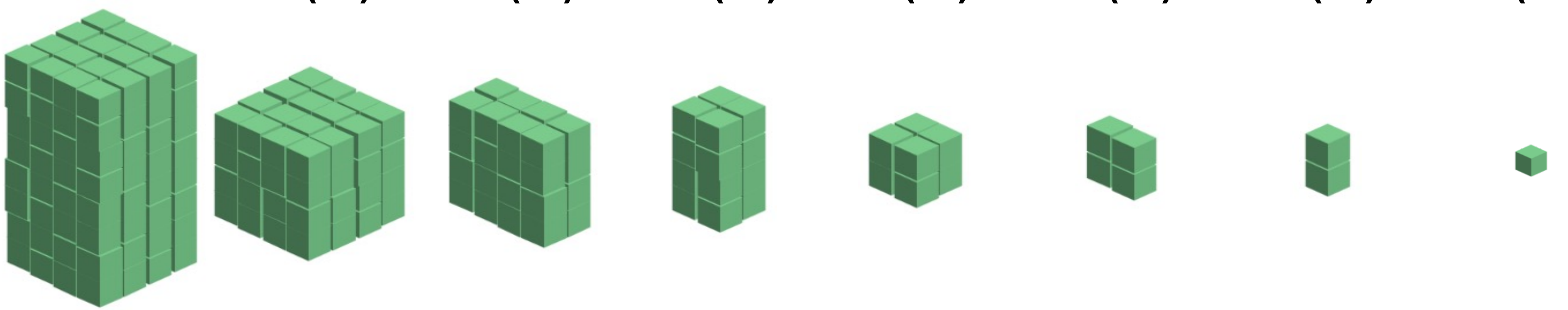
Lecture 4: Floats

CS 105

Representing Integers

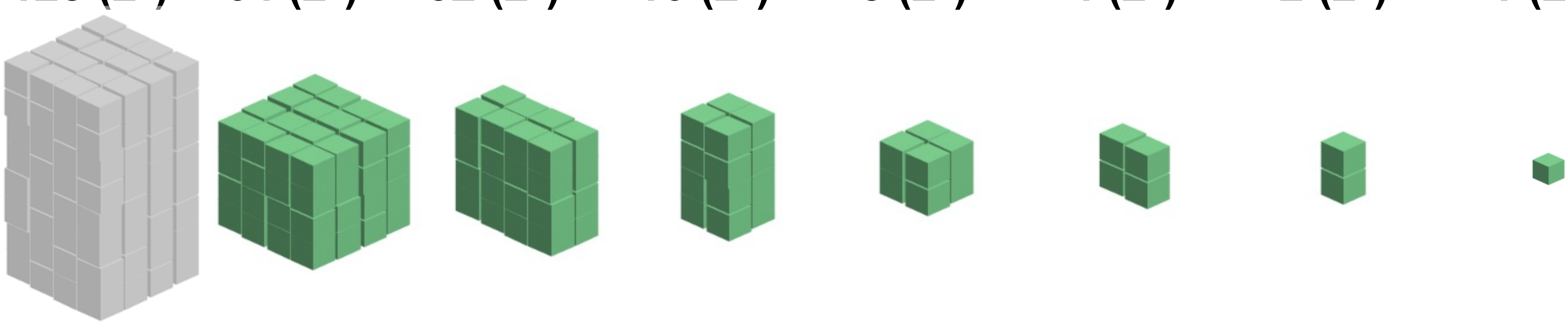
- unsigned:

128 (2^7) 64 (2^6) 32 (2^5) 16 (2^4) 8 (2^3) 4 (2^2) 2 (2^1) 1 (2^0)



- signed (two's complement):

-128 (2^7) 64 (2^6) 32 (2^5) 16 (2^4) 8 (2^3) 4 (2^2) 2 (2^1) 1 (2^0)



