Modulo: Approfondimenti sui Sistemi Aritmetici di un computer: tipo reale [P2_03]

Unità didattica: Errori di roundoff [3-AT]

Titolo: Accuratezza statica e dinamica del S. A. Standard IEEE 754

Argomenti trattati:

- ✓ Misure di errore (errore assoluto ed errore relativo) e loro significato
- ✓ Errore di roundoff di rappresentazione (accuratezza statica)
- ✓ Errore di roundoff delle operazioni aritmetiche floating-point (accuratezza dinamica)
- ✓ Ottimalità del S.A. Standard: massima accuratezza statica e massima accuratezza dinamica

Prerequisiti richiesti: Rappresentazione dei numeri floa-ting-point

PRECISIONE FINITA DEL SISTEMA ARITMETICO FLOATING-POINT

Errore di roundoff

statico

dinamico

rappresentazione in memoria

risultato di operazioni aritmetiche

Misure di errore

Se x indica il valore esatto ed $\tilde{x}$ una sua approssimazione ($\tilde{x}=fl(x)$), per misurare l'accuratezza di $\tilde{x}$ rispetto ad x si usano:

Errore assoluto $E_A(\tilde{x}) = |x - \tilde{x}|$

Errore relativo $E_R(\tilde{x}) = \left|\frac{x - \tilde{x}}{x}\right|$ per $x \neq 0$

Quali informazioni danno gli errori E_A ed E_R sul grado di approssimazione?

Errore assoluto E_A ⇒ ***cifre decimali corrette***

Errore relativo E_R ⇒ ***cifre significative corrette***

... cosa significa ???

$$x = 12,34567 = 1.234567\,e{+}02$$

cifre intere

cifre decimali o frazionarie

cifre significative

ordine di grandezza

Proprietà 1

Se $\tilde{x}$ è un'approssimazione di x corretta a p cifre decimali, allora si ha $|x-\tilde{x}| < 10^{-p}$

Proprietà 2

Se $\tilde{x}$ è un'approssimazione di x corretta a p cifre significative, allora si ha $\dfrac{|x-\tilde{x}|}{|x|} < 10^{-p+1}$

Esempio: proprietà 1

$$x = 12.34999$$

$$y = \tilde{x} = 12.34$$

p=2 cifre decimali corrette

Errore assoluto in y $= E_A(y) =$ 9.9900e-003 < $10^{-2} = 10^{-p}$

$$x = 12.34999$$

$$y = \tilde{x} = 12.35$$

p=1 cifra decimale corretta

? ← p=4

?

Errore assoluto in y $= E_A(y) =$ 1.0000e-005 < $10^{-4} = 10^{-p}$???

Esempio: proprietà 2

$x = 12.34999$

$y = \tilde{x} = 12.34$ p=4 cifre significative corrette

Errore relativo $= E_R(y) =$ 8.0891e-004 < $10^{-3} = 10^{-p+1}$

$x = 12.34999$

$y = \tilde{x} = 12.35$ p=3 cifre significative corrette ? ← p=5

?

Errore relativo $= E_R(y) =$ 8.0972e-007 < $10^{-6} = 10^{-p+1}$???

Anche se le due proprietà precedenti valgono per una sola implicazione ($\Rightarrow$), nella pratica si suppone l'equivalenza ($\Leftrightarrow$), nel senso che dall'ordine di grandezza degli errori si hanno informazioni sulle cifre (decimali o significative) corrette di un'approssimazione rispetto al corrispondente valore esatto.

Esempio: il seguente programma

```
#include <stdio.h>
#include <math.h> // per pow() e atan() [arcotangente]

void main()
{double double_x, rel_err, ass_err; float float_x; char i;
 for (i=-4; i<5; i=i+2)
     {double_x=4*atan(1.0)*pow(10,i);
      float_x=(float) double_x;

      ass_err=fabs(double_x-float_x);
      rel_err=ass_err/fabs(double_x);

      printf("double = %22.16e\n",double_x);
      printf("single = %22.16e\n",float_x);
      printf("errore assoluto = %8.3e\n",ass_err);
      printf("errore relativo = %8.3e\n",rel_err);
      puts("\n\n");
      }
}
```

calcola gli ***errori di rappresentazione*** del tipo **`float`** …

double(valore esatto!)	**float**	**double**(valore esatto!)
0.00031415926535…	3.1415926059…e–004	3.1415926535…e–004
0.03141592 6535…	3.1415928155…e–002	3.1415926535…e–002
3.1415926535…	3.1415927410…e–000	3.1415926535…e–000
314.15926535…	3.1415927124…e+002	3.1415926535…e+002
31415.926535…	3.1415925781…e+004	3.1415926535…e+004

