CSE 210: Computer Architecture
Lecture 22: Floating Point

Stephen Checkoway
Slides from Cynthia Taylor

CS History: IBM 704 Data-Processing Machine

Man and woman working with IBM type 704 electronic data processing machine used for making computations for aeronautical research. By NASA, Public Domain

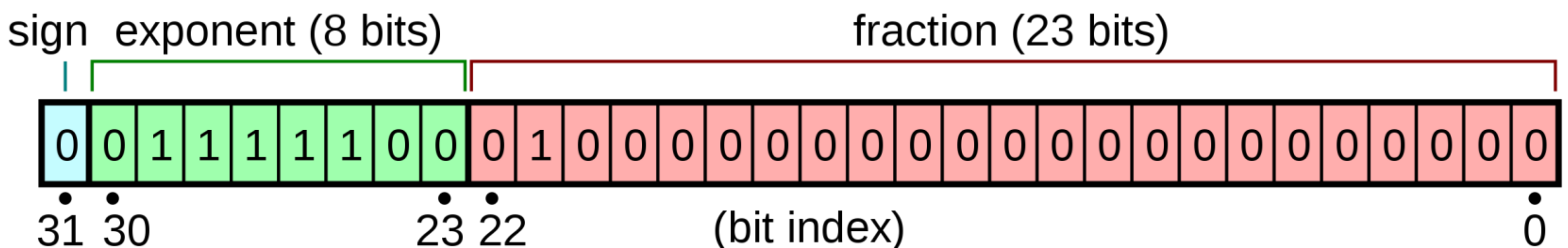
- First mass-produced computer with floating point arithmetic
- Introduced in 1954
- Had 36 bit words
- Floating point had
  - 1 sign bit
  - 8 bit exponent (biased by 127)
  - **27** bit fraction (no hidden bit)
- "pretty much the only computer that could handle complex math" at the time

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



1.001100101 * 2^7 as a single word

- 1.001100101 * 2^7 as a single word becomes
  - Sign =
  - Exponent =
  - Significand =

If we gave more bits to the exponent, and fewer to the fraction, we could represent

A. Fewer individual numbers

B. More individual numbers

C. Numbers with greater magnitude, but less precision

D. Numbers with smaller magnitude, but greater precision

Want To Make Comparisons 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 = $1.1 * 2^8$

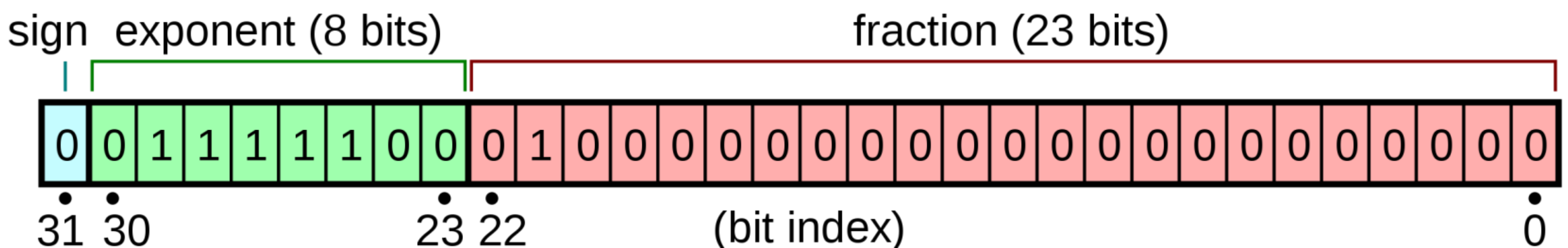
Problem with Two's Compliment

- 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 things like infinity
  - 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 x



Encode 1.000000001 * 2^7 in 32-bit Floating Point

A. 0 00000111 00000000100000000000000

B. 0 00000111 10000000100000000000000

C. 0 10000110 00000000100000000000000

D. 0 10000110 10000000100000000000000

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

	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 and an exponent of 2^{-126}
- NaN: Not a Number. Results from 0/0, 0 * ∞, (+∞) + (–∞) , etc.