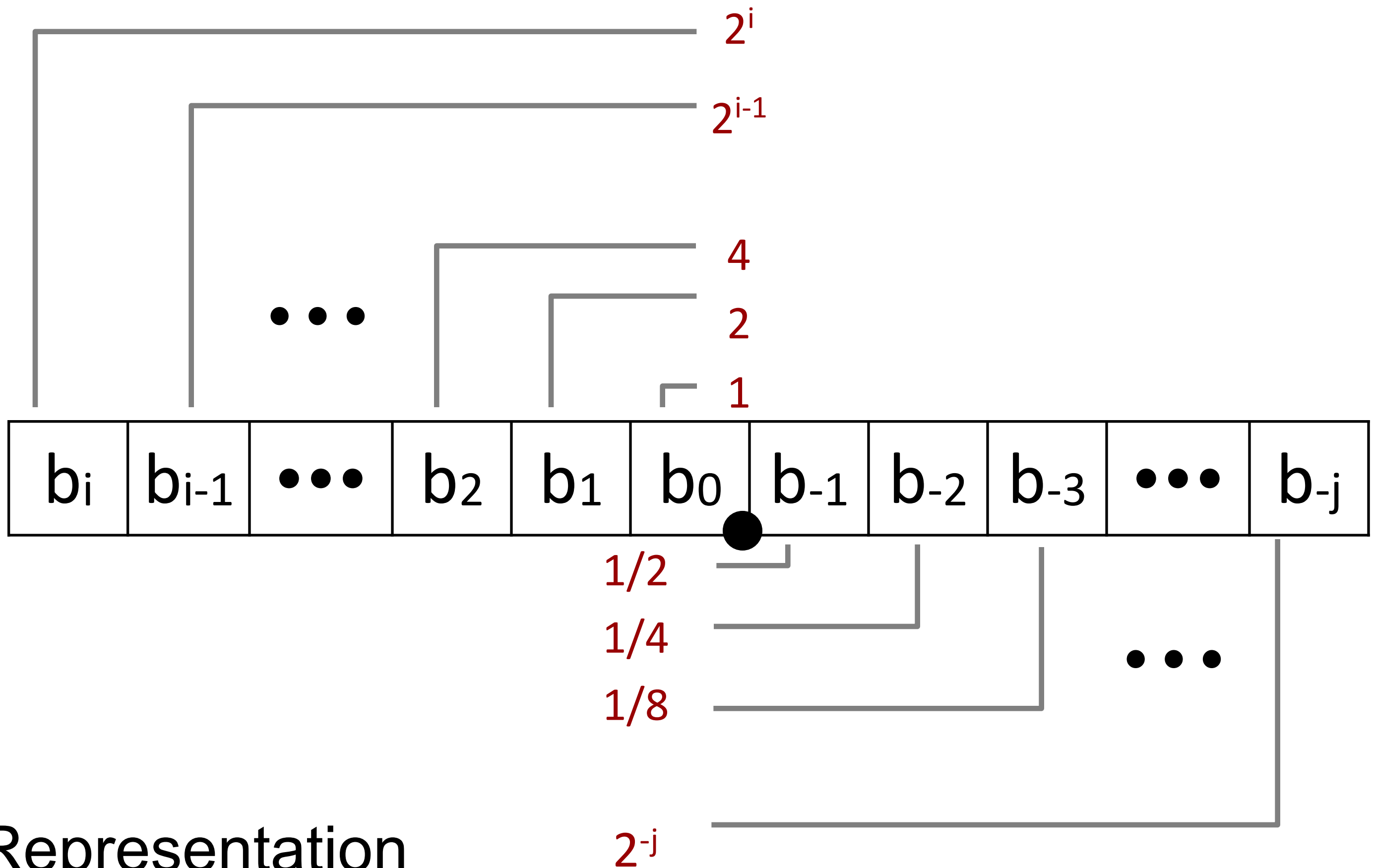
Note: to compute –x for a signed int x, flip all the bits, then add 1

$$x + \sim x = 11\ldots1 = -1, \text{ so } x + (\sim x + 1) = 0$$

Fractional binary numbers

- What is 1001.101_2?

