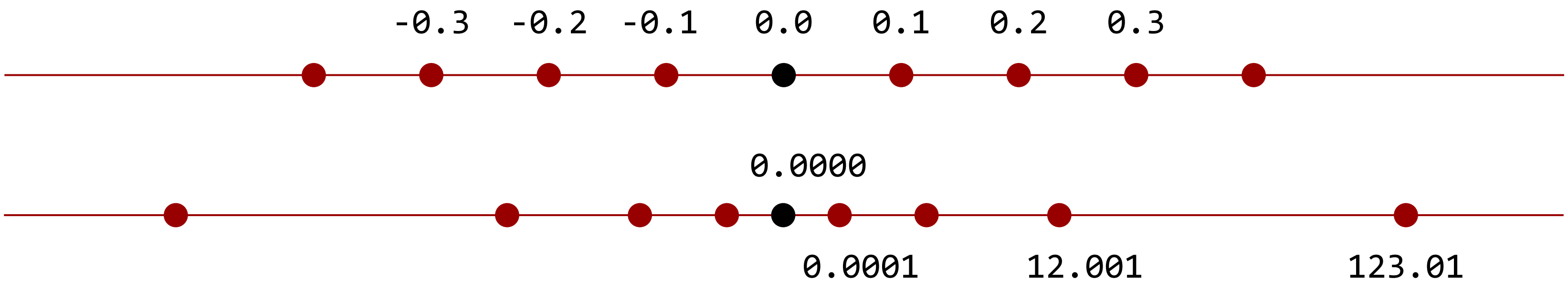
Unit 3

IEEE 754 Floating Point Representation

Floating Point

- Used to represent **very small** numbers (fractions) and **very large** numbers
  - Avogadro's Number: $+6.022 \times 10^{23}$
  - Boltzmann's Constant: $+1.38 \times 10^{-23}$
  - 32 or 64-bit integers can't represent this range!
- **`float`** / **`double`**: 32-bit and 64-bit floating-point in C

Same **number of combinations** given **32 bits**, so `float` **must space values differently** to have more range than `int`



Fixed Point, Base 10

- Let's say that we can use **only 6 digits** base 10

Unsigned Integers	**Fixed-Point, 1 decimal**	**Fixed-Point, 3 decimals**
`000000`	`00000.0`	`000.000`
`000001`	`00000.1`	`000.001`
`000002`	`00000.2`	`000.002`
`...`	`...`	`...`
`000150`	`00015.0`	`000.150`
`000151`	`00015.1`	`000.151`
`...`	`...`	`...`
`999998`	`99999.8`	`999.998`
`999999`	`99999.9`	`999.999`
Range: $[0, 10^6 - 1]$	**Range**: $[0, 10^5 - 0.1]$	**Range**: $[0, 10^3 - 0.001]$
Abs. rounding error $\leqslant 1/2$	**Abs. rounding error** $\leqslant 0.1/2$	**Abs. rounding error** $\leqslant 0.001/2$

Representation error (e.g., 2.1 rounded to 2), **add/sub are error-free** (except for **overflow**), **mul/div are not**

Floating Point, Base 10

- **Very large/small** numbers, same 6 digits?

Normal Notation
Don't start with 0

$$1.2345 \times 10^{5}$$

If exponent is `-1`	If exponent is `0`	If exponent is `1`
`.10000`	`1.0000`	`10.000`
`.10001`	`1.0001`	`10.001`
`.10002`	`1.0002`	`10.002`
`...`	`...`	`...`
`.99998`	`9.9998`	`99.998`
`.99999`	`9.9999`	`99.999`
Range: [0.1, 10^0-0.00001]	**Range**: [1, 10^1 - 0.0001]	**Range**: [10, 10^2 - 0.001]
ABS_ERR ⩽ 0.00001/2	**ABS_ERR** ⩽ 0.0001/2	**ABS_ERR** ⩽ 0.001/2

Biased Exponent
To represent positive and negative exponents using 1 decimal digit, we subtract BIAS=4 from stored digit

- stored digit `0`, .. , `9`
- exponent `-4`, .., `5`

Stored as

`123459`

If exponent is `5`
`100000.` to `999990.`
Range: [10^5, 10^6 - 10]
ABS_ERR ⩽ 10/2

We can use the **exponent to move the point**, and **pick large range or low representation error**

Perils of Floating Point

1.2345×10^{5} `123459`

1.0000×10^{-1} `100003`

What is the result of `123450 + 0.10000`?

