Slide 1

Administrivia

- (This set of “slides” includes many not used in class. Meant as highlights of material we don’t really have time for.)

Slide 2

Numbers and Arithmetic — Review/Recap

- 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.)

Slide 3

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. This is (almost) enough to do addition and subtraction — just need to figure out about overflow.
- Multiplication and division, however, may need to be more complex, involving multiple steps and control-flow logic.

Slide 4

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, figure 3.5. (Work through example if time permits.)

  Notice also that overflow could be a lot worse here — so normally we'll compute a result twice as big as the inputs.

  (We can do better — later.)
- What about signs? Algorithm works, if we extend the sign bit when we shift right.

Slide 5

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 6

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 in figure 3.10. (Work through example if time permits.)

  (Here too we can do better — later).

- 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 7

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 8

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`.

Slide 9

Minute Essay

- The following C code

```
float f1 = 0.0;
for (int i = 0; i < 10; ++i) {
    f1 += 0.1;
}
float f2 = 1.0;
printf("f1 = %f, f2 = %f, f1=f2? %c\n",
        f1, f2, (f1==f2) ? 'y' : 'n');
```

  prints

  `f1 = 1.000000, f2 = 1.000000, f1=f2? n`

  which seems somewhat surprising, no? Why doesn't it think the two floating-point quantities are equal?

Slide 10

Minute Essay Answer

- The quantity 0.1 can't be represented exactly in binary floating-point, so it shouldn't be a complete surprise that the two quantities aren't exactly equal, though apparently the rounded values used in printing *are* equal.