

# **Modulo:** Approfondimenti sui Sistemi Aritmetici di un computer: tipo reale [P2\_03]

## **Unità didattica:** Sistema Aritmetico Reale Floating-Point [2-AT]

### **Titolo:** Rappresentazione in memoria dei numeri reali

#### Argomenti trattati:

- ✓ Visualizzazione dei campi segno, esponente, mantissa
- ✓ Variabili predefinite dell'ambiente floating-point del C (in float.h)
- ✓ Schemi di rounding del Sistema Aritmetico Standard

**Prerequisiti richiesti:** Sistema Aritmetico Floating Point IEEE  
Standard 754

## Esempio: Estrazione di alcuni bit dal contenuto di una variabile

```
#include <stdio.h>
#define bias 127
int estraeEsp(int *);
void main()
{
    float x; short xesp;
    scanf("%f", &x);
    xesp = (short) estraeEsp(&x);
    printf("esp = %d\n", xesp);
}
int estraeEsp(int *n)
/* estrae da un float il campo esponente */
{
    int xesp, mask;
    mask = 0x7f800000; xesp = *n & mask;
    xesp = xesp >> 23; xesp = xesp - bias;
    return xesp;
}
```

Nota: il programma non è ISO C STANDARD! Infatti usando Dev-C++ o MS Visual C++ v.6, Visual Studio 2008 si ha un WARNING, usando CodeBlocks (-pedantic) si ha un ERROR!

Per  $x = 8.5$  produce  $esp = 3$  (Bias s.p.=127)



Per estrarre i campi (s,e,m) da un numero floating-point è utile usare una struttura di campi di bit in C

### Esempio

```
struct F_fields /** STRUTTURA DI CAMPI DI BIT **/  
{  
    unsigned int m: 23; // dai bit meno significativi  
    unsigned int e:  8;  
    unsigned int s:  1; // a quelli più significativi  
};
```

numero di bit per il campo della struttura

vedere: [P2\\_03\\_02\\_AC.pdf](#)

Il seguente programma visualizza in hex un float ed un double ...  
... si può aggiungere anche la visualizzazione dei bit

### mostra\_float\_double.c

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
```

per usare le variabili predefinite del sistema aritmetico floating-point

```
void mostra_sp(int );
void mostra_dp(int [ ]);
```

**Warn o Err:** parametro formale ed attuale non sono dello stesso tipo.

```
void main()
{float a; double b;
 scanf("%le",&b); a=(float)b;
 printf("float =%+e,\t", a);
 printf("double=%+e,\t", b);
}
void mostra_sp(int n)
{ printf("float esadecimale=%08x\n",n);
}
void mostra_dp(int n[ ])
{ printf("double esadecimale=%08x %08x\n",n[1],n[0]);
}
```

Versione che non provoca errori: usa **union**

```
#include ... #define MAX_LEN 64
void estrae_64_bit(short , char [], short []);
void mostra_sp(int );
void mostra_dp(int []);
void main() {short k, bit[MAX_LEN];
```

```
union sp
{
    float fa;
    int la;
    char C[4]; // solo per estrae_64_bit
} a;
```

```
union dp
{
    double db;
    int lb[2];
    char C[8]; // solo per estrae_64_bit
} b;
```

```
scanf("%le",&b.db); a.fa=(float)b.db;
```

```
printf("\n\nfloat= %+e,", a.fa); mostra_sp(a.la);
```

```
estrarre_64_bit(sizeof(a.fa), a.C, bit); printf("\tbit = ");
```

```
for (k=31; k>=0; k--) (k==31 | k==23) ? printf("%1d ",bit[k]) : printf("%1d",bit[k]);
```

```
printf("\n\ndouble= %+e,", b.db); mostra_dp(b.lb);
```

```
estrarre_64_bit(sizeof(b.db), b.C, bit); printf("\tbit = ");
```

```
for (k=63; k>=0; k--) (k==63 | k==52) ? printf("%1d ",bit[k]) : printf("%1d",bit[k]);
```

```
void mostra_sp(int n)
{
    printf("float esadecimale = %08x",n);
}
```

```
void mostra_dp(int n[])
{
    printf("double esadecimale = %08x %08x",n[1],n[0]);
}
```

```
void estrae_64_bit(short len, char ch[], short bit[MAX_LEN])
{
    ...
}
```

```
union dp
{
    double db;
    long int lb;
    char C[8]; // solo per estrae_64_bit
} b;