- `123450 + 0.1 = 123450.1`
- How do we encode this large number using 5+1 digits?
- Same encoding as `123450`! The `0.1` is lost...
- Extended range but less density around large numbers

Reality Check: Floating-Point Numbers

```c
#include <stdio.h>

int main() {
    float x = 1000000.0f;     // large float (32-bit IEEE 754)

    // adding 0.01 does not increase x
    printf("%d ", x + 0.01f  == x);

    printf("%d %d\n",
            -x  + (x  +  0.01f) == 0.0f,    // bad: large + small
           (-x  +  x) +  0.01f  == 0.01f);  // better
}
```

```
$ gcc -Wall -Wextra -pedantic -std=c11 reality.c -o reality; ./reality
1 1 1
```

Finite number of significant digits: $1,000,000 + 0.01 \approx 1,000,000$

Addition is not associative: $x + (y + z) \neq (x + y) + z$

Fixed Point, Base 2

- Unsigned and 2’s complement fall under a category of representations called “Fixed Point”
- Radix point assumed to be in a **fixed location for all numbers**
  - Integers: 10011101. (binary point to right of LSB)
    - Range [0, 255], absolute error of 0.5
  - Fractions: .10011101 (binary point to left of MSB)
    - Range [0, 1 - 2^{-8}], absolute error of 2^{-9}
- **Trade-off**: range vs absolute representation error
  - Many fraction digits limit the range
  - Few fraction digits increase the representation error

Bit storage

Fixed point rep.

Floating point allows the radix point to be in a different location for each value!

Floating Point, Base 2

CS:APP 2.4.2

- Similar to scientific notation base-10

  $\pm D.DDD \times 10^{\pm exp}$

- … but using base 2

  $\pm\, b.bbbb \times 2^{\pm\, exp}$

  3 fields: **sign**, **exponent**, **fraction**

  (fraction is also called *mantissa* or *significand*)

S	**Exp.**	**Fraction**

Normalized Floating-Point

- In decimal
  - +**0**.754 × 10^{15} not correct scientific notation
  - +**7**.54 × 10^{14} correct: one significant digit before point
- In binary, the only significant digit is ‘1’
  Thus, normalized FP format is:

$$\pm 1.\text{bbbbbb} \times 2^{\pm \text{exp}}$$

  - Floating-point numbers are always normalized: if hardware calculates a result of 0.00**1101** × 2^5 it must normalize to **1.101**000 × 2^2 before storing
  - The **1.** is actually **not stored but assumed** since we always will store normalized numbers

IEEE 754 Floating Point Formats

- **Single Precision** (32-bit)
  - `float` in C
  - 1 sign bit (0=pos / 1=neg)
  - 8 exponent bits
    - Excess-127 representation
    - value = stored - 127
  - 23 fraction bits (after 1.)
  - Equivalent decimal range:
    - **7 digits** × $10^{\pm 38}$

1	8	23
S	**Exp.**	**Fraction**

- **Double Precision** (64-bit)
  - `double` in C
  - 1 sign bit (0=pos / 1=neg)
  - 11 exponent bits
    - Excess-1023 representation
    - value = stored - 1023
  - 52 fraction bits (after 1.)
  - Equivalent decimal range:
    - **16 digits** × $10^{\pm 308}$

1	11	52
S	**Exp.**	**Fraction**

Excess-N Exponent Representation

- Exponent needs its own sign (+/-)
- Use **Excess-N** instead of 2's complement
  - *w*-bit exponent ⇒ Excess-(2^{w-1}-1) encoding
  - **`float`**: 8-bit exponent ⇒ Excess-127
  - **`double`**: 11-bit exponent ⇒ Excess-1023
  - Why? So that comparisons x < y are simple (compare each corresponding bit left-to-right)
- Rule: **true value = stored value - N**
- For single-precision, **N=127**
  - … × 2^{1} ⇒ stored value $(1+127)_{10}$ = $1000\ 0000_2$
- For double-precision, **N=1023**
  - … × 2^{-2} ⇒ stored value $(-2 + 1023)_{10}$ = $(011\ 1111\ 1101)_2$

2's comp.	Stored Value	Excess-127
-1	`1111 1111`	+128
-2	`1111 1110`	+127
-128	`1000 0000`	+1
+127	`0111 1111`	0
+126	`0111 1110`	-1
+1	`0000 0001`	-126
0	`0000 0000`	-127

Comparison of
2's comp. & Excess-N

Q: Why don't we use 2's comp. to represent negative #'s?