Errore ass.	**Errore rel.**
4.76…**e–012**	1.51…**e–008**
1.61…**e–009**	5.15…**e–008**
8.74…**e–008**	2.78…**e–008**
5.88…**e–006**	1.87…**e–008**
7.54…**e–004**	2.40…**e–008**

cifre decimali corrette

cifre significative corrette

Errore di rappresentazione (tipo `float`)

Errore ass.	Errore rel.
4.76…e–012	1.51…e–008
1.61…e–009	5.15…e–008
8.74…e–008	2.78…e–008
5.88…e–006	1.87…e–008
7.54…e–004	2.40…e–008

Perché l'errore relativo ha sempre lo stesso ordine di grandezza?

Perché dipende dalle cifre significative della mantissa (nel tipo float del C la precisione è di 24 bit)!

Perché l'**ordine di grandezza** è 10^{-8} ?

precisione = numero di cifre$_\beta$ significative rappresentate

rappresentazione (binaria) fl.p. a bit implicito su **t** bit per la mantissa

⇒ precisione (binaria) = **t+1**

Nell'Aritmetica Standard l'epsilon macchina $\varepsilon_{mach} = 2^{-t}$

è tale che $\frac{|x - fl(x)|}{|x|} \leq \frac{\varepsilon_{mach}}{2}$

Qual'è la precisione binaria?

$$-\log_2\left(\frac{|x - fl(x)|}{|x|}\right) \geq -\log_2\left(\frac{\varepsilon_{mach}}{2}\right) = t+1$$

precisione = numero di cifre$_\beta$ significative rappresentate

rappresentazione (binaria) fl.p. a bit implicito su **t** bit per la mantissa

⇒ precisione (binaria) = **t+1**

Qual'è la precisione decimale equivalente?

$$-\log_{10}\left(\frac{|x-fl(x)|}{|x|}\right) \geq -\log_{10}\left(\frac{\varepsilon_{mach}}{2}\right) \quad \left(\varepsilon_{mach} = 2^{-t}\right)$$

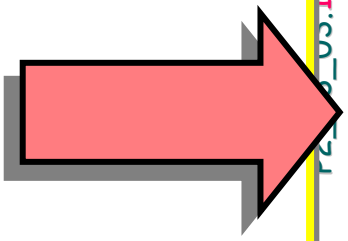
β=2, bit implicito		precisione decimale equivalente	
`float`	t=23 bit	**s.p.**	$-\log_{10}(\ldots) \approx 6.9 \approx 7 \div 8$ cifre decimali
`double`	t=52 bit	**d.p.**	$-\log_{10}(\ldots) \approx 15.9 \approx 16 \div 17$ cifre decimali

perciò l'errore relativo nel tipo float è sempre dell'ordine di 10^{-8}

... a che serve?

È inutile visualizzare più cifre della precisione decimale equivalente

```
#include <stdio.h>
#include <stdlib.h>
void main()
{float a; double b;
 b=0.66666666666666666; a=(float)b;
 printf("a = %...     b = %... \n",a,b);
}
```



formato	s.p.	d.p.
%f	0.666667	0.666667
%e	6.666667e-001	6.666667e-001
%8.3f	0.667	0.667
%8.3e	6.667e-001	6.667e-001
%21.15f	0.666666686534882	0.666666666666667
%21.15e	6.666666865348816 e-001	6.666666666666666 e-001

$$\frac{|x - fl(x)|}{|x|}$$

accuratezza statica

Errore di roundoff di rappresentazione

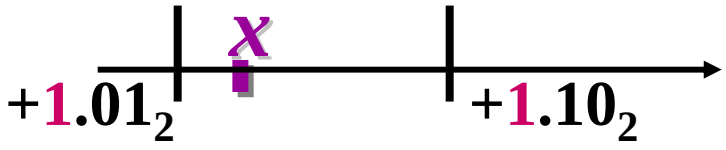
È l'errore introdotto nel passare da un numero reale x a precisione infinita al suo rappresentante floating-point in memoria $fl(x)$

