CSC 373 Winter 2020 Prof. Lytinen
Representation of floating point numbers

Reading: Bryant and O'Hallaron, section 2.4

- **Motivation**
  - What is the difference between integer and floating point numbers?
  - How does their representation affect their range (smallest and largest possible values)?
  - When does one use `int`, and when `double` or `float`? What are the advantages/disadvantages of each?

Example of floating point behavior

```
#include <stdio.h>

int main() {
  double x = 1.2;
  printf("x = : %0.50f\n\n", x);
}
```

```
$ gcc -o test test.c ./test
x = 1.19999999999999955591079014993738383054733276367 19
```

Binary floating point

For our purposes, a binary floating point number is of the form
$(0|1)^{+}.(0|1)^{*}$ | $(0|1)^{*}.(0|1)^{+}$
where

| means or
* means zero or more
+ means one or more

Examples of binary floating point numbers:

1.
.1
10.1
11111.0000101

Not binary floating point:

10

Meaning of a binary floating point number

- Each 1 represents a power of 2
- Each digit to the right of the '.' corresponds to a negative power of 2
  - The first digit to the right is 2^{-1} , etc.
- Each digit to the left of '.' is a non-negative power of 2, starting with 2^0
- Simple examples

Binary	Base 10
1000.0	8
0.1	0.5
0.01	0.25

- Example: 1001.101

$2^3 + 2^0 + 2^{-1} + 2^{-3}$
$8 + 1 + .5 + .125 = 9.625$

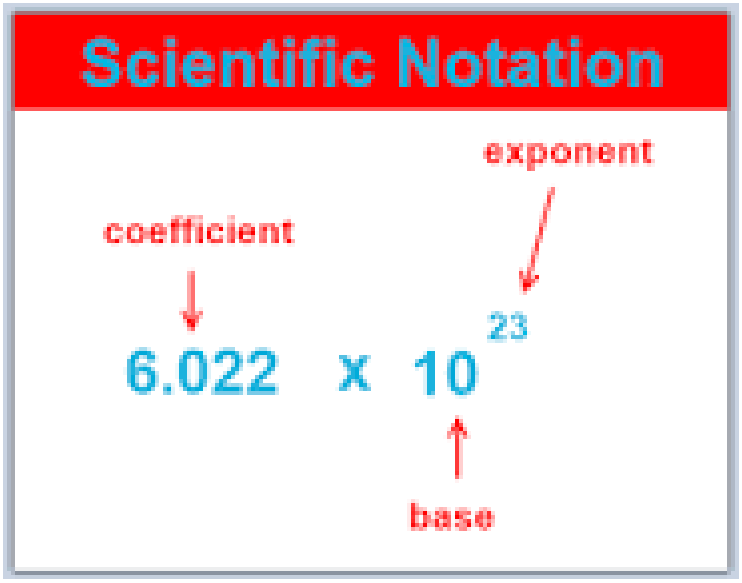
- **Exercise** What is 1011.011_2 in base 10?

Binary floating point	1	0	1	1	.	0	1	1
Each digit represents	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}

$1011.011_2 = 11.375$

Scientific notation

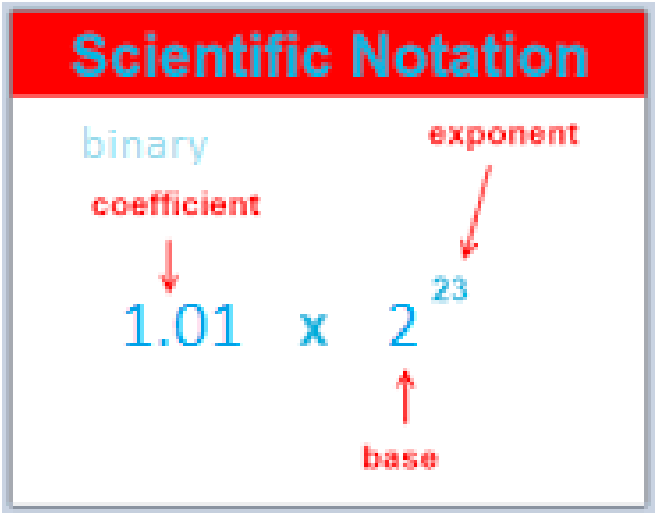
https://www.boundless.com/physics/textbooks/boundless-physics-textbook/the-basics-of-physics-1/significant-figures-and-order-of-magnitude-33/scientific-notation-201-6207/



For base 10:
(Coefficient) * $10^{exponent}$
$1 \leq$ coefficient < 10
(or $-1 \geq$ coefficient > -10
for negative numbers)

