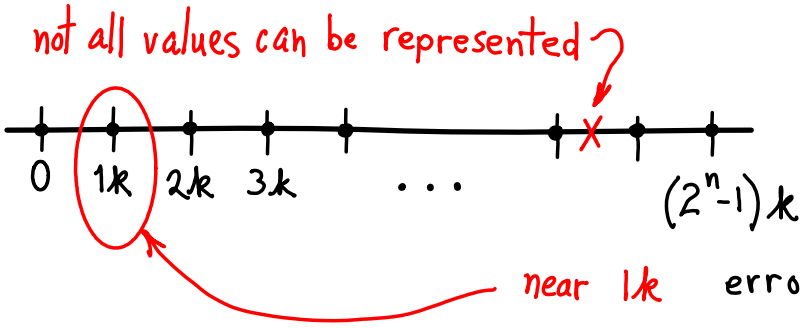
Floating Point

Would it be handy to have Real number arithmetic?

Can we approximate that?

Use n-bit scaled-number representation? Integer i represents k X i; k is some fraction.



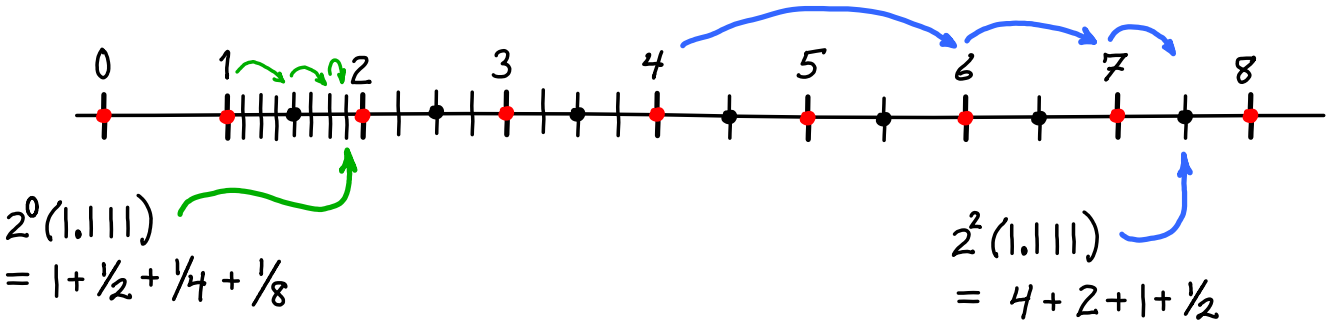
Discretization Error

worst case error $\approx \frac{1}{2}k$

error/value $= \frac{\frac{1}{2}k}{1k} = 50\%$

Exponential Scaling

$2^m(1.xyz)$



Discretization Error

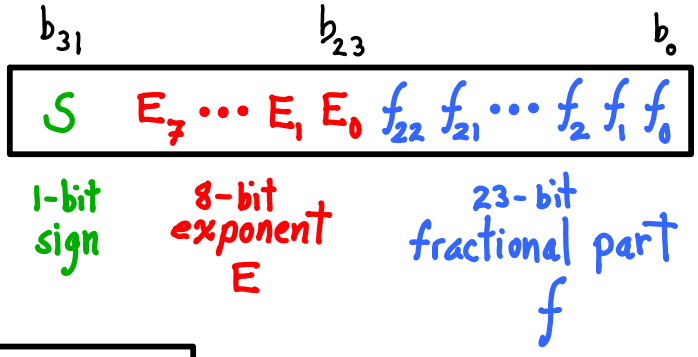
$\frac{1/16}{1} = \frac{1/4}{4} = 1/16$

===> **geometrical-progression of scaled integers**

% Error is consistent everywhere.

FP Format, 32-bit

$value = S\,2^{E} \times (1.f)$



0 00000010 10100000000000000000000

$+ 2^2 \times 1.1010\ldots0 = + 4(1 + 1/2 + 1/8)$

S: 0 = + 1 = -

E is pos. or neg. 8-bit 2s-comp.

Represent 0?