Fractional Binary Numbers



- Representation
  - Bits to right of "binary point" represent fractional powers of 2
  - Represents rational number: $\sum_{k=-j}^{i}(b_k \cdot 2^k)$

Example: Fractional Binary Numbers

- What is 1001.101_2?

$$= \mathbf{8} + \mathbf{1} + \ \ldots$$

Example: Fractional Binary Numbers

- What is 1001.101_2?

$$= 8 + 1 + \frac{1}{2} + \frac{1}{8}$$

Example: Fractional Binary Numbers

- What is 1001.101_2?

$$= 8 + 1 + \frac{1}{2} + \frac{1}{8} = 9\ \frac{5}{8} = 9.625$$

- What is the binary representation of 13 9/16?

1101.1001

Exercise 1: Fractional Binary Numbers

- Translate the following fractional numbers to their binary representation
  - `5 3/4`
  - `2 7/8`
  - `1 7/16`

- Translate the following fractional binary numbers to their decimal representation
  - `000.01100`
  - `000.11000`
  - `001.10000`

Exercise 1: Fractional Binary Numbers

- Translate the following fractional numbers to their binary representation
  - 5 3/4 **101.11000**
  - 2 7/8 **010.11100**
  - 1 7/16 **001.01110**

What do you notice about shifting?

- Translate the following fractional binary numbers to their decimal representation
  - 000.01100 $= \frac{1}{4} + \frac{1}{8} = \frac{3}{8} = .375$
  - 000.11000 $= \frac{1}{2} + \frac{1}{4} = \frac{3}{4} = .75$
  - 001.10000 $= 1 + \frac{1}{2} = \frac{3}{2} = 1.5$

Representable Numbers

- Limitation #1
  - Can only exactly represent numbers of the form $x/2^k$
  - Other rational numbers have repeating bit representations
  - Value Representation
    - 1/3 `0.0101010101[01]`…2
    - 1/5 `0.001100110011[0011]`…2
    - 1/10 `0.0001100110011[0011]`…2
- Limitation #2
  - Just one setting of binary point within the *w* bits
  - Limited range of numbers (Very small values? Imprecise values?)

Floating Point Representation

- Numerical Form: $(-1)^s \cdot M \cdot 2^E$
  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0)
  - Exponent E weights value by power of two

Exercise 2: Floating Point Numbers

- For each of the following numbers, specify a binary fractional number M in [1.0,2.0) and a binary number E such that the number is equal to $M \cdot 2^E$
  - 5 3/4
  - 2 7/8
  - 1 1/2
  - 3/4

Exercise 2: Floating Point Numbers

- For each of the following numbers, specify a binary fractional number M in [1.0,2.0) and a binary number E such that the number is equal to $M \cdot 2^E$
  - 5 3/4 **`M = 1.0111 E = 2`**
  - 2 7/8 **`M = 1.0111 E = 1`**
  - 1 1/2 **`M = 1.1000 E = 0`**
  - 3/4 **`M = 1.1000 E = -1`**

Floating Point Representation

- Numerical Form: $(-1)^s \cdot M \cdot 2^E$
  - Sign bit s determines whether number is negative or positive
  - Significand M normally a fractional value in range [1.0,2.0)
  - Exponent E weights value by power of two

- Encoding:

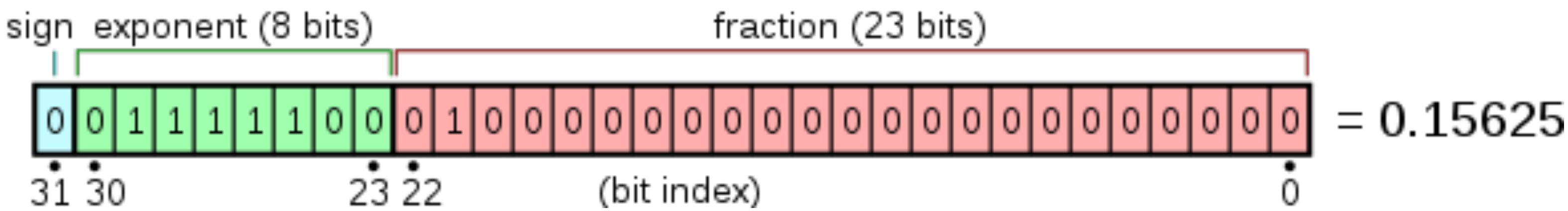
s	$\text{exp} = e_{k-1} \ldots e_1 e_0$	$\text{frac} = f_{n-1} \ldots f_1 f_0$

- s is sign bit s
- exp field encodes E (but is not equal to E)
  - normally $E = e_{k-1} \ldots e_1 e_0 - (2^{k-1} - 1)$ — **bias**
- frac field encodes M (but is not equal to M)
  - normally $M = 1.f_{n-1} \ldots f_1 f_0$

Float (32 bits):
- k = 8, n = 23
- bias = 127

Double (64 bits)
- k=11, n = 52
- bias = 1023

IEEE 754 single-precision binary floating-point format



$$(-1)^{b_{31}} \times 2^{(b_{30}b_{29}\ldots b_{23})_2 - 127} \times (1.b_{22}b_{21}\ldots b_0)_2$$

Exercise 3: Floating Point Representations

- What are the values of s, exp, and frac that correspond to the float representation of 5 3/4, assuming 1-bit s, 3-bit exp, and 4-bit frac?

s	$\text{exp} = e_{k-1} \ldots e_1 e_0$	$\text{frac} = f_{n-1} \ldots f_1 f_0$

  - $(-1)^s \cdot M \cdot 2^E$, M = 1.0111, E = 2
  - s is sign bit s
  - exp field encodes E (but is not equal to E)
    - normally $E = e_{k-1} \ldots e_1 e_0 - (2^{k-1} - 1)$
  - frac field encodes M (but is not equal to M)
    - normally $M = 1.f_{n-1} \ldots f_1 f_0$
- Under those assumptions, what is the full representation of 5 3/4 as a one-byte floating point value?

Exercise 3: Floating Point Representations

- What are the values of s, exp, and frac that correspond to the float representation of 5 3/4, assuming 1-bit s, 3-bit exp, and 4-bit frac?

s	$\text{exp} = e_{k-1} \ldots e_1 e_0$	$\text{frac} = f_{n-1} \ldots f_1 f_0$

  - $(-1)^s \cdot M \cdot 2^E$, M = 1.0111, E = 2
  - s is sign bit s
  - exp field encodes E (but is not equal to E)
    - normally $E = e_{k-1} \ldots e_1 e_0 - (2^{k-1} - 1)$
  - frac field encodes M (but is not equal to M)
    - normally $M = 1.f_{n-1} \ldots f_1 f_0$

s = 0

exp = 101

frac = 0111

- Under those assumptions, what is the full representation of 5 3/4 as a one-byte floating point value?

01010111 = 0x57

Example: Floats

- What fractional number is represented by the bytes 0x3ec00000? Assume big-endian order.

s	$\text{exp} = e_{k-1} \ldots e_1 e_0$	$\text{frac} = f_{n-1} \ldots f_1 f_0$

- s is sign bit s
- exp field encodes E (but is not equal to E)
  - normally $E = e_{k-1} \ldots e_1 e_0 - (2^{k-1} - 1)$
- frac field encodes M (but is not equal to M)
  - normally $M = 1.f_{n-1} \ldots f_1 f_0$