Dipende dalla precisione del Sistema Aritmetico Floating-point e dallo schema di *rounding* utilizzato

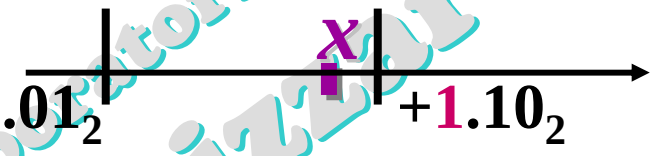
Esempio: schemi IEEE St. 754

$x \notin F(\beta=2, t=2$ (bit implicito)$, E_{min}=0, E_{max}=3)$

$+1.01_2 < x = +1.0101_2 < +1.10_2$



$+1.01_2 < x = +1.0111_2 < +1.10_2$



$fl(x) =$

1.01	RZ
1.01	RN
1.10	RP
1.01	RM

$fl(x) =$

1.01	RZ
1.10	RN
1.10	RP
1.01	RM

L'errore è sempre il minimo per **RN**

$|x-fl(x)| =$

0.0001	RZ
0.0001	RN
0.0011	RP
0.0001	RM

$|x-fl(x)| =$

0.0011	RZ
0.0001	RN
0.0001	RP
0.0011	RM

Proprietà matematica: $\forall a \in \mathbb{R}, \forall \varepsilon > 0 \Rightarrow a + \varepsilon > a$

Nel Sistema Aritmetico Floating-point non vale!

$$\forall a \in \mathbf{F}(\beta, t, E_{\min}, E_{\max}),$$
$$\exists \varepsilon > 0 \quad : \quad a \oplus \varepsilon = a$$

↑ addizione floating-point

Fissato un numero floating-point a: qual è il più piccolo numero floating-point che dà contributo nell'addizione floating-point con a, cioè

$$ulp(a) = min\{0 < \varepsilon \in \mathbf{F}(\beta, t, E_{\min}, E_{\max}) : a \oplus \varepsilon > a\}$$

u.l.p. = **U**nit in the **L**ast **P**lace

Approssimazione di *ulp*(1)

```
#include <stdio.h>
void main()
{ float eps, epsp1; int n;
    eps=1; n=0;
    epsp1=eps+1;
    while (epsp1>1)
    { eps=eps/2; n++;
      epsp1=eps+1;
    }
    eps=2.0f*eps; n--;
printf("num. divisioni per 2=%d\nepsilon=%e \n",n,eps);
}
```

genera successione $\{2^{-n}\}$

calcola $\{1+2^{-n}\}$

```
num. divisioni per 2=23
epsilon=1.192093e-007
```

u.l.p.(1) = ***Epsilon macchina*** ε_{mach}

$$\varepsilon_{mach} = min\{0<\varepsilon\in F(\beta,t,E_{min},E_{max}) : 1 \oplus \varepsilon > 1\}$$

dove $\oplus$ sta per *addizione floating-point*

bit implicito + round to nearest → $\varepsilon_{mach} = 2^{-t}$

Esempio: *Epsilon macchina ε_{mach} in singola precisione*

mostra_float_double.c

FLT_EPSILON **1.19…e–7**	**34000000**		
	0	**011 0100 0**	**000 0000 0000 0000 0000 0000**
+1	**3f800000**		
	0	**011 1111 1**	**000 0000 0000 0000 0000 0000**
1+FLT_EPS	**3f800001**		
	0	**011 1111 1**	**000 0000 0000 0000 0000 0001**
1+.6FLT_EPS	**3f800001**		
	0	**011 1111 1**	**000 0000 0000 0000 0000 0001**
1+.5FLT_EPS	**3f800000**		
	0	**011 1111 1**	**000 0000 0000 0000 0000 0000**

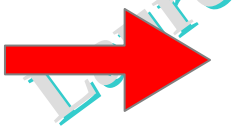
ESEMPIO 7b: Epsilon macchina ε_{mach} in doppia precisione

DBL_EPSILON	**3cb00000 00000000**		
2.22…e–16	**0**	**011 1100 1011**	**0000 0000 0000… 0000 0000**
+1	**3ff00000 00000000**		
	0	**011 1111 1111**	**0000 0000 0000… 0000 0000**
1+DBL_EPS	**3ff00000 00000001**		
	0	**011 1111 1111**	**0000 0000 0000… 0000 0001**
1+DBL_EPS/2	**3ff00000 00000000**		
	0	**011 1111 1111**	**0000 0000 0000… 0000 0000**

