

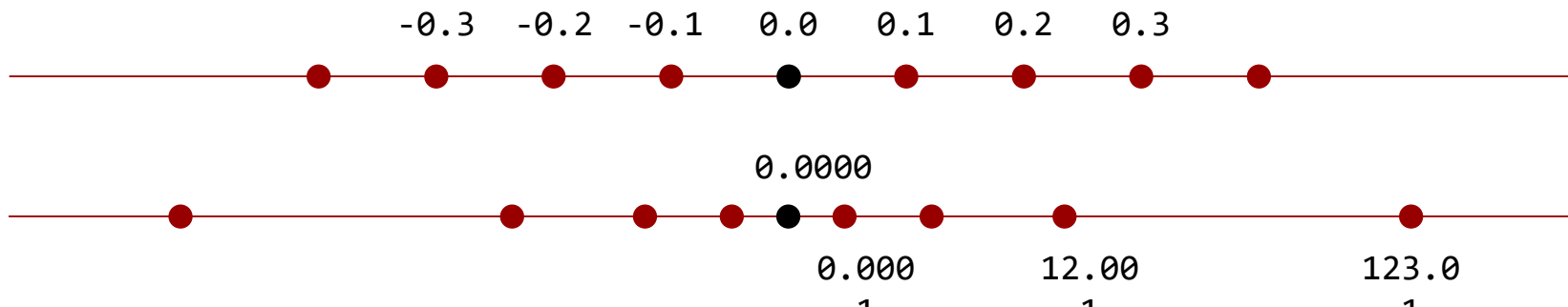
# Unit 3

## IEEE 754 Floating Point Representation

# Floating Point

- Used to represent **very** \_\_\_\_\_ numbers (fractions) and **very** \_\_\_\_\_ 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

\_\_\_\_\_ of combinations given 32 bits, so float must \_\_\_\_\_ 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:**  $[0, 10^6 - 1]$   
**Abs. rounding error**  $\leq 1/2$

**Range:**  $[0, 10^5 - 0.1]$   
**Abs. rounding error**  $\leq 0.1/2$

**Range:**  $[0, 10^3 - 0.001]$   
**Abs. rounding error**  $\leq 0.001/2$

**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

.10000  
.10001  
.10002  
...  
.99998  
.99999

Range:  $[0.1, 10^0 - 0.00001]$   
ABS\_ERR  $\leq 0.00001/2$

If exponent is 0

1.0000  
1.0001  
1.0002  
...  
9.9998  
9.9999

Range:  $[1, 10^1 - 0.0001]$   
ABS\_ERR  $\leq 0.0001/2$

If exponent is 1

10.000  
10.001  
10.002  
...  
99.998  
99.999

Range:  $[10, 10^2 - 0.001]$   
ABS\_ERR  $\leq 0.001/2$

Exponent

To represent positive and negative exponents using 1 decimal digit, we subtract \_\_\_\_\_ from stored digit