Comparisons & Excess-N

- Why put the exponent field before the fraction?
  - Q: Which FP number is bigger?
    0.9999×2^2 or 1.0000×2^1
  - A: We should **look at the exponent first** to compare FP values; only look at the fraction if the exponents are equal
- By placing the exponent field first we can compare entire FP values as single bit strings (i.e., as if they were unsigned numbers)

0	10000010	0000001000
0	10000001	1110000000

→

010000010000000001000

010000000111110000000

< > = ???

Reserved Exponent Values

- FP formats reserve the exponent values of **all 1's** and **all 0's** for special purposes
- Thus, for single-precision the range of exponents is **-126 to + 127**

Stored Value (range of 8-bits shown)	**Excess-127 Value and Special Values**
255 = 11111111	Reserved
254 = 11111110	254-127=+127
…	
128 = 10000000	128-127= +1
127 = 01111111	127-127= 0
126 = 01111110	126-127= -1
…	
1 = 00000001	1-127=-126
0 = 00000000	Reserved

IEEE Exponent Special Values

Exp. Field	Fraction Field	Meaning
000…00	0000...0000	±0
	Non-Zero	Denormalized (**±0.bbbbbb × 2^{-126}**)
111…11	0000...0000	± ∞
	Non-Zero	NaN (Not-a-Number) - 0/0, 0*∞,SQRT(-x)

Transition to denormalized

- When the exponent is all 0's and the fraction is nonzero, the number is denormalized
  - An implicit **0.**(fraction) is assumed
  - The exponent value **-126** is used, which is the same excess-127 value of an exponent field equal to 1
- This produces a smooth transition from normalized to denormalized numbers
  - `0 00000001 0000..0` is $(1.0)_2$ x 2^-126
  - `0 00000000 1000..0` is $(0.1)_2$ x 2^-126
  - `0 00000000 0100..0` is $(0.01)_2$ x 2^-126

A nice tool: http://evanw.github.io/float-toy/

Single-Precision Examples

CS:APP 2.4.3

①

2^7=128 **2^1=2**

1	**1000 0010**	**110 0110 0000 0000 0000 0000**

130-127 = 3

$$-1.1100110 \times 2^3$$

$$= -1110.011 \times 2^0$$

$$= -14.375$$

②

+0.6875 = +0.1011

$$= +1.\underbrace{011} \times 2^{-1}$$

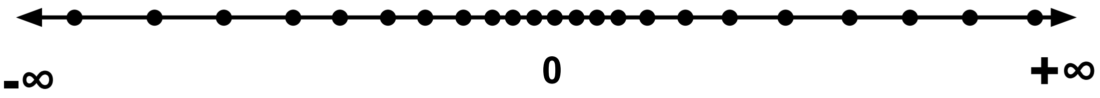
-1 +127 = 126

0	0111 1110	**011** 0000 0000 0000 0000 0000

`3 F 3 0 0 0 0 0`

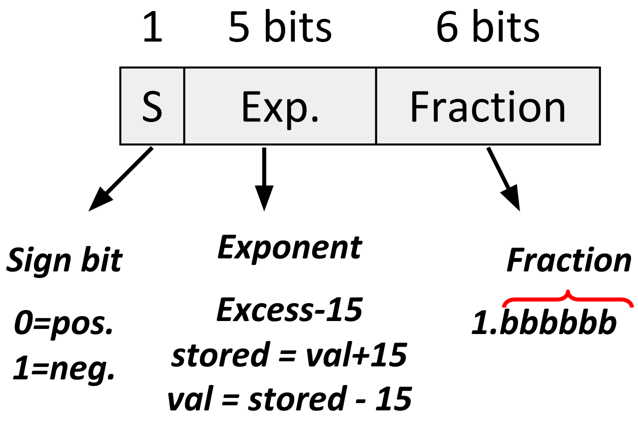
Floating Point vs. Fixed Point

- Single-precision (32-bits) equivalent decimal range
  - 7 significant decimal digits × $10^{\pm 38}$
  - Compare that to 32-bit signed integer where we can represent ±2 billion. How does a 32-bit float allow us to represent such a greater range?
  - FP allows for **range** but sacrifices **precision** (can't represent all numbers in its range)
- Double Precision (64-bits) Equivalent Decimal Range:
  - 16 significant decimal digits × $10^{\pm 308}$



12-bit "IEEE Short" Format

- 12-bit format defined just for this class (doesn't really exist)
  - 1 sign bit
  - 5 exponent bits (using Excess-15)
    - Same reserved codes
  - 6 fraction bits



1	5 bits	6 bits
S	Exp.	Fraction

Sign bit

0=pos.
1=neg.

