Lecture 3

Representing Data on the Computer

Ramani Duraiswami

AMSC/CMSC 662

Fall 2009

Bits and Bytes; Hexadecimal

- A bit is a single binary digit that can take on one of the two values 0 and 1.
- A byte is a group of eight bits.
- Since a hexadecimal digit (base 16) can be represented by four bits, bytes can be described by pairs of hexadecimal digits.
- 0, 1, 2, 3, 4, 5, 6, 7, 8,
- 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000
- 9, A (10), B(11), C(12), D(13), E(14), F(15)
- 1001, 1010, 1011, 1100, 1101, 1110, 1111
- 01011110_2 may be represented by the number $5E_{16}$,

Words & Addresses

- Memory locations on a 32 bit machine, usually consist of 4 bytes => called a word
- Relationship between words and data of various sizes:
  - byte 8bits, 1 byte
  - short or half word 16bits, 2 bytes
  - word 32bits, 4 bytes
  - long or double word 64 bits, 8 bytes
- Memory is addressed using an index, which is itself a binary number
- Addresses, usually are available for every byte
- Addresses can be grouped by bit-shifts
  - byte xx...xxxx
  - half word xx...xxx0
  - word xx...xx00
  - double word xx...x000
- Recall that words/memory are shipped across a bus
  - Contiguous blocks can be loaded easier

Memory fragmentation

- Usually memory is allocated in chunks of a word or of two words
- If the data, e.g. a C-struct or a Fortran 90 Type may consists of a mixture of a four byte variable, a two byte variable and a four byte variable
- This will cause wastage of two bytes due to memory fragmentation

low order bits of the address	memory
$x \ldots x00$	A
$x \ldots x10$	
$x \ldots x00$	B
$x \ldots x10$	X
$x \ldots x00$	C
$\vdots$	$\vdots$

Little Endian/Big Endian Ordering

- Ordering of bytes in a word
- we could store its bytes in any order, as long as the particular ordering is consistent from word to word.
- In practice, there are only two orders used: big endian and little endian.
- Consider a four byte word ABCD.
  - The byte A is called the leading or most significant byte.
  - The byte D is called the trailing or least signicant byte.
- In big-endian representation the bytes are stored in increasing memory locations beginning with the leading byte.
- In little-endian representation the bytes are stored in increasing memory locations beginning with the trailing byte.
- ABCD is stored as follows in the two methods
- Way of storing is usually unimportant, except for two situations
  - Transferring binary data between machines
  - Using bit shift operations

address	BE	LE
a	A	D
a+1	B	C
a+2	C	B
a+3	D	A

Bit operations

- Very efficient set of operations that are provided in processors, and that have representations in programming languages
- Will return to these in a later class

Unsigned Integers

- On a machine nonnegative integers can be represented by regarding the bits in a word as a binary number, that is, an unsigned integer.
- Integers can be added, subtracted, multiplied, and divided.
- Addition and subtraction are the fastest operations.
- Multiplication can be almost as fast as addition.
- Division is much slower.
- However, multiplication and division by two can be implemented using shifts
- Exceptions
- However, the result of these operations cannot always be represented in the computer.
- $13_{10} + 5_{10} = 1101_2 + 0101_2 = 10010_2$
- If we stay with 4 bit memory locations, the above sum cannot be represented
- This situation is called an arithmetic exception. Arithmetic exceptions can be handled by an automatic default or by trapping to an exception handler.
- In some situations, when we are performing calculations modulo some number, we may discard the extra bit.
- This gives the answer $0010_2 = 2_{10}$ which is just 13 + 5 (mod 16). In many applications this is just what we want.