Note that 0 must have a special value: $0 * 10^0$

Binary scientific notation



(Coefficient) $* 2^{e}$

e must be an integer
$1_2 \leq$ coefficient $< 10_2$
($1_{10} \leq$ coefficient $< 2_{10}$)

Or for negative numbers:

$-1_2 \geq$ coefficient $> -10_2$

For our purposes, we will still write the exponent in base 10.
Also, a special format is needed to represent 0

$0 * 2^0$

Converting a binary floating point number into binary scientific notation

- The binary floating point number must be non-zero
- Adjust the location of the point so that it appears in the right place (i.e., just to the right of the leftmost 1)
- Adjust the value of the exponent (e) accordingly. If the point has been moved to the left, then e is a positive number. If it's been moved to the right, e is a negative number. If the point has not been moved, then e=0.

- Examples:

$111.01 = 1.1101 * 2^{2}$
$0.00101 = 1.01 * 2^{-3}$
$1.001 = 1.001 * 2^{0}$

Exercises

Binary Floating Point	Binary Scientific Notation
1101.011	1.101011 * 2^3
.01011	$1.011 * 2^{-2}$
-1010100	-1.0101 * 2^6
101.01	$1.0101 * 2^{2}$
1	1 * 2^0

Note: 2s complement is not used in floating point numbers

Exercises

Decimal Fractional	Binary Floating Point	Binary Scientific Notation
1/4	.01	$1.0 * 2^{-2}$
1 7/16	1.0111	1.0111 * 2^0
3/8	0.011	1.1 * 2^-2
5/8	.101	$1.01 * 2^{-1}$
-4 5/16	-100.0101	-1.00101 * 2^2
3 15/16	11.1111	1.11111 * 2^1
-7	-1111	$-1.111 * 2^3$

Note: 2s complement is not used in floating point numbers

IEEE Floating Point Representation (Institute of Electrical and Electronics Engineers)

We modify our definition of binary floating point scientific notation slightly as follows:

$\mathbf{(-1)^s * C * 2^E}$

s indicates whether number is negative or non-negative (1 = negative)

C is a binary floating point number; where $1 \leq C < 10_2$ $(1 \leq C < 2_{10})$.

Note the coefficient cannot be negative, because the sign is indicated by **s**.

Alternate way to represent negative integers

In Binary Scientific Notation, the exponent must be an integer (could be +, -, or 0)

We could use 2s complemement but there is an alternative, which uses a ***Bias.***

Counting: 2s complements vs biased

In general, if we use *n* bits to represent an integer, then the ***bias*** is $2^{n-1}-1$. *Example: 4 bits (bias = 7)*

Number	2s compl	w/bias
7	0111	1110
6	0110	1101
5	0101	1100
4	0100	1011
3	0011	1010
2	0010	1001
1	0001	1000

Number	2s compl	w/bias
0	0000	0111
-1	1111	1110
-2	1110	1101
-3	1101	1000
-4	1100	0111
-5	1101	0110
-6	1110	0001

In biased form, we'll reserve 0000 and 1111 to have special meanings

Counting: 8-bit 2s complements vs biased

In general, if we use *n* bits to represent an integer, then the ***bias*** is $2^{n-1}-1$. For 8-bit numbers, the bias is 2^7-1 (127)

Number	2s compl	w/bias
3	0x03	0x82
2	0x02	0x81
1	0x01	0x80
0	0x00	0x7f
-1	0xff	0x7e
-2	0xfe	0x7d
-3	0xf2	0x7c

In biased form, we'll reserve 0x00 and 0xff to have special meanings

IEEE Encoding

- Leftmost bit indicates sign
- Remaining bits are divided between `exp` and `frac` according to size of datatype
- `exp` = E + the **bias**, which depends on the data size (in bits)
- frac = C - 1
- We can represent a range of large (positive) and small (negative) exponents; for 32-bit, the exponent is between 11111110_2 and 00000001_2 (11111111 and 00000000 have special meanings)

Data type	# bits in s	# bits in exp	bias	# bits in frac
double	1	11	2^{10}-1	52
float	1	8	2^7 -1 (127)	23