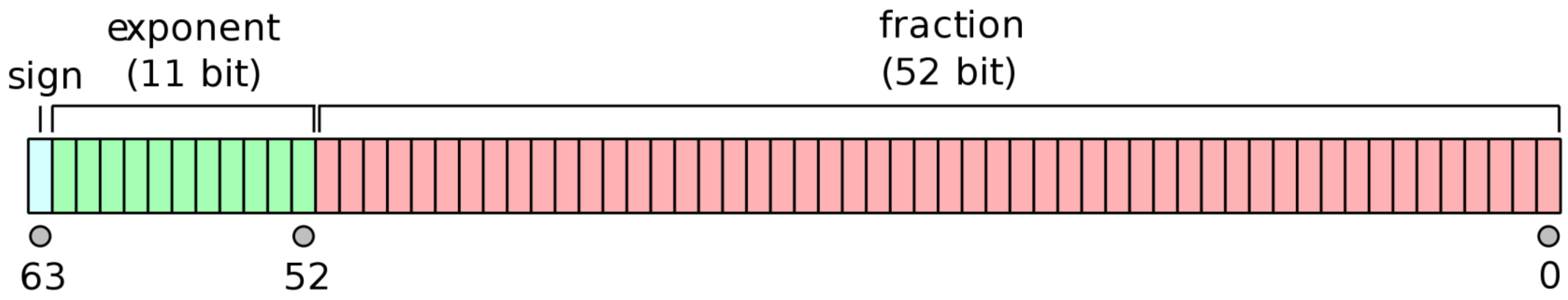
Overflow/underflow

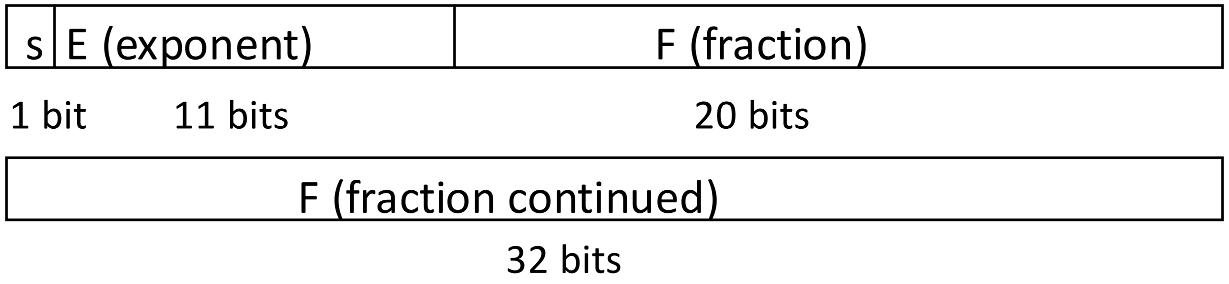
- Overflow happens when a positive exponent becomes too large to fit in the exponent field

- 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 32-bit words

Double precision in IEEE Floating Point





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

Which of these numbers does not exist in JavaScript?

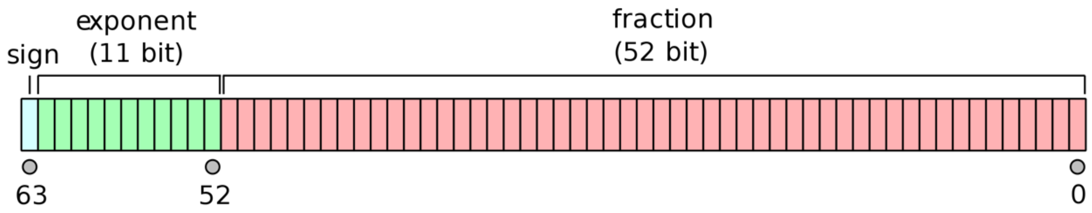
A. 9007199254740991

B. 9007199254740992

C. 9007199254740993

D. 9007199254740994

E. More than one of the above



Hint: 9007199254740992 is 2^{53}

There are always 2^{52} evenly spaced doubles between 2^n and 2^{n+1}. How many **floats** will there be between 2^n and 2^{n+1}?

