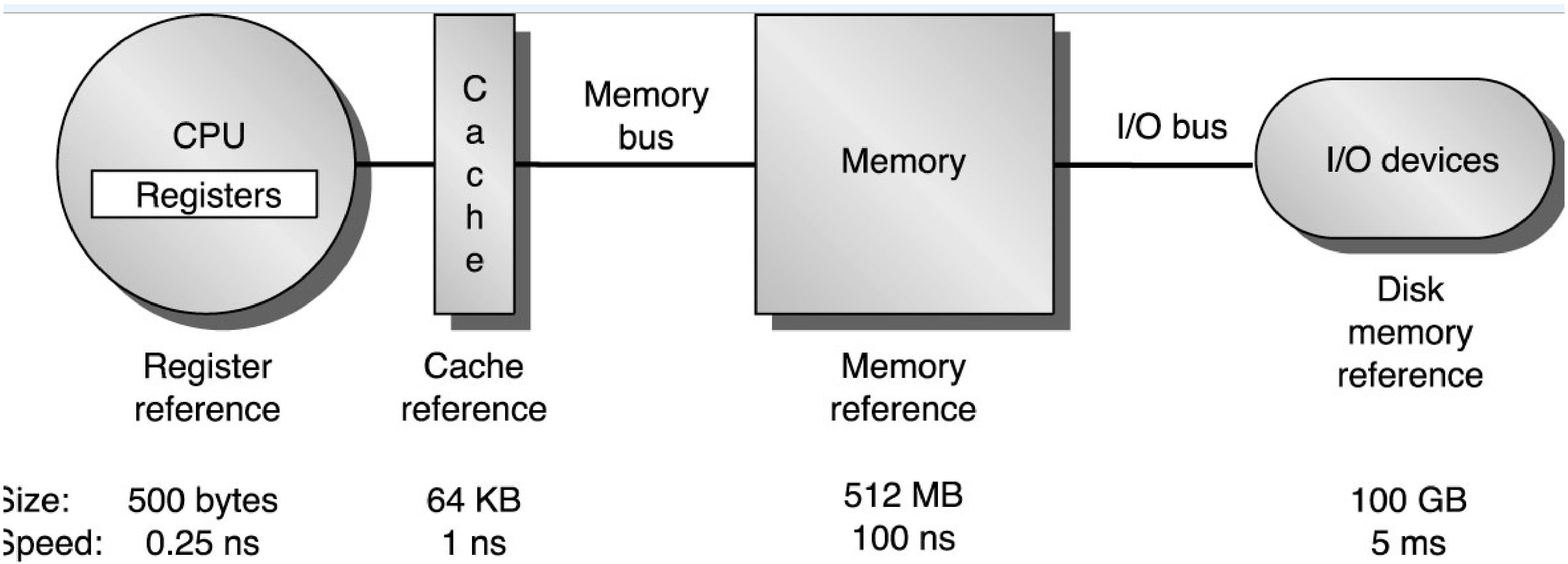
Matrix-vector product

- Matrix-vector multiplication

$$\boxed{\mathbf{M} \bullet \mathbf{v} = \begin{bmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} & \mathbf{M}_{13} \\ \mathbf{M}_{21} & \mathbf{M}_{22} & \mathbf{M}_{23} \\ \mathbf{M}_{31} & \mathbf{M}_{32} & \mathbf{M}_{33} \end{bmatrix} \begin{bmatrix} \mathbf{v}_x \\ \mathbf{v}_y \\ \mathbf{v}_z \end{bmatrix}}$$

- Recall how to do matrix multiplication
- How many operations does this matrix vector product take?
- How many operations does a general matrix vector product take?

