Computational Methods
CMSC/AMSC/MAPL 460
Representing numbers in floating point

Ramani Duraiswami,
Dept. of Computer Science

Fixed point representation

- How can we represent a number in a computer's memory?
- Fixed point is an obvious way:
- Used to represent integers on computers, and real numbers on some DSPs:
- Each **word** (storage location) in a machine contains a fixed number of digits.
- Example: A machine with a 6-digit word might represent 2005 as

0	0	2	0	0	5

- This only allows us to represent integers and uses a decimal system

Binary/Decimal/Octal/Hexadecimal

- Numbers can be represented in different bases
- Usually humans use decimal
  - Perhaps because we have ten fingers
- Computer memory often has two states
  - Assigned to 0 and 1
  - Leads to a binary representation
- Octal and Hexadecimal representations arise by considering 3 or 4 memory locations together
  - Lead to 2^3 and 2^4 numbers

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

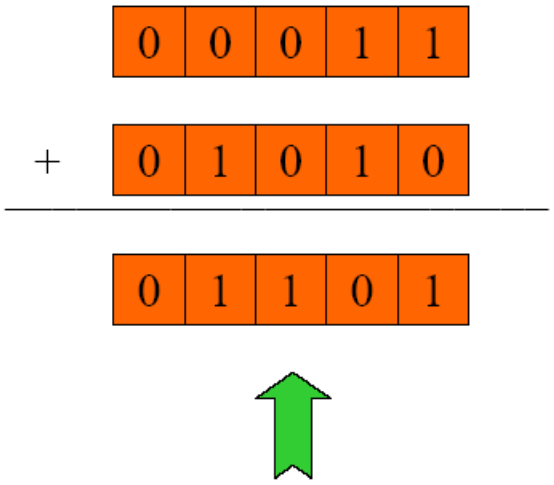
- Memory locations on a 32 bit machine, usually consist of 4 bytes => called a word
- Relationship between words and data of various sizes:
  - byte 8bits, 1 byte
  - short or half word 16bits, 2 bytes
  - word 32bits, 4 bytes
  - long or double word 64 bits, 8 bytes
- Internally, by default, Matlab stores all numbers in double words
  - Can specify other types of storage

Binary Representation

- Most computers use **binary (base 2)** representation.

0 1 0 1 1 0

- Each digit has a value 0 or 1.
- If the number above is binary, its value is
- $1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$. (or 22 in base 10)
- Adding numbers in binary



Note the "**carry**" here!

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 10$ (binary) $= 10_2 = 2$
$1 + 1 + 1 = 11$ (binary) $= 11_2 = 3$

Unsigned Integers

- Integers can be added, subtracted, multiplied, and divided.
- **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 some applications this is just what we want.

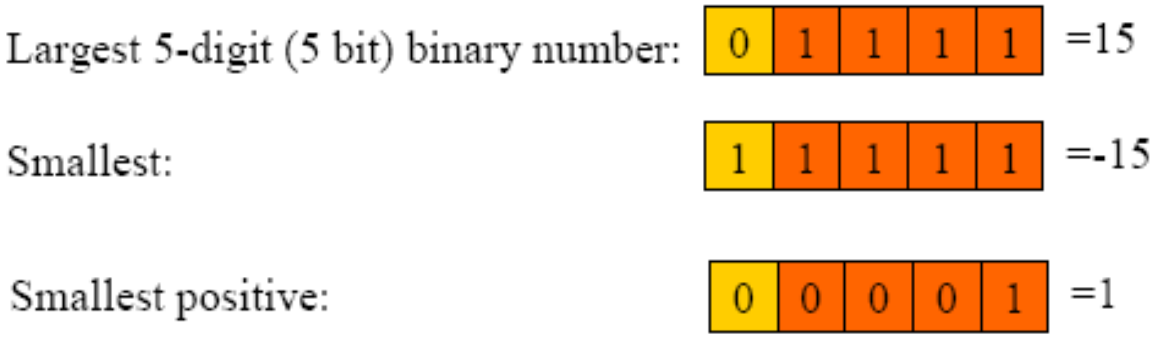
Exception handling

- 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.

Negative numbers

- One way computers represent negative numbers is using the *sign-magnitude representation:*
- **Sign magnitude**: if the first bit is zero, then the number is positive. Otherwise, it is negative.
- 0 0 0 1 1 Denotes +11.
- 1 0 0 1 1 Denotes -11.

Range of fixed point numbers



Signed Integers

- 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 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 the way that is used in computer processors, as it is easier to implent in hardware.