Float (32 bits):
- k = 8, n = 23
- bias = 127

$$(-1)^s \cdot M \cdot 2^E$$

0	011 1110 1	100 0000 0000 0000 0000 0000

s=0 exp=125 frac = $10000000000000000000000_2$

s=0 E = -2 M = $1.10000000000000000000000_2 = 1.5_{10}$

$$(-1)^0 \cdot 1.5_{10} \cdot 2^{-2} = 1 \cdot \frac{3}{2} \cdot \frac{1}{4} = \frac{3}{8} = \mathbf{.375_{10}}$$

$$(-1)^0 \cdot 1.1_2 \cdot 2^{-2} = .011_2 = \frac{1}{4} + \frac{1}{8} = \mathbf{.375_{10}}$$

C Float Demo

Exercise 4: Floats

- What fractional number is represented by the bytes 0x423c0000? Assume big-endian order.

s	$\text{exp} = e_{k-1} \dots e_1 e_0$	$\text{frac} = f_{n-1} \dots f_1 f_0$

  - s is sign bit s
  - exp field encodes E (but is not equal to E)
    - normally $E = e_{k-1} \dots e_1 e_0 - (2^{k-1} - 1)$
  - frac field encodes M (but is not equal to M)
    - normally $M = 1.f_{n-1} \dots f_1 f_0$

Float (32 bits):
- k = 8, n = 23
- bias = 127

$$(-1)^s \cdot M \cdot 2^E$$

0	100 0010 0	011 1100 0000 0000 0000 0000

s=0 exp=132 frac = $01111000000000000000000_2$

s=0 E = 5 M = $1.01111000000000000000000_2$

$$(-1)^0 \cdot 1.011110_2 \cdot 2^5 = 101111.0_2 == \mathbf{47_{10}}$$

Limitation so far…

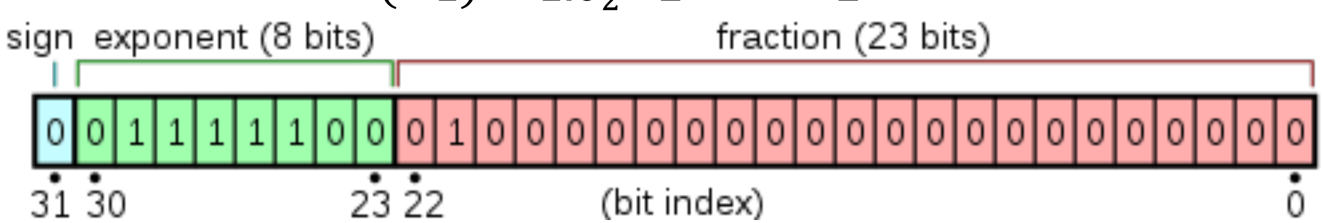
- What is the smallest non-negative number that can be represented?

0000 0000 0000 0000 0000 0000 0000 0000

s=0 exp=0 frac = $00000000000000000000000_2$

s=0 E = -127 M = $1.00000000000000000000000_2$

$$(-1)^0 \cdot 1.0_2 \cdot 2^{-127} = 2^{-127}$$



- What we like the smallest non-negative number to be?