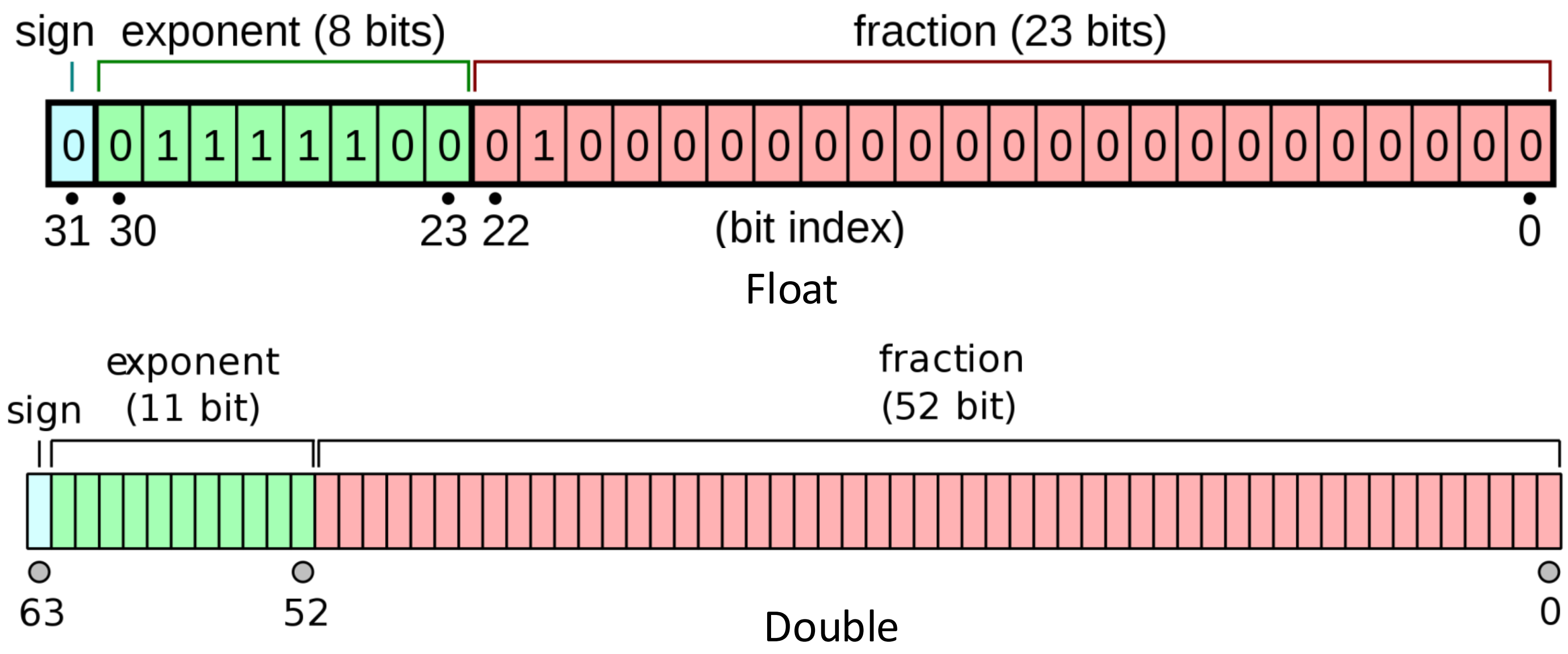
A. 2^8

B. 2^{23}

C. 2^{32}

D. 2^{52}

E. None of the above



Weird Float Tricks

- For floats of the same sign:
  - Adjacent floats have adjacent integer representations
  - Incrementing the integer representation of a float moves to the next representable float, moving away from zero
- This is specific to the IEEE 754 implementation of floating point!
- Want to play around with floats?
  - https://float.exposed/

Adding in floating point (assuming 4 fractional bits)

- Add together $1.1011 * 2^0$ and $1.0110 * 2^2$
- Normalize so both have the larger exponent
  - $.0110 * 2^2 + 1.0110 * 2^2$
- Add significands taking sign of numbers into account
  - $1.1100 * 2^2$
- Normalize to a single leading digit
  - $1.1100 * 2^2$

What problems could we run into doing this in hardware with 32-bit floats?

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. The inputs could be zero or the result could be zero
E. More than one of the above

Floating point addition algorithm

Input: two single-precision, floating point numbers x, and y

Output: x + y

1. If either x or y is 0, return the other one
2. Denormalize x or y to give them both the larger exponent
3. Add the significands (as integers; hidden bit + 23-bit fraction), taking sign into account
4. If the result is 0, return 0
5. Normalize the result by shifting the added significands left/right and increasing/decreasing the exponent
   Ex: $10011.101 * 2^{-1} = 1001.1101 * 2^{0} = 100.11101 * 2^{1}$

In Javascript, you perform the operation 9007199254740992 + 1. What is the result?

