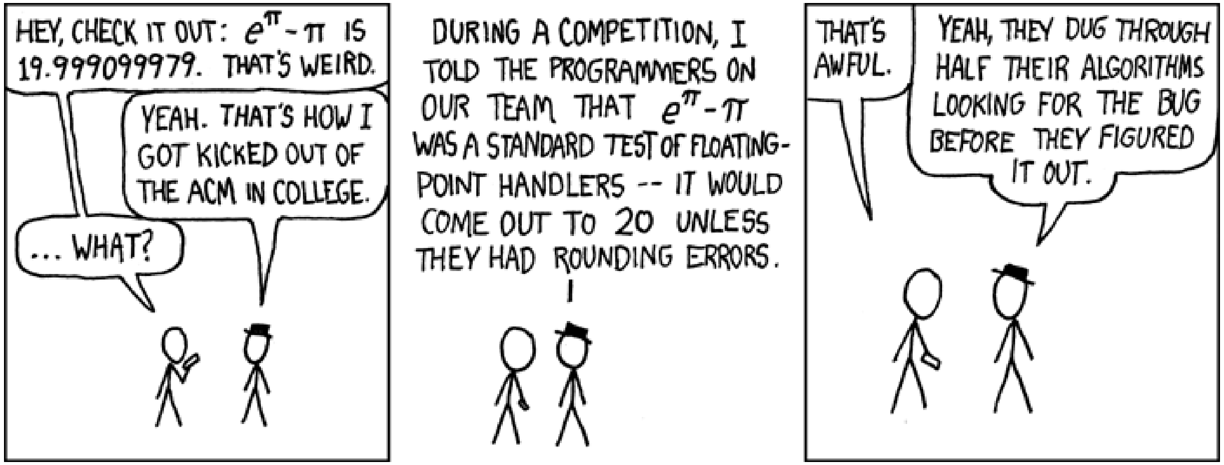
CS 261
Fall 2019

Mike Lam, Professor



https://xkcd.com/217/

Floating-Point Numbers

Floating-point

- Topics
  - Binary fractions
  - Floating-point representation
  - Conversions and rounding error

Binary fractions

- Now we can store integers
  - But what about general real numbers?
- Extend positional binary integers to store fractions
  - Designate a certain number of bits for the fractional part
  - These bits represent negative powers of two
  - (Just like fractional digits in decimal fractions!)

1	0	1	.	1	0	1
4	2	1		1/2	1/4	1/8

4 + 1 + 0.5 + 0.125 = **5.625** *(alternatively: 5 + 5/8)*

Another problem

- For scientific applications, we want to be able to store a wide *range* of values
  - From the scale of galaxies down to the scale of atoms
- Doing this with fixed-precision numbers is difficult
  - Even signed 64-bit integers
    - Perhaps allocate half for whole number, half for fraction
    - Range: ~2 x 10^{-9} through ~2 x 10^{9}

Floating-point demonstration using Super Mario 64:

https://www.youtube.com/watch?v=9hdFG2GcNuA

Floating-point numbers

- Scientific notation to the rescue!
  - Traditionally, we write large (or small) numbers as $x \cdot 10^e$
  - This is how floating-point representations work
    - Store exponent and fractional parts (the significand) separately
    - The decimal point “floats” on the number line
    - Position of point is based on the exponent

$$
1.23 = \begin{array}{r} 0.0123 \times 10^{2} \\ 0.123 \times 10^{1} \\ 1.23 \times 10^{0} \\ 12.3 \times 10^{-1} \\ 123.0 \times 10^{-2} \end{array}
$$

Floating-point numbers

- However, computers use binary
  - So floating-point numbers use base 2 scientific notation ($x \cdot 2^e$)
- Fixed width field
  - Reserve one bit for the sign bit (0 is positive, 1 is negative)
  - Reserve n bits for biased exponent (bias is $2^{n-1} - 1$)
    - Avoids having to use two's complement
  - Use remaining bits for normalized fraction (implicit leading 1)
    - Exception: if the exponent is zero, don't normalize

2.5 → 0 1000 010

Sign (+)

Significand: (1).01 = 2.5

Exponent (8 - 7 = 1)