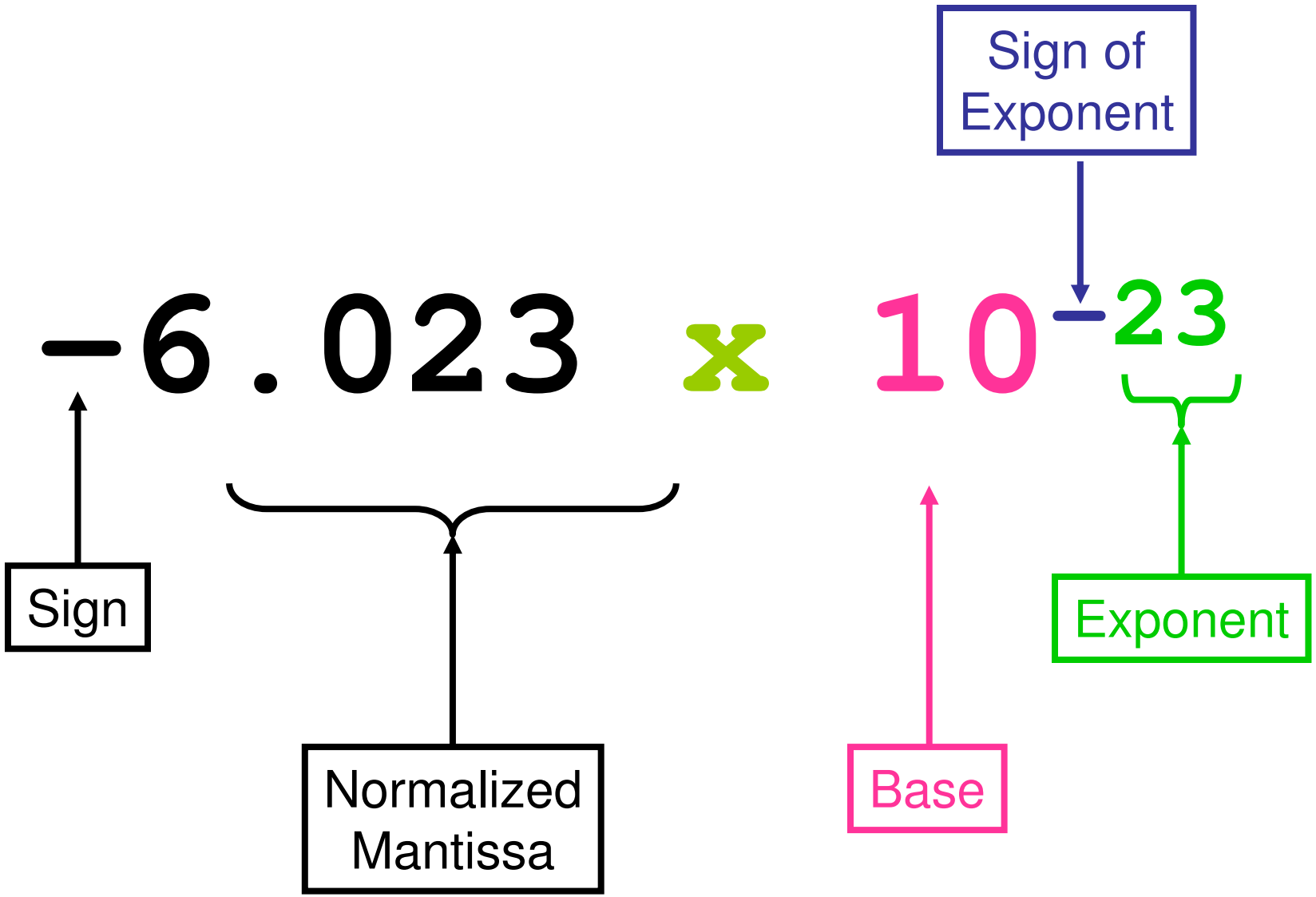
x	$+x$	$-x$
0	0000	
1	0001	1111
2	0010	1110
3	0011	1101
4	0100	1100
5	0101	1011
6	0110	1010
7	0111	1001
8		1000

- Fixed point arithmetic:
  - Easy: always get an integer answer.
  - Either we get exactly the right answer, or we can detect
  - overflow.
  - The numbers that we can store are equally spaced.
  - Disadvantage: **very** limited range of numbers.

