Chapter 5 | Floating Points

● Representing Floating Points in Binary ● Constructing Numbers ● Reference Values ● Examples Using 32-bit Floats

CS102
Computer
Environment

Representing floating points in binary

Background

- All floating-point numbers consist of three parts:

1. A *sign* - Indicates whether the number is positive (0 or above) or negative (below zero)
2. An *exponent* - This can be positive (absolute value of the number is above 1) or negative (absolute value of the number is between 0 and 1).
3. A *mantissa* - This is the *precision* of the number (also called the *significant* in our usage). Usually this is the fractional portion of the number after it has been *normalized* in scientific notation.

- Some examples in decimal:

Number	Sign	Mantissa	Exponent
3.763×10^{3}	+	3.763	3
1.2345×10^{-11}	+	1.2345	-11
-4.45×10^{5}	-	4.45	5
-2.6795×10^{-7}	-	2.6795	-7

- Representing the value 12,345 in decimal:

Number	Sign	Mantissa	Exponent
12345×10^{0}	+	12345	0
1234.5×10^{1}	+	1234.5	1
123.45×10^{2}	+	123.45	2
12.345×10^{3}	+	12.345	3
1.2345×10^{4}	**+**	**1.2345**	**4**
$.12345 \times 10^{5}$	+	.12345	5
$.012345 \times 10^{6}$	+	.012345	6

Notes:

- The **bold** row above is the *normalized* value for 12,345. We will use this normalization with binary floating-point numbers.
- In computing, we usually see scientific notation displayed like: -1.2345e004 or something similar.
- All of the necessary information (sign, exponent, mantissa) are present in the number.

IEEE 754

- The IEEE (Institute of Electrical and Electronics Engineers) sets the standard for floating point arithmetic.
- This standard specifies how single precision (32 bit) and double precision (64 bit) floating point numbers are represented and how arithmetic should be carried out on them.

32-bit precision (float type)

bit 31 => Sign bit
bits 23…30 => Exponent
bits 0…22 => Mantissa

Parts:

- ***Sign*** (1 bit) it is like binary integers, a 0 means positive and a 1 means negative.
- ***Mantissa*** (23 bits) is the "fractional" portion of the number.
  - There is an implied leading 1.
  - So the mantissas are actually 24 bits, even though only 23 bits are stored.
  - The mantissa is stored as a binary fraction of the form 1.ddddd.... This fraction has a value greater than or equal to 1 and less than 2.
  - Real numbers are always stored in normalized form
    - That is, the mantissa is left-shifted such that the high-order bit of the mantissa is always 1.
    - Because of this bit is always 1, it is assumed (not stored) in the real*4 (float) and real*8 (double) formats.
- ***Exponent*** (8-bit) It is biased by half of its possible values this is the power of 2 (binary)
  - The exponent is biased by half of the possible value.
  - In a 32-bit precision float type, the exponent is biased by 127.
    - This means you subtract this bias from the stored exponent to get the actual exponent.
    - When the stored exponent is less than the bias, it is actually a negative exponent.
  - These exponents are not powers of ten; they are powers of two. That is, 8-bit stored exponents can be up to 127.
  - To get the actual value,

- We subtract 127 from the stored exponent, and we get the actual exponent.
- Then two is raised to the power value of the actual exponent, and we get the exponent value.
- Then, finally the mantissa multiplied by the exponent value to get the real number.