Exponent

Excess-15
stored = val+15
val = stored - 15

Fraction

$1.\overbrace{bbbbbb}$

Examples

①

1	10100	101101

20-15=5

$$-1.101101 \times 2^5$$

$$= -110110.1 \times 2^0$$

$$= -110110.1 = -54.5$$

②

$$+21.75 = +10101.11$$

$$= +1.\underbrace{010111} \times 2^4$$

4+15=19

0	10011	010111

③

1	01101	100000

13-15=-2

$$-1.100000 \times 2^{-2}$$

$$= -0.011 \times 2^0$$

$$= -0.011 = -0.375$$

④

$$+3.625 = +11.101$$

$$= +1.\underbrace{110100} \times 2^1$$

1+15=16

0	10000	110100

ROUNDING

The Need To Round

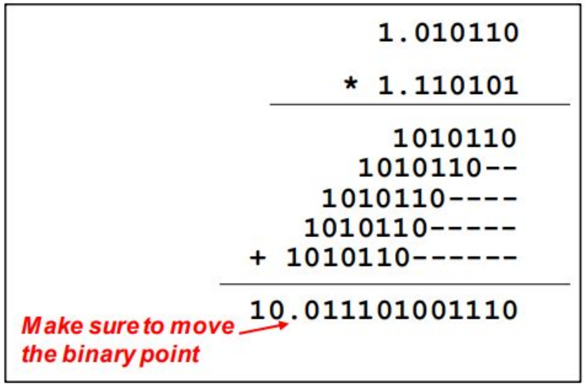
CS:APP 2.4.4

- Integer to FP
  - $+725 = 1011010101 = 1.\mathbf{011010}101 \times 2^9$
    - If we only have **6 fraction bits**, we can't keep all fraction bits
- FP ADD / SUB

$$\begin{array}{r} 5.9375 \times 10^1 \\ \underline{+\ 2.3256 \times 10^5} \end{array} \Rightarrow \begin{array}{r} .00059375 \times 10^5 \\ \underline{+\ 2.3256 \qquad\quad \times 10^5} \end{array}$$

- FP MUL / DIV

$$\begin{array}{lr} & 1.010110 \\ \underline{*} & \underline{1.110101} \\ \multicolumn{2}{r}{10.011101001110} \end{array}$$



```
          1.010110
        * 1.110101
    --------------
           1010110
         1010110--
       1010110----
      1010110-----
  +  1010110------
  ----------------
    10.011101001110
```

Make sure to move the binary point

Rounding Methods

- Methods of Rounding (you are only responsible for the first 2)

Round to Nearest, Half to Even	Round to the nearest representable number. If exactly halfway between, round to representable value with 0 in LSB (i.e., nearest **even** fraction).
Round towards 0 (Chopping)	Round the representable value closest to but not greater *in magnitude* than the precise value. Equivalent to just **dropping the extra bits**.
Round toward +∞ (Round Up / Ceiling)	Round to the closest representable value greater than the number
Round toward -∞ (Round Down / Floor)	Round to the closest representable value less than the number