Normalized and Denormalized

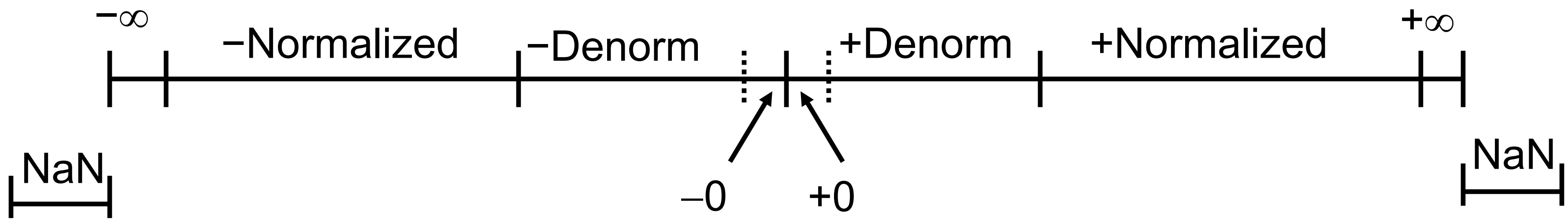
$$(-1)^s \cdot M \cdot 2^E$$

Normalized Values

- exp is neither all zeros nor all ones (normal case)
- exponent is defined as $\mathrm{E} = e_{k-1} \ldots e_1 e_0 - \mathrm{bias}$, where $\mathrm{bias} = 2^{k-1} - 1$ (e.g., 127 for float or 1023 for double)
- significand is defined as $M = 1.f_{n-1}f_{n-2} \ldots f_0$

- ## Denormalized Values
  - exp is either all zeros or all ones
  - if all zeros: $\mathrm{E} = 1 - \mathrm{bias}$ and $M = 0.f_{n-1}f_{n-2} \ldots f_0$
  - if all ones: infinity (if frac is all zeros) or NaN (if frac is non-zero)

Visualization: Floating Point Encodings



Exercise 5: Limits of Floats

- What is the difference between the largest (non-infinite) positive number that can be represented as a (normalized) float and the second-largest?

s	exp	frac
1	8-bits	23-bits

Exercise 5: Limits of Floats

- What is the difference between the largest (non-infinite) positive number that can be represented as a (normalized) float and the second-largest?

0 | 111 1111 0 | 111 1111 1111 1111 1111 1111

s=0 E = 127 M = $1.11111111111111111111111_2$

$$\text{largest} = 1.11111111111111111111111_2 \cdot 2^{127}$$

$$\text{second\_largest} = 1.11111111111111111111110_2 \cdot 2^{127}$$

$$\text{diff} = 0.00000000000000000000001_2 \cdot 2^{127} = 1_2 \cdot 2^{127-23} = \mathbf{2^{104}}$$

Any number between these two gets rounded.

Another Way to View Them

S	WINDOW	OFFSET

- Sign bit
- Window into two consecutive powers of two
  - [0.5, 1], [1, 2], [2, 4], …, [2^{127}, 2^{128}]
- Offset dividing the window into 2^{23} buckets
  - Finer grained near zero

Credit to Fabien Sanglard: https://fabiensanglard.net/floating_point_visually_explained/

Correctness

- **Example 1: Is (x + y) + z = x + (y + z)?**
  - Ints: Yes!
  - Floats:
    - (2^30 + -2^30) + 3.14 → 3.14
    - 2^30 + (-2^30 + 3.14) → 0.0

Floating Point in C

- C Guarantees Two Levels
  - **`float`** single precision (32 bits)
  - **`double`** double precision (64 bits)
- Conversions/Casting
  - Casting between **`int`**, **`float`**, and **`double`** changes bit representation
  - **`double`**/**`float`** → **`int`**
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range or NaN: Generally sets to TMin
  - **`int`** → **`double`**
    - Exact conversion,
  - **`int`** → **`float`**
    - Will round

Exercise 6: Casting with Floats

- Assume you have three variables: an int x, a float f, and a double d. Assume that all three variables store numeric values (not $+\infty, -\infty$, or `NaN`). Which of the following expressions are guaranteed to evaluate to True?
  1. `x == (int)(double)(x)`
  2. `x == (int)(float)(x)`
  3. `d == (double)(float) d`
  4. `f == (float)(double) f`

Exercise 6: Casting with Floats

- Assume you have three variables: an int x, a float f, and a double d. Assume that all three variables store numeric values (not $+\infty$, $-\infty$, or `NaN`). Which of the following expressions are guaranteed to evaluate to True?
  1. `x == (int)(double)(x)` **True**
  2. `x == (int)(float)(x)` **False**
  3. `d == (double)(float) d` **False**
  4. `f == (float)(double) f` **True**

Floating Point Operations

- All the bitwise and logical operations still work
- Float arithmetic operations done by separate hardware unit (FPU)