Se si elimina la variabile di appoggio **epsp1** dal programma …

```
#include <stdio.h>
void main()
{ float eps; int n;
    eps=1; n=0;
    while (eps+1>1)
       eps=eps/2; n++;
    eps=2.0f*eps; n--;
printf("num. divisioni per 2=%d\nepsilon=%e \n",n,eps);
}
```

epsilon macchina



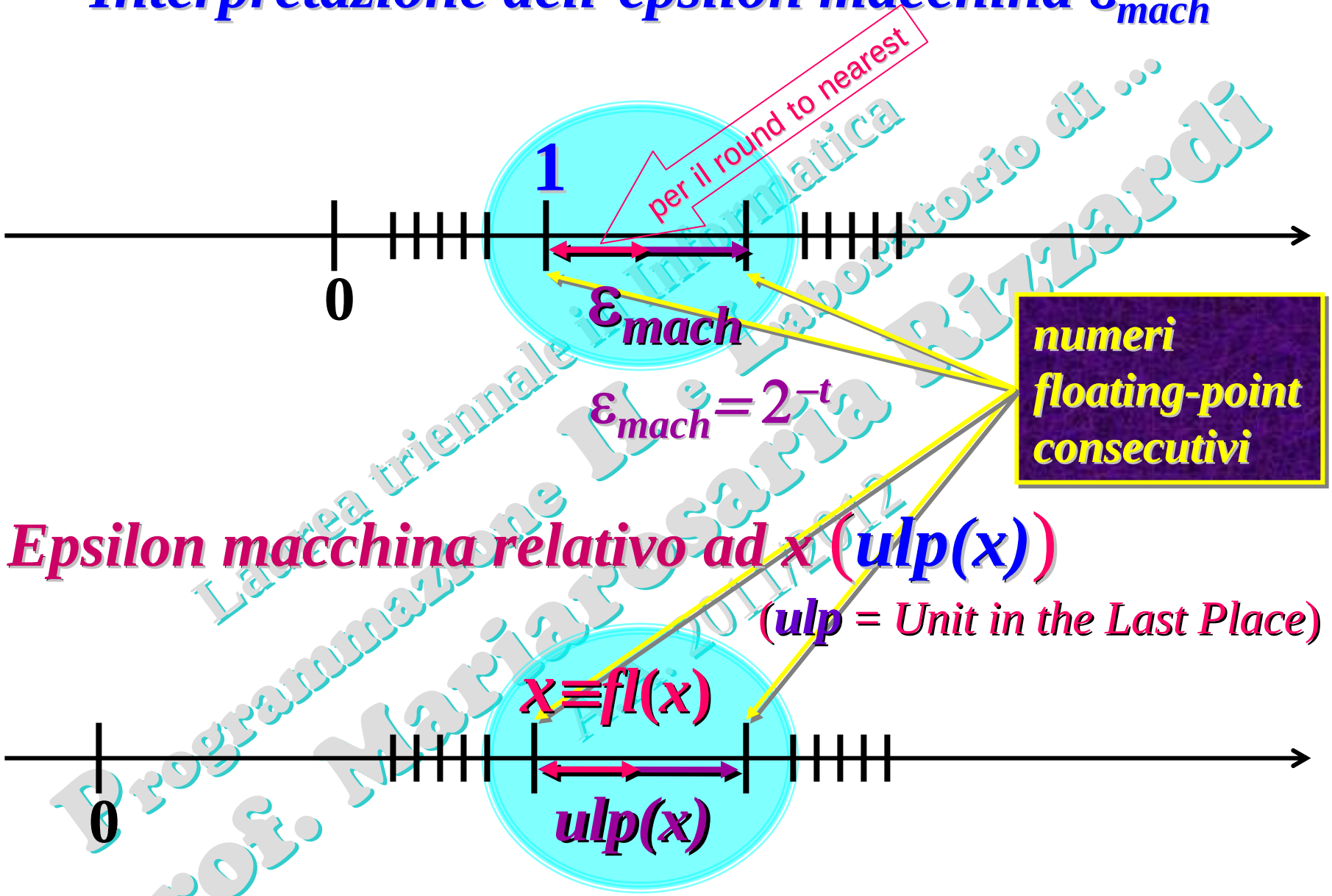
```
num. divisioni per 2=63
epsilon=1.084202e-019
```

???

