CSCI 210: Computer Architecture
Lecture 22: Floating Point

Stephen Checkoway

Oberlin College

Nov. 29, 2021

Slides from Cynthia Taylor

Announcements

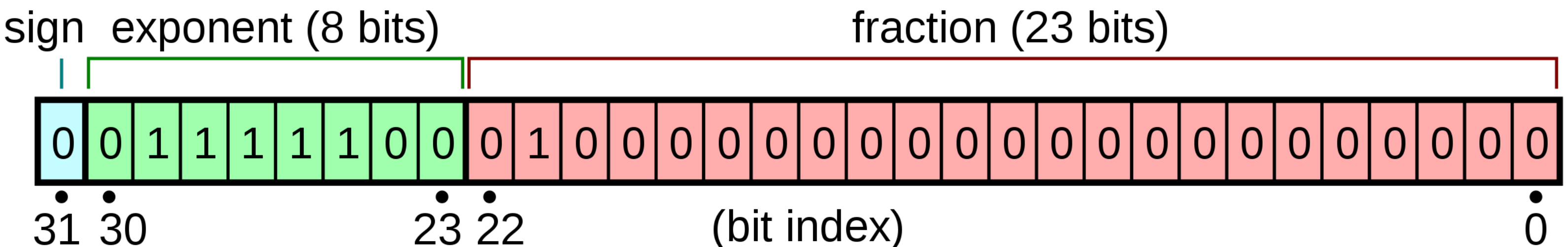
- Problem Set 7 due Friday
- Lab 6 due Sunday (it’ll be up tonight)
- Office Hours tomorrow 13:30 – 14:30

Review

- Unsigned 32-bit integers let us represent 0 to $2^{32} – 1$
- Signed 32-bit integers let us represent $– 2^{31}$ to $2^{31} – 1$
- 32-bit floating point numbers let us represent a wider range of values: larger, smaller, fractional

$$(-1)^s * 1.x * 2^e$$

- 1 bit for sign s (1 = negative, 0 = positive)
- 8 bits for exponent e
- 0 bits for implicit leading 1 (called the “hidden bit”)
- 23 bits for significand (without hidden bit)/fraction/~~mantissa~~ x



Want To Make Sorting Easy

- Can easily tell if number is positive or negative
  - Just check MSB bit

- Exponent is in higher magnitude bits than the fraction
  - Numbers with higher values will look bigger
  - 0 00000111 10000000000000000000000 = $1.1 * 2^7$
  - 0 00001000 10000000000000000000000 = $1.1 * 2^8$

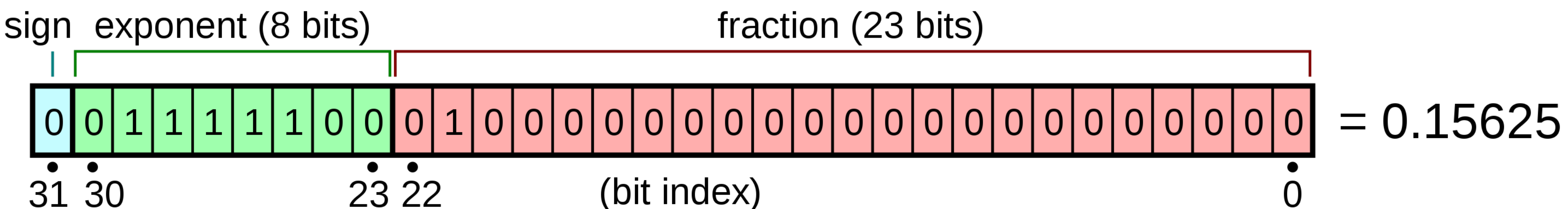
Problem with Two's Compliment

- 0 00000111 10000000000000000000000 = $1.1 * 2^{7}$
- 0 00001000 10000000000000000000000 = $1.1 * 2^{8}$
- 0 11111000 10000000000000000000000 = $1.1 * 2^{-8}$

- Solution: Get rid of negative exponents!
  - We can represent 2^8 = 256 numbers: normal exponents -126 to 127 and two special values for zero, infinity, (and NaN and subnormals)
  - Add 127 to value of exponent to encode it, subtract 127 to decode

$(-1)^s * 1.x * 2^e$

- 1 bit for sign s (1 = negative, 0 = positive)
- 8 bits for exponent e + 127
- 0 bits for implicit leading 1 (called the “hidden bit”)
- 23 bits for significand (without hidden bit)/fraction/~~mantissa~~ x



1.000000001 * 2^7 in Floating Point

A. 0 00000111 00000000100000000000000

B. 0 00000111 10000000010000000000000

C. 0 10000110 00000000100000000000000

D. 0 10000110 10000000010000000000000

E. None of the above

How Can We Represent 0 in Floating Point (as described so far)?

