Slide 1

Administrivia

- (None?)

Slide 2

Computer Multiplication, Revisited

- Terminology: In $a \times b$, call a the "multiplicand" and b the "multiplier".
- (Relatively) simple "humans can understand this" algorithm based on how humans do this without calculators shown in Figures 3.3 and 3.4 (Figure 3.5 is an optimized version). Note that Figure 3.3 also says how to initialize. What is all of this doing . . .

  ("ALU" here is something that can do simple arithmetic and logic operations.)

Slide 3

Multiplication — Big Picture(?)

- Set up work area to hold running total of partial products.
- Compute for each bit of multiplier its product with the multiplicand (i.e., a partial product). Easy since it's either the multiplicand or 0. Shift appropriate number of positions left and add to running total.

  Do this by repeatedly shifting multiplicand left and multiplier right. (Use additional work areas to do this.)
- (Work through example.)

Slide 4

Multiplication, One More Thing

- What about signs? Algorithm works, if we extend the sign bit when shifting right.

Slide 5

Computer Division, Revisited

- Terminology: Divide "dividend" a by "divisor" b to produce quotient q and remainder r, where $a{=}bq{+}r$ and $0{\leq}|r|{<}b$.
- (Relatively) simple "humans can understand this" algorithm loosely based on how humans do this without calculators shown in Figures 3.8 and 3.9. Note that Figure 3.8 also says how to initialize. What is all of this doing . . .

  ("ALU" here is something that can do simple arithmetic and logic operations.)

Slide 6

Division — Big Picture(?)

- Keep a sort of running total that reflects part of dividend we haven't divided yet ("running remainder"?). Also keep a shifted copy of divisor, initially shifted to match high-order bits, and a work area to build the quotient in.
- Repeatedly try subtracting shifted divisor from running remainder. If it "goes into", record a bit in the quotient and keep the result of the subtraction. If it doesn't, undo the subtraction. Either way, then shift the divisor to the right and the quotient left and repeat (fixed number of times).
- (Work through example.)

Slide 7

Division, One More Thing

- 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 8

Representing Non-Integer Numbers (Review)

- Usual approach is "floating-point", based on binary version of "scientific notation":

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

  E.g., $428{=}4.28{\times}10^2$, or $-.0012{=}{-}1.2{\times}10^{-3}$.

- 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).

Slide 9

Floating Point (Review)

- In base 10, 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 sets of limits: 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 10

Floating Point (Review), Continued

- Most architectures these days use one or more of the floating-point formats defined by the IEEE 754 standard. (Wikipedia article seems good. Many "who knew?" details!) Two things worth noting:
- Since first bit is (almost!) always 1, can omit it and get one extra bit. (Exception? special representation for that case.)
- Exponent is stored in "biased" form. Why? because then all exponents are non-negative, and comparisons are faster. (This speeds up sorting — perhaps why it's done this way?)
- (Work through example of conversion; use sample program `show-float.c` to check result.)

Slide 11

Floating Point (Review), Continued

- 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 12

Floating Point Arithmetic

- Arithmetic on floating-point values is, maybe no surprise, a bit complicated.
- Textbook shows algorithms in flowchart form. For now, skim the discussion of steps (Figures 3.14, etc.) but skip more-detailed explanation (Figures 3.15, etc.). (We may come back to the detailed versions later when they may make more sense.)

Slide 13

Minute Essay

- Now that you know what's involved in multiplication and division, does it make more sense to use shifts rather than multiplication when you can?
- It turns out that a smart compiler could also optimize multiplication by constants other than powers of 2. Any thoughts on how that might work?

Slide 14

Minute Essay Answer

- I hope so!
- If the compiler is smart enough, it could for example compile

```
n *= 5;
```

as, e.g.,

```
sll $t0, $s0, 2      # n*4
add $s0, $t0, $s0    # +n
```