...
void mostra_dp(long int n)
{
    printf("double esadecimale = %16lx", n);
}
```

C 64 bit

output

1.25

float =+1.250000e+000, float esadecimale=3fa00000

bit 0 01111111 010000000000000000000000

double=+1.250000e+000, double esadecimale=3ff40000 00000000

bit 0 011111111111 01000000000000000000000000000000...

# Che relazione c'è tra il *Sistema Aritmetico Floating-Point* in **singola** precisione e quello in **doppia** precisione?

- ◆ **Aumenta l'intervallo di rappresentabilità** (perché aumenta il campo esponente da 8 a 11 bit  $\Rightarrow$  range: da  $[-127, +128]$  a  $[-1023, +1024]$ ).
- ◆ **Aumenta la densità dei numeri** (perché aumenta il campo mantissa da 23 a 52 bit  $\Rightarrow$  range: da  $[0, 8388607]$  a  $[0, 4503599627370495]$ ).

## Quiz

- ❖ Quanti numeri double ci sono tra due float consecutivi?
- ❖ Quanti numeri double ci sono oltre FLT\_MAX (massimo float normalizzato rappresentabile)?

# ***ESEMPIO 1: Numeri normalizzati*** (bit implicito=1)

Numeri normalizzati

$$E_{\min} < \mathbf{e} < E_{\max}$$

$$\mathbf{m} \geq 0$$

b.db = +1.0;

<b>+1</b>	<b>s.p.</b> (32bit)	3f800000		
		0	011 1111 1	000 0000 0000 0000 0000 0000
	<b>d.p.</b> (64bit)	3ff00000 00000000		
		0	011 1111 1111	0000 0000 0000 .... 0000 0000
long d.	(80bit)	0	011 ..... 1111	1000 0000 0000 .... 0000 0000

bit esplicito

b.db = -1.0;

<b>-1</b>	<b>s.p.</b> (32bit)	bf800000		
		1	011 1111 1	000 0000 0000 0000 0000 0000
	<b>d.p.</b> (64bit)	bff00000 00000000		
		1	011 1111 1111	0000 0000 0000 .... 0000 0000



Per estrarre da un **float** i **bit** dei campi **segno**, **esponente**,  
**mantissa** ....

### EstraeFloat\_Bit.c

```
#include <stdio.h>
#include <stdlib.h>
#include <float.h>
#include <math.h> /*per pow()
void estrae_bit(int ,char [32]);

int main()
{
    char i, bit[32];
    union basic_single {float fa;
                        int ia; } a;

/*definisce float di cui visualizzare i bit*/
```

per usare: FLT\_MIN, ...

**a.fa= ...**

```
    estrae_bit(a.ia, bit);
    printf("\tfloat    = %e\n",a.fa);
    printf("\tint hex = %08x\n",a.ia);
    puts("bit corrispondenti segno esponente mantissa");
    printf("\t\t %1d ",bit[0]);
    for (i=1; i<=8; i++)
        printf("%1d",bit[i]);
    printf(" ");
    for (i=9; i<32; i++)
        printf("%1d",bit[i]);
    return 0;}
```