Senza la variabile di appoggio **epsp1**, il ciclo **while** viene eseguito nei registri dell'unità aritmetica che hanno un campo mantissa maggiore di quello della memoria:

precisione_registro = precisione_formato IEEE std Extended (cioè del tipo **long double** del *C*).

Interpretazione dell'epsilon macchina ε_{mach}



Epsilon macchina relativo ad x ($ulp(x)$)

(*ulp* = *Unit in the Last Place*)

$$x = s \times (\ell . m) \times 2^{e} \quad \rightarrow \quad ulp(x) = \varepsilon_{mach} \cdot 2^{e}$$

Nel S.A. Std., se x è un numero reale rappresentabile e $fl(x)$ indica il corrispondente numero floating point:

$fl(x)$ risultato di massima accuratezza $\Leftrightarrow$ **def** $fl(x) = x(1+\delta)$, $|\delta| \leq \frac{1}{2}\varepsilon_{mach}$

dove $\delta = \dfrac{|x - fl(x)|}{|x|}$, δ **errore relativo**

Mediante il bit implicito e lo schema del round to nearest, il ***S.A. IEEE Std 754*** assicura la massima accuratezza nella rappresentazione in memoria dei numeri reali (***massima accuratezza statica***).

Come assicurare nelle operazioni aritmetiche risultati di massima accuratezza (***massima accuratezza dinamica***) ?

accuratezza dinamica

Si dimosta che nei registri dell'unità aritmetica per assicurare risultati di massima accuratezza (***massima accuratezza dinamica***) sono sufficienti:

- 1 bit per rappresentare il bit implicito (ℓ)
- 2 guard-bit (g) oltre la mantissa
- 1 sticky-bit *

* sticky bit = bit che vale 1 se per esso transita almeno un bit=1

In tal modo con ***registri lunghi*** (*t*+3+ 1 ***sticky***) ***bit*** si ottengono gli stessi risultati aritmetici che si otterrebbero con registri di lunghezza …"infinita" (economia di costi).

Esempio 1: uso dei guard bit

cancellazione in s.p. x-y

$x=1$		3f800000
0	011 1111 1	000 0000 0000 0000 0000 0000
$y=1-2^{-24}$		3f7fffff
0	011 1111 0	111 1111 1111 1111 1111 1111

nei registri, dopo l'allineamento degli operandi …

			ℓ		guard bit		sticky bit
x	0	011 1111 1	1	000 0000 0000 0000 0000 0000	0	0	0
y	0	011 1111 1	0	111 1111 1111 1111 1111 1111	1	0	0
x-y	0	011 1111 1	0	000 0000 0000 0000 0000 0000	1	0	0
x-y	0	011 0011 1	1	000 0000 0000 0000 0000 0000	0	0	0

normalizzazione

Esempio 1: uso dei guard bit

```
1   /* Esempio 1: cancellazione in s.p. x-y */
2   #include <stdio.h>
3   #include <math.h>
4   #define MAX_LEN 32
5
6   void mostra_32_bit(int num, short bit[32])
7   {   char k;
8       for (k=0; k<32; k++) bit[k]=0;
9       for (k=0; k<32; k++)
10      {    bit[k] = num & 1;
11           num = num >> 1;
12      }
13      for (k=31; k>=0; k--)
14          (k==31 | k==23) ? printf("%1d ",bit[k]) : printf("%1d",bit[k]);
15      puts("\n");
16  }
17
18  int main()
19  {   union sp
20      {   float f;
21          int   n;
22      } x, y, z;
23      short bit[MAX_LEN];
24
25      x.f=1.0f;
26      y.f=1.0f-(float)pow(2.0, -24);
27      z.f=x.f-y.f;
28
29      printf("\nx = %e\thex = %08x\n\n",x.f,x.n); mostra_32_bit(x.n, bit);
30      printf("\ny = %e\thex = %08x\n\n",y.f,y.n); mostra_32_bit(y.n, bit);
31      printf("\nz = %e\thex = %08x\n\n",z.f,z.n); mostra_32_bit(z.n, bit);
32
33  return 0;
34  }
```

```
x = 1.000000        hex = 3f800000
0 01111111 00000000000000000000000

y = 9.999999e-001   hex = 3f7fffff
0 01111110 11111111111111111111111

z = 5.960464e-008   hex = 33800000
0 01100111 00000000000000000000000
```

attenzione: cosa cambia se si usa il formato %f?

