Firmware Division
&
Floating pointer adder

Prof. Alberto Borghese
Dipartimento di Scienze dell'Informazione
borghese@di.unimi.it

Università degli Studi di Milano

Riferimenti sul Patterson: 3.4, 3.5

Sommario

- Divisione intera
- Somma in virgola mobile

La divisione decimale

Dividendo Divisore

```
— - -
2516 : 12 = 209
 116
   8
```

Quoziente

Resto

Dividendo = Divisore * Quoziente (quoto) + Resto

La divisione decimale

```
- - - -
2516 : 12 = 0209
0 =
----
25 -
24=
-----
  11 -
   0=
-------
   116 -
   108 =
   -------
       8      Resto
```

Dividendo = Divisore * Quoziente (quoto) + Resto

La divisione decimale::algoritmo

Passo 1) 2 5 1 6 : 0012 = 0209 — Il 12 nel 2 ci sta 0 volte.
-0 — 12 x 0 = 0
--
2 — **Resto parziale - I**

Passo 2) 2 5 — Considero il 5 del divisore e lo aggiungo al resto parziale. Il 12 nel 25 ci sta 2 volte.
-2 4 — 12 x 2 = 24
-----
1 — **Resto parziale - II**

Passo 3) 1 1 — Considero l'1 del dividendo e lo aggiungo al resto parziale. Il 12 nell'11 ci sta 0 volte.
-0 — 12 x 0 = 0
-----
1 1 — **Resto parziale – III**

Passo 4) 1 1 6 — Considero il 6 del dividendo e lo aggiungo al resto parziale. Il 12 nel 16 ci sta 1 volta.
-1 0 8 — 12 x 9 = 108
------
8 — Resto parziale – IV = RESTO

La divisione tra numeri binari

Divisione decimale fra i numeri a = 1.001.010 e b = 1.000   a : b = ?

1001010 : 1000 = 1   Dividendo : Divisore   74 : 8 = 9 resto 2
1000-

1

1001010 : 1000 = 1001   Quoziente
1000-

1010 Resto parziale
1000-
-----
10 Resto

Dividendo = Quoziente x Divisore + Resto

Confronto tra divisione tra numeri binari e decimali

In DECIMALE:

Ad ogni passo devo verificare QUANTE VOLTE il resto parziale contiene il divisore. Il risultato è un numero che va da 0 a 9 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

In BINARIO:

Ad ogni passo devo verificare SE il resto parziale contiene il divisore. Ovverosia se lo contiene 0 o 1 volta. Il risultato è un numero che può valere {0, 1}.

In DECIMALE:

Il numero che viene sottratto al resto parziale è ottenuto moltiplicando il divisore per una delle cifre da 0 a 9.

In BINARIO:

Il numero che viene sottratto al resto parziale può essere solamente 0 o il DIVISORE stesso

In entrambi i casi il quoziente si forma dalla cifra più significativa, cioè da sinistra a destra.

Peculiarità della divisione

- Come rappresento la condizione “il divisore è contenuto nel resto parziale”?

Esempi:

10 (resto parziale) non è contenuto in 1000 (divisore)

1001 (resto parziale è contenuto in 1000 (divisore)

Per tentativo.

Eseguo la sottrazione.

Risultato >= 0 → il resto parziale è contenuto nel divisore.

Risultato < 0 → il resto parziale non è contenuto nel divisore (è più piccolo del divisore).

Osserviamo che nel secondo caso abbiamo fatto in realtà una sottrazione che non avremmo dovuto effettuare.

NB il calcolatore non può sapere se il resto parziale contiene il divisore fino a quando non ha effettuato la sottrazione.

La divisione tra numeri binari

Divisione decimale fra i numeri a = 100 1010 e b = 000 1000 a : b = ? 74 : 8=?

Nel primo passaggio allineo il divisore con la prima cifra significativa (=1) del dividendo.

```
 _                      _
000 000100 1010 -       000 000100 1010 +       (Resto parziale = dividendo)-divisore * 2^6
000 100000 0000 =       111 100000 0000 =
----------------------  ------------------------
111100100 1010          111100100 1010          ( =  -438|10)      Nuovo resto parziale
```


Note e strategia di implementazione

I bit del dividendo vengono analizzati da sx a dx. Il divisore viene spostato verso dx di 1 bit ad ogni passo.

