Slide 1

Administrivia

- Headline news is getting scary these days. Almost makes me wonder whether to just ignore it? though that doesn't seem like a great idea either. We in CS understand what "exponential growth" is, so we worry more?
- Agreed? Let's try a Zoom "show of hands . . .

Slide 2

More Administrivia

- Quiz 4 deferred.
- I'm realizing that we only have four weeks left and some important material. So I'll try to resist temptation to spend too much time on non-essentials — such as the fairly complete discussion of how to do multiplication and division with plausible hardware, and more details about floating-point. Summary of all that today, then move on.
- One thing that wil help us is a move from truly in-class exams to — whatever. I'm not quite sure about Exam 2, but I think it doesn't make sense to make it happen before the end of classes as planned. Possibly sort-of-in-class during scheduled exam period(s).

Slide 3

Endianness, Revisited

- Recall from last time that architectures can differ with regard to the order in which they store bytes of integer values.

  (Sample program `show-int.c` shows which one x86 apparently uses.)

Slide 4

Multiplication, Revisited

- (First discuss simple "humans can understand this" / proof of concept approach.)
- Terminology: In $a \times b$, call a the "multiplicand" and b the "multiplier".
- As with addition, first think through how we do this "by hand" in base 10. (Example last time.)
- (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 5

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 6

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 7

Division, Revisited

- (First discuss simple "humans can understand this" / proof of concept approach.)
- 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. (Example, briefly.) 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 8

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 9

Division, Continued

- Here too, approach works but is slow. Speeding it up ...
- Not as simple as with multiplication (is it apparent why?). Textbook says current hardware can still take some advantage of parallelism by computing some things speculatively. More in textbook if you're curious!

Slide 10

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 11

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 12

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 13

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 14

Floating Point in MIPS Architecture

- 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 15

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).

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

Slide 16

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 17

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.)

Slide 18

Example Programs

- Several examples on course Web site. For reasons of time I won't work through them in class, at least not now!

Slide 19

Designing a Processor — Overview

- 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). (Actually only a small subset of instructions, but enough to give you the idea?)
- May be useful to keep in mind the goal. We need something that can
  - Provide short-term storage of values (registers).
  - Perform arithmetic and logical operations on these values.
  - Provide longer-term storage of values (memory).
  - Transfer data between registers and memory.
  - Repeatedly fetch and execute instructions, allowing for both sequential execution and branching/jumps.

Slide 20

Circuit Design — Overview

- AND and OR gates implement Boolean-algebra functions of the same names; inverter implements “not”.
- Short executive-level summary of how this works in current technology next time I hope!

Slide 21

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 22

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
```