Value = $(-1)^s \times 1.f \times 2^E$

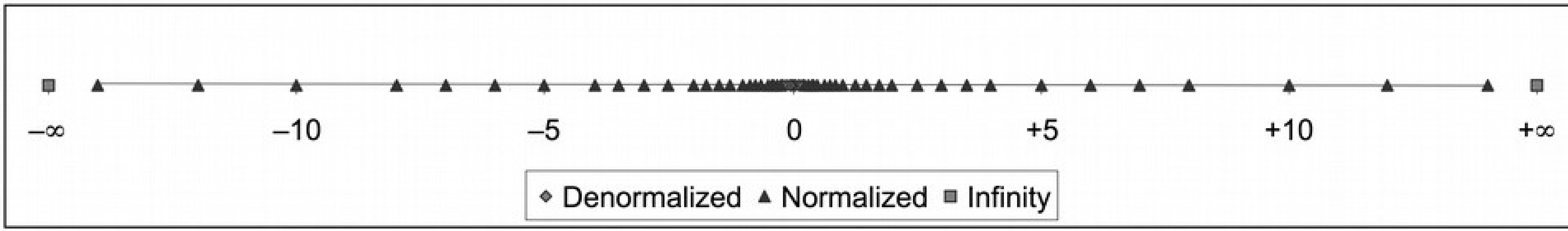
Aside: Offset binary

- Alternative to two's complement
  - Actual value is stored value minus a constant K (in FP: $2^{n-1} - 1$)
  - Also called biased or excess representation
  - Ordering of actual values is more natural

Example range (`int8_t`):

Binary	Unsigned	Two's C	Offset-127
0000 0000	0	0	-127
0000 0001	1	1	-126
…	…	…	…
0111 1110	126	126	-1
0111 1111	127	127	0
1000 0000	128	-128	1
1000 0001	129	-127	2
…	…	…	…
1111 1110	254	-2	127
1111 1111	255	-1	128

Floating-point numbers



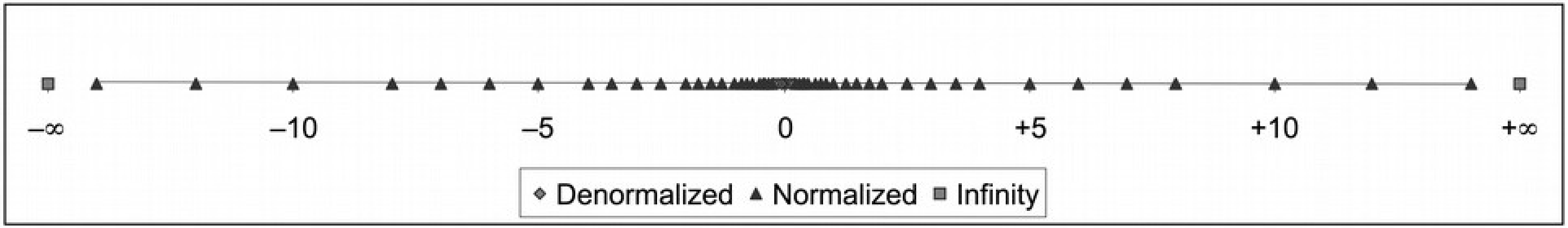
Not evenly spaced! (as integers are)

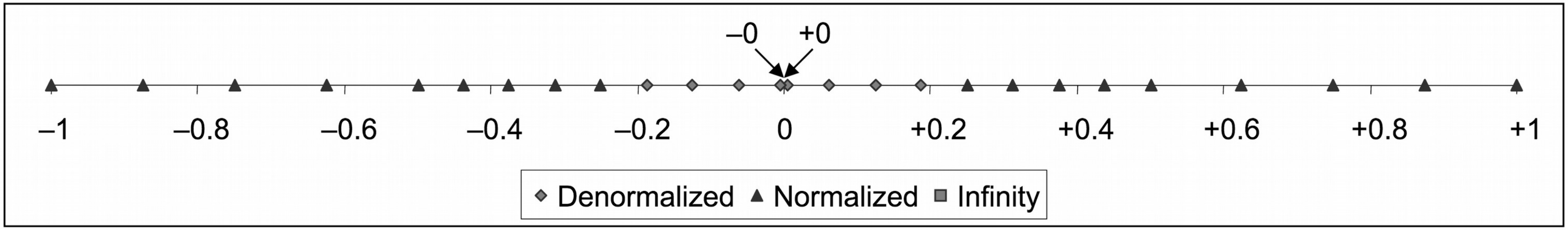
Consider these examples:

$1.00000 \times 2^{0} \rightarrow 1.00001 \times 2^{0}$
$1.00000 \times 2^{100} \rightarrow 1.00001 \times 2^{100}$

Adding a least-significant digit adds more value with a higher exponent than with a lower exponent

Floating-point numbers





Representable values for 6-bit floating-point format. There are k = 3 exponent bits and n = 2 fraction bits. The bias is 3.

