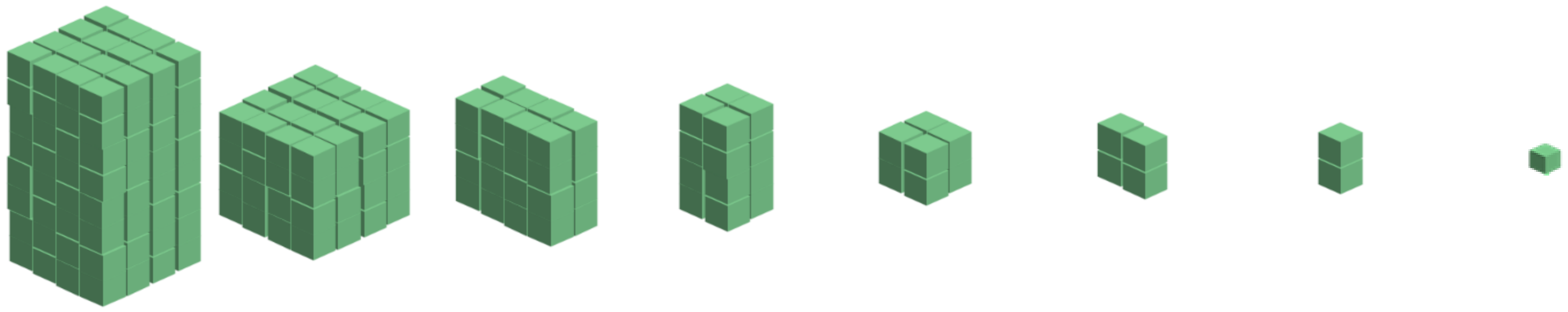
Lecture 3: Floats

CS 105 Fall 2024

Review: Representing Integers

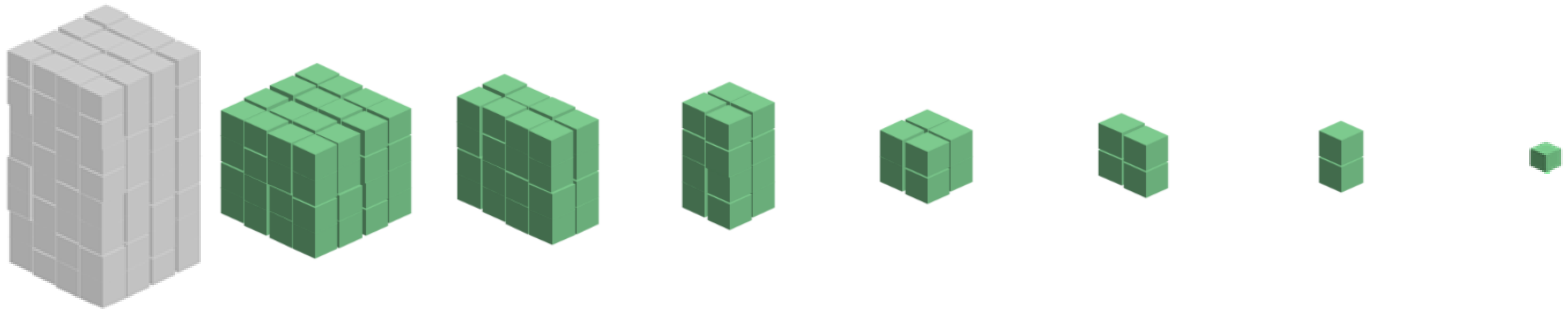
- unsigned:

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

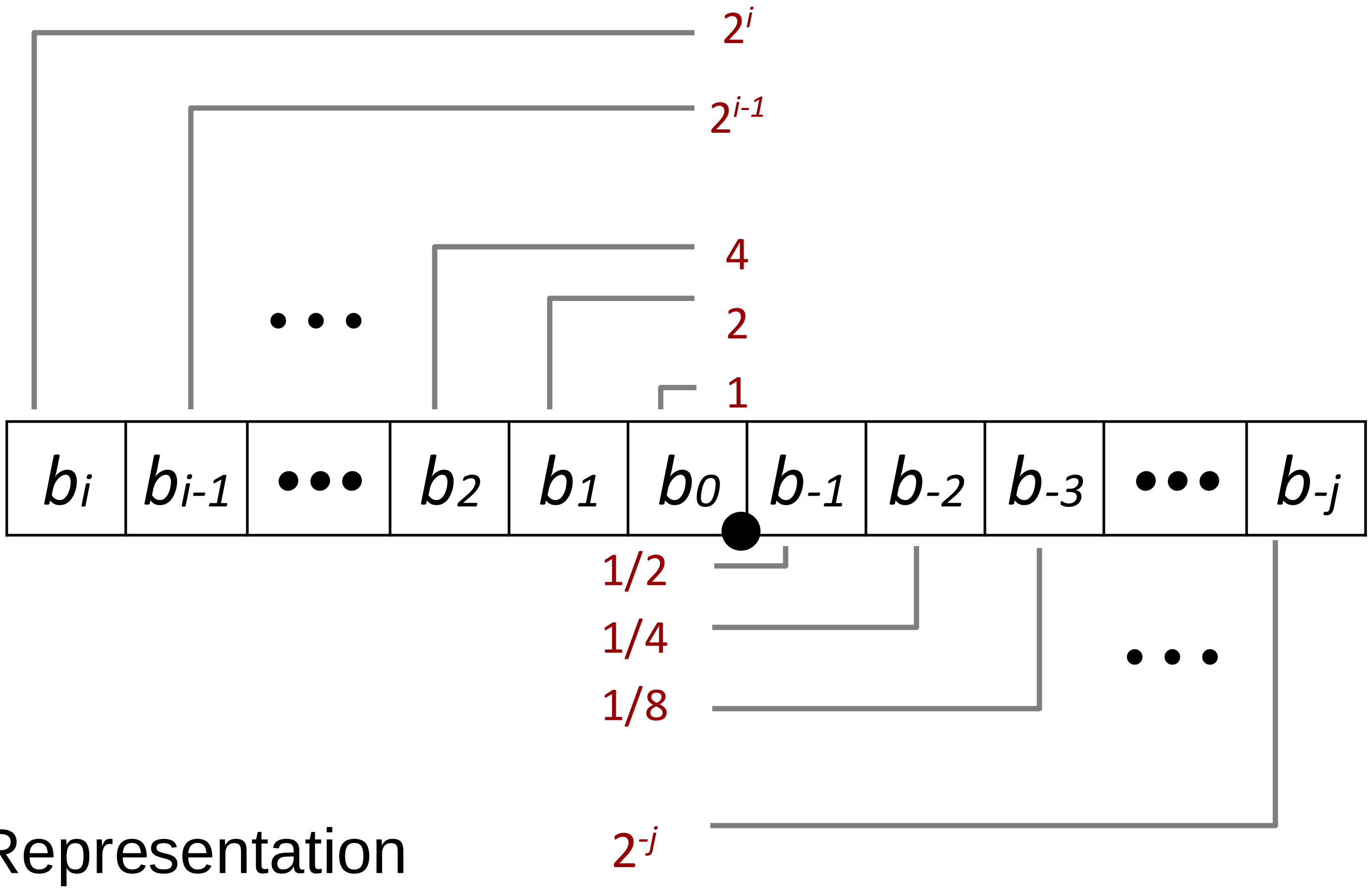


- signed (two's complement):

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



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?

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

- 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
  - `.011`
  - `.11`
  - `1.1`

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? very large?)