Memory Hierarchy



Ways to implement a matrix vector product

- Access matrix
  - Element-by-element along rows
  - Element-by-element along columns
  - As column vectors
  - As row vectors
- Homework 1: Upload a single log-log graph of your results (with five different curves) on to Piazza. Use the code snippet provided, or your own

```
[m,n]=size(A);
y = zeros(m,1);
for i=1:m,
    for j=1:n,
        y(i) = y(i) + A(i,j)*x(j);
    end
end
```

```
[m,n]=size(A);
y = zeros(m,1);
for i=1:m,
    y(i) = A(i,:) * x;
end
```

```
[m,n]=size(A);
y = zeros(m,1);
for j=1:n,
    y = y + A(:,j)*x(j);
end
```

Binary Representation: Bits

- Computer memory is binary
- A bit is a single binary digit that can take on one of the two values 0 and 1.

$$437=1\times256+1\times128+0\times64+1\times32+1\times16+0\times8+1\times4+0\times2+1\times1$$

$$=2^8+2^7+2^5+2^4+2^2+2^0=110110101$$

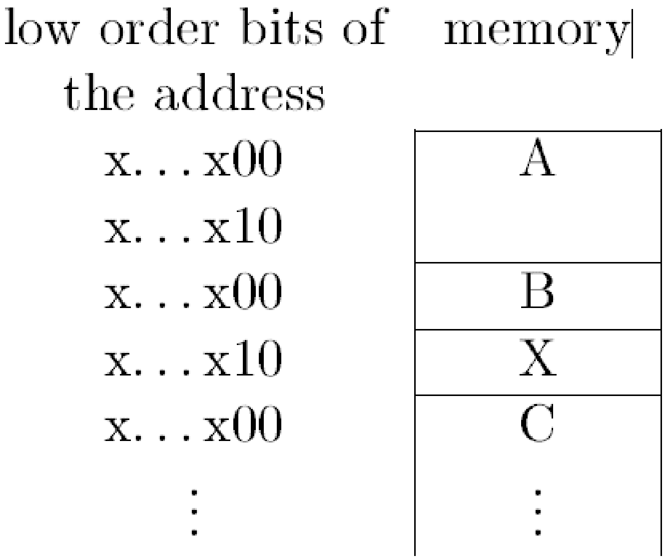
- $0.625=1\times0.5+0\times0.25+1\times0.125=2^{-1}+2^{-3}=0.101$
- A byte is a group of eight bits.
  - Since a hexadecimal digit (base 16) can be represented by four bits, bytes can be described by pairs of hexadecimal digits.
  - 0, 1, 2, 3, 4, 5, 6, 7, 8,
  - 0000, 0001, 0010, 0011, 0100, 0101, 0110, 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
  - 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



Bit operations

- Very efficient set of operations that are provided in processors, and that have representations in programming languages
- May 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

- We have an arithmetic system which is not closed under the normal set of operations.
- Consider 4 bit arithmetic
- $13_{10} + 5_{10} = 1101_2 + 0101_2 = 10010_2$
- the above sum is not representable in 4 bits
- 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.