```

void estrae_bit(int reg, char B[32])
/* rappresentazione binaria dell'int reg
nell'array B
bit +signif.    <--- bit -signif.
B[0] B[1] B[2]  ...  B[30] B[31]
*/
{
    short i;
    for (i=31; i>=0; i--)
    {
        B[i]=(char)(1&reg); /* estrae bit -significat.*/
        reg=reg>>1;        /* shift a destra di 1 bit*/
    }
}

```

ricorda che ...

Oggetto IEEE Std.754	Caratterizzazione esponente <b>e</b> mantissa <b>m</b>		<i>ℓ</i>
<b>Numeri normalizzati</b> valore = $(-1)^s [l.m] \times 2^{e-Bias}$	$E_{min} < e < E_{max}$	$m \geq 0$	1
<b>Infinito con segno</b>	$e = E_{max}$	$m = 0$	-
<b>NaN (Not A Number)</b>	$e = E_{max}$	$m \neq 0$	-
<b>Zero con segno</b>	$e = E_{min}$	$m = 0$	-
<b>Numeri denormalizzati</b> valore = $(-1)^s [l.m] \times 2^{e-Bias+1}$	$e = E_{min}$	$m \neq 0$	0

output

**a.fa = FLT\_MAX;**

float = 3.402823e+038

long hex = 7f7fffff

bit corrispondenti segno esponente mantissa

0 11111110 111111111111111111111111

massimo numero normalizzato rappresentabile in s.p.

**a.fa = FLT\_MIN;**

float = 1.175494e-038

long hex = 00800000

bit corrispondenti segno esponente mantissa

0 00000001 000000000000000000000000

minimo numero normalizzato rappresentabile in s.p.

**a.fa = FLT\_MIN/pow(2,23);**

float = 1.401298e-045

long hex = 00000001

bit corrispondenti segno esponente mantissa

0 00000000 000000000000000000000001

minimo numero denormalizzato

**a.fa = FLT\_MIN/pow(2,24);**

bit corrispondenti segno esponente mantissa

0 00000000 000000000000000000000000

**underflow**

FLT\_MAX, FLT\_MIN,  
DBL\_MAX, DBL\_MIN:  
variabili predefinite

$$+1.5$$
$$= 1 + \frac{1}{2}$$

**s.p.**

(32bit)

**3fc00000**

0

011 1111 1

100 0000 0000 0000 0000 0000

**d.p.**

(64bit)

**3ff80000 00000000**

0

011 1111 1111

1000 0000 0000 .... 0000 0000

$$+1.25$$
$$= 1 + \frac{1}{4}$$

**s.p.**

(32bit)

**3fa00000**

0

011 1111 1

010 0000 0000 0000 0000 0000

**d.p.**

(64bit)

**3ff40000 00000000**

0

011 1111 1111

0100 0000 0000 .... 0000 0000

$$+1.125$$
$$= 1 + \frac{1}{8}$$

**s.p.**

(32bit)

**3f900000**

0

011 1111 1

001 0000 0000 0000 0000 0000

**d.p.**

(64bit)

**3ff20000 00000000**

0

011 1111 1111

0010 0000 0000 .... 0000 0000

# ESEMPIO 2a: *estremi normalizzati* (bit implicito=1) in singola precisione

Numeri normalizzati

$$E_{\min} < e < E_{\max}$$

$$m \geq 0$$

**FLT\_  
MIN**

1.1...e-38

**s.p.**

(32bit)

00800000

0

000 0000 1

000 0000 0000 0000 0000 0000

**d.p.**

(64bit)

38100000 00000000

0

011 1000 0001

0000 0000 0000 .... 0000 0000

**FLT\_  
MAX**

3.4...e+38

**s.p.**

(32bit)

7f7fffff

0

111 1111 0

111 1111 1111 1111 1111 1111

**d.p.**

(64bit)

47efffff e0000000

0

100 0111 1110

1111 1111 1111 .... 0000 0000

# ESEMPIO 2b: *estremi normalizzati* (bit implicito=1) in doppia precisione

## Underflow → 0

**DBL\_  
MIN**

2.2...e-308

**s.p.**

(32bit)

00000000

0

000 0000 0

000 0000 0000 0000 0000 0000

**d.p.**

(64bit)

00100000 00000000

0

000 0000 0001

0000 0000 0000 ..... 0000 0000

## Overflow → Inf

**DBL\_  
MAX**

1.7...e+308

**s.p.**

(32bit)

7f800000 (+inf)

0

111 1111 1

000 0000 0000 0000 0000 0000

**d.p.**

(64bit)

7fefffff ffffffff

0

111 1111 1110

1111 1111 1111 ..... 1111 1111

# ESEMPIO 3: Zero con segno (bit implicito=0)

<b>+0</b>	<b>s.p.</b> (32bit)	000000000		
		0	000 0000 0	000 0000 0000 0000 0000 0000
	<b>d.p.</b> (64bit)	000000000 000000000		
		0	000 0000 0000	0000 0000 0000 .... 0000 0000

<b>-0</b>	<b>s.p.</b> (32bit)	800000000 (*)		
		1	000 0000 0	000 0000 0000 0000 0000 0000
	<b>d.p.</b> (64bit)	800000000 000000000 (**)		
		1	000 0000 0000	0000 0000 0000 .... 0000 0000

(\*) per esempio **a.fa=-DBL\_MIN**

(\*\*) per esempio **b.db=-DBL\_MIN/DBL\_MAX**



# ESEMPIO 4: *Infinito con segno (affine mode)*

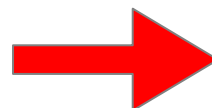
Infinito con segno

$e = E_{\max}$

$m = 0$

<b>+INF</b>	<b>s.p.</b>	<b>7f800000 (+inf)</b>		
	(32bit)	0	111 1111 1	000 0000 0000 0000 0000 0000
	<b>d.p.</b>	<b>7ff00000 00000000 (+1.#INF00e+000)</b>		
	(64bit)	0	111 1111 1111	0000 0000 0000 .... 0000 0000
<b>-INF</b>	<b>s.p.</b>	<b>ff800000 (-inf)</b>		
	(32bit)	1	111 1111 1	000 0000 0000 0000 0000 0000
	<b>d.p.</b>	<b>fff00000 00000000 (-inf)</b>		
	(64bit)	1	111 1111 1111	0000 0000 0000 .... 0000 0000

per es. **a.fa=-FLT\_MAX/FLT\_MIN**  
**b.db=-DBL\_MAX/DBL\_MIN**



**a float = -1.#INF00e+000**  
**b double= -1.#INF00e+000**



## ESEMPIO 5: NaN

NaN (Not A Number)	$e = E_{\max}$	$m > 0$
--------------------	----------------	---------

<b>+NaN</b>	<b>s.p.</b> (32bit)	7fc00000
		0 111 1111 1 100 0000 0000 0000 0000 0000
	<b>d.p.</b> (64bit)	7ff80000 00000000
		0 111 1111 1111 1000 0000 0000 ..... 0000 0000

per es.