- stored digit 0, ..., 9
- exponent \_\_\_\_\_, .., \_\_\_\_\_

Stored as

123459

If exponent is 5

100000. to 999990.  
Range:  $[10^5, 10^6 - 10]$   
ABS\_ERR  $\leq 10/2$

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

# Perils of Floating Point

$$1.2345 \times 10^5$$

123459

$$+ 1.0000 \times 10^{-1}$$

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

```
#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 "\_\_\_\_\_"
- 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<sup>-8</sup>], absolute error of 2<sup>-9</sup>
- **Trade-off:** \_\_\_\_\_ vs absolute representation \_\_\_\_\_
  - Many fraction digits limit the \_\_\_\_\_
  - Few fraction digits increase the representation \_\_\_\_\_

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

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

- ... but using base 2

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

3 fields: \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_

(fraction is also called \_\_\_\_\_ or *significand*)

S	Exp.	Fraction
---	------	----------



# Normalized Floating-Point

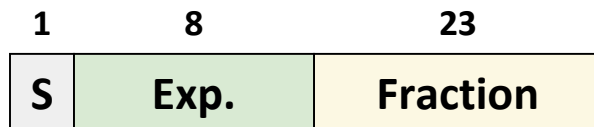
- In decimal
  - $+0.754 \times 10^{15}$  \_\_\_\_\_ scientific notation
  - $+7.54 \times 10^{14}$  \_\_\_\_\_: one significant digit before point
- In binary, the only significant digit is \_\_\_\_\_  
Thus, normalized FP format is:

$$\pm \underline{\hspace{2cm}} \times 2^{\pm \text{exp}}$$

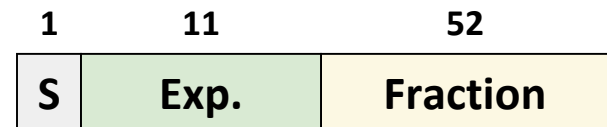
- Floating-point numbers are always normalized: if hardware calculates a result of  $0.001101 \times 2^5$  it must \_\_\_\_\_ to  $1.101000 \times 2^2$  before storing
- The **1.** is actually **not** \_\_\_\_\_ since we always will store normalized numbers

# IEEE 754 Floating Point Formats

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



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



# Excess-N Exponent Representation

- Exponent needs its own sign (+/-)
- Use **Excess-N** instead of 2's complement
  - $w$ -bit exponent  $\Rightarrow$  Excess- $(2^{w-1}-1)$  encoding
  - **float**: 8-bit exponent  $\Rightarrow$  Excess-127
  - **double**: 11-bit exponent  $\Rightarrow$  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**
  - $\dots \times 2^1 \Rightarrow$  stored value  $(1+127)_{10} = 1000\ 0000_2$
- For double-precision, **N=1023**
  - $\dots \times 2^{-2} \Rightarrow$  stored value  $(-2 + 1023)_{10} = (011\ 1111\ 1101)_2$

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

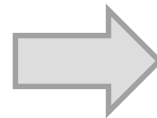
Comparison of  
2's comp. & Excess-N

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?  
 $0.9999 \times 2^2$  or  $1.0000 \times 2^1$