Signed Integers

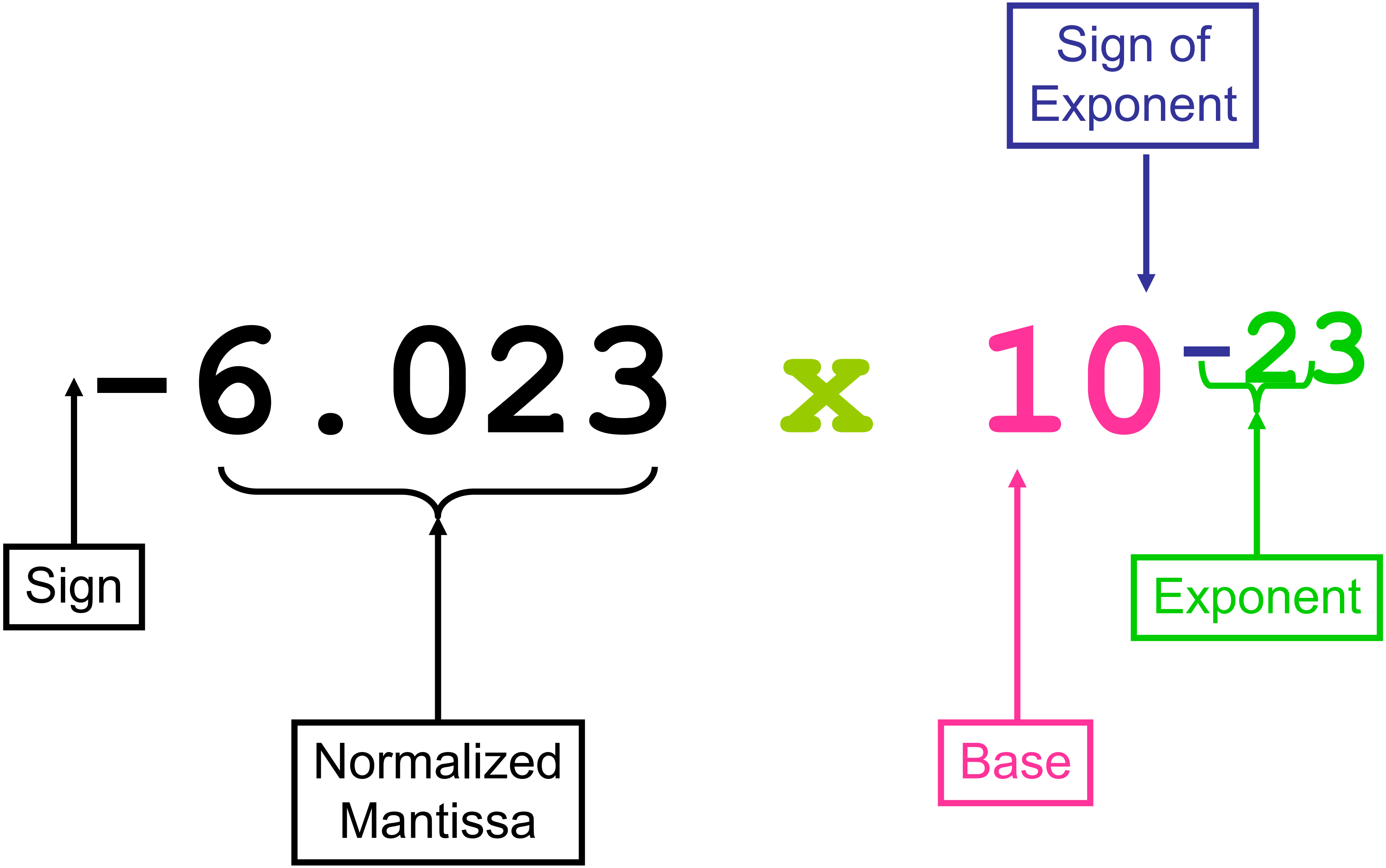
- Stored in a four byte word
- Can have two byte, byte, and 8 byte versions
- Need to figure out how to represent sign:
- Two approaches
  - **Sign magnitude**: if the first bit is zero, then the number is positive. Otherwise, it is negative.
    - 0 0 1 1 Denotes +11.
    - 1 0 1 1 Denotes -11.
    - Zero: Both 0 0 0 0 and 1 0 0 0 represent zero
  - **Two's complement**: If first bit is zero the number is positive,
    and it is the same as sign-magnitude
  - **Negative numbers** have a 1 in the first place
  - Value determined by subtraction of the number from 2^n.
  - There is one more negative number possible
- Two's complement representation seems unnatural, but in fact it is preferred
- Exceptions: Overflow/underflow