Exception handling

- In others this is a wrong result and we need to use exception handling
- Operations leading to exceptions
  - a + b: Overflow
  - a - b: Negative result, i.e., a < b
  - a*b: Overflow
  - a/b: Division by zero or noninteger result
- This may need to bring in logic that causes the process to stop, and bring in further information from main memory and may be computationally expensive.
- Fatal exceptions: cause process to abort
- Default handling: may be turned on
- For division it is generally agreed that division by zero is fatal
- There is also agreement about what to do when the result is not an integer
- E.g., 17/3 = 5.6667 -> 5
- The exact quotient should be truncated toward zero.

Signed Integers

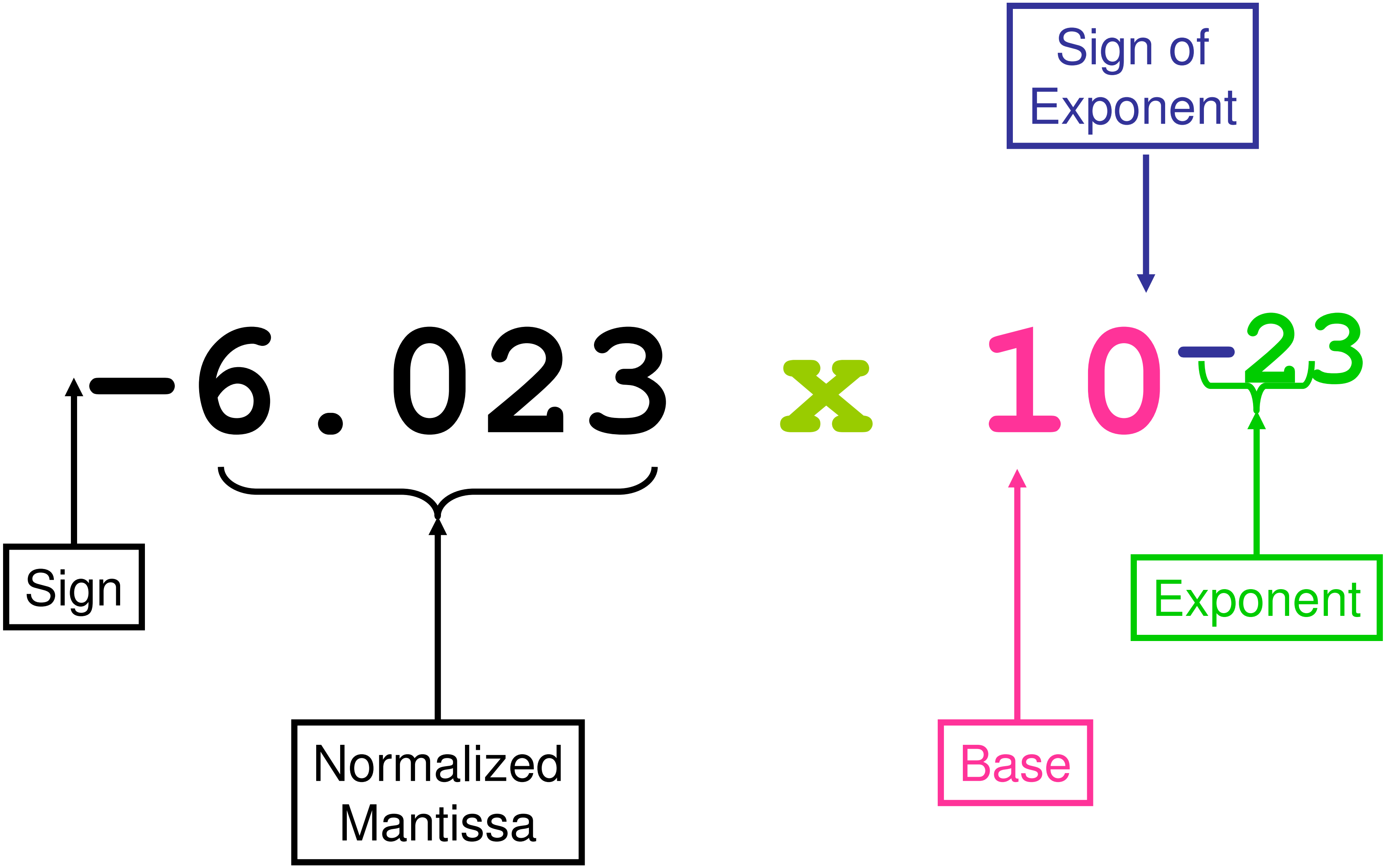
x	$+x$	$-x$
0	0000	
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8		1000