- Example:

Binary	Actual exponent (decimal)
01111111	0 (127 - 127)
10000000	1 (128 - 127)
10000010	3 (130 - 127)
01111100	-3 (124 - 127)

- Note that you would add 127 to the decimal exponent to get the binary value.
- There is no two's complement issue,
  - So a negative number looks exactly like a positive number
  - with the exception of the sign bit
- Example:

Decimal	Binary
0.65625	0 01111110 01010000000000000000000
-0.65625	1 01111110 01010000000000000000000
0.2	0 01111100 10011001100110011001101
-0.2	1 01111100 10011001100110011001101

Special numbers:

0 00000000 00000000000000000000000 = +0
(Sign is 0, exponent is 0, and mantissa is 0)

1 00000000 00000000000000000000000 = -0
(Sign is 1, exponent is 0, and mantissa is 0)

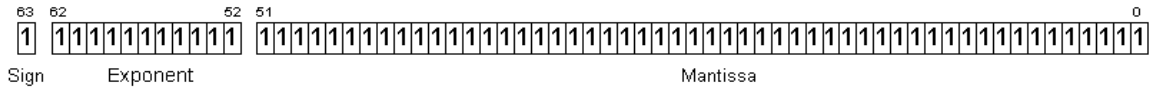
0 11111111 00000000000000000000000 = +Infinity
(Sign is 0, exponent is 255, and mantissa is 0)

1 11111111 00000000000000000000000 = -Infinity
(Sign is 1, exponent is 255, and mantissa is 0)

0 11111111 00000100000000000000000 = NaN
(Sign is 0 or 1, exponent is 255, and mantissa is non-zero)

1 11111111 00100010001001010101010 = NaN
(Sign is 0 or 1, exponent is 255, and mantissa is non-zero)

64-bit double-precision



bit 63 => Sign Bit
bits 52…62 => Exponent
bits 0…51 => Mantissa

- Same as double-precision, except there are more bits, 11 for the exponent and 52 for the mantissa.
- Exponent has a bias of 1023.

Summary:

- The sign bit is 0 for positive, 1 for negative.
- The exponent field is 127 (single-precision) plus the actual exponent or 1023 (double-precision) plus the actual exponent.
- The first bit of the mantissa is an implied 1, so only the fractional portion is represented in the remaining 23 bits (single-precision) or 52 bits (double-precision).

Constructing numbers

- First, we check and update the sign,
  - If sign is negative, we convert the number to positive, and
  - Set bit-0 to 1.
- Second, we need to get the exponent value and normalize the number, so it becomes in the form of 1.ddddddd…,
  - If the number is less than 1 **(float < 1.0f)**:
    - Starting from exponent value equal to 1,
    - We **multiply** the number with 2 to the power exponent,
    - First by 2 then by 4, then by 8 etc…
    - If the result is less than 1.0f
    - We increment the exponent and multiply the number with the new exponent.
    - If the result is not less than 1.0f,
    - We return the exponent value.
  - If the number is greater or equal to 2.0f:
    - Starting from exponent value equal to 1,
    - We **divide** the number with 2 to the power exponent,
    - First by 2 then by 4, then by 8 etc…
    - If the result is greater or equal to 2.0f
    - We increment the exponent and divide the number with the new exponent.
    - If the result is not greater or equal to 2.0f,
    - We return the exponent value.

- Third we calculate the exponent value biased by 127,
  - If the number is less than 1.0f
  - We subtract from 127 the exponent value
  - Otherwise, we add to the exponent 127.
- Forth, we calculate the mantissa value, so it becomes in the form of 0.dddddd…
  - If the number is less than 1.0f
  - The mantissa value is the number **multiplied** by 2 to the power exponent, and the whole value -1
  - Otherwise, the mantissa value is the number **divided** by 2 to the power exponent, and the whole value -1
- Fifth we get the binary value of the mantissa according to the reference table.
- Finally we display the number.

