COMP 122/L Lecture 3

Kyle Dewey

Outline

- Floating point numbers

Question

How might we represent floating point numbers?

```
1.25
47.9
0.82
```

-A lot of different ways possible
-A whole lot of problems related to precision arise. Just about any representation devisable will be complex.

Enter IEEE-754

- Standardized floating point representation and operations
- Modern systems all use this
- Complex and *weird*

Enter IEEE-754

- Standardized floating point representation and operations
- Modern systems all use this
- Complex and *weird*

```
min(X, Y) =? min(Y, X)
```

Enter IEEE-754

- Standardized floating point representation and operations
- Modern systems all use this
- Complex and *weird*

```
min(X, Y) =? min(Y, X)
```

May or may not be true...

-Standard doesn’t enforce that this is true in general. Implementations are permitted to make it so this isn’t true in all cases.

Basis

Based on the idea of *scientific notation*

Basis

Based on the idea of *scientific notation*

$4.23 * 10^{7}$ $-8.7 * 10^{2}$ $14.6 * 10^{-5}$ $-9.4 * 10^{-18}$

Basis

Based on the idea of *scientific notation*

$4.23 * 10^{7}$ $-8.7 * 10^{2}$ $14.6 * 10^{-5}$ $-9.4 * 10^{-18}$

Caveat: use powers of two

$1.1 * 2^{-1}$

Basis

Based on the idea of *scientific notation*

$4.23 * 10^{7}$ $-8.7 * 10^{2}$ $14.6 * 10^{-5}$ $-9.4 * 10^{-18}$

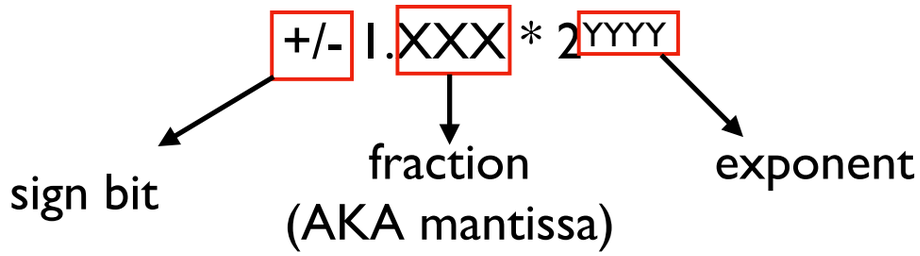
Caveat: use powers of two

$1.1 * 2^{-1}$

Additional caveat: numbers are always in the form:

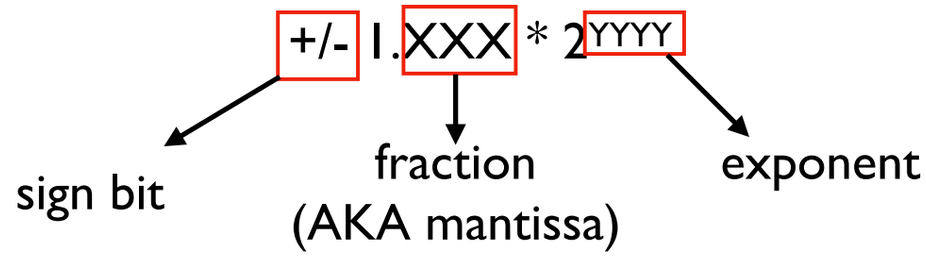
$+/- 1.XXX * 2^{YYYY}$

Components



-Image from: https://en.wikipedia.org/wiki/IEEE_754

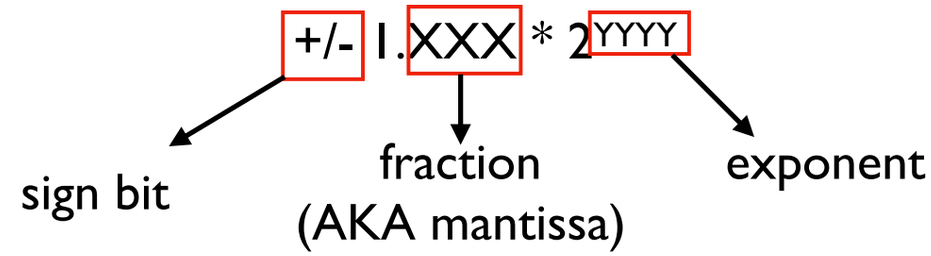
Components



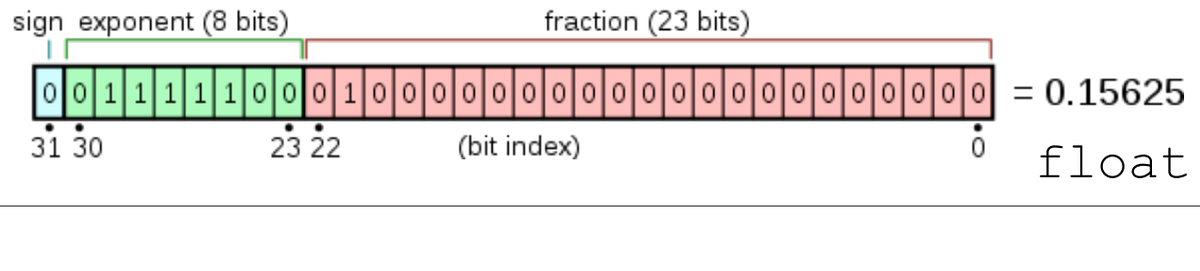
- More bits for fraction and exponent mean greater range and precision
- Most common: 32 bits (binary32 / `float`) and 64 bits (binary64 / `double`)

-Image from: https://en.wikipedia.org/wiki/IEEE_754

Components

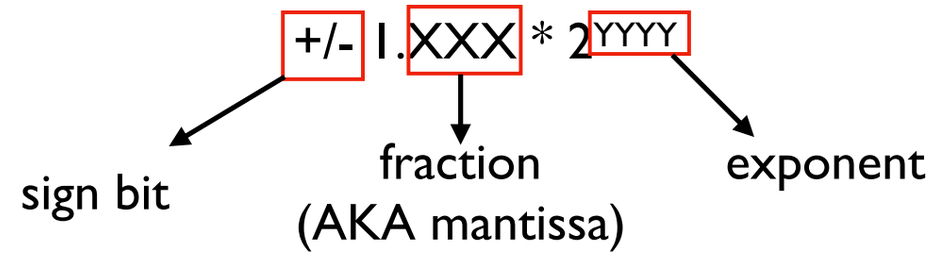


- More bits for fraction and exponent mean greater range and precision
- Most common: 32 bits (binary32 / `float`) and 64 bits (binary64 / `double`)



-Image from: https://en.wikipedia.org/wiki/IEEE_754

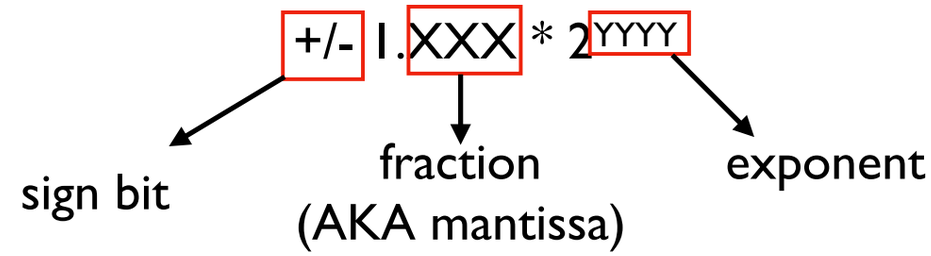
Question



Standard says:

Why this order?

Question



Standard says:

Why this order?

(Usually) preserves order even if compared as a two's complement integer

Sign Bit

0 = positive; 1 = negative

Question: What about 0?

Sign Bit

0 = positive; 1 = negative

Question: What about 0?

Both positive and negative zero (quirk).

Fraction Value

Recall: each bit for integers represents a power of two:

1001

1	0	0	1
1 * 2^3	0 * 2^2	0 * 2^1	1 * 2^0
8	0	0	1

= 9

Same idea for fractional part, but with *negative exponents:*

XXXX.0110

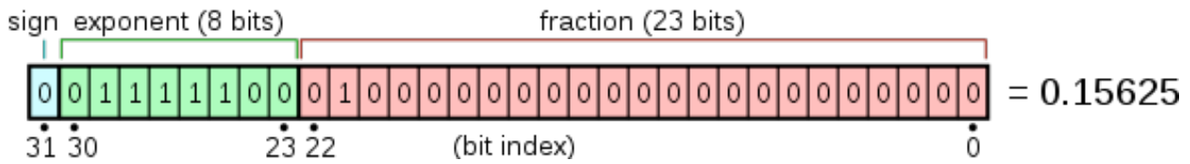
0	1	1	0
0 * 2^-1	1 * 2^-2	1 * 2^-3	1 * 2^-4
0	0.25	0.125	0

