Slide 1

Administrivia

- Reminder: Homework 4 due Monday.
- Review sheet for Exam 1 (possibly a better name than “midterm” because the second exam won’t be a comprehensive final) on the Web. I will try to say more about what might be on the exam Monday.
- (Reminder: Sample solutions to quizzes online. Sample solutions to written problems distributed in hardcopy, with graded work. I will make available sample solutions to programming problems in hardcopy too. I don’t put them online because after all many of them are problems from a published textbook.)

Slide 2

Minute Essay From Last Lecture

- Most (though not all!) found writing actual programs at least a little educational. Some said it was even a little fun, if tedious (true enough).
- One person mentioned a stumbling block that’s interesting and sort of important . . .

Data Alignment

- Like many (but not all) architectures, MIPS load/store instructions require that the address be appropriately "aligned" — words on word boundaries, etc.
- Assembly directives such as `.word` do the right alignment (skipping space if necessary — e.g, a `.word` after a string). `.space`, however, does not. Can use `.align` to force alignment.

Slide 3

Data Representation and Arithmetic — Review

- (The next few slides are ones I used in a previous year, and they're sufficiently better than what I did this year that I'm going to include them here but go through them quickly.)
- (The last slide, though, about floating-point formats, makes a point I should have mentioned.)

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.

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

Slide 7

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 8

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

Slide 9

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 10

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 11

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 12

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 13

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 14

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 15

Addition/Subtraction and Overflow, Continued

- 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 16

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 17

Representing Real (Non-Integer) Numbers

- Approach is based on a binary version of “scientific notation”:

  In base 10, we can write numbers in the form $+/-x.yyyy \times 10^z$.

  E.g., $428 = 4.28 \times 10^2$, or $-.0012 = -1.2 \times 10^{-3}$.
- We can do the same thing in base 2. Examples:

  $32 = 1.0_2 \times 2^5$

  $-3 = -1.1_2 \times 2^1$

  $1/2 = 1.0_2 \times 2^{-1}$

  $3/8 = 1.1_2 \times 2^{-2}$
- This is “floating point” (as opposed to “fixed point”, which would allow for non-integers but wouldn’t allow as much flexibility — wide range, all with reasonable precision).

Slide 18

Representing Real Numbers, Continued

- In base 10, we can completely specify a number by giving its sign, a number in the range $0 \leq x < 10$ (the “significand” or “mantissa”), and the exponent for 10. Same idea applies in base 2.
- So, most/all “floating-point formats” have a bit for the sign, some bits for the significand, and some bits for the exponent. Different choices are possible, even with the same total number of bits; (at least) one architecture (VAX) even supported more than one format with the same number of bits(!).
- With integers, number of bits limits the range of numbers that can be represented. With “floating-point” numbers, two limiting factors — number of bits for the significand (which limits what?), and number of bits for the exponent (which limits what?).

  (Does this suggest why the VAX designers offered two formats?)

Slide 19

Floating Point in MIPS — A Little More

- (Example of program using floating-point instructions.)
- Some of the instruction names include `c1`. Short for “coprocessor 1”. What’s that? well, as textbook mentions, once upon a time chips for PC-class machines didn’t have enough transistors to implement floating-point arithmetic, so if it was included in the hardware at all, it was as a separate chip (“coprocessor”). This may also explain why there are distinct floating-point registers. Now a thing of the past, but the name stuck.
- “If at all”? was it not possible on machines without floating-point hardware to do floating-point arithmetic?

Slide 20

Floating Point in MIPS — A Little More

- (Can you not do floating-point arithmetic without hardware support?) Sure you can — in software. (Eek! slow but if packaged in libraries better than nothing.)

Slide 21

A Little About Circuit Design

- Goal of Chapter 4 — sketch design of a (hardware) implementation of MIPS architecture in terms of some simple building blocks (AND and OR gates, inverters).
- Key components of the design (Figures 4.1 and 4.2):
  - Something to implement memory.
  - Something to implement instructions: “ALU” (arithmetic/logic unit).
  - Something to implement registers: “register file”.
  - Something to implement fetch/decode/execute cycle: “control logic”.

  The first three together make up the “data path”. Analogy — it’s a puppet, with “control” pulling its strings.

Slide 22

Minute Essay

- None – quiz.