Il quoziente cresce dal bit più significativo verso il bit meno significativo. Cresce verso dx.

All'inizio il divisore è allineato alla sinistra del dividendo: le cifre del dividendo sono allineate agli 0 del divisore e gli 0 del dividendo sono allineati alle cifre del divisore.

Ci sono N+1 passi di divisione, il primo darà sempre 0 e si potrebbe omettere.

Occorre quindi effettuare:

- Shift quoziente a sx ad ogni passo.
- Scrittura di 1 o 0 nel registro quoziente.
- Shift del divisore verso dx ad ogni passo..

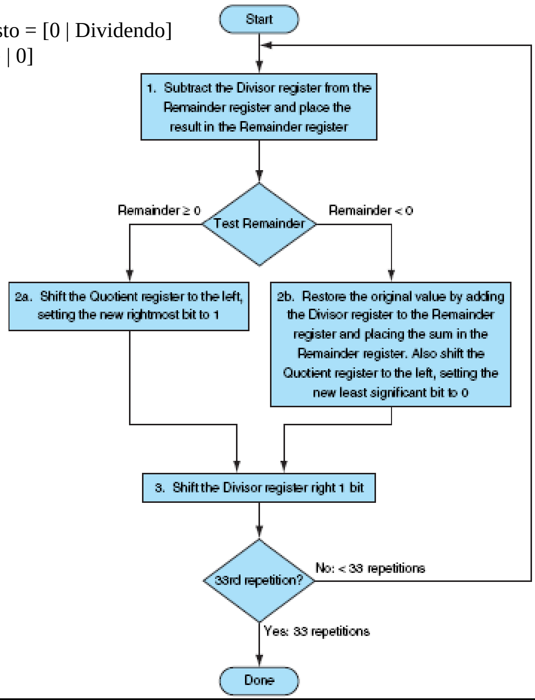
Utilizzo un unico registro per dividendo e resto.

Considero il primo resto parziale uguale al dividendo (inizializzazione).

Il primo passo sarà "a vuoto" perchè produrrà sempre quoziente 0.

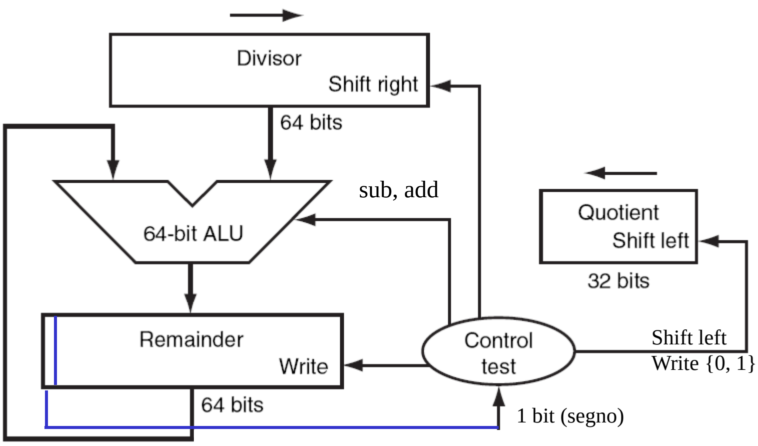
Divisione:: algoritmo per 32 bit

Inizializzazione: Resto = [0 | Dividendo]
Divisore = [Divisore | 0]
Quoziente = 0
k = 0



Il circuito firmware della divisione

Inizializzazione: Resto = 0 | Dividendo



Esempio

Divisione decimale fra i numeri a = 7 e b = 2   a : b = ?

Inizializzo il divisore alla sinistra delle quattro cifre significative. La prima cifra del quoziente sarà sempre 0.

Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

Esempio – step 1

Il resto parziale è inizializzato a: [0 | dividendo]: 0000 0111

0000 0111 : 0010 =

-0010 0000

<0

Il divisore non è contenuto nel resto parziale

Osservando i registri, in pratica, eseguiamo la differenza tra:

0000 0111 –   7   - Resto parziale

0010 0000 =   32 = 2*2^4 - Divisore allineato al di fuori e alla sx dei 4 bit del dividendo.

1110 0111+ -25_{10} < 0

0010 0000=

0000 0111   +7

Quoziente al passo 1: 0

Storno la sottrazione

Esempio – step 2

Il resto parziale è ancora uguale al dividendo: 0000 0111

Il divisore (0001 0000) non è contenuto nel resto parziale

```
 0000 0111 : 0010 =
-0001 0000
---------------
   <0
```

Osservando i registri, in pratica, eseguiamo la differenza tra:

0000 0111 – 7 - Resto parziale
0001 0000 = 16 = $2*2^3$ - Divisore allineato al MSB dei 4 bit del dividendo.

```
-------------
1111 0111+    -9_10 +    < 0
0001 0000=    16
-------------
0000 0111     +7
```

Quoziente al passo 2: 00
Storno ancora la sottrazione

Esempio – step 3

Il resto parziale è ancora uguale al dividendo: 0000 0111

Il divisore (0000 1000) non è contenuto nel resto parziale (0111)

```
 0000 0111 : 0010 =
-0000 1000
   -----------
   <0
```

Osservando i registri, in pratica, eseguiamo la differenza tra:

0000 0111 – 7 - Resto parziale
0000 1000 = 8 = $2 * 2^2$ - Divisore allineato al terzo dei 4 bit del dividendo.

```
-------------
1111 1111+    -1_10    < 0
0000 1000=    +8
-------------
0000 0111     +7
```

Quoziente al passo 3: 000
Storno ancora la sottrazione

Esempio – step 4

Il resto parziale è ancora uguale al dividendo: 0000 0111

0000 0111 : 0010 =
-0000 0100
----
0 001 > 0

Il divisore (0000 0010) è contenuto nel resto parziale

Osservando i registri, in pratica, eseguiamo la differenza tra:

0000 0111 – Resto parziale
0000 0100 = Divisore allineato al secondo dei 4 bit del dividendo.

0000 0011 3_{10} E' il nuovo resto parziale $7 - 1*2^2 = 3$

Quoziente al passo 4: 0001
Non storno la sottrazione.
Il resto non è più uguale al dividendo.

Esempio – step 5

Il resto parziale = 0000 0011

0000 0111 : 0010 =
-0000 0010

0001 > 0

Il divisore (0010) è contenuto nel resto parziale

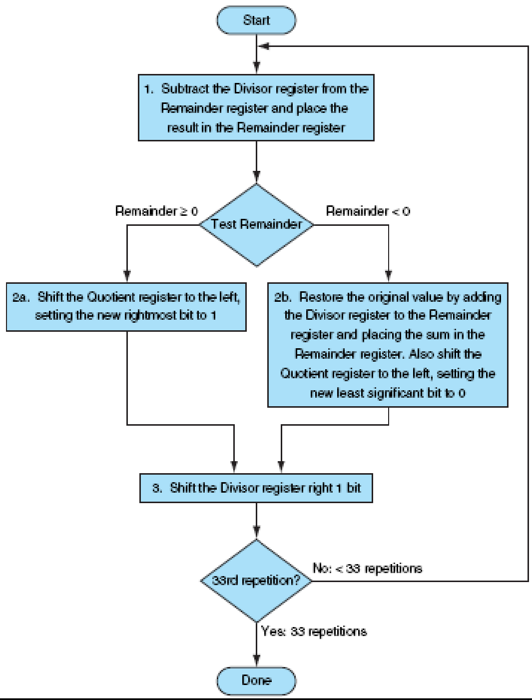
Osservando i registri, in pratica, eseguiamo la differenza tra:

0000 0011 – Resto parziale
0000 0010 = Divisore allineato LSB dei 4 bit del dividendo.

