CSD1100

IEEE754

Vadim Surov

So what is the Problem?

Given the following binary representation:

$37.25_{10} = 100101.01_2$

$7.625_{10} = 111.101_2$

$0.3125_{10} = 0.0101_2$

How we can represent the whole and fraction part of the binary rep. in 4 bytes?

Solution is Normalization

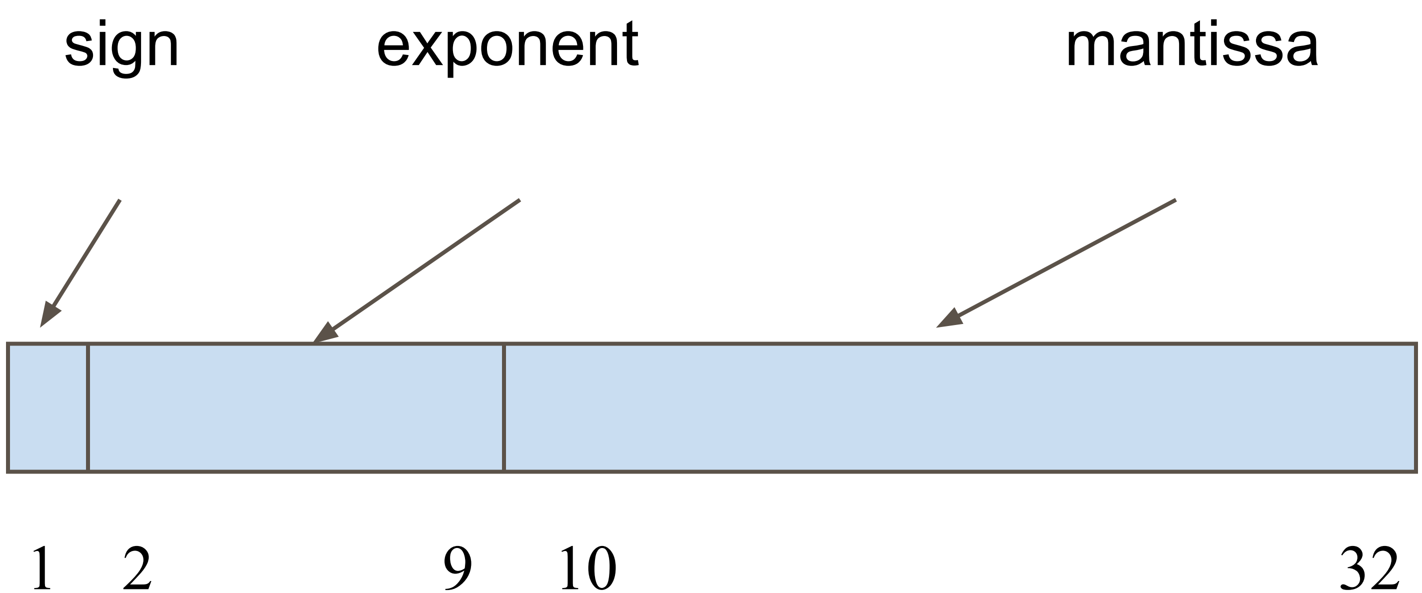
- Every binary number, **except the one corresponding to the number zero**, can be normalized by choosing the exponent so that the radix point falls to the right of the leftmost 1 bit.
- Ex:
  - $37.25_{10} = 100101.01_2 = 1.0010101 \times 2^5$
  - $7.625_{10} = 111.101_2 = 1.11101 \times 2^2$
  - $0.3125_{10} = 0.0101_2 = 1.01 \times 2^{-2}$
- After normalizing, the numbers now have different mantissas and exponents.