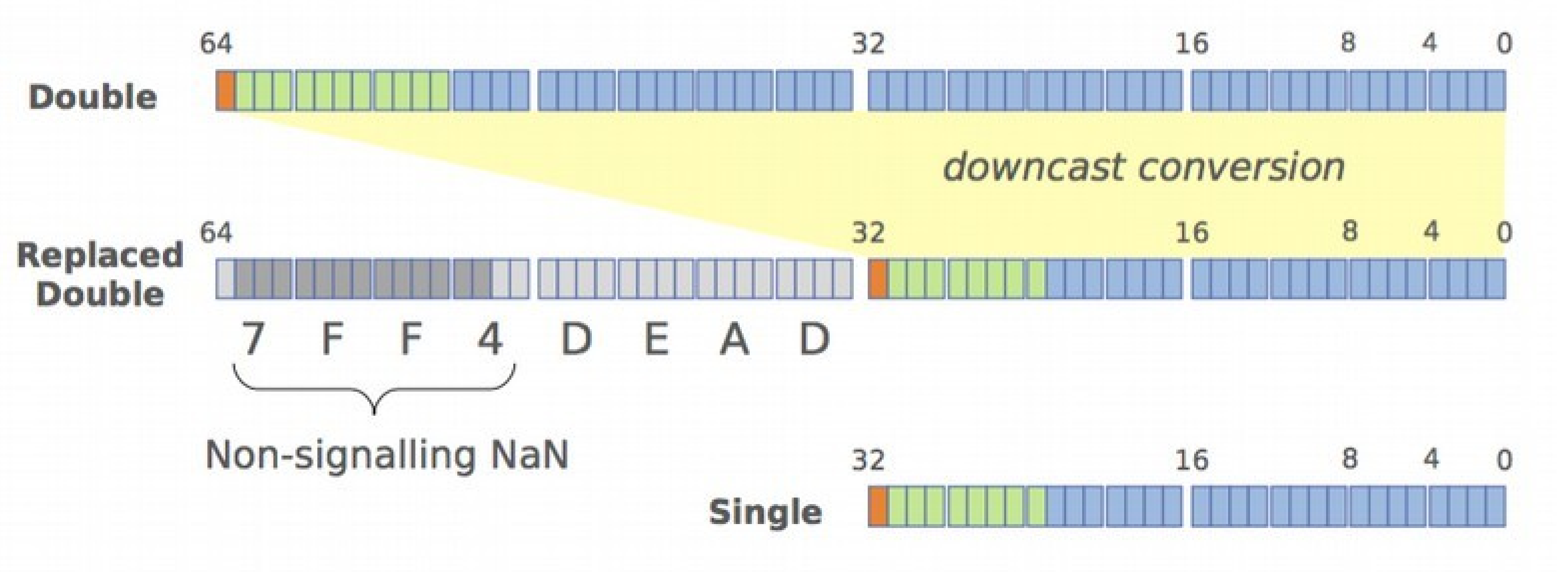
Description	Bit representation	Exponent e	E	2^E	Fraction f	M	Value $2^E \times M$	V	Decimal
Zero	0 0000 000	0	-6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{0}{512}$	0	0.0
Smallest positive	0 0000 001	0	-6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$	$\frac{1}{512}$	0.001953
	0 0000 010	0	-6	$\frac{1}{64}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$	$\frac{1}{256}$	0.003906
"denormal" numbers provide gradual underflow near zero	0 0000 011	0	-6	$\frac{1}{64}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$	$\frac{3}{512}$	0.005859
	⋮								
Largest denormalized	0 0000 111	0	-6	$\frac{1}{64}$	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$	$\frac{7}{512}$	0.013672
Smallest normalized	0 0001 000	1	-6	$\frac{1}{64}$	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{512}$	$\frac{1}{64}$	0.015625
	0 0001 001	1	-6	$\frac{1}{64}$	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$	$\frac{9}{512}$	0.017578
	⋮								
values < 1	0 0110 110	6	-1	$\frac{1}{2}$	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$	$\frac{7}{8}$	0.875
	0 0110 111	6	-1	$\frac{1}{2}$	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$	$\frac{15}{16}$	0.9375
One	0 0111 000	7	0	1	$\frac{0}{8}$	$\frac{8}{8}$	$\frac{8}{8}$	1	1.0
	0 0111 001	7	0	1	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	$\frac{9}{8}$	1.125
	0 0111 010	7	0	1	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$	$\frac{5}{4}$	1.25
values > 1	⋮								
	0 1110 110	14	7	128	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{1792}{8}$	224	224.0
Largest normalized	0 1110 111	14	7	128	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{1920}{8}$	240	240.0
Infinity	0 1111 000	—	—	—	—	—	—	∞	—