  - A: We should **look at the** \_\_\_\_\_ 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 \_\_\_\_\_ numbers)

0	10000010	0000001000
0	10000001	1110000000



0100000100000001000

0100000011110000000

< > = ???

# 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	_____
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	_____

# IEEE Exponent Special Values

Exp. Field	Fraction Field	Meaning
000...00	0000...0000	
	Non-Zero	
111...11	0000...0000	
	Non-Zero	

# Transition to denormalized

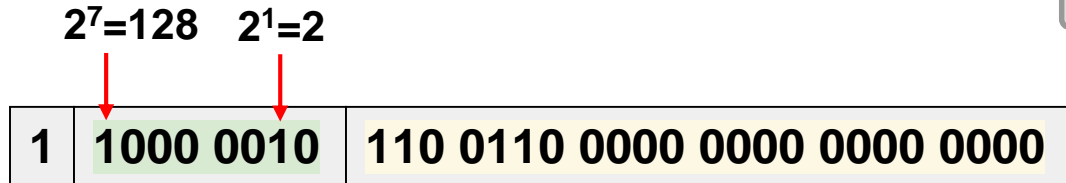
- When the exponent is all 0's and the fraction is nonzero, the number is denormalized
  - An implicit **0.**(fraction) is assumed
  - The exponent value **-126** is used, which is the same excess-127 value of an exponent field equal to 1
- This produces a smooth transition from normalized to denormalized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

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

# Single-Precision Examples

CS:APP 2.4.3

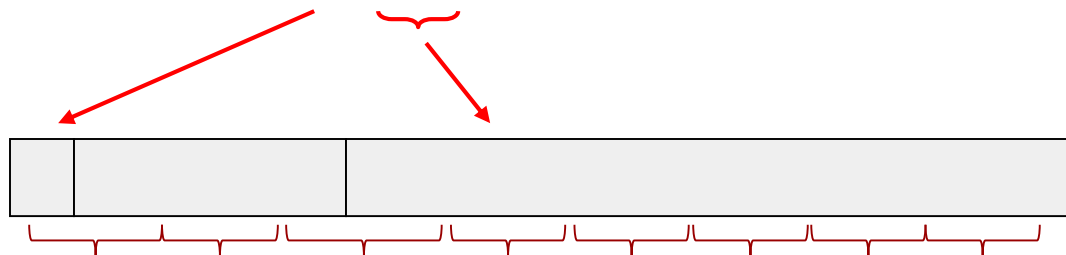
1



=  
=

2

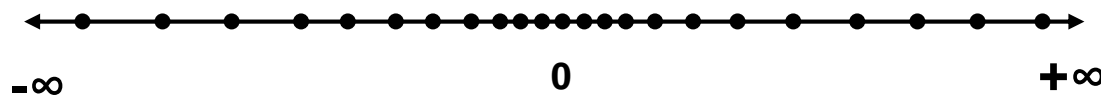
**+0.6875 = +0.1011**





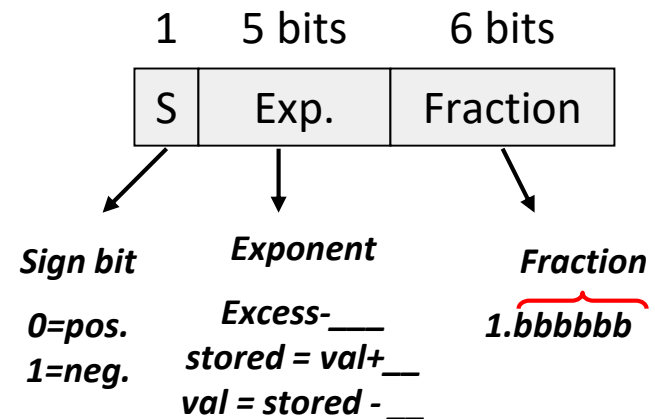
# Floating Point vs. Fixed Point

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



# 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-\_\_\_\_)
    - Same reserved codes
  - 6 fraction bits



# Examples

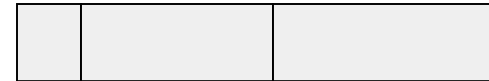
① 

1	10100	101101
---	-------	--------

  
 20

=  
 =

②  $+21.75 = +10101.11$



③ 

1	01101	100000
---	-------	--------

=  
 =

④  $+3.625 = +11.101$

$= +1.110100 \times 2^1$

↙ **1+15=16** ↘

0	10000	110100
---	-------	--------

# ROUNDING

# The Need To Round

CS:APP 2.4.4

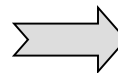
- Integer to FP

– +725 = 1011010101 = **1.011010101** × 2<sup>9</sup>

- If we only have **6 fraction bits**, we can't keep all fraction bits

- FP ADD / SUB

$$\begin{array}{r} 5.9375 \times 10^1 \\ + 2.3256 \times 10^5 \\ \hline \end{array}$$



$$\begin{array}{r} .00059375 \times 10^5 \\ + 2.3256 \times 10^5 \\ \hline \end{array}$$

- FP MUL / DIV

$$\begin{array}{r} 1.010110 \\ * 1.110101 \\ \hline 10.011101001110 \end{array}$$

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

Make sure to move the binary point →

# Rounding Methods

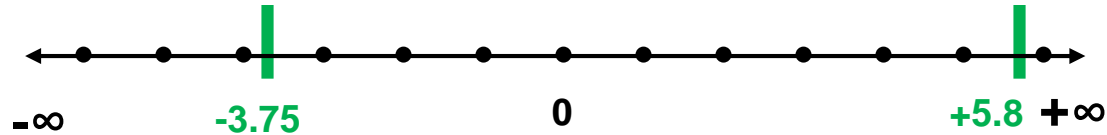
- Methods of Rounding (you are only responsible for the first 2)

<p><b>Round to _____,</b>  <b>Half to Even</b></p>	<p>Round to the nearest representable number.                  If exactly halfway between, round to representable value with _____ (i.e., nearest ____ fraction).</p>