IEEE 754

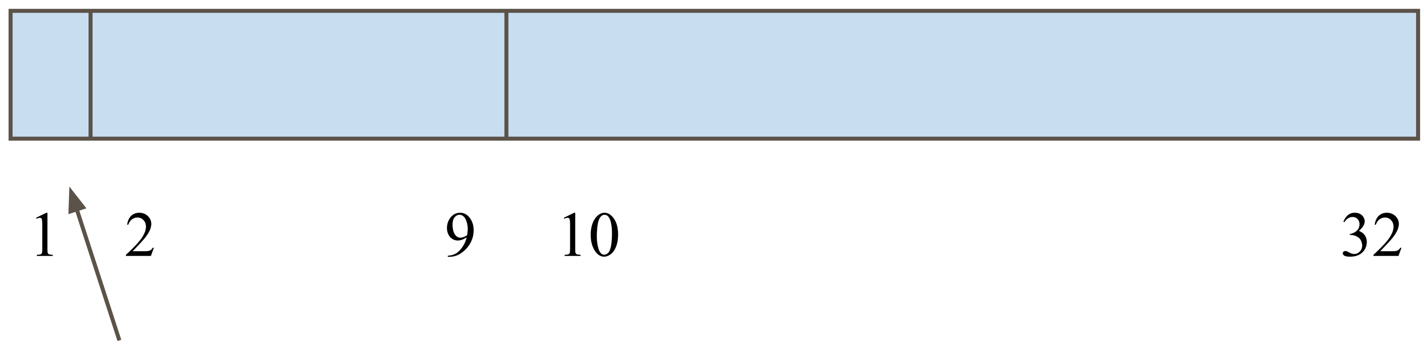
- Established in 1985 as uniform standard for floating point arithmetic
- Supported by all major CPUs
- Variations
  - Single precision: 8-bit exponent, 23-bit mantissa
    - 32 bits total
  - Double precision: 11 bits, 52 bits
    - 64 bits total

IEEE 754

- Floating point numbers can be represented by binary codes by dividing them into three parts:

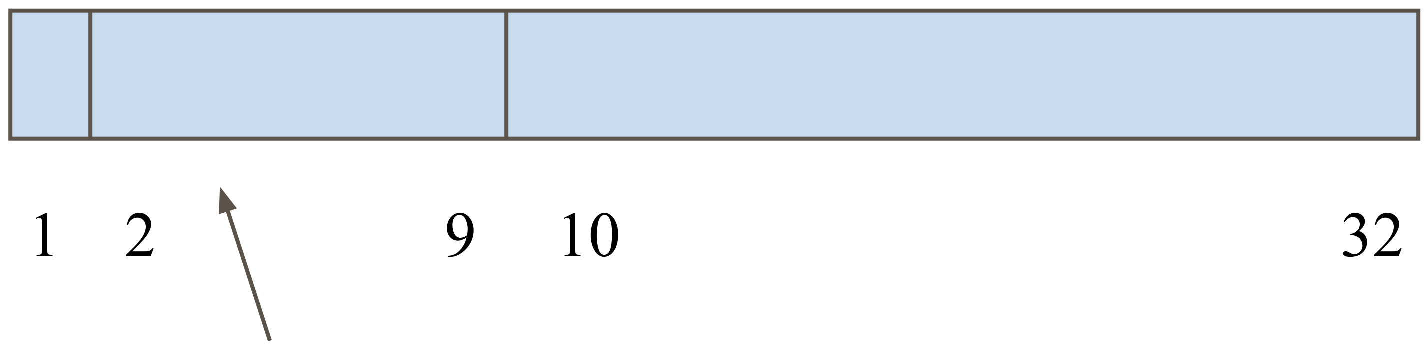


IEEE 754. Sign



- The first, or leftmost, field of our floating point representation will be the sign bit:
  - 0 for a positive number,
  - 1 for a negative number.