Figure 2.35 Example nonnegative values for 8-bit floating-point format. There are $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is 7.

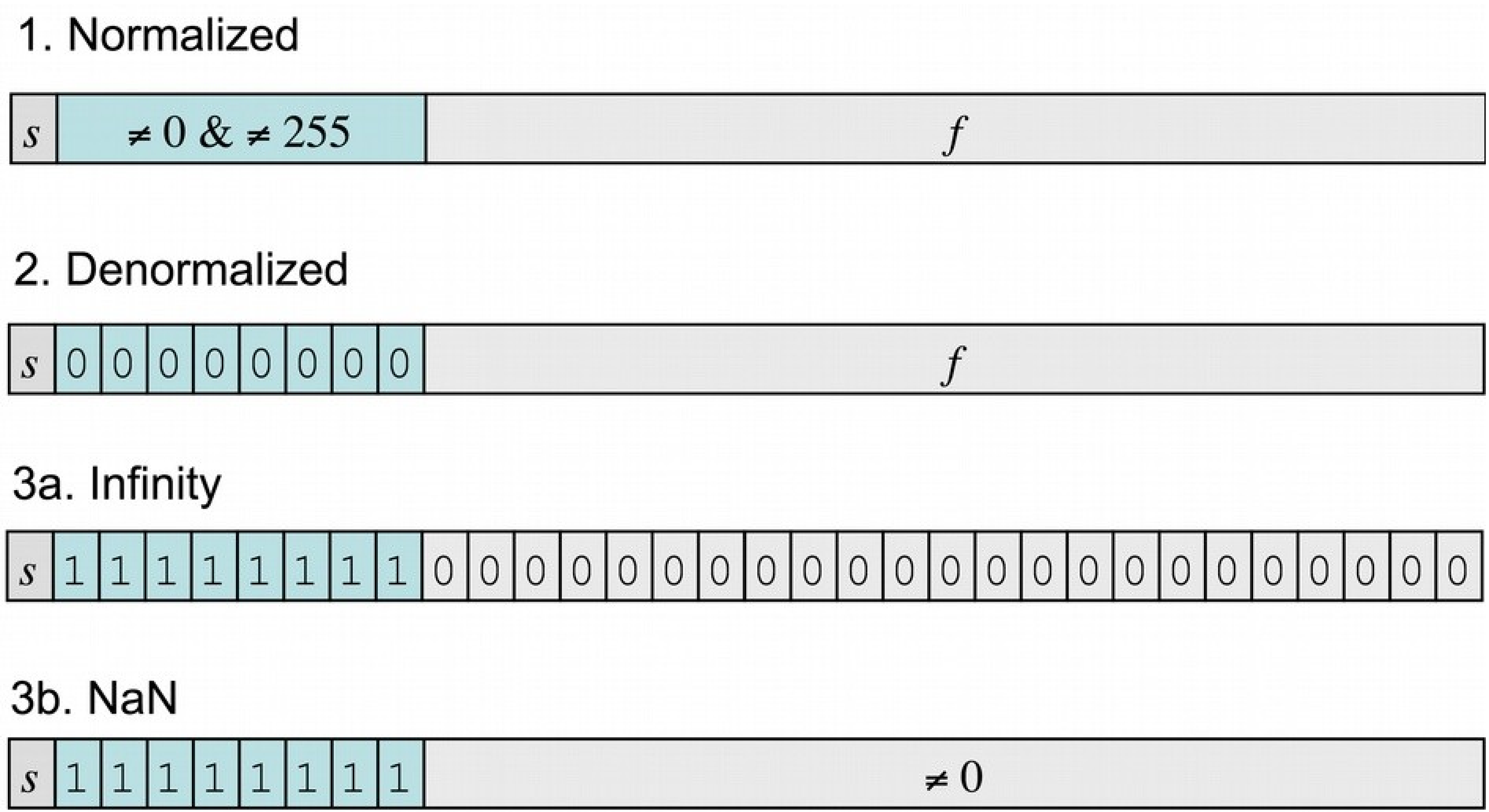
what about values higher than this one?