= 0.375

Exponent Value

- Always written as an unsigned number, *even for negative exponents*
- Uses a *biased representation*: the actual exponent is always (written exponent - 127)
  - If written exponent is 120, the actual exponent is (120 - 127) = -7

Putting it All Together



- Sign: 0 (positive)
- Exponent: 4 + 8 + 16 + 32 + 64 = 124; 124 - 127 = -3
- Fraction: $0 * 2^{-1} + 1 * 2^{-2} = 0.25$
- Overall magnitude: $(1 + 0.25) * 2^{-3} = 0.15625$

The 1 is implicit in the encoding

+/- $1.XXX * 2^{YYYY}$

Decimal Floating-point to Binary Floating-point

Floating-point Conversion

- Basic idea: determine the correct sign bit, exponent, and fractional part to use, and stitch them together
- Eight-step algorithm for this
- Running example: -9.5625

Step 1: Determine Sign Bit

- -9.5625 is negative
- Sign bit is 1 for negative values

Step 2: Convert Integral Part to Unsigned Binary

- -9.5625's integral portion is 9
- 9 = 1001
- No need to add padding or anything else (yet)

Step 3: Convert Fractional Part to Binary

- -9.5625's fractional portion is 0.5625
- Determine which negative powers of two (2^{-1}, 2^{-2}, 2^{-3}, ...) will sum up to this number, or at least as closely as possible to this number
- Pseudocode algorithm on next side can be used for this

```
fraction = 0.5625
num_iterations = 0
bits = ""
while fraction != 0 and
      num_iterations < 23:
  fraction *= 2
  num_iterations++
  if fraction >= 1.0:
    bits += "1"
    fraction -= 1.0
  else:
    bits += "0"
```

Step 3: Algorithm with Example

0.5625

Iteration	Calculation	>= 1.0?	Output Bit
1	0.5625 * 2 = 1.125	yes	1
2	0.125 * 2 = 0.25	no	0
3	0.25 * 2 = 0.5	no	0
4	0.5 * 2 = 1.0	yes	1

Step 4: Normalize Value

- Recall: the encoding assumes numbers are always in the format +/- 1.XXX * 2^{YYYY}
- We need to put the number in this form
- Integral part (step 2): 1001
- Fractional part (step 3): 1001
- Number overall: integral.fractional: 1001.1001

Step 4: Normalize Value

- To get 1001.1001 into the expected 1.XXX * 2^{YYYY} format, we need to move the dot to the left 3 positions
- Moves to the left denote positive exponents, moves to the right are negative
- Overall: exponent: 3

Step 5: Add Bias to Exponent

- Exponent (from step 4): 3
- Recall: exponents are stored in biased form, and we will always subtract 127 from this value **later**
- ...so here, we always **add** 127: 3 + 127 = 130

Step 6: Convert Biased Exponent to Binary

- Biased exponent (step 5): 130
- 130 in binary: 1000 0010

Step 7: Determine Final Mantissa Bits

- Needed: exactly 23 mantissa bits
- From step 4, we initially had 1001.1001, then moved the dot to the left to get 1.0011001
- First bits of mantissa here will thus be 0011001

Step 7: Remaining Mantissa Bits

- Initial bits: 0011001
- If algorithm in step 3 terminated with 1.0, the number is getting represented exactly, with no precision loss. Pad zeros on right until 23 bits.
    - May need additional algorithm iterations if some precision loss happened
- Depending on how exponent moved, some bits on the right may also need to be removed (precision loss)

Step 7: Remaining Mantissa Bits

- Initial bits: 0011001
- Algorithm terminated exactly, so we can pad with zeros. Final mantissa:
  - 0011 0010 0000 0000 0000 000

Step 8: Combine Everything

- Sign bit (step 1): 1
- Exponent bits (step 6): 1000 0010
- Mantissa bits (step 7): 0011 0010 0000 0000 0000 000
- Overall: 1 1000 0010 0011 0010 0000 0000 0000 000 (copy/pasting all components)
    - Or: 1100 0001 0001 1001 0000 0000 0000 0000 (spaces every nibble)

Further Examples / Explanation

- There is also a link named "Instructions for Converting Between Decimal and Binary Floating-Point Numbers" linked off of the course website
    - Covers the same thing, but says it a little differently, and has additional examples and links