A. 0 00000000 00000000000000000000000

B. 0 01111111 00000000000000000000000

C. 1 00000000 00000000000000000000000

D. More than one of the above

E. We can’t represent 0

Special Cases

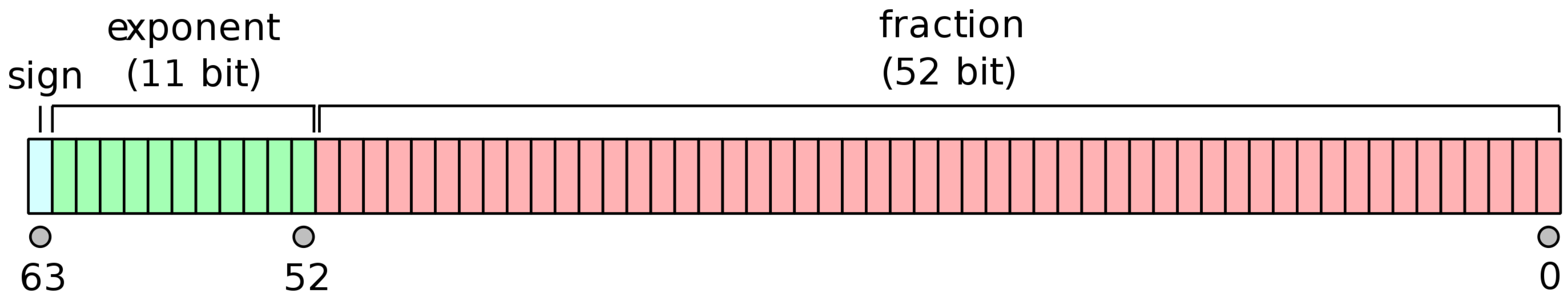
Object	Exponent	Significand
Zero	0	0
Subnormal	0	Nonzero
Infinity	255	0
NaN	255	Nonzero

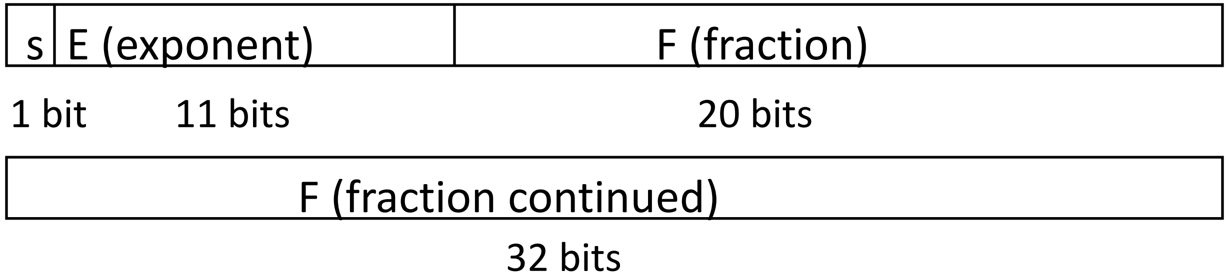
- Subnormal number: Numbers with magnitude smaller than 2^{-126}
  - They have an implicit leading 0 bit
- NaN: Not a Number. Results from 0/0, $0 * \infty$, $(+\infty) + (-\infty)$, etc.

Overflow/underflow

- Overflow happens when a positive exponent becomes too large to fit in the exponent field
- Underflow happens when a negative exponent becomes too large (in magnitude) to fit in the exponent field
- One way to reduce the chance of underflow or overflow is to offer another format that has a larger exponent field
  - Double precision – takes two MIPS words

Double precision in MIPS





Adding

- Add together $2.34 * 10^3$ and $4.56 * 10^5$
- Normalize so both have the larger exponent
  - $0.0234*10^5 + 4.56 * 10^5$
- Add significands taking sign of numbers into account
  - $4.5834 * 10^5$
- Normalize to a single leading digit
  - $4.5834 * 10^5$

$$1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$$

A. $0.001_2 \times 2^{-1}$

B. $1.111_2 \times 2^{-1}$

C. $1.011_2 \times 2^{-2}$

D. $1.000_2 \times 2^{-4}$

E. None of the above

What problems could we run into doing this in binary?