0000 0001 1_{10} E' il nuovo resto parziale $3 - 1*2^1 = 1$ RESTO FINALE

Quoziente al passo 5: 00011

Divisione:: algoritmo

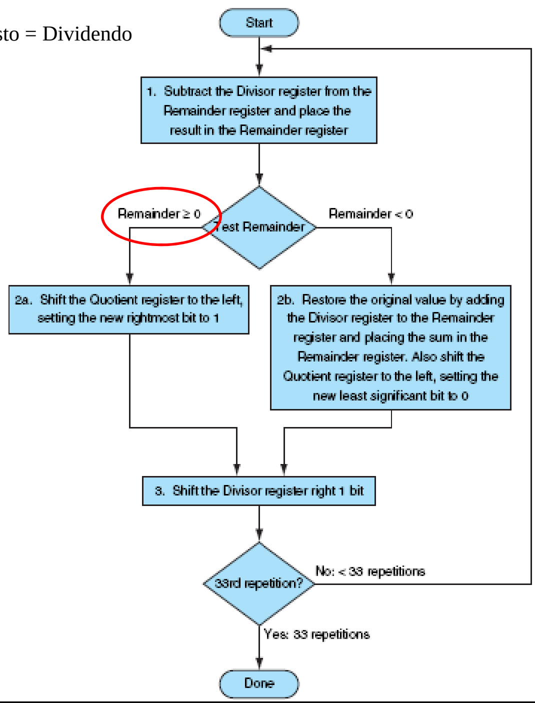


Divisione:: passo 1

Inizializzazione: Resto = Dividendo

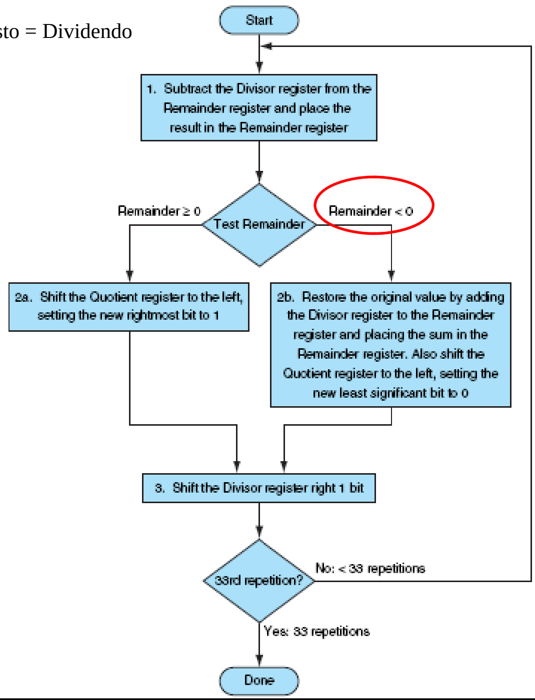
1001010 : 1000 = 1
-1000

1010



Divisione:: passo 2

Inizializzazione: Resto = Dividendo



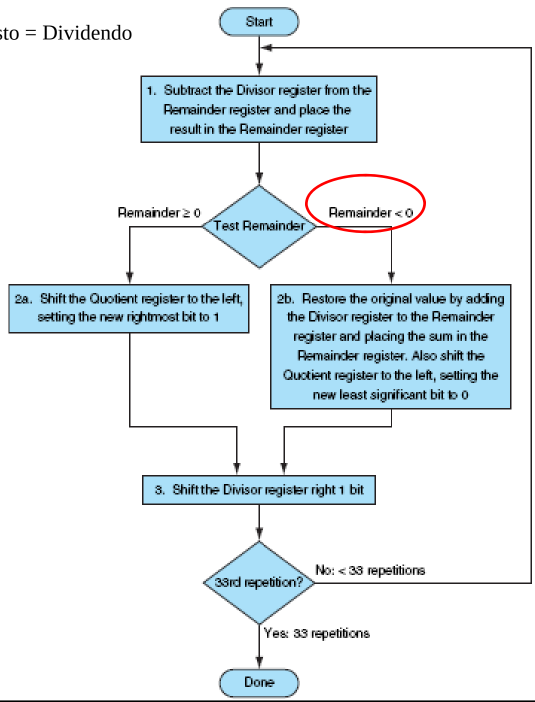
1001010 : 1000 = 10
-1000-

1010
-1000

< 0

Divisione:: passo 3

Inizializzazione: Resto = Dividendo



1001010 : 1000 = 100
-1000-

1010
-1000

< 0

Inizializzazione: Resto = Dividendo

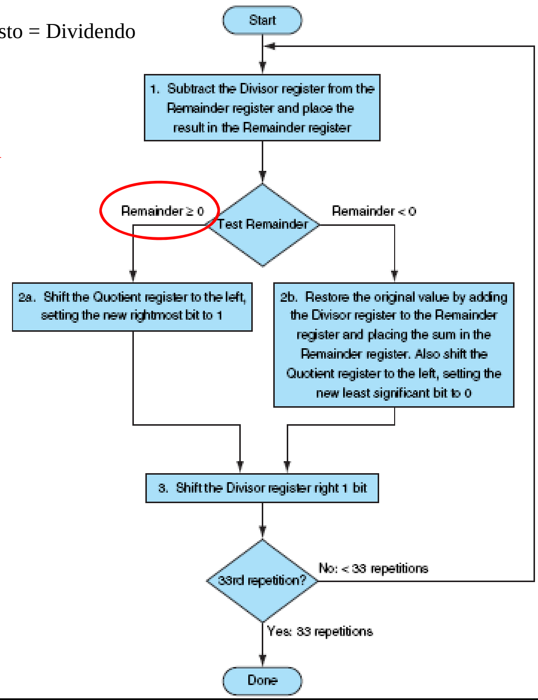
1001010 : 1000 = 1001

-1000-

1010

-1000

10