Reference values

Bit Position	Exponent Exponent	Decimal Fraction	Decimal Number
22	$1 / 2^{1}$	1 / 2	0.50000000000000000000000
21	$1 / 2^{2}$	1 / 4	0.25000000000000000000000
20	$1 / 2^{3}$	1 / 8	0.12500000000000000000000
19	$1 / 2^{4}$	1 / 16	0.06250000000000000000000
18	$1 / 2^{5}$	1 / 32	0.03125000000000000000000
17	$1 / 2^{6}$	1 / 64	0.01562500000000000000000
16	$1 / 2^{7}$	1 / 128	0.00781250000000000000000
15	$1 / 2^{8}$	1 / 256	0.00390625000000000000000
14	$1 / 2^{9}$	1 / 512	0.00195312500000000000000
13	$1 / 2^{10}$	1 / 1024	0.00097656250000000000000
12	$1 / 2^{11}$	1 / 2048	0.00048828125000000000000
11	$1 / 2^{12}$	1 / 4096	0.00024414062500000000000
10	$1 / 2^{13}$	1 / 8192	0.00012207031250000000000
9	$1 / 2^{14}$	1 / 16384	0.00006103515625000000000
8	$1 / 2^{15}$	1 / 32768	0.00003051757812500000000
7	$1 / 2^{16}$	1 / 65536	0.00001525878906250000000
6	$1 / 2^{17}$	1 / 131072	0.00000762939453125000000
5	$1 / 2^{18}$	1 / 262144	0.00000381469726562500000
4	$1 / 2^{19}$	1 / 524288	0.00000190734863281250000
3	$1 / 2^{20}$	1 / 1048576	0.00000095367431640625000
2	$1 / 2^{21}$	1 / 2097152	0.00000047683715820312500
1	$1 / 2^{22}$	1 / 4194304	0.00000023841857910156250
0	$1 / 2^{23}$	1 / 8388608	0.00000011920928955078125

Examples using 32-bit floats

Example 1

- Representing decimal numbers in binary works great if the decimal number makes a nice "round" number in binary.
- Most decimal numbers can't be represented exactly in binary.
- The value 0.2 can be represented exactly in decimal, but in binary, it can't.
- The output from the table shows this imperfection clearly, since not all the mantissa of 0.2 which is 0.6 could be represented within 23 bits.
- First, we need to change the format so it becomes 1.dddd….
- If we multiply 0.2 by 2^1 we get 0.4 which is still less than 1.
- If we multiply 0.2 by 2^2 we get 0.8 which is still less than 1.
- If we multiply 0.2 by 2^3 we get 1.6 which is more than 1.
- Now we have the exponent which is 3.
- $0.2 \times 2^3 = 1.6$, and the equivalent value to 0.2 is 1.6×2^{-3}
- The biased exponent value is 127 -3 = 124, since when we subtract 124 from 127 we get -3.
- The mantissa is 1.6, but since the 1 is implied within the IEEE format we subtract one from the mantissa so we get the form of 0.6
- Now according to the reference table we convert the mantissa value to binary by subtracting the maximum value while the number remains positive.

Bit number	Decimal number	Used bits	Mantissa
			0.6000000000
22	0.5000000000	1	0.1000000000
21	0.2500000000	0	0.1000000000
20	0.1250000000	0	0.1000000000
19	0.0625000000	1	0.0375000000
18	0.0312500000	1	0.0062500000
17	0.0156250000	0	0.0062500000
16	0.0078125000	0	0.0062500000
15	0.0039062500	1	0.0023437500
14	0.0019531250	1	0.0003906250
13	0.0009765625	0	0.0003906250
12	0.0004882813	0	0.0003906250
11	0.0002441406	1	0.0001464844
10	0.0001220703	1	0.0000244141
9	0.0000610352	0	0.0000244141
8	0.0000305176	0	0.0000244141
7	0.0000152588	1	0.0000091553
6	0.0000076294	1	0.0000015259
5	0.0000038147	0	0.0000015259
4	0.0000019073	0	0.0000015259
3	0.0000009537	1	0.0000005722
2	0.0000004768	1	0.0000000954
1	0.0000002384	0	0.0000000954
0	0.0000001192	0	0.0000000954