A. Added fraction could be longer than 23 bits

B. Normalized exponent could be greater than 127 or less than -126

C. Shifting fraction to match largest exponent could take more than 23 bits

D. More than one of the above

Floats in higher-level languages

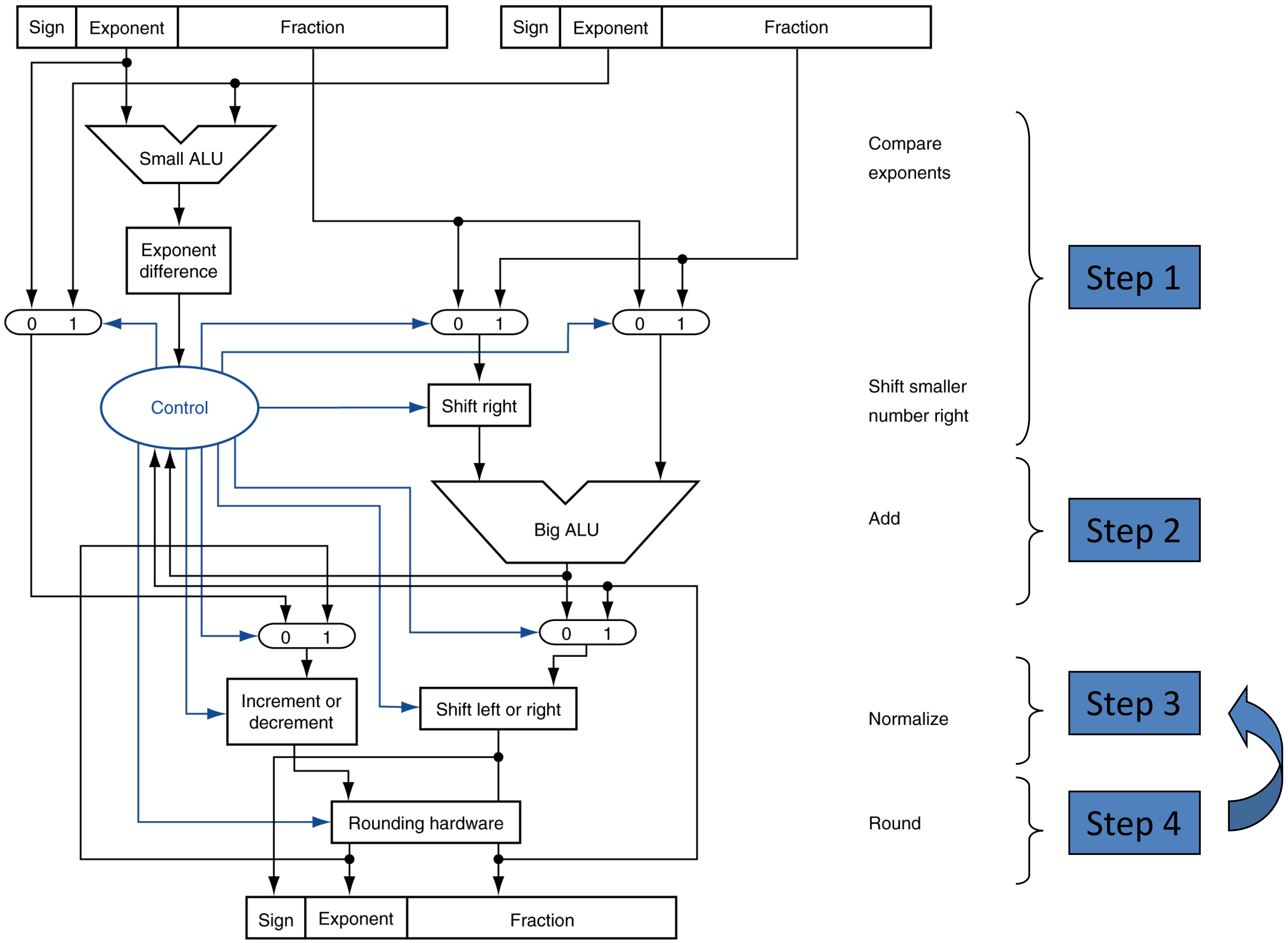
- C, Java: float, double
- JavaScript: numbers are always 64-bit double precision
- Rust: f32, f64

- Sometimes intermediate values (e.g., x*y in x*y + z) may be doubles (or larger types!) even when the inputs are all floats

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles

FP Adder Hardware



Multiplication

- Multiply 2.34 * 10^3 and 4.56 * 10^5
- Add together exponents
  - 10^8
- Multiply fractions (with appropriate signs)
  - 10.6704 * 10^8
- Normalize
  - 1.06704 * 10^9

$$1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$$

A. $-1.110_2 \times 2^{-1}$

B. $-1.110_2 \times 2^{-2}$

C. $-1.110_2 \times 2^{-3}$

D. $-1.110_2 \times 2^{1}$

What problems could we run into doing this in binary floating point?

A. Adding bias in exponent in twice

B. Shifted exponent could be greater than 127 or less than -126

C. Multiplied fraction could be longer than 23 bit

D. More than one of the above

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
- FP instructions operate only on FP registers
  - Programs generally don’t do integer ops on FP data, or vice versa
- FP load and store instructions
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`
  - Psuedoinstructions are easier to read: l.s, l.d, s.s, s.d

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic (operates on paired registers)
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`

Reading

- Next Lecture: Floating Point/Performance

- Problem Set 7