NaNs

- NaN = "Not a Number"
  - Result of 0/0 and other undefined operations
  - Propagate to later calculations
  - Quiet and signaling variants (qNaN and sNaN)
  - Allowed a neat trick during my dissertation research:



Floating-point numbers



Floating-point issues

- Rounding error is the value lost during conversion to a finite significand
  - Machine epsilon gives an upper bound on the rounding error
    - (Multiply by value being rounded)
  - Can compound over successive operations
- Lack of associativity caused by intermediate rounding
  - Prevents some compiler optimizations
- Cancellation is the loss of significant digits during subtraction
  - Can magnify error and impact later operations

```
double a = 100000000000000000000.0;
double b = -a;
double c = 3.14;
if (((a + b) + c) == (a + (b + c))) {
    printf ("Equal!\n");
} else {
    printf ("Not equal!\n");
}
```

```
  2.491264  (7)
- 2.491252  (7)
  0.000012  (2)

(5 digits cancelled)
```

```
  1.613647  (7)
- 1.613647  (7)
  0.000000  (0)

(all digits cancelled)
```

Floating-point issues

- Many numbers cannot be represented exactly, regardless of how many bits are used!
  - E.g., $0.1_{10} \rightarrow 0.00011001100110011001100_2 \ldots$

- This is no different than in base 10
  - E.g., 1/3 = 0.333333333 …

- If the number can be expressed as a sum of negative powers of the base, it can be represented exactly
  - Assuming enough bits are present