```
a.fa = ((float)DBL_MAX)+((float)-DBL_MAX);
```

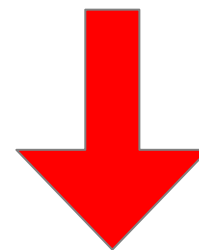
forma indeterminata  $+\infty - \infty$

```
a float = 1.#QNAN0e+000
b double= 1.#QNAN0e+000
```

<b>-NAN</b>	<b>s.p.</b> (32bit)	<b>ffc00000</b>	
		<b>1</b> <b>111 1111 1</b> <b>100 0000 0000 0000 0000 0000</b>	
	<b>d.p.</b> (64bit)	<b>fff80000 00000000</b>	
		<b>1</b> <b>111 1111 1111</b> <b>1000 0000 0000 ..... 0000 0000</b>	

per es.

```
a.fa = pow(-2,1.5);
```



operazione invalida  $(-2)^{1.5}$

```
a float = -1.#IND00e+000
b double= -1.#IND00e+000
```

... perché  $(-2)^{1.5} = e^{1.5 \log(-2)}$  e non è definito  $\log(-2)$ !!!

# ESEMPIO 6a: Numeri denormalizzati (bit implicito=0) in singola precisione

Numeri denormalizzati

$e = E_{\min}$

$m > 0$

Per i denormalizzati il **valore** dell'esponente (costante) è dato da:  $e - \text{Bias} + 1$  (-126 in single)

$$\text{FLT\_MIN}/2 = 5.8...e-39_{10} = 00400000_{16}$$

0	000 0000 0	100 0000 0000 0000 0000 0000
---	------------	------------------------------

$$\text{FLT\_MIN}/4 = 2.9...e-39_{10} = 00200000_{16}$$

0	000 0000 0	010 0000 0000 0000 0000 0000
---	------------	------------------------------