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
- Use Scientific Notation as a guide

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: Need to represent conditions which arise during calculations
    - Infinity
    - Not a number (NaN)
    - Underflow

Floating point on a computer

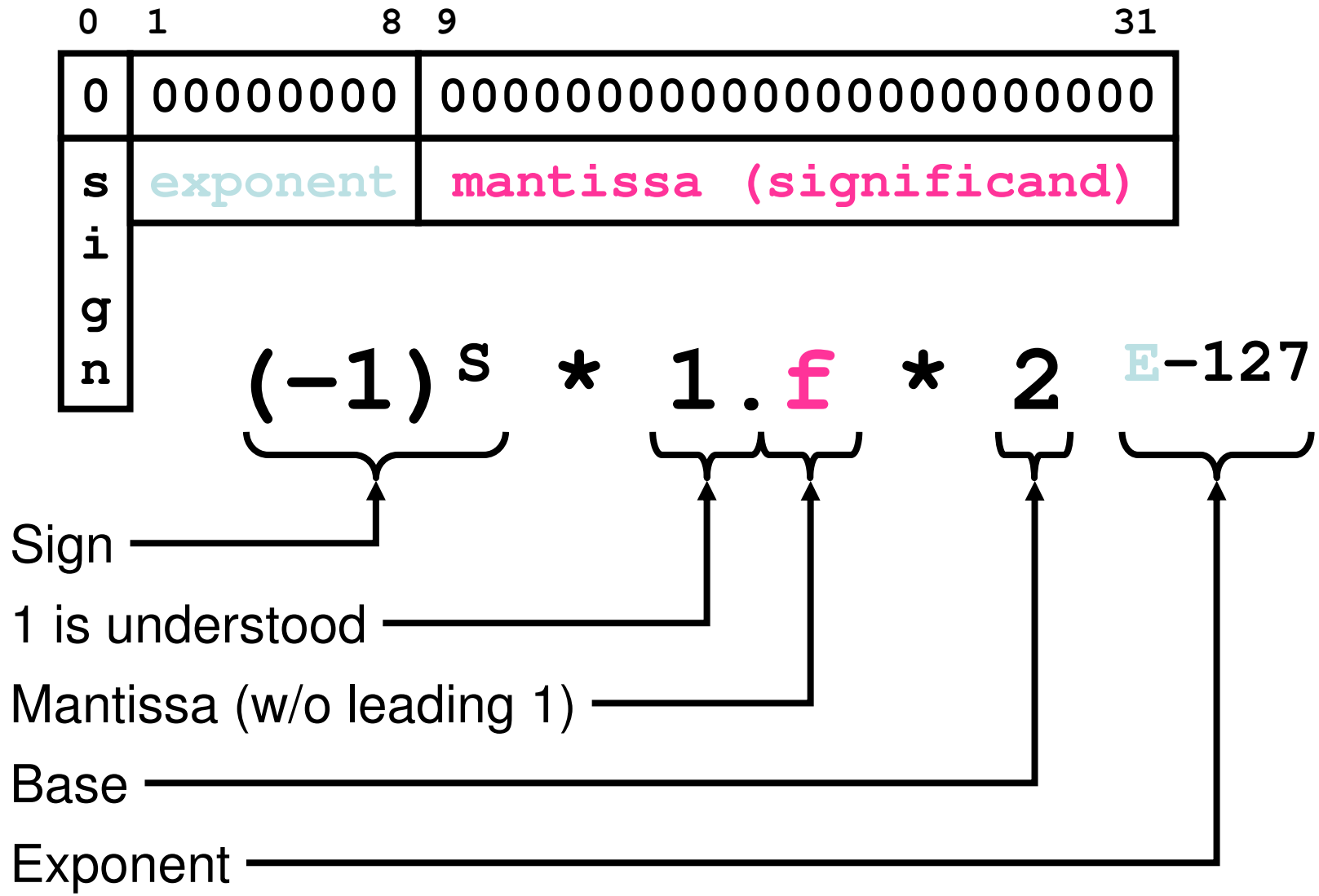
- If we wanted to store 15×2^{11} , we would need 16 bits:

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

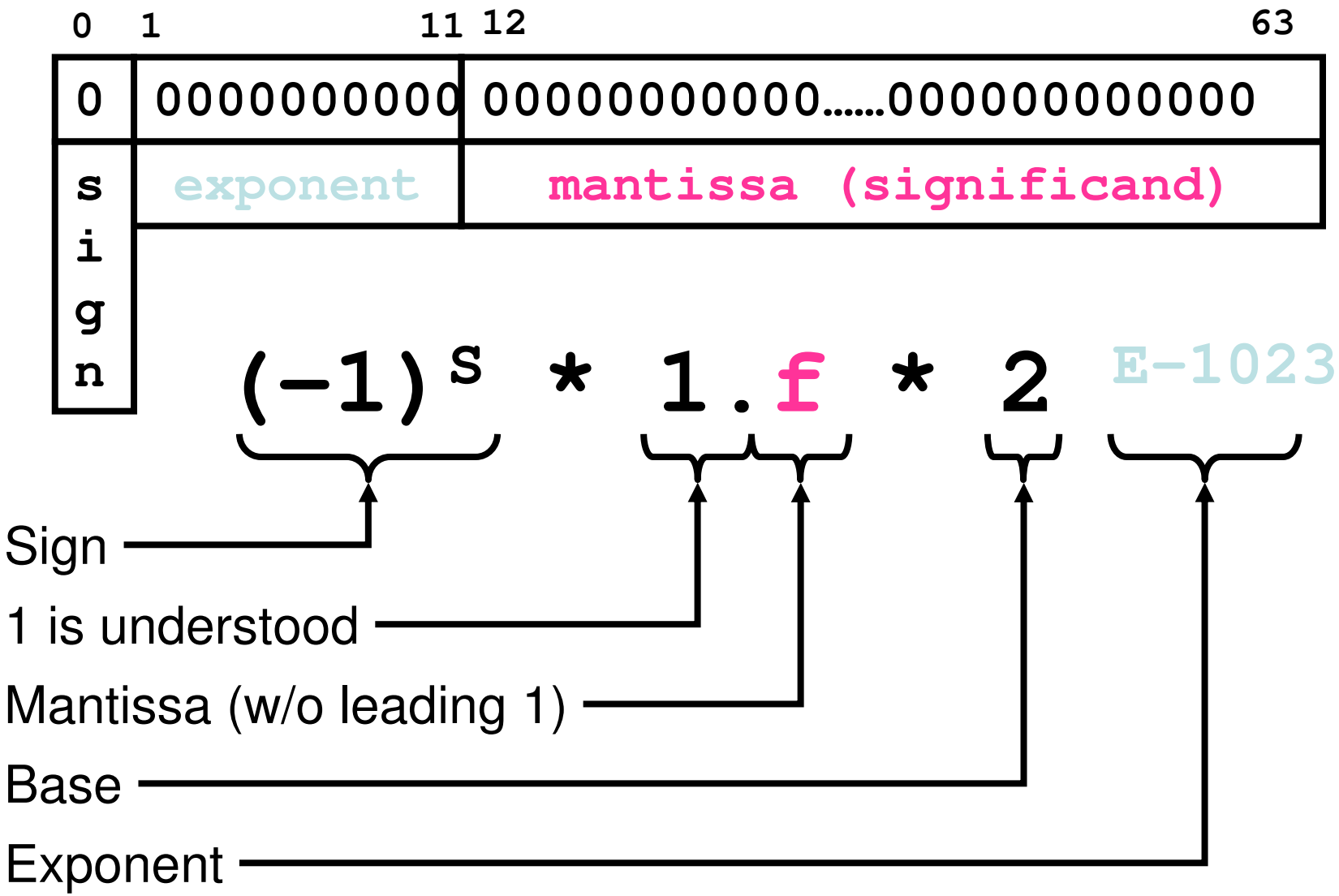
- Instead we store it as three numbers
- $(-1)^S \times F \times 2^E$, with F = 15 saved as 01111 and E = 11 saved as 01011.
- Now we can have fractions/decimals, too:

  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

$$0 \le f < 1$$

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

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

The exponent e is an integer in the interval

$$-1022 \le e \le 1023$$

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	00000000000.....00000000000000
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

Some numbers cannot be exactly represented

$$\frac{1}{10} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{0}{2^6} + \frac{0}{2^7} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{0}{2^{10}} + \frac{0}{2^{11}} + \frac{1}{2^{12}} + \cdots$$

$$t = (1 + \frac{9}{16} + \frac{9}{16^2} + \frac{9}{16^3} + \ldots + \frac{9}{16^{12}} + \frac{10}{16^{13}}) \cdot 2^{-4}$$

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

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

Effects of floating point

Finite f implies finite *precision*.

Finite e implies finite *range*

Floating point numbers have discrete spacing,
a maximum and a minimum.

Effects of floating point

- eps *is the distance from* 1 *to the next larger floating-point number.*
- eps = 2^{-52}
- In Matlab

	Binary	Decimal
eps	2^(-52)	2.2204e-16
realmin	2^(-1022)	2.2251e-308
realmax	(2-eps)*2^1023	1.7977e+308

Rounding vs. Chopping

- **Chopping**: Store x as c, where |c| < |x| and no machine number lies between c and x.
- **Rounding**: Store x as r, where r is the machine number closest to x.
- **IEEE standard arithmetic uses rounding**.

Machine Epsilon

- **Machine epsilon** is defined to be the smallest positive number which, when added to 1, gives a number different from 1.
  - Alternate definition (1/2 this number)
- **Note:** Machine epsilon depends on d and on whether rounding or chopping is done, but does not depend on m or M!

```
x = 1; while 1+x > 1, x = x/2, pause(.02), end
```

```
x = 1; while x+x > x, x = 2*x, pause(.02), end
```

```
x = 1; while x+x > x, x = x/2, pause(.02), end
```