0 0 ··· 0 0 ··· 0

$+2^{0} \times 1.0 \cdots 0 = 1$?

We need 1.

Is there some other value we can do without?

Use it to represent 0?

How about sacrificing the smallest value?
Most negative 8-bit exponent? 10000000 = -128

$2^{-128} \times 1.0$

0 1 0 0 0 0 0 0 0 0 ··· 0 means +0

Maybe we can live with that?
Maybe use -127?

Here's a nice number for physics/chemistry: 6.023×10^{23}

Do we have enough bits for this?

6023×10^{20} w/ error ± 0.5

$\approx 6 \times 2^{10}$

$\approx 8 \times 2^{10}$

$= 2^{13}$ → 13-bit fractional part

we have 23 bits ✓

exponent

$10^{23} \approx 10^{24} = 10^{3 \cdot 8} \approx (2^{10})^{8} = 2^{80}$ → E = +80

8-bit 2s-comp. exponent range is -127 to +127 ✓

Sorting FP

Sorting is common.

Check x > y seems hard.

Check n > m for ints is easier: (n - m) and check sign bit.

Check x > y using integer hardware?

Treat x and y as integers, do integer subtraction?

Let's look at FP format, bit by bit

sign bit

x [0]

y [1]

Do (x - y):

x > y as 2's comp. integers

pos. FPs > neg. FP

It works, so far.

Now we check case when x and y have same sign.

exponent bits

x [00]

y [01]

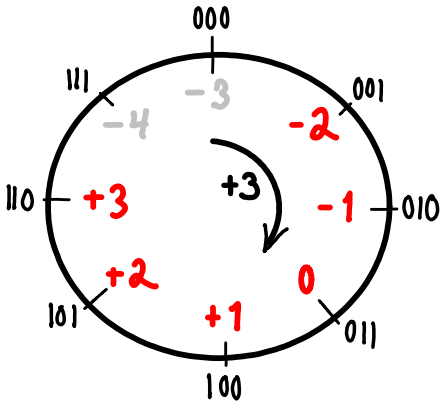
Not good:

negative exponent makes it look like x < y

But, as FP, x > y

→ Make all exponents so that neg. are less as unsigned

→ re-code 2s-comp



value	3-bit 2s-comp		new code	
+3	011		110	
+2	010		101	
+1	001		100	
0	000		011	
−1	111		010	
−2	110		001	
−3	101	reserved	000	for 0
−4	100	reserved	111	for NaN

New code = 2s-comp. + 011
"excess 3 code"

3-bit exponents AS **unsigned ints.**

Negative exponents
look **smaller** than
Positive exponents

for n-bit exponent

add $01\cdots1 = 2^{n-1} - 1$

Fractional parts are already unsigned:

We can sort FPs!

8-bit exponent

add 127 "excess 127 code"