<p><b>Round towards _____</b>  <b>(Chopping)</b></p>	<p>Round the representable value closest to but not greater <i>in magnitude</i> than the precise value.                  Equivalent to just _____ <b>the extra bits.</b></p>
<p>Round toward _____                  (Round Up / Ceiling)</p>	<p>Round to the closest representable value greater than the number</p>
<p>Round toward _____                  (Round Down / Floor)</p>	<p>Round to the closest representable value less than the number</p>

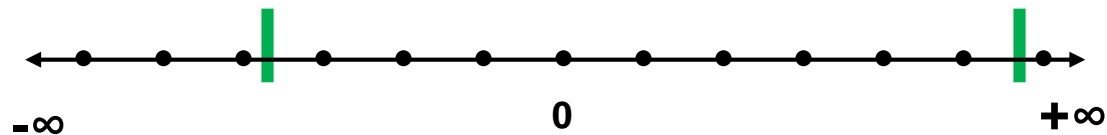
# Number Line View Of Rounding Methods

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

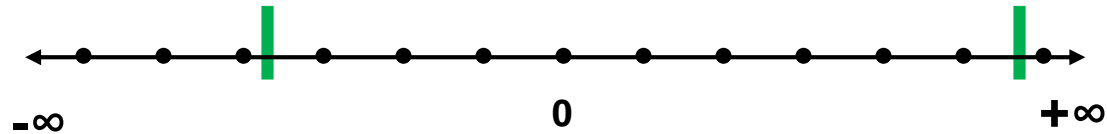
**Round to Nearest**



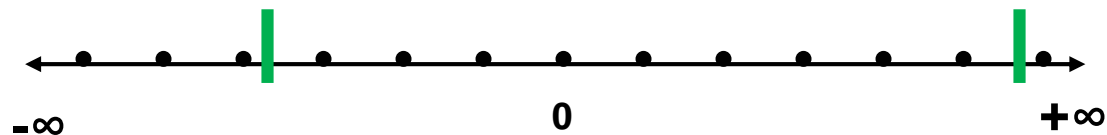
**Round to Zero**



**Round to +Infinity**



**Round to -Infinity**



# ... and many more!

RoundingMode (Java SE 12 & J x +

docs.oracle.com/en/java/javase/12/docs/api/java.base/java/math/RoundingMode.html

OVERVIEW MODULE PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP Java SE 12 & JDK 12

SUMMARY: NESTED | ENUM CONSTANTS | FIELD | METHOD DETAIL: ENUM CONSTANTS | FIELD | METHOD

SEARCH:  X

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  $1/100^{\text{th}}$ 
  - **1.2351** to **1.2399**  $\Rightarrow$  round up to **1.24**
  - **1.2301** to **1.2349**  $\Rightarrow$  round down to **1.23**
  - **1.2350**  $\Rightarrow$  Rounding options 1.23 or 1.24
    - Choose the option with an **even digit** in the LS place (i.e., **1.24**)
  - **1.2450**  $\Rightarrow$  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 \_\_\_\_\_ probability of leading to rounding up vs. rounding down
  - Attempt to \_\_\_\_\_ 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

$$0.5 = \underline{0.} \quad \underline{1} \quad \underline{0} \quad \underline{0}$$

*Bits that fit in FRAC field*

$$1.010010 \overset{\text{GRS}}{\text{101}} \times 2^4$$

***Additional bits: 101***

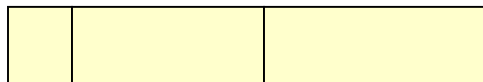
# Round to Nearest, Base 2

$$1.001100110 \times 2^4$$

*Additional bits: 110*



*Round up (fraction + 1)*



$$1.111111101 \times 2^4$$

*Additional bits: 101*



*Round up (fraction + 1)*

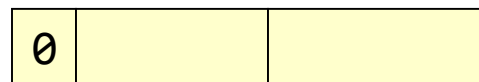
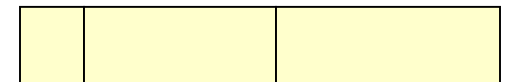


$$1.001101001 \times 2^4$$

*Additional bits: 001*



*Leave fraction*



*Requires renormalization* →

# 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**

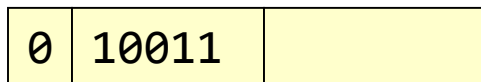
$$1.001100100 \times 2^4$$

Additional bits: 100



Rounding options are:  
 1.001100 or 1.001101

In this case, round \_\_\_\_\_



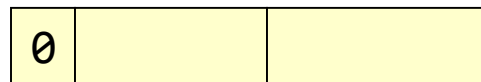
$$1.111111100 \times 2^4$$

Additional bits: 100



Rounding options are:  
 1.111111 or 10.000000

In this case, round \_\_\_\_\_



