Slide 1

Administrivia

- Reminder: Homework 3 due today. Written problems in hardcopy by 5pm (or so). Programming problems by e-mail by 11:59pm.

  Don't forget the "honor code statement" — the Honor Code pledge (or just "pledged"), and whether you worked with anyone else. For programming problems, put it in the source code.

- Homework 4 on the Web. Due in a week.

Slide 2

Conditional Execution, Revisited

- We've done at least one example of compiling an if/else, and there are others in the textbook.
- Surprisingly few people, however, were able to do this correctly on the quiz:

  Most people didn't seem to realize that after the code for the "if" part, you need an explicit "jump" to skip the "else" part. If you think about it a minute, it should be obvious why — how else can the processor know to skip?

Slide 3

Integer Arithmetic — Recap/Review

- Addition is more or less straightforward: Build single-bit "add" circuit with carry-in and carry-out and chain 32 (or whatever) of them together. Can basically use this same circuit for subtraction and even for `slt`.
- Multiplication much more complicated, but based on how it can be done with pencil and paper, but keeping a "running total" to hold sum of partial products so far. "Real" MIPS instruction to multiply puts result in two special-purpose registers `lo` and `hi`, and you can then move values into general-purpose registers.

Slide 4

Division

- As with other arithmetic, first think through how we do this "by hand" in base 10. (Review terminology: We divide "dividend" a by "divisor" b to produce quotient q and remainder r, where $a = bq + r$ and $0 \leq |r| < b$.) Example?

  We can do the same thing in base 2; this gives the algorithm shown in textbook figures 3.8 through 3.10. (Work through example?)
- What about signs? Simplest solution is (they say!) to perform division on non-negative numbers and then fix up signs of the result if need be.

Slide 5

Division, Continued

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

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

  Assembler provides a "pseudoinstruction":

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

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

Slide 6

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 7

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 8

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.

  So a real C compiler for MIPS would use unsigned arithmetic — C ignores overflow, so why bother to look for it. Examples in the textbook don't do this, perhaps to keep things simpler.

Slide 9

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 10

Representing Real Numbers, Continued

- In base 10, we can completely specify a nonzero number by giving its sign, a number in the range $1 \leq x < 10$ (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?)

Floating-Point, Continued

Slide 11

- Most architectures these days use one or more of the floating-point formats defined by the IEEE 754 standard. MIPS uses two, 32-bit single precision and 64-bit double precision.
- (Work through example, checking result with `show-float` program from "sample programs" page.)

Floating-Point, Continued

Slide 12

- Recall also that this way of representing real numbers means they aren't quite the real numbers of math.
- (Review "floating point is strange" examples from CSCI 1120.)

Slide 13

Floating-Point, Continued

- Arithmetic on floating-point values is, maybe no surprise, a bit complicated.
- Textbook shows algorithms (in flowchart form). Probably useful/interesting to skim, but we won't discuss.

Slide 14

Floating Point in MIPS Architecture

- Architecture defines 32 floating-point registers (`$f0` through `$f31`), used singly for single-precision, in pairs for double-precision.
- Instruction set includes:
  - Arithmetic instructions:
    `add.s`, `sub.s`, `mul.s`, `div.s`; `add.d`, `sub.d`, `mul.d`, `div.d`
  - Load/store instructions (single-precision):
    `lwc1`; `swc1`
  - Comparisons:
    `c.eq.s`, `c.lt.s`, etc.; `c.eq.d`, `c.lt.d`, etc.
    These set a bit true/false, which can be used by `bc1t`, `bc1f`.
- (Example program(s) next time.)

Floating Point in MIPS, Continued

Slide 15

- 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?

Floating Point in MIPS, Continued

Slide 16

- (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.)

Minute Essay

- Anything noteworthy about Homework 3? For most of you the programming part was your first try at producing complete-for-SPIM MIPS programs; was it interesting, tedious, educational, . . . ?