Reserved

1...1	0	$\pm\infty$

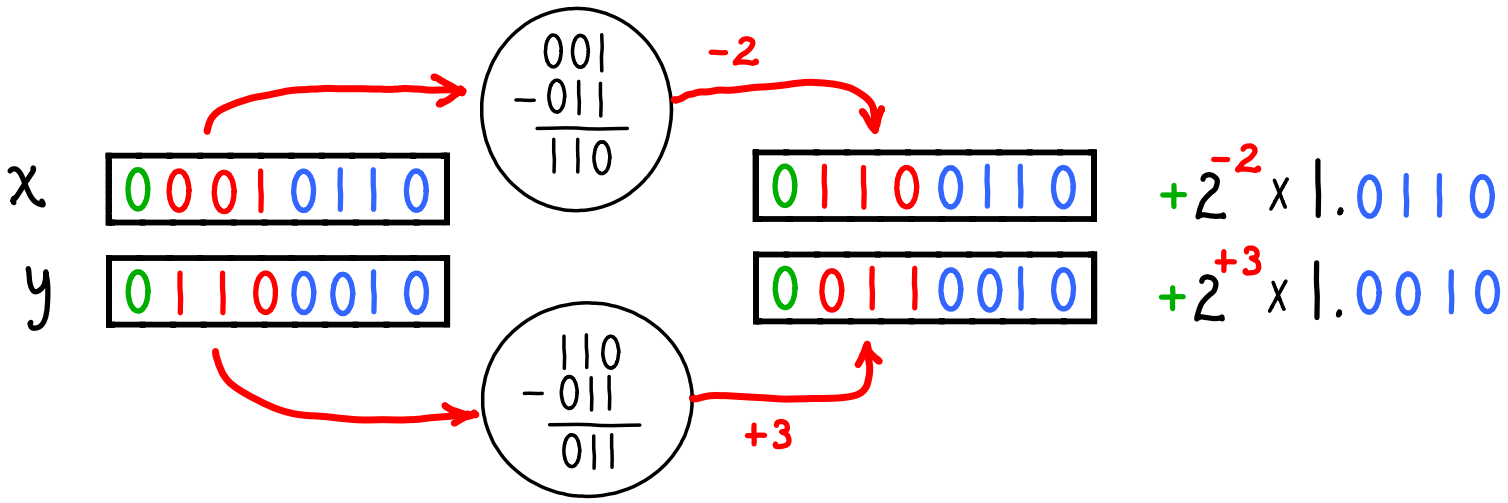
1...1	$f \neq 0$	NaN

0...0	0	0 (± 0)

0...0	$f \neq 0$	not normalized $2^{-126}\ 0.f$

8-bit FP, ADD

1. Convert exponents: excess-3 ===> 2s-comp.



2. Shift/Align fractional parts: make exponents same, shift x's fractional part right 5 places

x $+2^{+3} \times 0.0000|0110$

y $+2^{+3} \times 1.0010$

3. ADD $+2^{+3} \times 1.0010|0110$

4. re-Normalize: shift fraction and adjust exponent (not needed in this example)

$+2^{+3} \times 1.0010|0110$

5. Round to 4-bit fraction: round-to-nearest (or round-to-zero or ...)

$+2^{+3} \times 1.0010$

6. Convert exponent (+3) → 0 1 1 0 0 0 1 0

Uh Oh. That's y ??!

Errors: $x + y = y$!

Be Careful: discretization, rounding errors can add up ===> big problems.