IEEE 754. Exponent



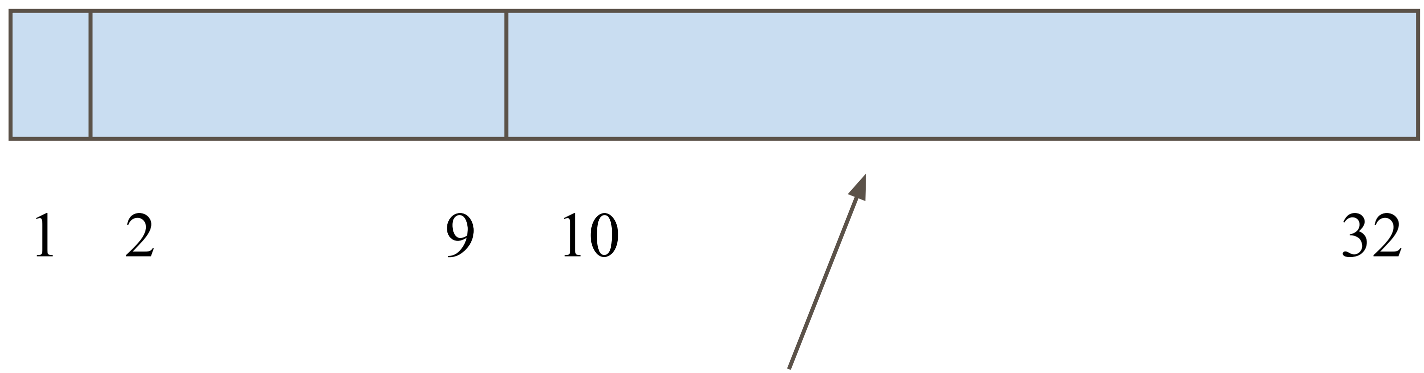
- Exponent in 8-bit field
- Since we must be able to represent both positive and negative exponents, we will use a convention which uses a value known as a **bias of 127 (**or **excess 127)** to determine the representation of the exponent.
  - $127 = 2^{8-1}-1$, where 8 is the number of bits

IEEE 754. Exponent

- Examples:
  - An exponent of 5 is stored as 127 + 5 or 132
  - An exponent of -5 is stored as 127 + (-5) = 122

- The biased exponent, the value actually stored, will range from 0 through 255. This is the range of values that can be represented by 8-bit, unsigned binary numbers.

IEEE 754. Mantissa



- Mantissa in 23-bit field
- The mantissa is the set of 0's and 1's to the left of the radix point of the **normalized** binary number.
  - Ex: if 1.00101 X 2^3 then mantissa is 00101
  - First 1 (as in 1.00101) is always there, so we don't need to waste one of our precious bits on it. **Just assume it is always there.**

Special case: Zero

- The number zero is represented specially:
  - Sign bit is 0 for positive zero, 1 for negative zero.
  - Exponent bits are all 0.
  - Mantissa bits are all 0.

Special case: Infinity

- Positive and negative infinities are represented thus:
  - sign = 0 for positive infinity, 1 for negative infinity.
  - Exponent bits are all 1.
  - Mantissa bits are all 0.

Special case: NaN

- Some operations of floating-point arithmetic are invalid, such as taking the square root of a negative number.
- The act of reaching an invalid result is called a floating-point exception. An exceptional result is represented by a special code called a NaN, for "Not a Number".
- All NaNs in IEEE 754 have this format:
    - Sign bit is either 0 or 1.
    - Exponent its are all 1.
    - Mantissa bits are anything except all 0 bits (since all 0 bits represents infinity).

Double Precision

- Double precision uses more space, allows greater magnitude and greater precision.
- Other than that, it behaves just like single precision.
- We will use only single precision in following examples, but any could easily be expanded to double precision.

DEC to IEEE: 40.15625

Step 1.

Compute the binary equivalent of the whole part and the fractional part

40 = 1*32+0*16+1*8+0*4+0*2+0*1

→ 101000

.15625 = 0*0.5+0*0.25+1*0.125+0*0.0625+1*0.03125

→ .00101

DEC to IEEE: 40.15625

Step 2.

Normalize the number by moving the decimal point to the right of the leftmost one.

$$101000.00101 = 1.0100000101 \times 2^5$$

Step 3.

Convert the exponent to a biased exponent

$$127 + 5 = 132 \quad \Rightarrow \quad 132_{10} = 10000100_2$$

Step 4.

Store the results from above

Sign Exponent Mantissa

0 10000100 01000001010000000000000

DEC to IEEE: –24.75

Step 1. Convert

$24_{10} = 11000_2 \quad .75_{10} = .11_2$

So: $-24.75_{10} = -11000.11_2$

Step 2. Normalize

$-11000.11 = -1.100011 \times 2^4$

Step 3. Convert the exponent to a biased exponent

$127 + 4 = 131 \quad => \quad 131_{10} = 10000011_2$

Step 4. Store the results from above

Sign Exponent Mantissa

1 10000011 10001100000000000000000

IEEE to DEC

- Do the steps in reverse order
- In reversing the normalization step move the radix point the number of digits equal to the exponent.
  - If exponent is positive move to the right,
  - if negative move to the left.