Number Line View Of Rounding Methods

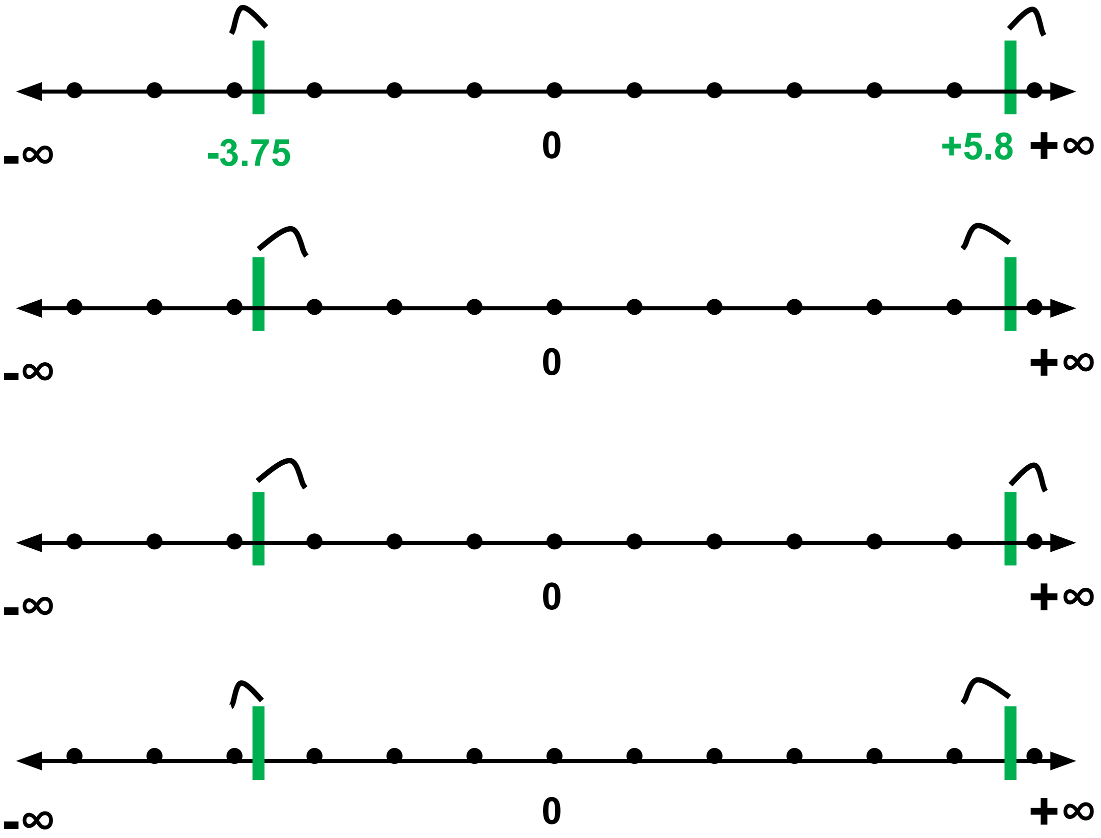
Green lines are FP results that fall between two representable values (dots) and thus need to be rounded