Divisione:: passo 4

Come ottimizzare il circuito della divisione

Il resto si sposta di 1 bit alla volta verso dx ma rimane pari al numero di bit della parola. Possiamo evitare di spostare il resto.

Il divisore si sposta verso dx di un bit ad ogni passo e divide il resto parziale. Otteniamo lo stesso risultato se spostiamo il resto parziale a sx di un bit ad ogni passo.

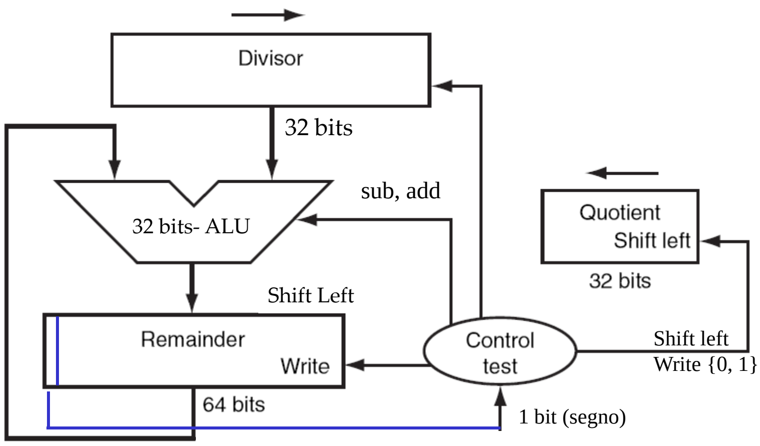
Il quoziente si sposta verso sx ad ogni passo.

Inizializziamo il resto come RESTO = 0 | DIVIDENDO

Ad ogni passo sposto il dividendo di una posizione a sx ed inserisco un bit del quoziente.

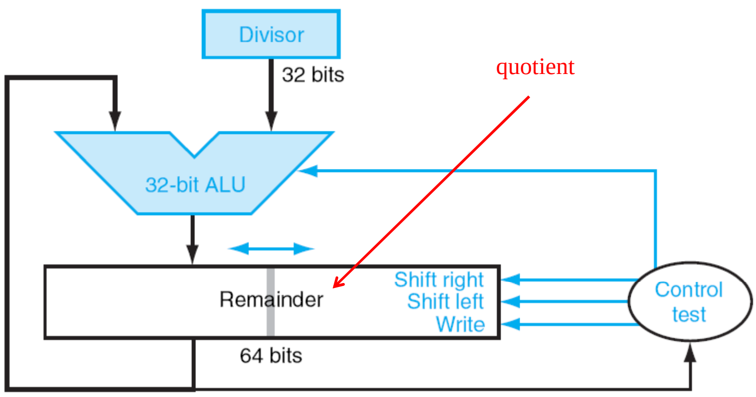
Il circuito firmware con un’ottimizzazione

