Programming Problem Essays

Slide 3

- Several people commented that actually having to write programs that can be run helped them understand. That was my goal! Several commented that my examples helped. Good; they're meant to!

  Several people mentioned spending a lot of time on the problem. Not my goal, but debugging in SPIM is a huge pain.
- Several people mentioned that the meaninglessness of the code in the textbook problem was — unsatisfactory? Agreed. "My bad", maybe.
- One person said it was interesting how the assembler reserves one register for pseudoinstructions. Maybe, but — "principle of least surprise"

Homework 4 Programming Problem Essays

Slide 4

- Several people mentioned being tripped up by registers being reused when making a recursive call. "Indeed"?
- Several people said this one also helped them understand better, particularly about how procedure calls work. Several said it was not particularly tough given the factorial example to work from. That was my intent! Others said debugging was hard/painful. Agreed.
- Several people said they enjoyed the assignment. (Good to hear!)

Slide 5

Numbers and Arithmetic — Overview

- Most architectures these days represent integers as fixed-length two's complement binary quantities.

  (But that there are/were architectures that support variable-length "packed decimal", with each byte storing representations of two base-10 digits.)

- 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).

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

Slide 6

Numbers and Arithmetic — Overview, Continued

- Arithmetic can (in principle anyway) be done using same techniques taught to grade-school children.

  (Well, I hope still taught? Fans of classic science fiction may know Asimov short story "The Feeling of Power" (1958?), which posits a world in which no humans can do simple arithmetic without a computer. But he didn't predict how pervasive and affordable computers would become!)

Slide 7

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.

  Brief example:

  $1011_2 = (8+2+1)_{10} = 11_{10}$

Slide 8

Binary Versus Decimal (Review), Continued

- Converting from decimal to binary? Repeatedly divide by 2 and record remainders . . .
- Why does this work? 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:

    Last bit of $bits(n)$ should be r.

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

Other Number Bases (Review)

Slide 9

- Binary is useful for showing real internal state but not very compact. Decimal is compact but not so easy to convert to/from binary.
- Easy to convert binary 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 to make the idea of positional notation work (tangent — very powerful idea! compare to Roman numerals) we use "A" through "F" (uppercase or lowercase).

  Conversion is based on some simple if tedious algebra: Group bits, right to left, in groups of 3 (for octal) or 4 (for hexadecimal), and factor out a power of 8 or 16 from each group.
- Note that we can also convert directly to/from decimal, much as we did for binary.

Binary Versus Decimal (Review?), Continued

Slide 10

- 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 from *Gulliver's Travels*).

  (Sample program `show-int.c` shows which one x86 apparently uses.)

Slide 11

Representing Integers (Review)

- Representing non-negative integers is straightforward: 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 analogy of a car odometer: Representable numbers form a circle, since adding 1 to largest number yields 0.

Slide 12

Representing Integers (Review), Continued

- 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.
- Note: With this scheme +1/-1 moves us "around the circle" — nothing special needed for negative numbers.

Slide 13

Representing Integers (Review), Continued

- Note: If we have n bits, adding 2^n to x gives us x again. Leads to an easy way to compute $-x$: Compute $2^n - x$, and note that

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

  which is very easy to compute . . .
- (This is the familiar(?) method of “flipping the bits” and adding 1. Not magic!)

Slide 14

Signed Versus Unsigned

- If we have n bits, can use them to represent signed values in. (What range?)

  Or can use them to represent non-negative values only (“unsigned values”). (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 (Review?)

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

Slide 16

Two's Complement and Addition/Subtraction (Review)

- Addition in binary works much like addition in decimal (taking into account the different bases). Note what happens if one number is negative.
- Subtraction could also be done the way we do in decimal. But could also compute $a-b$ as $a+(-b)$, which makes for simpler hardware (more about this soon).

Slide 17

Integer Addition/Subtraction and Overflow

- 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?

Slide 18

Addition/Subtraction and Overflow, Continued

- Note 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?
- If we add two negative numbers and get overflow, how can we tell this has happened?
- (Figure 3.8 in textbook summarizes.)

Slide 19

Addition/Subtraction and Overflow, Continued

- When we detect overflow, what do we do about it?
- From a HLL standpoint: 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 fixed address ("exception handler"), usually containing operating-system code to take appropriate action.

Slide 20

Addition/Subtraction and Overflow, Continued

- C ignores overflow (not sure why!). So a real C compiler for MIPS would use unsigned arithmetic.
- Examples in the textbook don't do this, perhaps to keep things simpler. SPIM also apparently ignores overflow.

Slide 21

Implementing Arithmetic — Preview

- In next chapter, start talking about hardware design (though still at a somewhat abstract level).
- For now, may be useful to know that the low-level building blocks are entities that can evaluate Boolean expressions(!).
- So for example, 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. (Figures B.5.2, B.5.7.)
- Multiplication and division, however, may need to be more complex, involving multiple steps and control-flow logic.

Slide 22

Multiplication

- (First discuss simple "humans can understand this" / proof of concept approach.)
- As with addition, first think through how we do this "by hand" in base 10. (Example, briefly.)
- Can do the same thing in base 2, but it's simpler, no? computing the partial results is easier. This gives textbook's first algorithm, shown in figures 3.3 through 3.6. (Example another time.)

  Note also: Overflow could be a lot more here, so normally compute a result twice as big as the inputs.

Slide 23

Multiplication, Continued

- Approach just discussed works and is implementable, but it's slow.
- Can do better by computing partial products in parallel and then combining them in a way that also takes advantage of obvious(?) opportunity for parallelism. Impractical when chips were less complex; became feasible when hardware designers had more transistors to work with!

  (A few more details in textbook, if you're curious. Reasonable summary in Figure 3.7.)

Slide 24

Multiplication, Continued

- In MIPS architecture, 64-bit product / work area kept in 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
  ```

- Note, however, that a "smart" compiler might turn some multiplications into shifts. (Which ones?)

Slide 25

Division

- (Again, first discuss simple "humans can understand this" / proof of concept approach.)
- As with other arithmetic, first think through how we do this "by hand" in base 10. (Example, briefly.)
- Can do the same thing in base 2; this gives the algorithm shown in textbook figures 3.8 through 3.10. (Example another time.)

Slide 26

Division, Continued

- Here too, approach works but is slow. Speeding it up . . .
- Not as simple as with multiplication (is it apparent why?). Textbook says current hardware can still take some advantage of parallelism by computing some things speculatively. More in textbook if you're curious!

Slide 27

Division, Continued

- In MIPS architecture, 64-bit work area for quotient and remainder kept in same two special-purpose registers used for multiplication (`lo` and `hi`). After division, quotient in `lo` and remainder in `hi`. Two (or more) instructions needed to do a division and get result:

```
div rs1, rs2
mflo rq
mfhi rr
```

Assembler provides a “pseudoinstruction”:

```
div rdest, rs1, rs2
```

- Note, however, that a “smart” compiler might turn some divisions into shifts. (Which ones?)

Slide 28

Minute Essay

- How did the exam compare to your expectations? with regard to length, difficulty, topics . . .
- (And if you feel inclined to tell me how you spent your spring break?)