Round to Nearest

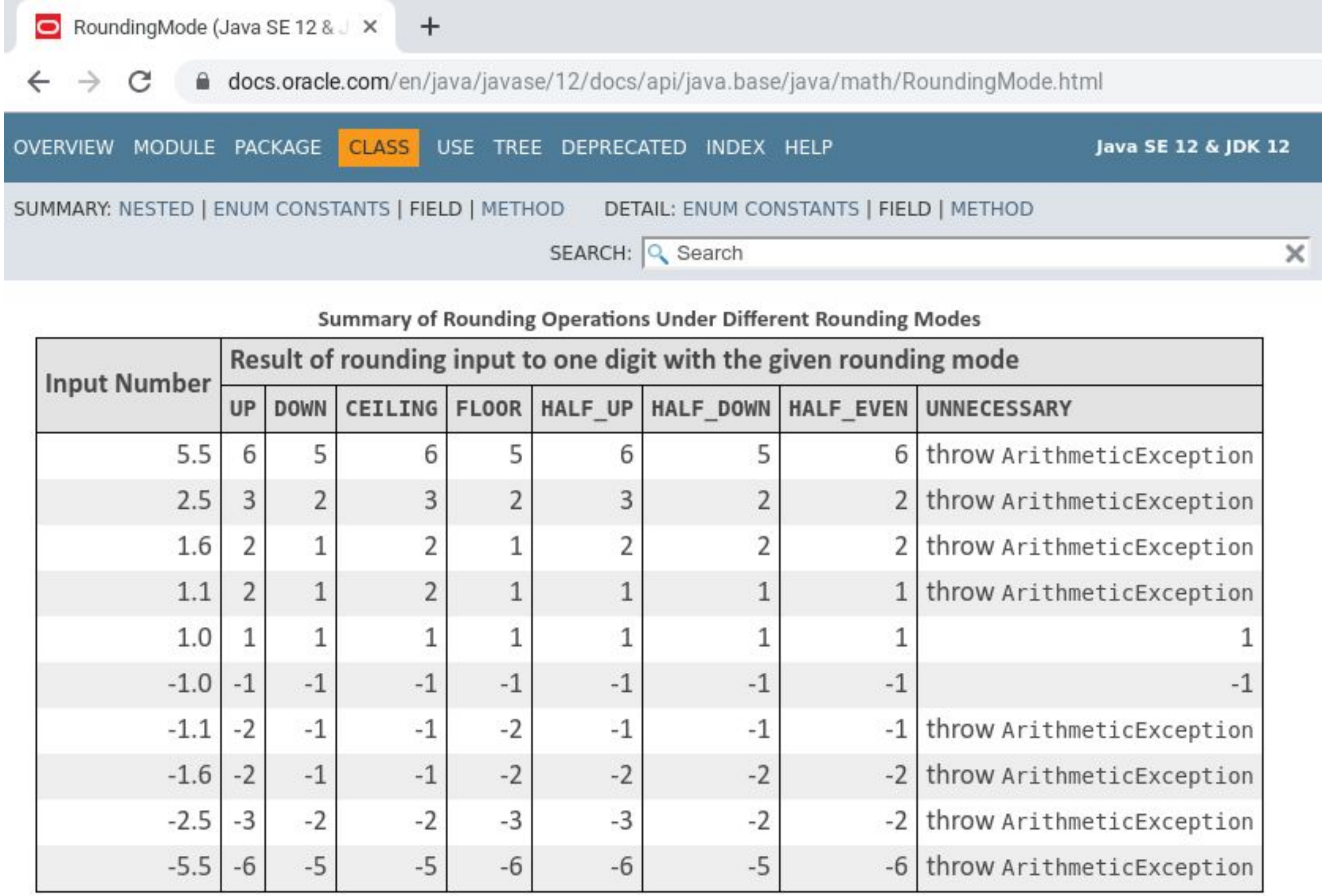
Round to Zero

Round to +Infinity

Round to -Infinity



… and many more!



Summary of Rounding Operations Under Different Rounding Modes

Input Number	Result of rounding input to one digit with the given rounding mode							
	UP	DOWN	CEILING	FLOOR	HALF_UP	HALF_DOWN	HALF_EVEN	UNNECESSARY
5.5	6	5	6	5	6	5	6	throw `ArithmeticException`
2.5	3	2	3	2	3	2	2	throw `ArithmeticException`
1.6	2	1	2	1	2	2	2	throw `ArithmeticException`
1.1	2	1	2	1	1	1	1	throw `ArithmeticException`
1.0	1	1	1	1	1	1	1	1
-1.0	-1	-1	-1	-1	-1	-1	-1	-1
-1.1	-2	-1	-1	-2	-1	-1	-1	throw `ArithmeticException`
-1.6	-2	-1	-1	-2	-2	-2	-2	throw `ArithmeticException`
-2.5	-3	-2	-2	-3	-3	-2	-2	throw `ArithmeticException`
-5.5	-6	-5	-5	-6	-6	-5	-6	throw `ArithmeticException`

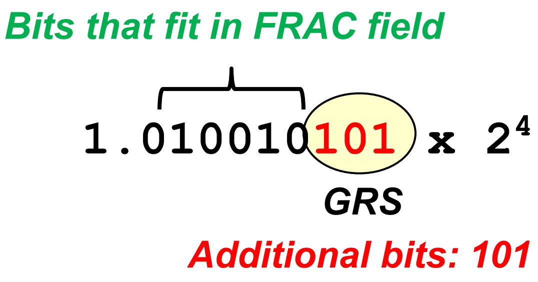
Rounding to Nearest, Base 10

- Same idea as rounding in decimal
- Round 1.23xx to the nearest 1/100th
  - **1.23**51 to **1.23**99 ⇒ round up to **1.24**
  - **1.23**01 to **1.23**49 ⇒ round down to **1.23**
  - **1.23**50 ⇒ Rounding options 1.23 or 1.24
    - Choose the option with an **even digit** in the LS place (i.e., **1.24**)
  - **1.24**50 ⇒ Rounding options 1.24 or 1.25
    - Choose the option with an **even digit** in the LS place (i.e., **1.24**)
- Which option has the even digit is essentially a 50-50 probability of leading to rounding up vs. rounding down
  - Attempt to reduce bias in a sequence of operations

Rounding to Nearest, Base 2

- What does "exactly" half-way correspond to in binary (i.e., 0.5 dec. = ??)
- Hardware will keep some additional bits beyond what can be stored to help with rounding
  - Guard bits, Round bit, and Sticky bit (GRS)
- Thus, if the additional bits are:
  - **10…0** = Exactly half way (**round to even**) $(10.10000)_2$ is $(2.5)_{10}$ rounded to 2
  - **1x…x** = More than half way (**round up**) $(10.10010)_2$ is $(2.5 + 1/16)_{10}$ rounded to 3
  - **0x…x** = Less than half way (**round down**) $(10.00010)_2$ is $(2 + 1/16)_{10}$ rounded to 2

`0.5 = 0. 1 0 0`



Bits that fit in FRAC field

1.010010101×2^4

GRS

Additional bits: 101

Round to Nearest, Base 2

$1.001100110 \text{ x } 2^{4}$

Additional bits: 110

⇩

Round up (fraction + 1)