Floating-point standards

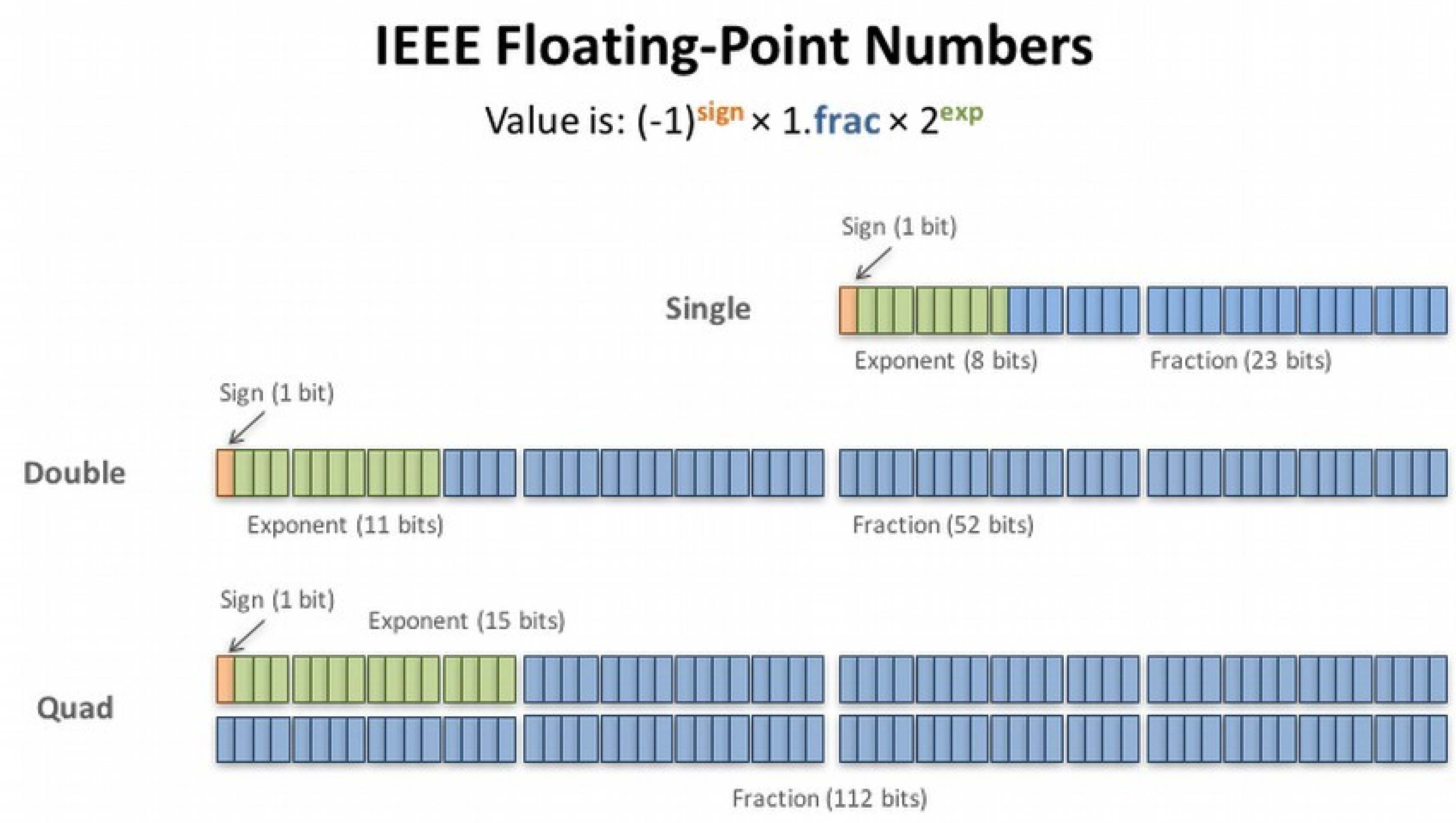
Name	Bits	Exp	Sig	Dec	M_Eps
bfloat16	16	8	7+1	2.408	7.81e-03
IEEE half	16	5	10+1	3.311	9.77e-04
IEEE single	32	8	23+1	7.225	1.19e-07
IEEE double	64	11	52+1	15.955	2.22e-16
IEEE quad	128	15	112+1	34.016	1.93e-34

NOTES:

- Sig is <*explicit*>[+<*implicit*>] bits
- Dec = $\log_{10}(2^{\mathrm{Sig}})$
- M_Eps (machine epsilon) = $b^{(-(p-1))} = b^{(1-p)}$

  (upper bound on relative error when rounding to 1)

Floating-point standards



Conversion and rounding

To:

From:

	Int32	Int64	Float	Double
Int32	-	-	R	-
Int64	O	-	R	R
Float	OR	OR	-	-
Double	OR	OR	OR	-

O = overflow possible
R = rounding possible

"-" is safe

Rounding

Mode	$1.40	$1.60	$1.50	$2.50	$–1.50
Round-to-even	$1	$2	$2	$2	$–2
Round-toward-zero	$1	$1	$1	$2	$–1
Round-down	$1	$1	$1	$2	$–2
Round-up	$2	$2	$2	$3	$–1

Figure 2.37 **Illustration of rounding modes for dollar rounding.** The first rounds to a nearest value, while the other three bound the result above or below.

Round-to-even: round to nearest, on ties favor even numbers to avoid statistical biases

In binary, to round to bit *i*, examine bit *i*+1:
- If 0, round down
- If 1 and any of the bits following are 1, round up
- Otherwise, round up if bit *i* is 1 and down if bit *i* is 0

```
10.00011 → 10.00  (down)
10.00100 → 10.00  (tie, round down)
10.10100 → 10.10  (tie, round down)
10.01100 → 10.10  (tie, round up)
10.11100 → 11.00  (tie, round up)
10.00110 → 10.01  (up)
```

Floating-point issues

- Single vs. double precision choice
  - Theme: **system design involves tradeoffs**
  - Single precision arithmetic is **faster**
    - Especially on GPUs (vectorization & bandwidth)
  - Double precision is **more accurate**
    - More than twice as accurate!
  - Which do we use?
    - And how do we justify our choice?
    - Does the answer change for different regions of a program?
    - Does the answer change for different periods during execution?
    - This is an open research question (talk to me if you’re interested!)