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

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

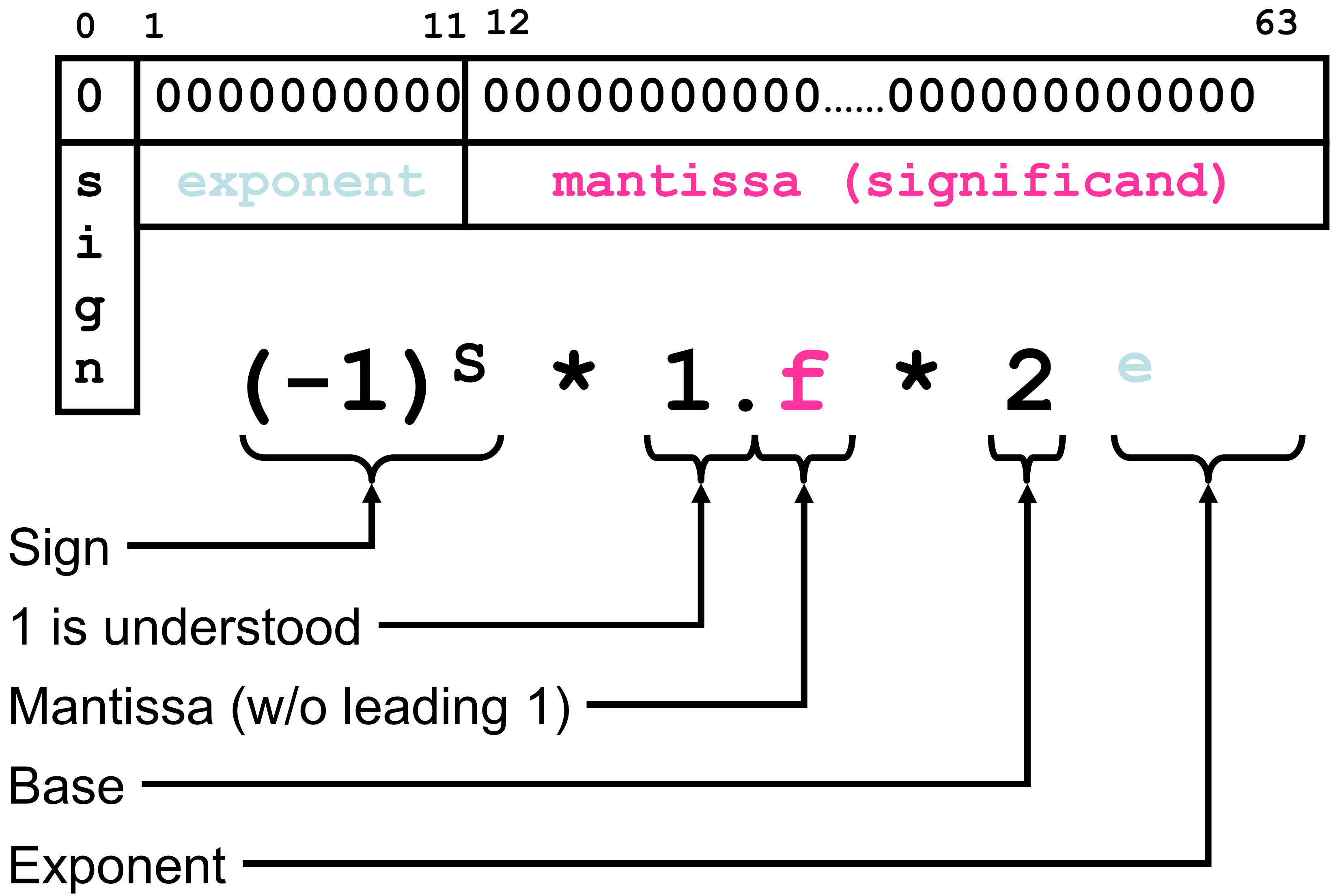
Scientific Notation



Floating point on a computer

- Need to represent numbers using fixed number of bits
- Base: Binary
- Divide bits into two numbers: mantissa and exponent
- Mantissa is “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)

IEEE-754 (double precision) – 64 bits



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

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

Special (Exceptional) Numbers

0	00000000000	0000000000000......000000000000000
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

Floating point exceptions

- Underflow
- Overflow
- Division by zero
- NaN: 0/0, $\infty\times 0$, or sqrt($-$number).

IEEE-754 (single precision) – 32 bits

0	1 … 8	9 … 31
0	00000000	00000000000000000000000
sign	exponent	mantissa (significand)

$$\underbrace{(-1)^{S}}_{\text{Sign}} * \underbrace{1}_{\text{1 is understood}}.\underbrace{M}_{\text{Mantissa (w/o leading 1)}} * \underbrace{2}_{\text{Base}}{}^{\underbrace{E-127}_{\text{Exponent}}}$$

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!

Some numbers cannot be exactly represented

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

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

```
x = 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
```