0	10011	001101

$1.111111101 \text{ x } 2^{4}$

Additional bits: 101

⇩

Round up (fraction + 1)

$$\begin{array}{rl} & 1.111111 \text{ x } 2^{4} \\ + & 0.000001 \text{ x } 2^{4} \\ \hline & 10.000000 \text{ x } 2^{4} \\ & 1.000000 \text{ x } 2^{5} \end{array}$$

Requires renormalization

0	10100	000000

$1.001101001 \text{ x } 2^{4}$

Additional bits: 001

⇩

Leave fraction

0	10011	001101

Round to Nearest: Halfway Case

- In all these cases, the numbers are halfway between the 2 round values
- Thus, we round **to the value with 0 in the LSB**

1.001100100 x 2^4

Additional bits: 100

Rounding options are:
1.001100 or 1.001101

In this case, round down

0	10011	001100

1.111111100 x 2^4

Additional bits: 100

Rounding options are:
1.111111 or 10.000000

In this case, round up

$$\begin{array}{rl} & 1.111111 \text{ x } 2^4 \\ + & 0.000001 \text{ x } 2^4 \\ \hline & 10.000000 \text{ x } 2^4 \\ & 1.000000 \text{ x } 2^5 \end{array}$$

Requires renormalization

0	10100	000000

1.001101100 x 2^4

Additional bits: 100

Rounding options are:
1.001101 or 1.001110

In this case, round up

0	10011	001110

Round to 0 (Chopping)

- Simply drop the G,R,S bits and take fraction as is

GRS

1.001100001 x 2^4

drop G,R,S bits

⇩

0	10011	001100

GRS

1.001101101 x 2^4

drop G,R,S bits

⇩

0	10011	001101

GRS

1.001100111 x 2^4

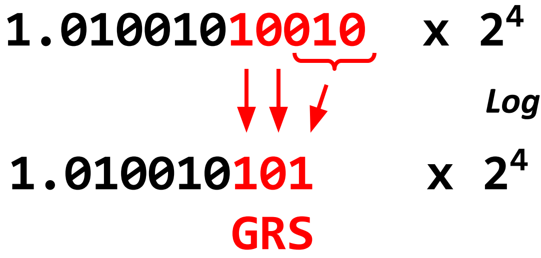
drop G,R,S bits

⇩

0	10011	001100

Rounding Implementation

- There may be a large number of bits after the fraction
- To implement any of the methods we can keep only a subset of the extra bits after the fraction
  - **G**uard bits: bits immediately after LSB of fraction (many HW implementations keep up to 16 additional guard bits)
  - **R**ound bit: bit to the right of the guard bits
  - **S**ticky bit: Logical OR of all other bits after Guard & R bits


1.01001010010 x 2^4

1.010010101 x 2^4

GRS

Logical OR (output is '1' if any input is '1', '0' otherwise

We can perform rounding to a 6-bit fraction using just these 3 bits.

Avoid large + small, or large - large

MAJOR IMPLICATIONS FOR PROGRAMMERS

FP Addition/Subtraction

CS:APP 2.4.5

FP add/sub are not associative! $(a+b)+c \neq a+(b+c)$

- **Rounding**

$$(0.0001 + 98475) - 98474 \neq 0.0001 + (98475-98474)$$
$$98475 - 98474 \neq 0.0001 + 1$$
$$1 \neq 1.0001$$

- **Infinity**

$$1 \quad + \quad 1.11\ldots1 \times 2^{127} \quad - \quad 1.11\ldots1 \times 2^{127}$$

- **Add similar, small magnitude numbers first**

Catastrophic Cancellation

- $9.999 - 9.998 = 1.000 \times 10^{-3}$ … 4 to 1 significant digits
- **Rearrange formulas!** (A goal of “numerical analysis”)

Floating point MUL/DIV

- Also not associative
- Doesn't distribute over addition
  - $a*(b+c) \neq a*b + a*c$
  - Example:
    - (big1 * big2) / (big3 * big4) ⇒ magnitude overflow on first mul.
    - 1/big3 * 1/big4 * big1 * big2 ⇒ magnitude underflow on first mul.
    - (big1 / big3) * (big2 / big4) ⇒ better
- Note: Careful with integer mul/div in C
  - `F = (9/5)*C + 32`
  - Should be `F = (9*C)/5 + 32`