- Stored in a four byte word
- Can have two byte, byte, and 8 byte versions
- Need to figure out how to represent sign:
- Two approaches
  - **Sign magnitude**: if the first bit is zero, then the number is positive. Otherwise, it is negative.
    - 0 0 1 1 Denotes +11.
    - 1 0 1 1 Denotes -11.
    - Zero: Both 0 0 0 0 and 1 0 0 0 represent zero
  - **Two's complement**: As before the if the first bit is zero the number is positive
  - However values for the negative numbers are determined by subtraction of the number from 2^n.
  - There is one more negative number possible
- Signed numbers can overflow or underflow.
- Two's complement representation seems unnatural, but in fact it is often preferred because it makes addition easier to implement in silicon.

Floating point

- Attempt to
  - Handle decimal numbers
  - increase the range of numbers that can be represented
  - Provide a standard by which exceptions are consistently handled

Scientific Notation



Floating point on a computer

- Using fixed number of bits represent real numbers on a computer
- Once a base is agreed we store each number as two numbers and two signs
  - Mantissa and exponent
- Mantissa is usually “normalized”
- If we have infinite spaces to store these numbers, we can represent arbitrarily large numbers
- With a fixed number of spaces for the two numbers (mantissa and exponent)

Binary Floating Point Representation

- Same basic idea as scientific notation
- Modifications and improvements based on
  – Hardware architecture
  – Efficiency (Space & Time)
  – Additional requirements
    - Infinity
    - Not a number (NaN)
    - Not normalized
    - etc.