IEEE to DEC:

1 01111101 01000000000000000000000

Step 1

Extract unbias exponent

Biased exponent: $01111101_2 = 125_{10}$

Unbias exponent: $125 - 127 = -2$

Step 2

Write normalized number

-1.01×2^{-2}

IEEE to DEC:
1 01111101 01000000000000000000000

Step 3:

Write the binary number (denormalize value from step 2)

-0.0101_2

Step 4:

Convert binary number to floating-point equivalent

$-0.0101_2 = -(0.25 + 0.0625) = -0.3125$

IEEE to DEC:

0 10000011 11010100000000000000000

Step 1

Extract exponent (unbias exponent)

biased exponent = 10000011 = 131

exponent: 131 - 127 = 4

Step 2

Write Normalized number

1.110101×2^{4}

IEEE to DEC:
0 10000011 11010100000000000000000

Step 3:

Write the binary number (denormalize value from step 2)

$$11101.01_2$$

Step 4:

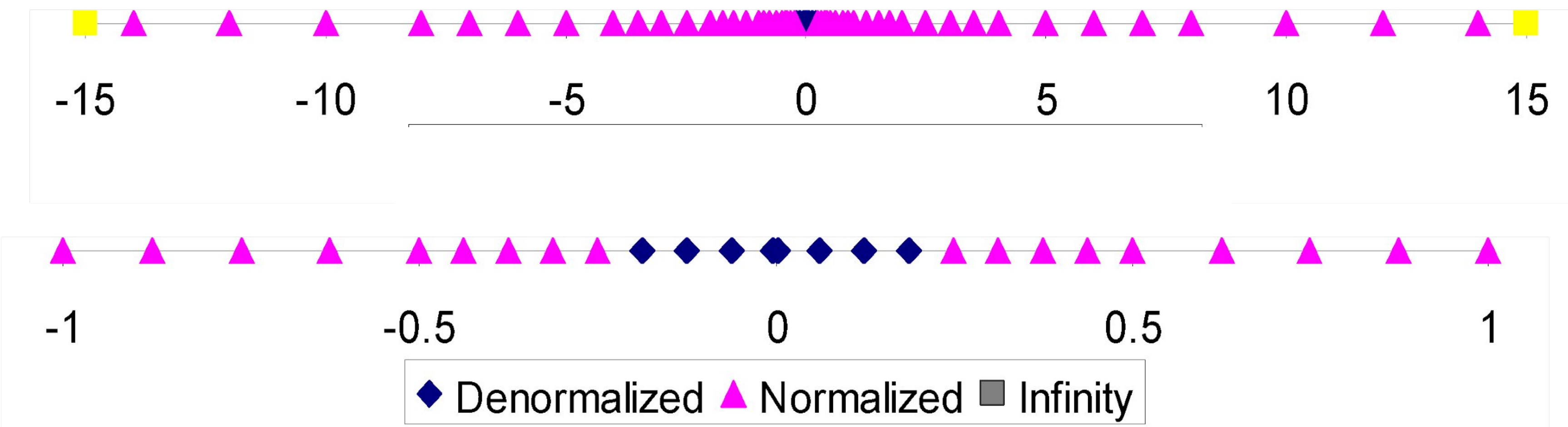
Convert binary number to floating point equivalent (add column values)

$$11101.01_2 = 16 + 8 + 4 + 1 + 0.25 = 29.25_{10}$$

Distribution of Values

- 6-bit IEEE-like format
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3 = 2^{4-1}-1

- Notice how the distribution gets denser toward zero.



Denormalized Values

- Normalized means represented in the normal, or standard, notation. Some numbers do not fit into that scheme and have a separate definition.
- Consider the smallest normalized value: 1.000...000 x 2^{-126}
  - How would we represent half of that number?
  - 1.000...000 x 2^{-127} (But we cannot fit 127 into the exponent field, bcs result is all 0, so represents 0)
  - 0.100...000 x 2^{-126} (But we are stuck with that implied 1 before the implied point)
- So, there are a lot of potentially useful values that don't fit into the scheme. The solution: special rules when the exponent has value 0 (which represents -127).