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 (binary) fractional value in range [1.0,2.0)
  - **Exponent** E weights value by power of two

- Examples:
  - 1.0
  - -1.25
  - 64
  - .625

Exercise 2: Floating Point Numbers

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

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 $E = e_{k-1} \ldots e_1 e_0 - \underbrace{\boxed{(2^{k-1} - 1)}}_{\text{bias}}$
- 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

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

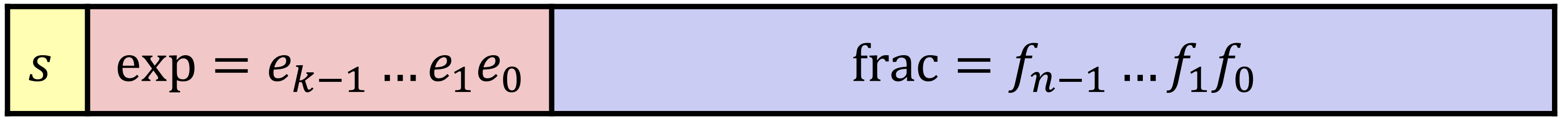
s=0 exp=125 frac = $100000000000000000000000_2$

s=0 E = -2 M = $1.100000000000000000000000_2 = 1.5_{10}$

$$(-1)^0 \cdot 1.5_{10} \cdot 2^{-2} = 1 \cdot \frac{3}{2} \cdot \frac{1}{4} = \frac{3}{8} = \mathbf{.375_{10}} \qquad (-1)^0 \cdot 1.1_2 \cdot 2^{-2} = .011_2 = \frac{1}{4} + \frac{1}{8} = \mathbf{.375_{10}}$$

Exercise 3: Floats

- What fractional number is represented by the bytes 0x423c0000? 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