$$\text{FLT\_MIN}/8 = 1.4...e-39_{10} = 00100000_{16}$$

0	000 0000 0	001 0000 0000 0000 0000 0000
---	------------	------------------------------

...

$$\text{FLT\_MIN}/2^{23} = 1.4...e-45_{10} = 00000001_{16}$$

0	000 0000 0	000 0000 0000 0000 0000 0001
---	------------	------------------------------

## ESEMPIO 6b: Numeri denormalizzati (bit implicito=0) in doppia precisione

Per i denormalizzati il **valore dell'esponente** (costante) è dato da:  $e - \text{Bias} + 1$   
(-1022 in double)

$$\text{DBL\_MIN}/2 = 00080000\ 00000000_{16}$$

0	000 0000 0000	1000 0000 0000	..... 0000 0000
---	---------------	----------------	-----------------

$$\text{DBL\_MIN}/4 = 00040000\ 00000000_{16}$$

0	000 0000 0000	0100 0000 0000	..... 0000 0000
---	---------------	----------------	-----------------

$$\text{DBL\_MIN}/8 = 00020000\ 00000000_{16}$$

0	000 0000 0000	0010 0000 0000	..... 0000 0000
---	---------------	----------------	-----------------

...

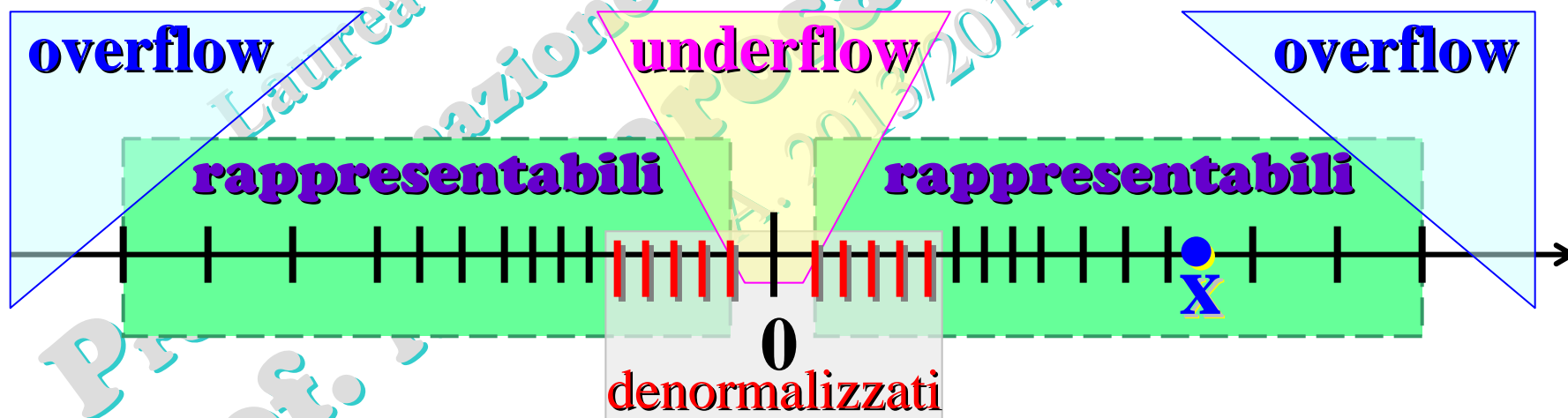
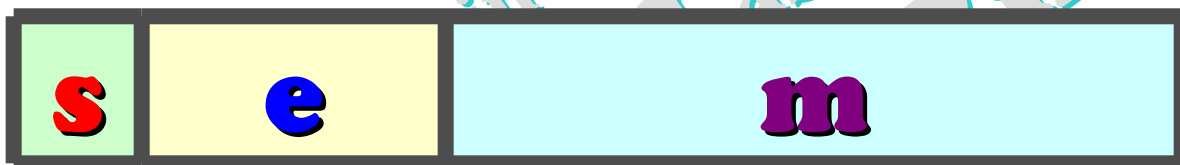
$$\text{DBL\_MIN}/2^{52} = 00000000\ 00000001_{16}$$

0	000 0000 0000	0000 0000 0000	..... 0000 0001
---	---------------	----------------	-----------------