Rounding

- The general rule when rounding binary fractions to the n-th place prescribes to check the digit following the n-th place in the number.
- If it's 0, then the number should always be rounded down.
- If, instead, the digit is 1 and any of the following digits are also 1, then the number should be rounded up.
- If, however, all of the following digits are 0's, then a tie breaking rule must be applied and usually it's the 'ties to even'. This rule says that we should round to the number that has 0 at the n-th place.

Rounding

- Let’s round some numbers to 2 places
  - 0.11001 — rounds down to 0.11, because the digit at the 3-rd place is 0
  - 0.11101 — rounds up to 1.00, because the digit at the 3-rd place is 1 and there are following digits of 1 (5-th place)
  - 0.11100—apply the ‘ties to even’ tie breaker rule and round up because the digit at 3-rd place is 1 and the following digits are all 0's.

Summary

Item	Single precision	Double precision
Bits in sign	1	1
Bits in exponent	8	11
Bits in fraction	23	52
Bits total	32	64
Exponent system	Excess 127	Excess 1023
Exponent Range	-126 to +127	-1022 to 1023
Smallest normalized number	2^{-126}	2^{-1022}
Largest normalized number	approx. 2^{128}	approx. 2^{1024}
Decimal range	approx. 10^{-38} to 10^{38}	approx. 10^{-308} to 10^{308}
Smallest denormalized number	approx. 10^{-45}	approx. 10^{-324}

Floating Point Puzzles

- Given

```
int x = …;
float f = …;
double d = …;
```

- Assume neither d nor f is NaN

- For each of the following C expressions, either:
    - Argue that it is true for all argument values
    - Explain why not true

Floating Point Puzzles

```
 1. x == (int)(float) x
 2. x == (int)(double) x
 3. f == (float)(double) f
 4. d == (float) d
 5. f == -(-f);
 6. 2/3 == 2/3.0
 7. d < 0.0 ⇒ ((d*2) < 0.0)
 8. d > f ⇒ -f > -d
 9. d * d >= 0.0
10. (d+f)-d == f
```

Floating Point Puzzles

1. `x == (int)(float) x`

   No: float has 24 bit significand, int has 31 bit

2. `x == (int)(double) x`

   No: Bigger 53 bit significand, but converting to an integer always rounds down

Answers To Floating Point Puzzles

3. f == (float)(double) f **Yes: increases precision**
4. d == (float) d **No: loses precision**
5. f == -(-f); **Yes: Just change sign bit**
6. 2/3 == 2/3.0 **No: 2/3 == 0**
7. d < 0.0 ⇒ ((d*2) < 0.0) **Yes!**
8. d > f ⇒ -f > -d **Yes!**
9. d * d >= 0.0 **Yes!**
10. (d+f)-d == f **No: Not associative**

Comparison

```
#include <stdio.h>

void main() {
   float a = 1.345f, b = 1.123f, c = a + b;

   if (c == 2.468f)
      printf("They are equal.\n");
   else
      printf("They are not equal!");
}
```

Comparison

```
#include <stdio.h>
#define EPSILON 0.0001  // Define tolerance
int equal(float x, float v) {
   return (((v - EPSILON) < x) &&
            (x < ( v + EPSILON))); }
void main() {
   float a = 1.345f, b = 1.123f, c = a + b;
   if (equal(c, 2.468f))
      printf("They are equal.\n");
   else
      printf("They are not equal!");
}
```

Comparison. Example 2 (C++)

```
double a = 1.1, b = 2.2, c = 3.3;
double sum = a + b;

cout << setprecision(18);
cout << "a   =   " << a   << endl;
cout << "b   =   " << b   << endl;
cout << "sum =   " << sum << endl;
cout << "c   =   " << c   << endl;
cout << "sum - c      = " << sum - c  << endl;
cout << "c - sum      = " << c - sum  << endl;
cout << "abs(c - sum) = " << abs(c - sum) << endl;

const double epsilon = 0.0000001;
if (abs(c - sum) < epsilon)
    cout << "Equal";
else
    cout << "Not equal";
```

```
a   =   1.10000000000000009
b   =   2.20000000000000018
sum =   3.30000000000000027
c   =   3.29999999999999982
sum - c      = 4.44089209850062616e-016
c - sum      = -4.44089209850062616e-016
abs(c - sum) = 4.44089209850062616e-016
Equal
```