Esempio 2: uso dello sticky bit

cancellazione in s.p. x-y

$x=1$		3f800000
0	011 1111 1	000 0000 0000 0000 0000 0000
$y=(1+2^{-23})2^{-25}$		33000001
0	011 0011 0	000 0000 0000 0000 0000 0001

nei registri, dopo l'allineamento degli operandi …

					guard bit		sticky bit
x	0	011 1111 1	1	000 0000 0000 0000 0000 0000	0	0	0
y	0	011 1111 1	0	000 0000 0000 0000 0000 0000	0	1	1
x-y	0	011 1111 1	0	111 1111 1111 1111 1111 1111	1	0	1
x-y	0	011 1111 0	1	111 1111 1111 1111 1111 1111	0	1	0

← = *normalizzazione*

Senza lo sticky-bit, nemmeno un registro a lunghezza doppia avrebbe fornito un risultato di massima accuratezza!

Esempio 2a: uso dello sticky bit

```c
/* Esempio 2a: cancellazione in s.p. x-y */
#include <stdio.h>
#include <math.h>
#define MAX_LEN 32

void mostra_32_bit(int num, short bit[32])
{    char k;
     for (k=0; k<32; k++) bit[k]=0;
     for (k=0; k<32; k++)
     {     bit[k] = num & 1;
           num = num >> 1;
     }
     for (k=31; k>=0; k--)
          (k==31 | k==23) ? printf("%1d ",bit[k]) : printf("%1d",bit[k]);
     puts("\n");
}

int main()
{    union sp
     {    float f;
          int   n;
     } x, y, z;
     short bit[MAX_LEN];

     x.f=1.0f;
     y.f=(1.0f+(float)pow(2.0, -23))*(float)pow(2.0, -25);
     z.f=x.f-y.f;

     printf("\nx = %e\thex = %08x\n\n",x.f,x.n); mostra_32_bit(x.n, bit);
     printf("\ny = %e\thex = %08x\n\n",y.f,y.n); mostra_32_bit(y.n, bit);
     printf("\nz = %e\thex = %08x\n\n",z.f,z.n); mostra_32_bit(z.n, bit);

return 0;
```

```
x = 1.000000e+000       hex = 3f800000
0 01111111 00000000000000000000000

y = 2.980233e-008       hex = 33000001
0 01100110 00000000000000000000001

z = 9.999999e-001       hex = 3f7fffff
0 01111110 11111111111111111111111
```

versione 2: passo … passo (simulazione senza sticky bit)

Esempio 2b: uso dello sticky bit

```
#include <stdio.h>
#include <math.h>
#define MAX_LEN 32
void mostra_32_bit(int num, s
{    char k;
     for (k=0; k<32; k++) bit[
     for (k=0; k<32; k++)
     {     bit[k] = num & 1; nu
     }
     for (k=31; k>=0; k--)
          (k==31 | k==23) ? pri
     puts("\n");
}
int main()
{    union sp
     {    float f;
          int   n;
     } x, y, z;
     short bit[MAX_LEN];

     puts("\nPasso passo ...\n");
     x.f=1.0f;
     printf("\nx = %e\thex = %08x\n\n",x.f,x.n); mostra_32_bit(x.n, bit);

     y.f=(1.0f+(float)pow(2.0, -23))*(float)pow(2.0, -25);
     printf("\ny = %e\thex = %08x\n\n",y.f,y.n); mostra_32_bit(y.n, bit);

     puts("Allinea y a x");
     y.f=y.f*(float)pow(2.0, -25); // allinea y a x
     printf("\ny = %e\thex = %08x\n\n",y.f,y.n); mostra_32_bit(y.n, bit);

     z.f=x.f-y.f;
     printf("\nz = %e\thex = %08x\n\n",z.f,z.n); mostra_32_bit(z.n, bit);
return 0;}
```

```
Passo passo ...
x = 1.000000e+000       hex = 3f800000
0 01111111 00000000000000000000000

y = 2.980233e-008       hex = 33000001
0 01100110 00000000000000000000001

Allinea y a x
y = 8.881785e-016       hex = 26800001
0 01001101 00000000000000000000001

z = 1.000000e+000       hex = 3f800000
0 01111111 00000000000000000000000
```

Gli esempi sull'uso dei guard bit e dello sticky bit continuano a valere anche quando si usano variabili long double