# SCHEMI DI ROUNDING

$$\forall x \in \mathbb{R} : x = \pm 1.x_1x_2x_3 \dots \times 2^p \xrightarrow{?} fl(x)$$

$$fl(x) = (s, e, m)$$



Per gli **x** rappresentabili come determinare la mantissa **m** di  $fl(x)$  ?

# SCHEMI DI ROUNDING

Ad ogni numero reale **rappresentabile** viene associato il suo rappresentante floating-point mediante uno schema di *rounding*:

$$\forall x \in \mathbb{R} \longrightarrow fl(x) \in F(2, t, E_{\min}, E_{\max})$$

Il **S.A. Standard IEEE** prevede 4 schemi di rounding:

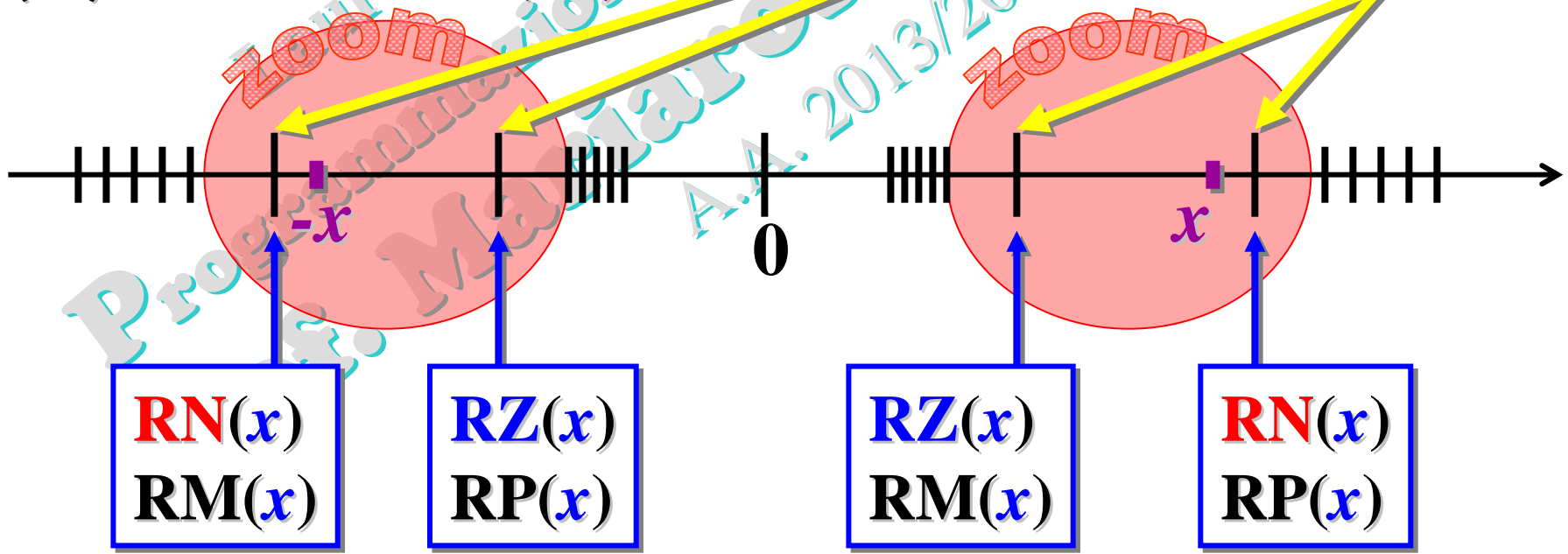
(RN) Round to nearest (round) ← (default)

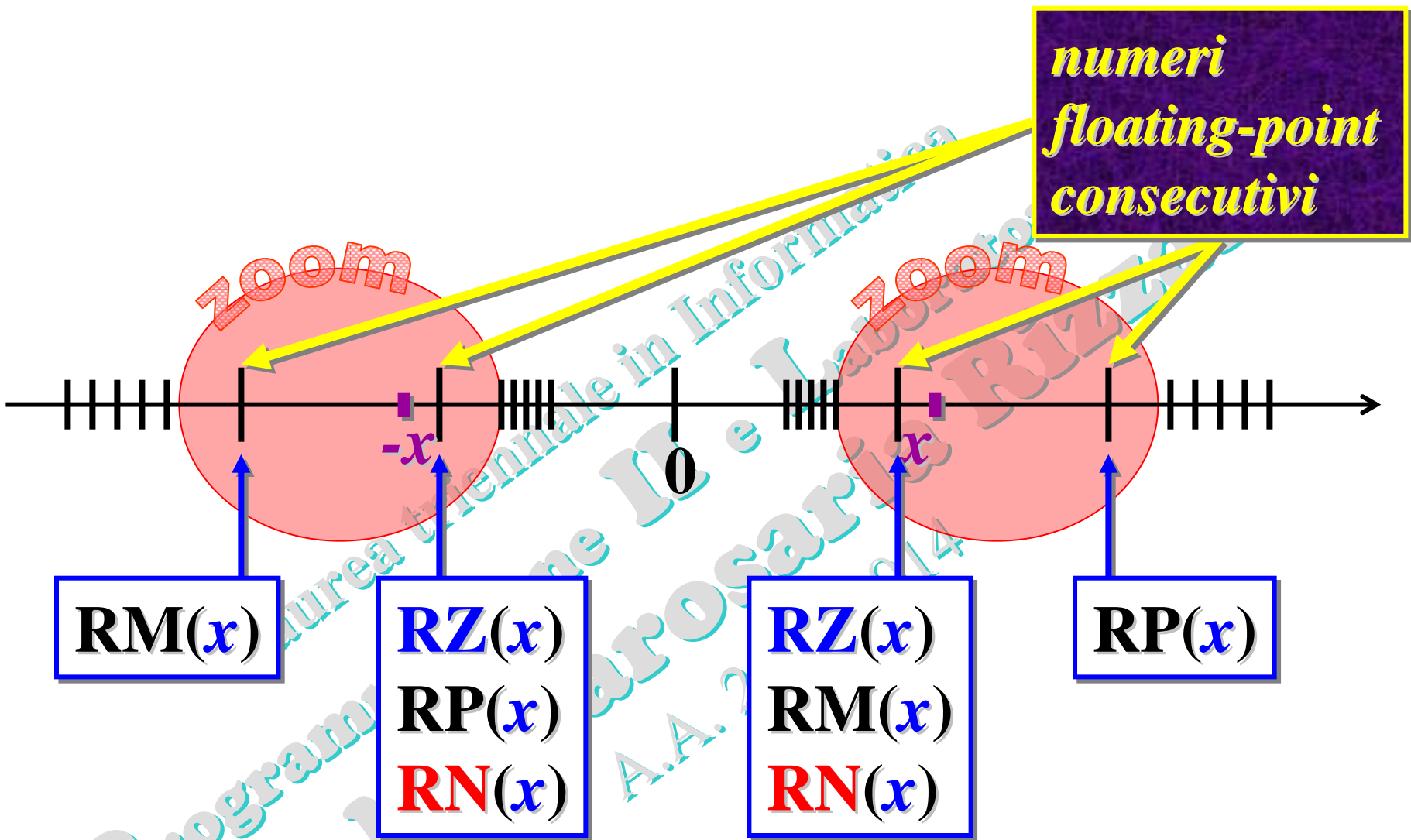
(RZ) Round toward 0 (fix) [troncamento]

(RM) Round toward  $-\infty$  (floor)

(RP) Round toward  $+\infty$  (ceil)

**numeri  
floating-point  
consecutivi**





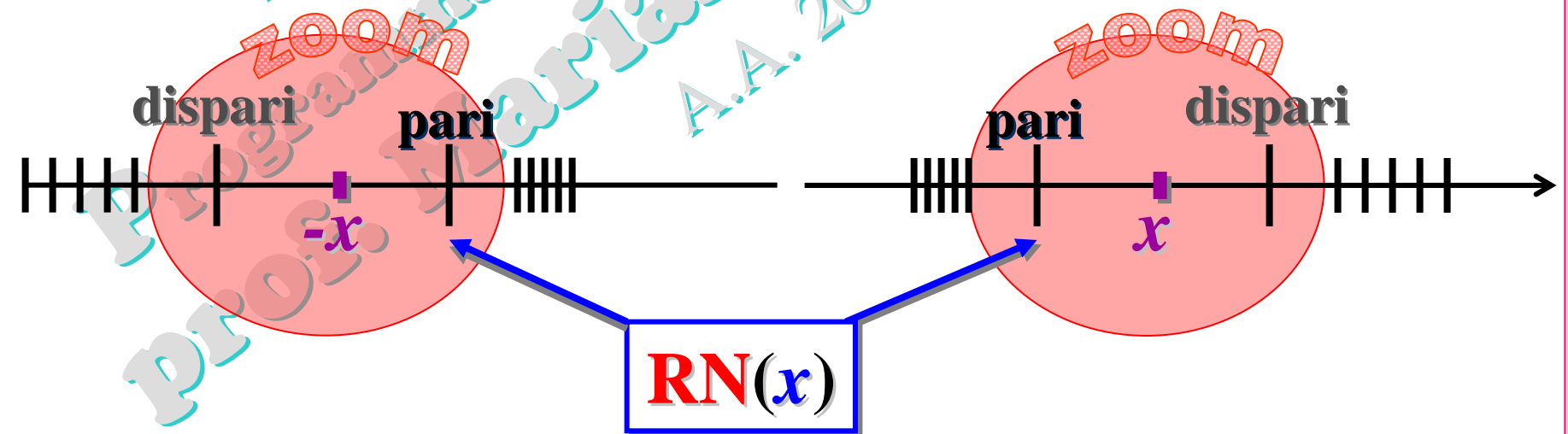
**i 4 SCHEMI DI ROUNDING sono diversi!**



# ROUND TO NEAREST (default)

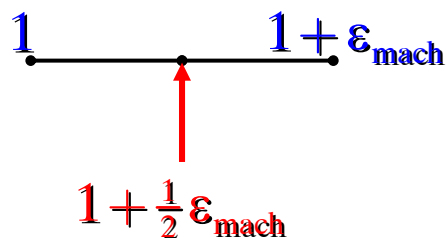
Lo schema **RN** (Round to nearest) è l'**arrotondamento** in base al quale il valore del primo bit della mantissa di  $x$  da eliminare [il  $(t+2)^{\text{esimo}}$ ] influenza la mantissa di  $fl(x)$ : se questo bit è 1 allora si aggiunge 1 al **bit meno significativo (ulp=Unit in the Last Place)** della mantissa di  $fl(x)$ .