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

s	exp	frac
1	8-bits	23-bits

Limitation so far…

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

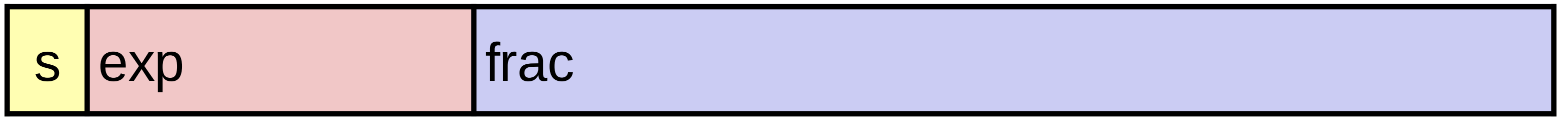
0000 0000 0000 0000 0000 0000 0000 0000

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

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

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

Normalized and Denormalized



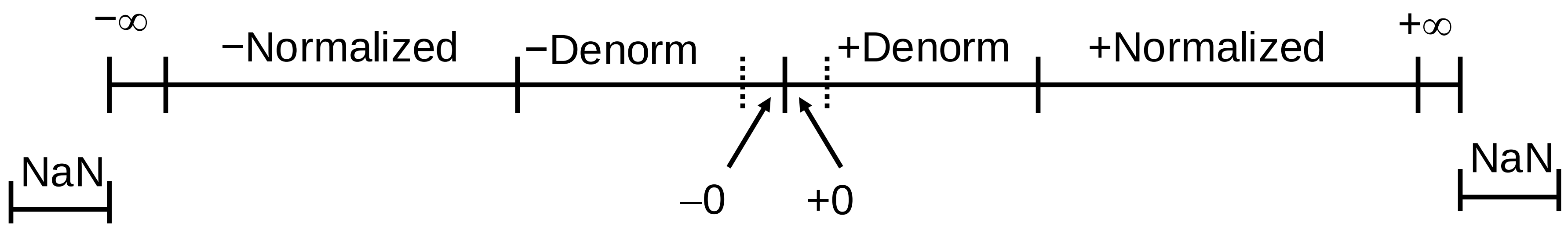
$$(-1)^s \cdot M \cdot 2^E$$

Normalized Values

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

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

Visualization: Floating Point Encodings



s	exp	frac
1	8-bits	23-bits

Exercise 4: Normalized and Denormalized

- Write a C function to compute a floating point representation of 2^x by directly constructing the IEEE float representation of the result. When x is too small, return 0.0 When x is too large, return $+\infty$

```
float fpwr2(int x){
  unsigned exp, frac, u;

  if(x < _____________){  /* Too small */
    exp = ____________;
    frac = ___________;
  } else if (x <= ____){  /* Denormalized */
    exp = ____________;
    frac = ___________;
  } else if (x <= ____){  /* Normalized */
    exp = ____________;
    frac = ___________;
  } else {                /* Too big */
    exp = ____________;
    frac = ___________;
  }

  u = exp << 23 | frac;  /* pack exp, frac */
  return u2f(u);  /* return as float */
}
```

s	exp	frac
1	8-bits	23-bits

Example: 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 $M = 1.11111111111111111111111_2$

$$\text{largest} = 1.11111111111111111111111_2 \cdot 2^{127}$$
$$\text{second\_largest} = 1.11111111111111111111110_2 \cdot 2^{127}$$

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

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

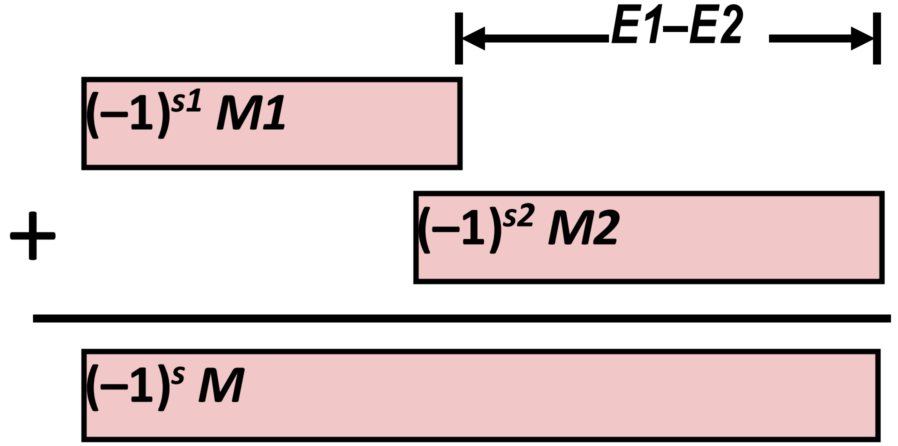
Floating Point Operations

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

Floating Point Addition

- Float operations done by separate hardware unit (FPU)
- $F_1 + F_2 = (-1)^{s_1} \cdot M_1 \cdot 2^{E_1} + (-1)^{s_1} \cdot M_1 \cdot 2^{E_1}$
  - Assume *E1* >= *E2*

Get binary points lined up



- Exact Result: $(-1)^s \cdot M \cdot 2^E$
  - Sign *s*, significand *M*:
    - Result of signed align & add
  - Exponent *E*: *E1*
- Fixing
  - If *M* ≥ 2, shift *M* right, increment *E*
  - if *M* < 1, shift *M* left *k* positions, decrement *E* by *k*
  - Overflow if *E* out of range
  - Round *M* to fit **`frac`** precision

Floating Point Multiplication

- $F_1 \cdot F_2 = (-1)^{s_1} \cdot M_1 \cdot 2^{E_1} \cdot (-1)^{s_1} \cdot M_1 \cdot 2^{E_1}$
- Exact Result: $(-1)^s \cdot M \cdot 2^E$

Sign *s*:	*s1* ^ *s2*
Significand *M*:	*M1* x *M2*
Exponent *E*:	*E1* + *E2*

- Fixing
  - If *M* ≥ 2, shift *M* right, increment *E*
  - If *E* out of range, overflow
  - Round *M* to fit **`frac`** precision

- Implementation
  - Biggest chore is multiplying significands

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