FP Comparison

- Beware of equality (==) check or even less- or greater-than
- Don't use FP as loop counters
- Common approach to replace equality check
  - Check if difference of two values is within some small epsilon
  - Many questions are raised by this… (what epsilon, what about sign, transitive equality, relative check)?
  - Interesting: Python's `isclose(x,y)` python.org/dev/peps/pep-0485

```
float x = 0.1;
float y = 0.2;
printf("%d\n", x+y == 0.3);  // 0
```

Why does it print 0?

```
int i = 0;
for(double t = 0.0;
    t < 1.0; t += 0.1) {
    printf("%d\n", i++);
}
```

Why does it print 0…10?

```
// better!
int equal(float x, float y,
          float epsilon) {
  return fabs(x-y) < epsilon;
}
```

FP & Compiler Optimizations

- Suppose we want to compute:

```
x = a + b + c;
y = b + c + d;
```

- Can the compiler optimize this as:

```
temp = b + c;
x = a + temp;
y = temp + d;
```

Re: What is acceptable for -ffast-math?
From: Linus Torvalds
"I used -ffast-math myself, when I worked on the quake3 port to Linux…"
https://gcc.gnu.org/ml/gcc/2001-07/msg02150.html

Casting and C

What about cast from long?

Cast	Overflow Possible?	Rounding Possible?	Notes
int to float	No	Yes	float uses 23+1 binary digits
int to double	No	No	double uses 52+1 binary digits
float to double	No	No	more digits for exp and fraction
double to float	Yes	Yes	fewer digits for exp and fraction
float/double to int	Yes	Yes	**Round to 0 is used to truncate fractional values** (i.e., 1.9 ⇒ 1) If overflow, use MAX_NEG int.

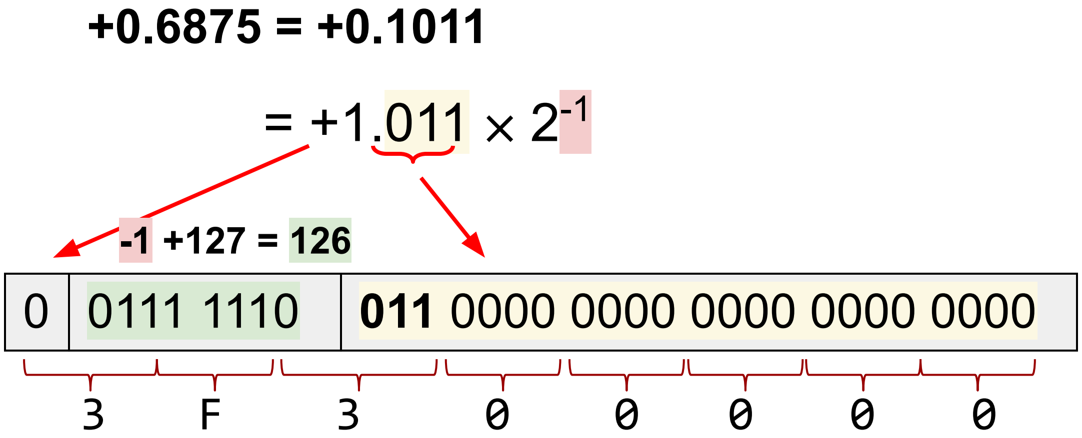
References (in addition to CSAPP)

THE FLOATING-POINT GUIDE
floating-point-gui.de

What Every Computer Scientist Should Know
About Floating-Point Arithmetic
bit.ly/2k8W2cB

Losing My Precision:
Tips For Handling Tricky Floating Point Arithmetic
bit.ly/2m4oH2Y

Hints for DataLab



+0.6875 = +0.1011

$= +1.011 \times 2^{-1}$

-1 +127 = 126

0	0111 1110	**011** 0000 0000 0000 0000 0000

3 F 3 0 0 0 0 0

Stored Value (range of 8-bits shown)	**Excess-127 Value and Special Values**
`255 = 11111111`	`+inf / -inf / NaN`
`254 = 11111110`	`254-127=+127`
…	
`128 = 10000000`	`128-127= +1`
`127 = 01111111`	`127-127= 0`
`126 = 01111110`	`126-127= -1`
…	
`1 = 00000001`	`1-127=-126`
`0 = 00000000`	`+0.0 / -0.0` `0.(frac) x 2^-126`

- How to take the absolute value?
- How to compare without “==” ?
- How to divide by 2 without “/” ?
  - Modify the exponent
  - But denormalized values have all 0’s
  - Then, modify the fraction (may need rounding!)