8-bit FP, MULT

$x = +2^{-2} \times 1.0110$

$y = +2^{+3} \times 1.0010$

x 00010110

y 01100010

1. convert exponents

x 01100110

y 00110010

$+2^{-2} \times 1.0110$

$\times \; +2^{+3} \times 1.0010$

2. ADD exponents

$+2^{+1}$

3. Shift fractions, mult. unsigned ints, add exponents

1.0110×2^{-4}

$\times \; 1.0010 \times 2^{-4}$

$1.0001111100 \times 2^{-8}$

4. Normalize

$+2^{+1} \times 1.0001111100 \times 2^{+1}$

5. Add exponents

$+2^{+2} \times 1.0001111100$

6. Round

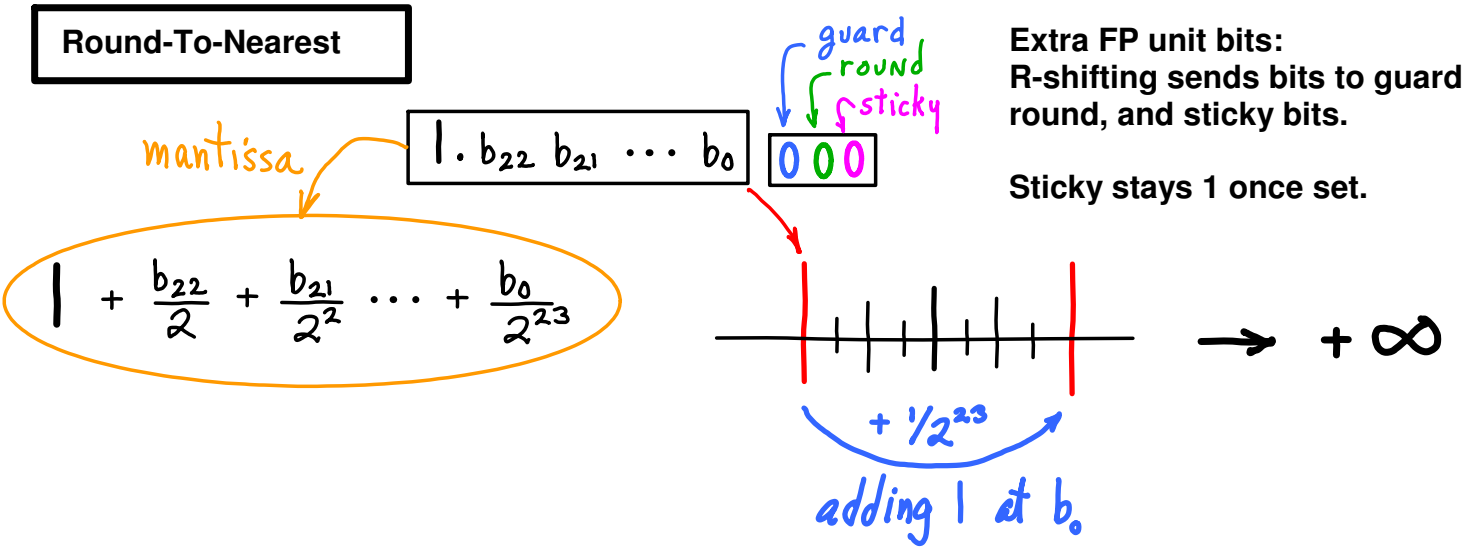
(might have to normalize, then round, then normalize again.)