- And the mantissa value is: 10011001100110011001101

- The exponent value is: 01111100
- The sign bit value is: 0
- float = 0.200000
- exp = 3
- mantissa = 0.600000
- IEEE 32-bit representation in binary
- 0 01111100 10011001100110011001101
- IEEE 32-bit representation in Hexadecimal Big Endian format
- 3E 4C CC CD
- IEEE 32-bit representation in Hexadecimal Little Endian format
- CD CC 4C 3E

A programming example where a float loses precision

```
#include <stdio.h>

#define EPSILON    0.0001 // Define your own tolerance
#define FLOAT_EQ(x,v) (((v - EPSILON) < x) && (x < ( v + EPSILON)))

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

     c = a + b;

   // if (FLOAT_EQ(c, 2.468))  // Remove comment for correct result
   if (c == 2.468)        // Comment this line for correct result
         printf("They are equal.\n");
   else
       printf("They are not equal! The value of c is %13.10f,or %f\n",c,c);
}
/*
Output Result
They are not equal! The value of c is 2.4679999352 or 2.468000. */
```

Example 2

- Let's represent 2.5 which will deal with numbers that are greater or equal to 2.
- First, we need to change the format so it becomes 1.dddd….
- If we divide 2.5 by 2^1 we get 1.25 which is less than 2.
- Now we have the exponent which is 1.
- 2.5 / 2^1 =1.25, and the equivalent value to 2.5 is 1.25 x 2^1
- The biased exponent value is 127 + 1 = 128.
- The mantissa is 1.25, but since the 1 is implied within the IEEE format we subtract one from the mantissa so we get the form of 0.25
- Now according to the reference table we convert the mantissa value to binary by subtracting the maximum value while the number remains positive.
- It is clear in this example that the number 2.5 can be represented exactly in binary without a loss of precision.

Bit number	Decimal number	Used bits	Mantissa
			0.2500000000
22	0.5000000000	0	0.2500000000
21	0.2500000000	1	0.0000000000
20	0.1250000000	0	0.0000000000
19	0.0625000000	0	0.0000000000
18	0.0312500000	0	0.0000000000
17	0.0156250000	0	0.0000000000
16	0.0078125000	0	0.0000000000
15	0.0039062500	0	0.0000000000
14	0.0019531250	0	0.0000000000
13	0.0009765625	0	0.0000000000
12	0.0004882813	0	0.0000000000
11	0.0002441406	0	0.0000000000
10	0.0001220703	0	0.0000000000
9	0.0000610352	0	0.0000000000
8	0.0000305176	0	0.0000000000
7	0.0000152588	0	0.0000000000
6	0.0000076294	0	0.0000000000
5	0.0000038147	0	0.0000000000
4	0.0000019073	0	0.0000000000
3	0.0000009537	0	0.0000000000
2	0.0000004768	0	0.0000000000
1	0.0000002384	0	0.0000000000
0	0.0000001192	0	0.0000000000

- And the mantissa value is: 01000000000000000000000
- The exponent value is: 10000000
- The sign bit value is: 0
- float=2.500000
- exp=1 /*-1*/
- mantissa=0.250000
- IEEE 32-bit representation in binary
- 0 10000000 01000000000000000000000
- IEEE 32-bit representation in Hexadecimal Big Endian format
- 40 20 00 00
- IEEE 32-bit representation in Hexadecimal Little Endian format
- 00 00 20 40

Example 3

- float=6.000000
- exp=2 /*-2*/
- mantissa=0.500000
- IEEE 32-bit representation in binary
- 0 10000001 10000000000000000000000
- IEEE 32-bit representation in Hexadecimal Big Endian format
- 40 C0 00 00
- IEEE 32-bit representation in Hexadecimal Little Endian format
- 00 00 C0 40

Example 4

- Float = -4.000000
- exp=2 /*-2*/
- mantissa=0.000000
- IEEE 32-bit representation in binary
- 1 10000001 00000000000000000000000
- IEEE 32-bit representation in Hexadecimal Big Endian format
- 40 80 00 00
- IEEE 32-bit representation in Hexadecimal Little Endian format
- 00 00 80 40