Slide 1

Administrivia

- (Via e-mail.)

Slide 2

Numbers and Arithmetic — Overview

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

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

- Most current architectures these days represent real numbers using one or more of the formats laid out by IEEE 754 standard. Based on a base-2 version of scientific notation, plus a few special values (e.g., for zero).

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

Slide 3

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. He didn't predict how pervasive and affordable computers would become!)

Slide 4

Integers (Review)

- This might be a good time to review what you've presumably been taught already about representing integers:
- Fixed-size binary, using two's-complement idea for negative values.
- Addition straightforward, based on how humans (can) do it (without a computer).
- Subtraction could be done similarly, but turns out to be easier to compute $a - b$ as $a + (-b)$.

Slide 5

Arithmetic Overflow

- You might notice that that it any program that produces a large value may just quietly gives wrong results.
- Compare to what happens if you write an equivalent program in a high-level language . . .
- When result-in-process gets too big to fit into available space (32-bit register here), two options: Hardware can signal exception, or it can just drop high-order bits. If it drops bits, result can look negative, or it can just be wrong.

Slide 6

Arithmetic Overflow, Continued

- "Signal exception"? Yes. We'll talk more about this later, but possible to build hardware that detects overflow and does something. (Apparently SPIM doesn't do this.)
- But since many programming languages ignore overflow, often instructions have signed form that checks and unsigned form that doesn't (e.g., `addu` versus `add`). (*Note* that despite the possibly-misleading names, this is the only difference between the two, and note further that this is also true of `addi` versus `addiu` — maybe not what you'd expect!)

Slide 7

Arithmetic Overflow, Continued

- Really careful programmers put in their own checks for overflow. May actually be *easier* in assembly language: `mult` instruction generates 64-bit result in special-purpose registers . . .

Slide 8

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 — next lecture.)
- Multiplication and division, however, may need to be more complex, involving multiple steps and control-flow logic. (Historical(?) aside: Early implementations may have just done the simple dumb thing — repeated additions or subtractions. (!))

Slide 9

Multiplication

- Textbook first discusses 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.
- (Some) details shortly, though a quick look at Figures 3.3 and 3.4 should give you the idea.

  May also explain why I keep saying "but don't do full multiplication to multiply by 4!" ?

Slide 10

Multiplication, A Bit More

- 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. What is all of this doing . . .

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

Slide 11

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.)
- (Working through example omitted for reasons of time.)

Slide 12

Multiplication, Continued

- Approach works and is implementable, but is 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 13

Division

- Again, textbook first discusses 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, and again it's somewhat simpler.
- (Some) details shortly, though a quick look at Figures 3.8 and 3.9 should give you the idea.

  Here too it explains why bit-shifting is a better option when possible.

Slide 14

Division, A Bit More

- 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 based on how humans do this without calculators shown in Figures 3.8 and 3.9. What is all of this doing . . .

Slide 15

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).
- (Working through example omitted for reasons of time.)

Slide 16

Multiplication in MIPS

- 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
  ```

- (Look briefly at sample program that uses this to check for overflow.)
- Note again that a "smart" compiler might turn some multiplications into shifts. (Which ones?)

Slide 17

Division in MIPS

- 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
```

- Here too, a "smart" compiler might turn some divisions into shifts. (Which ones?)

Slide 18

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). (Note that some old architectures included that, though.)

Slide 19

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 20

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?)
- (Working through an example attractive but for reasons of time we won't.)

Slide 21

Floating Point Arithmetic

- Arithmetic on floating-point values is, maybe no surprise, a bit complicated.
- Textbook shows algorithms in flowchart form. Figures 3.14 and 3.16 show high-level view; more details, but we'll skip those (though read if you're interested!)

Slide 22

Floating Point in MIPS

- Architecture supports IEEE 754 "single" (32 bits) and "double" (64 bits).
- Architecture defines 32 floating-point registers (`$f0` through `$f31`), used singly for single-precision, in pairs for double-precision.

Slide 23

MIPS Floating-Point Instructions

- Arithmetic instructions (single-precision):

  Basics: `add.s`, `sub.s`, `mul.s`, `div.s`.

  Interesting extras: `abs.s`, `neg.s`, `sqrt.s`.

  All have double-precision counterparts (replace `.s` in name with `.d`).

- Load/store instructions:

  Single-precision `lwc1`, `swc1`.

  Double-precision `ldc1`, `sdc1`.

  Pseudoinstructions `li.s`, `li.d`.

Slide 24

MIPS Floating-Point Instructions, Continued

- Comparisons:

  `c.eq.s`, `c.lt.s`, etc., plus double-precision counterparts.

  These set a bit true/false, which can be used by `bc1t`, `bc1f`.

- Data copying:

  `mov.s`, `mov.d` to copy from one (pair of) register(s) to another.

  `mtc1`, `mfc1` to copy from general-purpose register to floating-point register and vice versa. *NOTE* that this just copies bits!

- Conversion between integer and floating point:

  `cvt.w.s`, `cvt.s.w`, and double-precision counterparts.

Slide 25

Floating Point in MIPS, Continued

- Some 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? Well . . . (Minute-essay question.)
- (Examples under "sample programs".)

Slide 26

Floating Point Versus Real Arithmetic, Revisited

- In CSCI 1120 I show two "floating point is strange" examples.
- Revisit those . . .

Slide 27

Minute Essay

- It turns out that a smart compiler could also optimize multiplication by constants other than powers of 2. For example, can you think of how it might be possible to multiply by 10 without using full multiplication?
- Is there a way to implement floating-point arithmetic without hardware support?

Slide 28

Minute Essay Answer

- A smart-enough compiler could compile

```
n *= 10;
```

as, e.g.,

```
sll $t0, $s0, 2     # n*4
add $t0, $t0, $s0   # n*4 + 1
sll $s0, $t0, 1     # 2(n*4 + 1)

\item By emulating it in software!  it wouldn't be fast, but it could work!
```