Per evitare **errori sistematici**, nel caso in cui la mantissa di  $x$  sia equidistante dalle mantisse di due numeri Floating Point consecutivi, il **Round to nearest** la approssima con la mantissa che tra le due è pari.





# Esempio 7a: Round to nearest in singola precisione



1			3f800000					
0	011 1111 1		000 0000 0000 0000 0000 0000					

Epsilon-macchina

$\epsilon_{mach} = \min\{f > 0 \text{ numero normalizzato tale che } fl(1 + f) > 1\}$

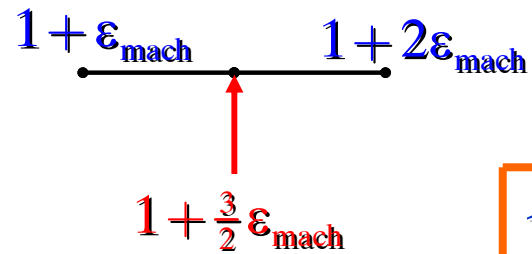
$1 + 1/2 \epsilon_{mach}$			3f800000		
0	011 1111 1	000 0000 0000 0000 0000 0000			



**RN**

$\epsilon_{mach} = \text{FLT\_EPSILON}$  (<float.h>)


$1 + \epsilon_{mach}$			3f800001		
0	011 1111 1	000 0000 0000 0000 0000 0001			



$1 + \epsilon_{mach}$		3f800001
0	011 1111 1	000 0000 0000 0000 0000 0001

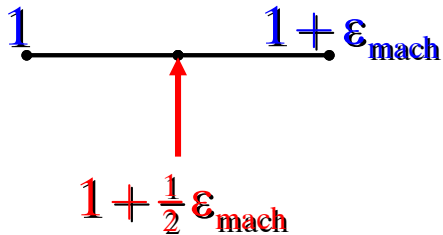
$1 + \frac{3}{2}\epsilon_{mach}$		3f800002
0	011 1111 1	000 0000 0000 0000 0000 0010

**RN**



$1 + 2\epsilon_{mach}$		3f800002
0	011 1111 1	000 0000 0000 0000 0000 0010

# Esempio 7b: Round to nearest in doppia precisione



1	3ff0000000000000	
0	011 1111 1111	0000 0000 0000 .... 0000 0000