Simple example

Represent ½ in IEEE 32-bit floating point

$½ = 0.1_2 = 1.0 * 2^{-1}$

B = 1.0
E = -1

sign = 0
frac = 1.0 – 1 = .0
exp = E + bias = -1 + 127 = 126

s	exp	frac
0	01111110	00000000000000000000000
1 bit	8 bits	23 bits

Simple example

Represent -2.25_{10} in IEEE 32-bit floating point

$-2.25_{10} = -10.01_2 = -1.001 * 2^1$

C = -1.001
E = 1

sign = 1
frac = 1.001 - 1 = .001
exp = E + bias = 1 + 127 = 128

s	exp	frac
1	10000000	00100000000000000000000
1 bit	8 bits	23 bits

Simple example

Represent ½ in IEEE 64-bit floating point

$½ = 0.1_2 = 1.0 * 2^{-1}$

C = 1.0
E = 0

sign = 0
frac = 1.0 – 1 = .0
exp = E – bias = -1 + 1023 = 1022 = 01111111110

s	exp	frac
0	01111111110	0000000000000000000…000
1 bit	11 bits	52 bits

Simple example

Represent -2.5_{10} in IEEE 32-bit and 64-bit floating point

$-2.5_{10} = -10.1_2 = -1.01 * 2^1$

C = -1.001
E = 1

sign = 1
frac = 1.01 – 1 = .01
exp (32-bit) = E + bias = 1 + 127 = 128
exp = E – bias = 1 + 1023 = 1024

32-bit

s	exp	Frac
1	10000000	01000000000000000000…000
1 bit	8 bits	23 bits

64-bit

s	exp	frac
1	10000000000	0100…000
1 bit	11 bits	54 bits

Exercises

Fill in the blanks.

Decimal	Binary Scientific Notation	IEEE 32-bit	IEEE 64-bit
4.5	$1.001 * 2^2$	0x40900000	0x4012000000000000
-2.375	-1.0011 * 2^1	0xc0180000	
	$-1.11 * 2^0$		
		0xbf000000	
			0x3ff4000000000000
-1.625	-1.101 * 2^0	1 01111111 10100000….	0xbfd00000

Exercise solutions

Fill in the blanks.

Decimal	Binary Scientific Notation	IEEE 32-bit	IEEE 64-bit
4.5	$1.001 * 2^2$	0x40900000	0x4012000000000000
-2.25			
	$-1.11 * 2^0$		
		0xbf000000	
			0x3ff4000000000000

Code: print the bits of a 32-bit IEEE floating point (in hex)

```
void print_float(float *fptr) {
  printf("0x");
  void *vptr = (void *) fptr; // pointer to void is typeless
  int i;
  for (i=7; i>=0; i-=1) {
     // shift 4 bits at a time, because 4 bits = 1 hex digit
    int x = (*((int *) vptr) >> 4*i) & 15;
    printf("%x", x);
  }
   printf("\n");
}
```

Range of doubles

Largest number 0 11111111110 11111...1111111111111111

exp = 11111111110 - bias = 01111111111
E = 01111111111 = 1023 = 2^10-1

largest M is approximately 2
largest number = $2 * 2^{1023} = 2^{1024} =$

Largest number is $1.7976931348623159077293051907890 * 10^{308}$

Smallest number 0 00000000001 0000 ... 00000000000000000001 =

Smallest M is 1.00000000000000000001
smallest E is 00000000001 – bias = -0111111110

Smallest number is approximately 2^{-1023}
$= 1.1125369292536006915451163586662 * 10^{-308}$

Denormalized Values

Condition: exp = 000…0

Value

Exponent value E = –Bias + 1

Significand value M = $0.xxx\ldots x_2$

xxx…x: bits of frac

Cases

exp = 000…0, frac = 000…0

Represents value 0

Note distinct values +0 and –0

exp = 000…0, frac != 000…0

Numbers very close to 0.0

Example: 0 00000000 00000000000000000000001 =

$2^{-126} * 2^{-52} = \mathbf{2^{-178} = 2.61012178719940981068411764370 26 * 10^{-54}}$

Rounding

In base 10, not all numbers can be expressed in a finite decimal notation

Example: 1/3 = 0.33333333333333333333333

The same is true for binary
1/5 = ???

Only numbers that are sums of integral powers of 2 (positive or negative) are exactly representable

Other numbers must be rounded

$1/5 = 1/8 + 1/16 + 1/64 + 1/128 + \ldots = 0.001100110011\ldots_2$

IEEE 32-bit representation:

$0.001100110011\ldots = 1.100110011\ldots * 2^{-3}$

32-bit IEEE: Exp = 127 - 3 = 124 = 01111100 Frac = 100110011... 0x3e4ccccd

Rounding Example

```
int main() {
  float frac = .6;
  printf("%.20f\n", frac);
  int i;
  for (i=1; i<10000; i++) {
    frac += (3.+13./(float)i);
  }
  for (i=1; i<10000; i++) {
    frac -= (3.+13./(float)i);
  }
  printf("%.20f\n", frac);
}

[slytinen@cdmlinux ieee]$ ./round
0.60000002384185791016
1.44581007957458496094
```