Esempio 2a

x = 1.0000000000000000000e+000 hex = 00003fff 80000000 00000000
bit=0 011111111111111 10000000000000000000000000000000000000000000000000000000000000000

y = 2.7105054312137611000e-020 hex = 00003fbe 80000000 00000001
bit=0 011111110111110 10000000000000000000000000000000000000000000000000000000000000001

≠

z = 1.0000000000000000000e+000 hex = ffff3ffe ffffffff ffffffff
bit=0 011111111111110 11111111111111111111111111111111111111111111111111111111111111111

Esempio 2b

Passo passo ...
x = 1.0000000000000000000e+000 hex = 00003fff 80000000 00000000
bit=0 011111111111111 10000000000000000000000000000000000000000000000000000000000000000

y = 2.7105054312137611000e-020 hex = 00003fbe 80000000 00000001
bit=0 011111110111110 10000000000000000000000000000000000000000000000000000000000000001

=

Allinea y a x
y = 7.3468396926392969000e-040 hex = 00003f7d 80000000 00000001
bit=0 011111101111101 10000000000000000000000000000000000000000000000000000000000000001

z = 1.0000000000000000000e+000 hex = ffff3fff 80000000 00000000
bit=0 011111111111111 10000000000000000000000000000000000000000000000000000000000000000

Esercizi

1 Scrivere delle function C per calcolare rispettivamente l'epsilon macchina della singola, della doppia precisione e della precisione long double, visualizzando ad ogni passo i singoli bit.

esempio: singola precisione

```
n= 0   eps+1 =  0   10000000  00000000000000000000000
n= 1   eps+1 =  0   01111111  10000000000000000000000
n= 2   eps+1 =  0   01111111  01000000000000000000000
n= 3   eps+1 =  0   01111111  00100000000000000000000
n= 4   eps+1 =  0   01111111  00010000000000000000000
n= 5   eps+1 =  0   01111111  00001000000000000000000
n= 6   eps+1 =  0   01111111  00000100000000000000000
n= 7   eps+1 =  0   01111111  00000010000000000000000
n= 8   eps+1 =  0   01111111  00000001000000000000000
n= 9   eps+1 =  0   01111111  00000000100000000000000
n=10   eps+1 =  0   01111111  00000000010000000000000
n=11   eps+1 =  0   01111111  00000000001000000000000
n=12   eps+1 =  0   01111111  00000000000100000000000
n=13   eps+1 =  0   01111111  00000000000010000000000
n=14   eps+1 =  0   01111111  00000000000001000000000
n=15   eps+1 =  0   01111111  00000000000000100000000
n=16   eps+1 =  0   01111111  00000000000000010000000
n=17   eps+1 =  0   01111111  00000000000000001000000
n=18   eps+1 =  0   01111111  00000000000000000100000
n=19   eps+1 =  0   01111111  00000000000000000010000
n=20   eps+1 =  0   01111111  00000000000000000001000
n=21   eps+1 =  0   01111111  00000000000000000000100
n=22   eps+1 =  0   01111111  00000000000000000000010
n=23   eps+1 =  0   01111111  00000000000000000000001
```

2 Scrivere una function C per calcolare dalla definizione l'ULP(x) dove x è il parametro reale float di input. [liv. 3]

Generando in modo random i bit(*) di un numero reale x (double x), determinare i bit della corrispondente variabile flx (float flx; flx=(float) x). Se il numero x è rappresentabile nel tipo *float* calcolarne l'errore assoluto E_A e relativo E_R (considerando come esatto il double x e come approssimante il float flx) dalle formule

$$E_A(flx) = |x - flx| \qquad E_R(flx) = \frac{|x - flx|}{|x|}$$

[liv. 2]

(*) La funzione C rand(), in stdlib.h, restituisce un intero pseudorandom minore o uguale di RAND_MAX (= $32767_{10} = 7fff_{16} = 0111\ 1111\ 1111\ 1111_2$ - massimo intero positivo su 16 bit).

Come generare a caso i 64 bit di una variabile double?

- Generando i singoli bit random:
  - come resto della divisione per 2 [... come?];
  - associando 0 agli interi < 16384 e 1 agli interi > 16383;
- Generando 4 sequenze random di 16 bit (**4 × 16 = 64**) [... come?]

vedere: Uso di `rand()` in Materiale di supporto