Epsilon-macchina

$\epsilon_{mach} = \min\{f > 0 \text{ numero normalizzato tale che } fl(1 + f) > 1\}$

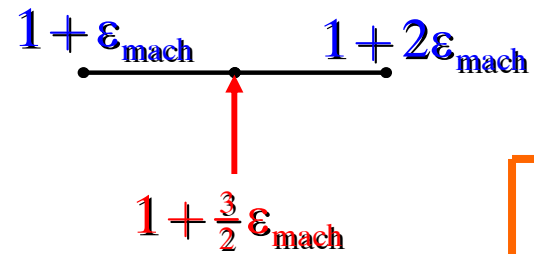
$1 + \frac{1}{2}\epsilon_{mach}$	3ff0000000000000	
0	011 1111 1111	0000 0000 0000 .... 0000 0000



**RN**

$\epsilon_{mach} = \text{DBL\_EPSILON}$  (<float.h>)

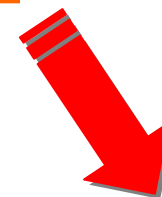
$1 + \epsilon_{mach}$	3ff0000000000001	
0	011 1111 1111	0000 0000 0000 .... 0000 0001



$1 + \epsilon_{\text{mach}}$		3ff000000000000001
0	011 1111 1111	0000 0000 0000 .... 0000 0001

$1 + 3/2\epsilon_{\text{mach}}$		3ff000000000000002
0	011 1111 1111	0000 0000 0000 .... 0000 0010

**RN**



$1 + 2\epsilon_{\text{mach}}$		3ff000000000000002
0	011 1111 1111	0000 0000 0000 .... 0000 0010

# Esercizi

1

Scrivere una *function* *C* per visualizzare la rappresentazione binaria (s,e,m) di un numero *float*. Verificare che il valore del numero ottenuto dalla terna coincida con il dato iniziale.

2

Scrivere una *function* *C* di conversione di un numero reale dalla base 10 alla rappresentazione floating-point IEEE Std 754 binaria (single o double). Come input viene fornita una stringa di caratteri contenenti il numero da convertire. [liv. 3]