Manual conversions

- To fully understand how floating-point works, it helps to do some conversions manually
  - This is unfortunately a bit tedious and very error-prone
  - There are some general guidelines that can help it go faster
  - You will also get faster with practice
  - Use the `fp.c` utility (posted on the resources page) to generate practice problems and test yourself!
    - Compile: `gcc -o fp fp.c -lm`
    - Run: `./fp <exp_len> <sig_len>`
    - It will generate all positive floating-point numbers using that representation
    - Choose one and convert the binary to decimal or vice versa

```
...
0 1011 000      58    normal:  sign=0  e=11  bias=7  E=4  2^E=16  f=0/8  M=8/8  2^E*M=128/8  val=16.000000
0 1011 001      59    normal:  sign=0  e=11  bias=7  E=4  2^E=16  f=1/8  M=9/8  2^E*M=144/8  val=18.000000
0 1011 010      5a    normal:  sign=0  e=11  bias=7  E=4  2^E=16  f=2/8  M=10/8  2^E*M=160/8  val=20.000000
0 1011 011      5b    normal:  sign=0  e=11  bias=7  E=4  2^E=16  f=3/8  M=11/8  2^E*M=176/8  val=22.000000
...
```

Textbook's technique

e: The value represented by considering the exponent field to be an unsigned integer

E: The value of the exponent after biasing

2^E: The numeric weight of the exponent

f: The value of the fraction

M: The value of the significand

$2^E \times M$: The (unreduced) fractional value of the number

V: The reduced fractional value of the number

Decimal: The decimal representation of the number

If this technique works for you, great!
If not, here's another perspective...

Converting floating-point numbers

- Floating-point → decimal:
  - 1) Sign bit (*s*):
    - Value is negative iff set
  - 2) Exponent (*exp*):
    - All zeroes: denormalized (E = 1-bias)
    - All ones: NaN unless *f* is zero (which is infinity) – DONE!
    - Otherwise: normalized (E = *exp*-bias)
  - 3) Significand (*f*):
    - If normalized: M = 1 + $f / 2^m$ (where *m* is the # of fraction bits)
    - If denormalized: M = $f / 2^m$ (where *m* is the # of fraction bits)
  - 4) Value = $\mathbf{(-1)^s \times M \times 2^E}$

Note:
bias = $2^{n-1} - 1$
(where *n* is the # of exp bits)

Converting floating-point numbers

- Decimal → floating-point (normalized only)
  - 1) Convert to unsigned fractional binary format
    - Set sign bit
  - 2) Normalize to 1.xxxxxx
    - Keep track of how many places you shift left (negative for shift right)
    - The "xxxxxx" bit string is the significand (pad with zeros **on the right**)
    - If there aren't enough bits to store the entire fraction, the value is rounded
  - 3) Encode resulting binary/shift offset (E) using bias representation
    - **Add** bias and convert to unsigned binary
    - If the exponent cannot be represented, result is zero or infinity

Note:
bias = $2^{n-1} - 1$

(where n is the # of exp bits)

Example (4-bit exp, 3-bit frac):

2.75 (dec) → 10.11 (bin) → 1.011×2^1 (bin) → 0 1000 011

Bias = $2^{4-1} - 1 = 7$

Exp: $1 + 7 = 8$

Example (textbook pg. 119)

$12345_{10} \rightarrow 11000000111001_2$

$\rightarrow 1.1000000111001_2 \text{ x } 2^{13}$

exp = 13 + 127 (bias) = 140 = 10001100_2

$\rightarrow$ 0 10001100 10000001110010000000000

(note the shared bits that appear in all three representations)

Exercises

- What are the values of the following numbers, interpreted as floating-point numbers with a 3-bit exponent and 2-bit significand?
  - What about a 2-bit exponent and a 3-bit significand?

```
001100          011001
```

- Convert the following values to a floating-point value with a 4-bit exponent and a 3-bit significand. Write your answers in hex.

```
-3        0.125        120        ∞
```