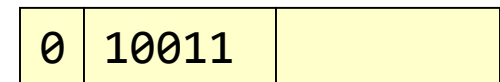
$$1.001101100 \times 2^4$$

Additional bits: 100



Rounding options are:  
 1.001101 or 1.001110

In this case, round \_\_\_\_\_



Requires renormalization

# Round to 0 (Chopping)

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

$$1.\overset{GRS}{001100001} \times 2^4$$

drop G,R,S bits



0	10011	001100
---	-------	--------

$$1.\overset{GRS}{001101101} \times 2^4$$

drop G,R,S bits



0	10011	001101
---	-------	--------

$$1.\overset{GRS}{001100111} \times 2^4$$

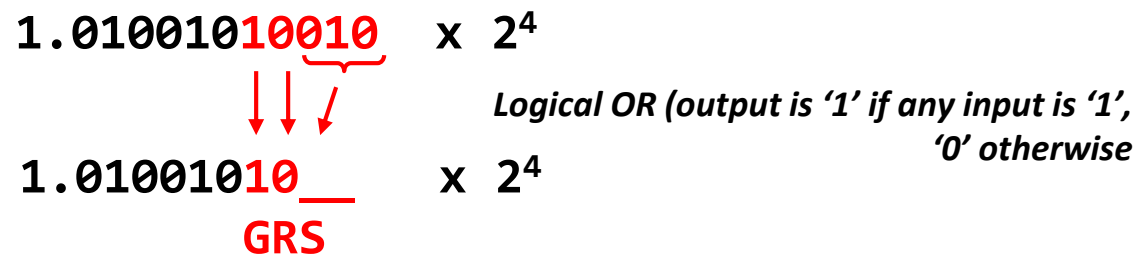
drop G,R,S bits



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
  - **Guard bits:** bits immediately after LSB of fraction (many HW implementations keep up to 16 additional guard bits)
  - **Round bit:** bit to the right of the guard bits
  - **Sticky bit:** \_\_\_\_\_ of all other bits after Guard & R bits



*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 \_\_\_\_\_!  $(a+b)+c \neq a+(b+c)$

- Rounding

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

$$98475 - 98474 \neq 0.0001 + 1$$

$$1 \neq 1.0001$$

- Infinity

$$1 + 1.11\dots1 \times 2^{127} - 1.11\dots1 \times 2^{127}$$

- Add \_\_\_\_\_ 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 \_\_\_\_\_ over addition
  - $a*(b+c) \neq$  \_\_\_\_\_
  - Example:
    - $(big1 * big2) / (big3 * big4) \Rightarrow$  magnitude overflow on first mul.
    - $1/big3 * 1/big4 * big1 * big2 \Rightarrow$  magnitude underflow on first mul.
    - $(big1 / big3) * (big2 / big4) \Rightarrow$  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](https://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:  
 $\text{temp} = b + c;$   
 $x = a + \text{temp};$   
 $y = \text{temp} + d;$

## 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			float uses 23+1 binary digits
int to double			double uses 52+1 binary digits
float to double			more digits for exp and fraction
double to float			fewer digits for exp and fraction
float/double to int			<b>Round to 0 is used to truncate fractional values</b> (i.e., $1.9 \Rightarrow 1$ ) If overflow, use MAX_NEG int.

# References (in addition to CSAPP)

THE FLOATING-POINT GUIDE

[floating-point-gui.de](http://floating-point-gui.de)

What Every Computer Scientist Should Know  
About Floating-Point Arithmetic

[bit.ly/2k8W2cB](http://bit.ly/2k8W2cB)

Losing My Precision:

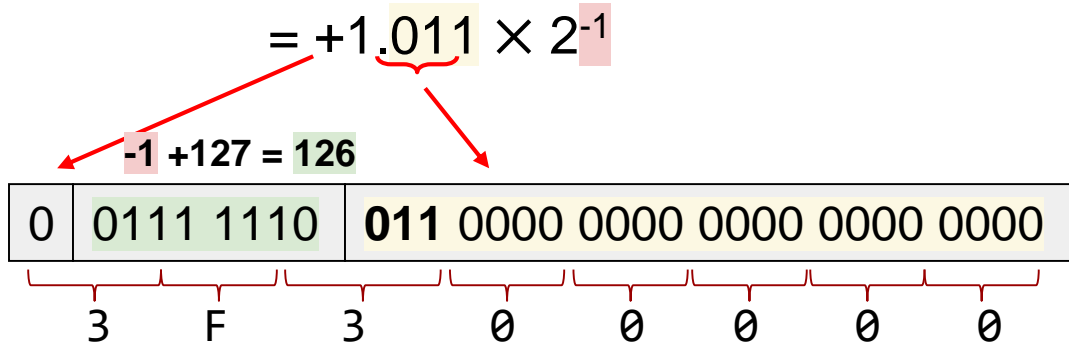
Tips For Handling Tricky Floating Point Arithmetic

[bit.ly/2m4oH2Y](http://bit.ly/2m4oH2Y)

# Hints for DataLab

**+0.6875 = +0.1011**

**= +1.011 × 2<sup>-1</sup>**



- 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!)

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 <sup>-126</sup>