$+2^{+2} \times 1.0010$

7. Convert exponent and encode

01010010

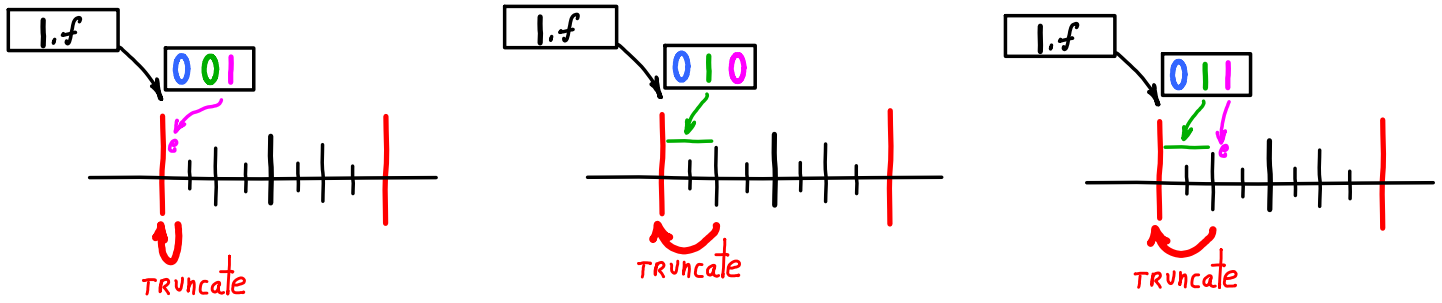
Round-To-Nearest

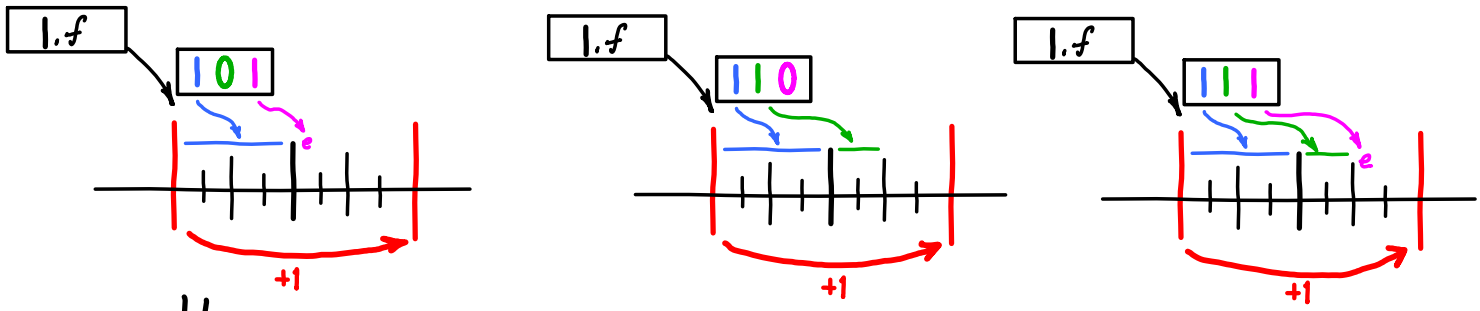


Extra FP unit bits:
R-shifting sends bits to guard, round, and sticky bits.

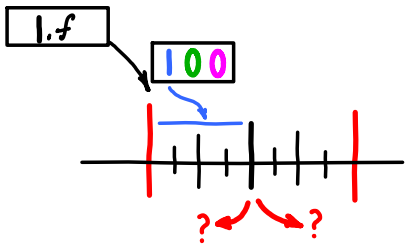
Sticky stays 1 once set.

The obvious cases





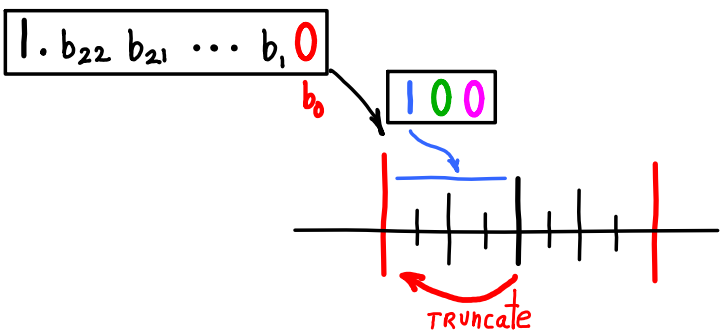
The problem case



Always round up? Always down?

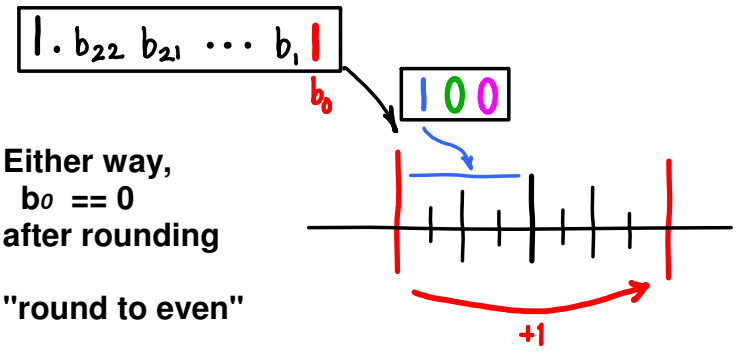
50-50 chance of either up or down?

Let b_0 decide ===> 50-50 it is 0 or 1



Either way,
b_0 == 0
after rounding

"round to even"



Convert to 32-bit FP

value = 28

1. convert to binary

11100

2. normalize

$2^4 \times 1.1100$

3. encode

23 bits

0 00000100 110…0

+ 01111111

4. Convert exponent to excess 127

0 10000011 110…0

Subtract largest possible power of 2:

28 - 16 == 12 ===> save 10000

Subtract next largest possible power of 2 from remainder:

12 - 8 == 4 ===> save 1000

Subtract next possible power of 2:

4 - 4 == 0 ===> save 100

- Latency in cycles of common arithmetic operations
- Source: *Software Optimization Guide for AMD Family 10h Processors, Dec 2007*
  - Intel "Core 2" chips similar

	Int 32	Int 64	Fp 32	Fp 64
Add/Subtract	1	1	4	4
Multiply	3	5	4	4
Divide	14 to 40	23 to 87	16	20

- Floating point divide faster than integer divide?
  - Why?