A. -9007199254740992

B. 9007199254740992

C. 9007199254740993

D. This will cause an error

E. None of the above

How many times will this loop run in python?

```
a = 1000
while a != 0:
   a -= 0.001
```

A. 1000 times
B. 100000 times
C. 1000000 times
D. It will run forever
E. None of the above

This will run forever

```
a = 1000
while a != 0:
    a -= 0.001
```

- a is never 0, instead it goes from 1.673494676862619e-08 to -0.0009999832650532314.
- Takeaway: Float equality is hard! Usually want to check within a small range

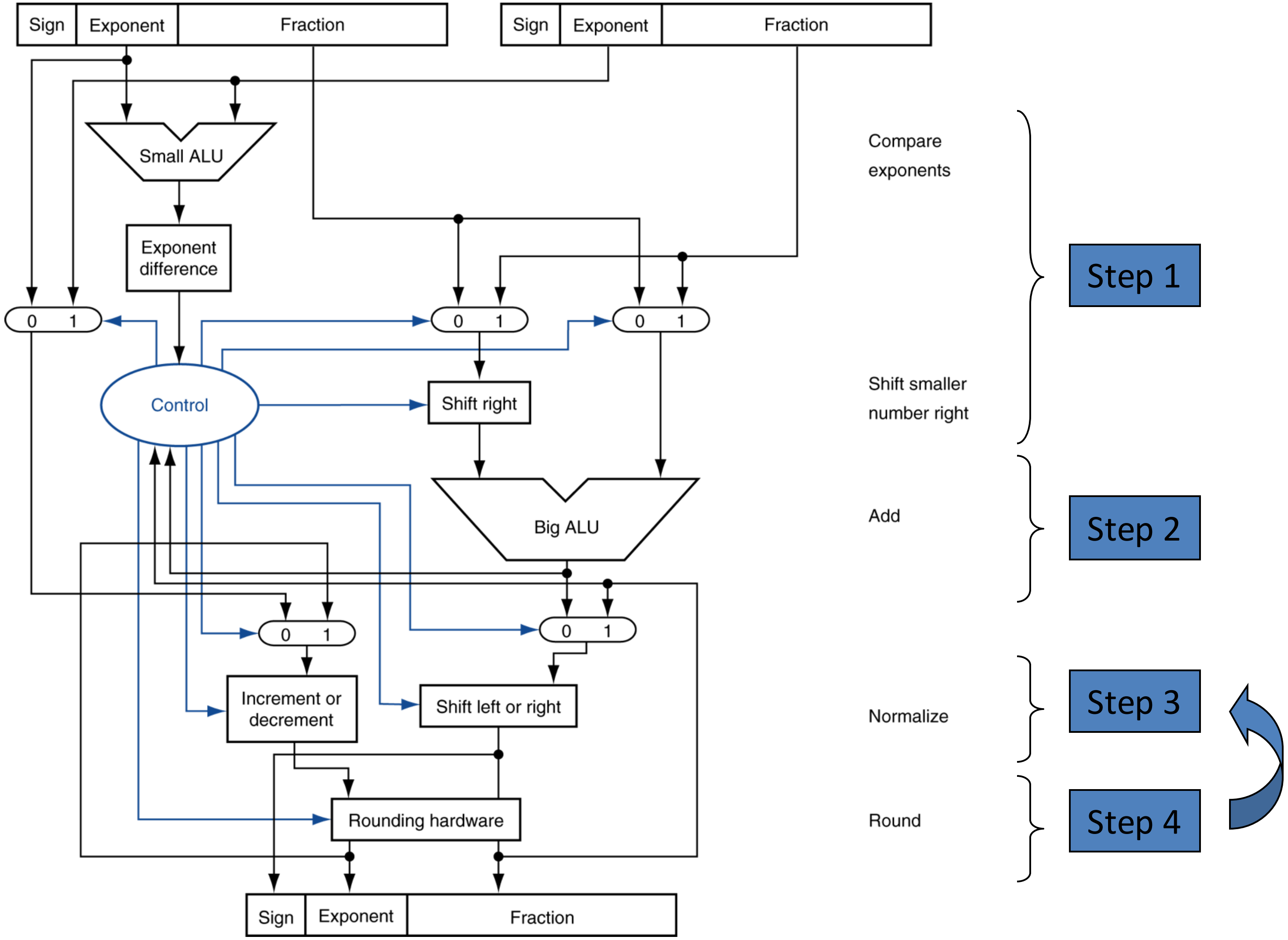
FP Adder Hardware

- Much more complex than integer adder

- Doing it in the general purpose ALU/CPU would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions

- FP adder usually takes several cycles

FP Adder Hardware



Reading

- Next lecture: Floating Point, addressing