Floating point on a computer

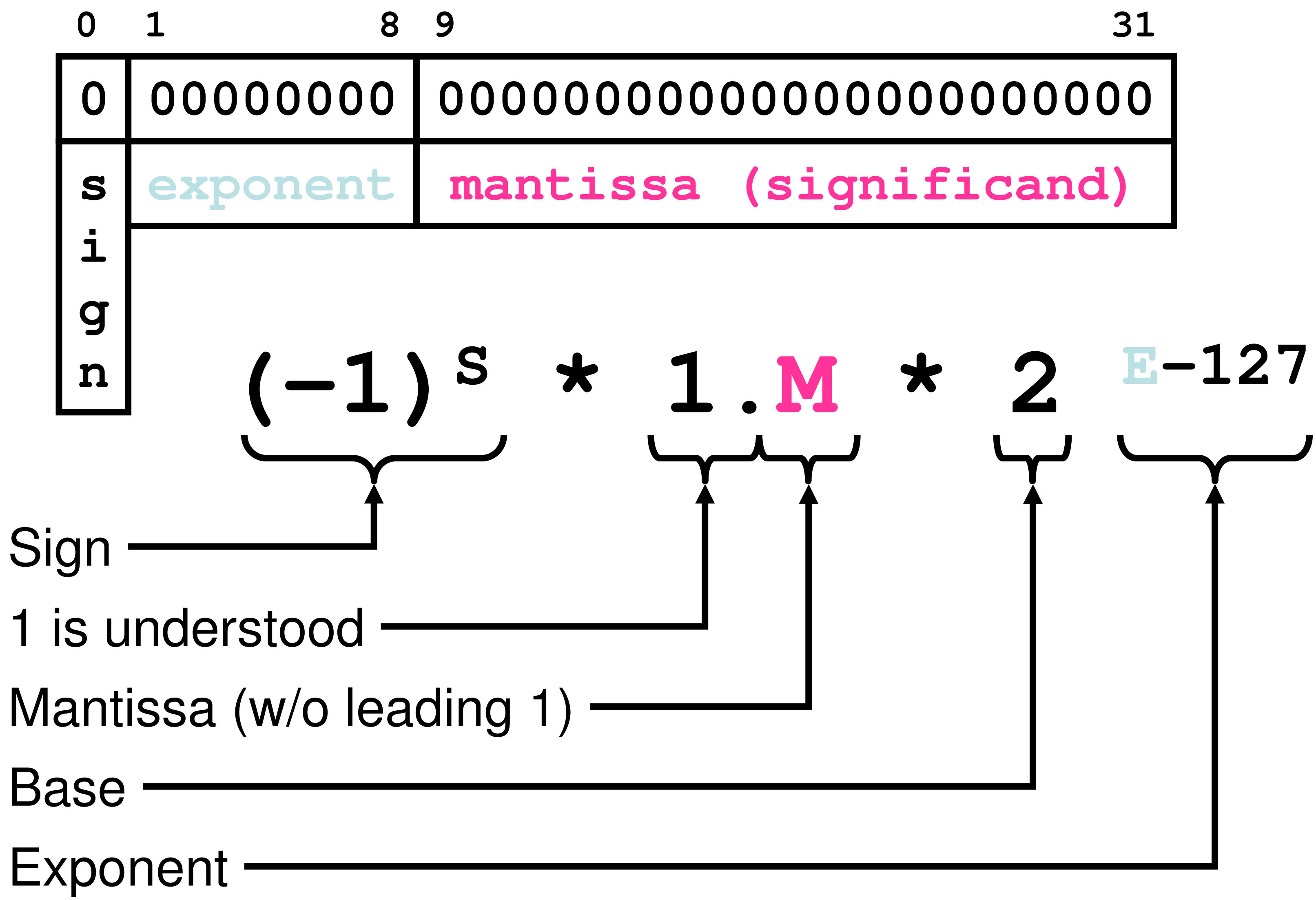
- If we wanted to store 15 x 2^{11} , we would need 16 bits:

  0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0

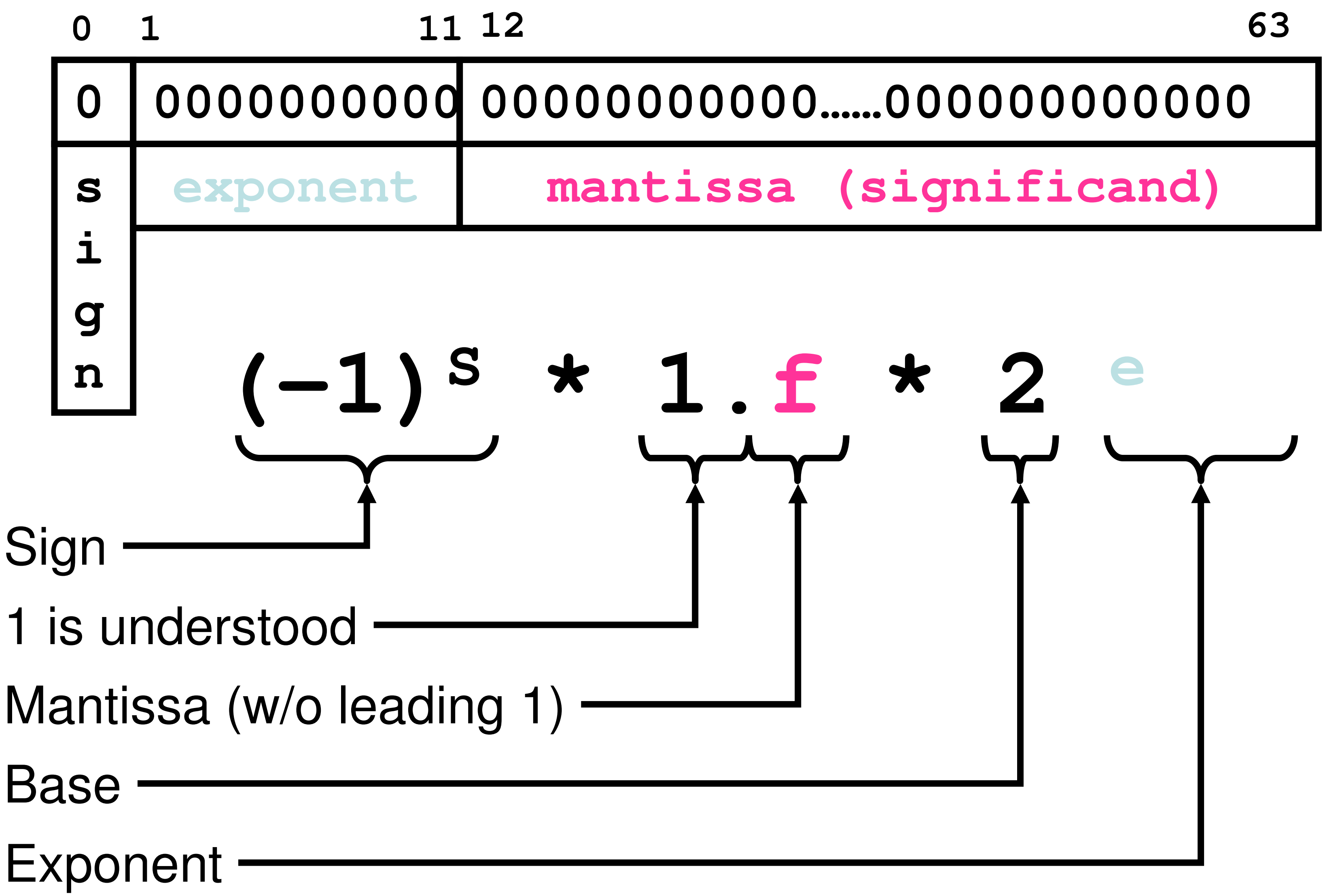
- Instead we store it as three numbers
- $(-1)^S \times F \times 2^E$, with F = 15 saved as 01111 and E = 11 saved as 01011.
- Now we can have fractions/decimals, too:

  binary .101 = $1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$.

IEEE-754 (single precision)



IEEE-754 (double precision)



IEEE - 754

Most nonzero floating-point numbers are normalized. This means they can be expressed as

$$x = \pm(1 + f) \cdot 2^e$$

The quantity f is the fraction or mantissa and e is the exponent. The fraction satisfies

$$0 \leq f < 1$$

and must be representable in binary using at most 52 bits. In other words, $2^{52}f$ is an integer in the interval

$$0 \leq 2^{52}f < 2^{52}$$

The exponent e is an integer in the interval

$$-1022 \leq e \leq 1023$$

The finiteness of f is a limitation on *precision*. The finiteness of e is a limitation on *range*. Any numbers that don't meet these limitations must be approximated by ones that do.

Double-precision floating-point numbers are stored in a 64 bit word, with 52 bits for f, 11 bits for e, and one bit for the sign of the number. The sign of e is accommodated by storing $e + 1023$, which is between 1 and $2^{11} - 2$. The two

Can be written...

0	00000000000	0000000000000......0000000000000000
sign	exponent	mantissa (significand)

$$(-1)^{S} * 2^{E} * 1.f$$

	E+1023 == 0	0 < E+1023 < 2047	E+1023 == 2047
f==0	0	Powers of Two	∞
f~=0	Non-normalized typically underflow	Floating point Numbers	Not A Number

- $x = \pm(1+f) \times 2^e$
- $0 \leq f < 1$
- $f = (\text{integer} < 2^{52}) / 2^{52}$

- $-1022 \leq e \leq 1023$
- $e = \text{integer}$