Inizializzazione: Resto = 0 | Dividendo

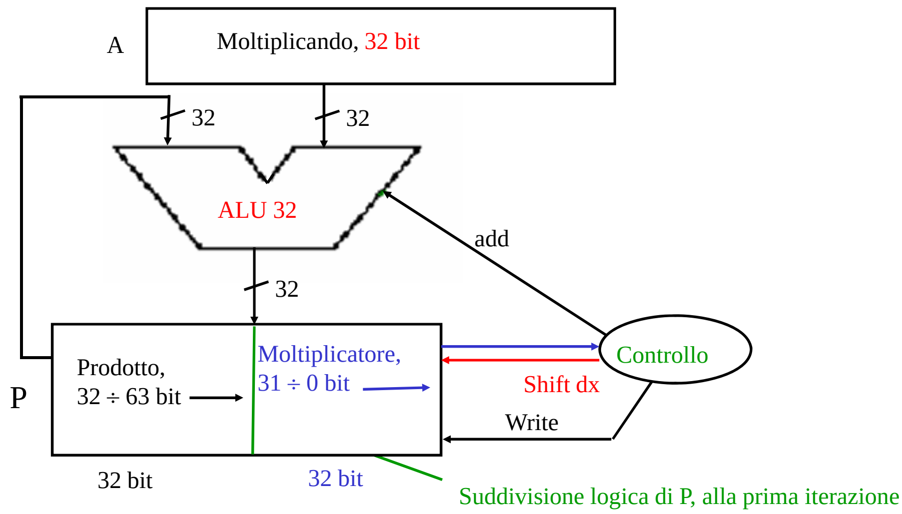


Il circuito firmware ottimizzato della divisione

Inizializzazione: Resto = 0 | Dividendo

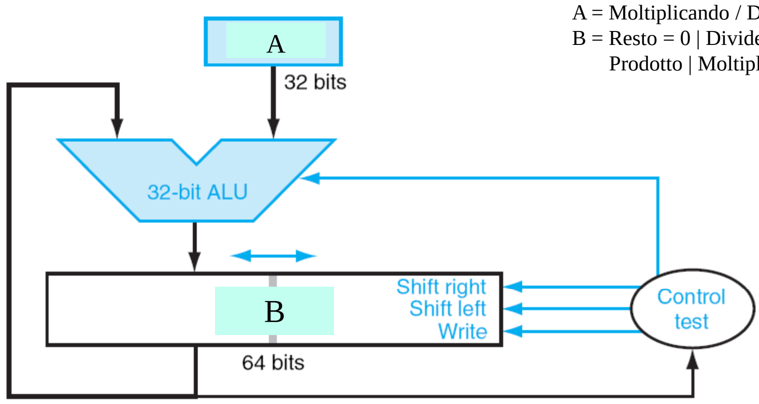


Circuito ottimizzato della moltiplicazione



Un unico circuito per moltiplicazione e divisione

Inizializzazione:
A = Moltiplicando / Divisore
B = Resto = 0 | Dividendo / Prodotto | Moltiplicatore



Il segno della divisione

Dividendo = Quoziente x Divisore + Resto

+	+	+	
+	-	-	Qual'è il segno del resto?
-	+	-	
-	-	+	

Esempio:

+7 = +2 x +3 + 1

-7 = -2 x +3 - 1 (sarebbe uguale anche a -2*+4 +1!!, ma cambierebbe il quoziente in funzione del segno del resto!)

+7 = -2 x -3 + 1

-7 = +2 x -3 - 1

Osserviamo che il quoziente è positivo se i segni di Divisore e Dividendo sono concordi.
Il resto ha il segno del Dividendo.

Sommario

- Divisione intera
- Somma in virgola mobile

Codifica in virgola mobile
Standard IEEE 754 (1980)

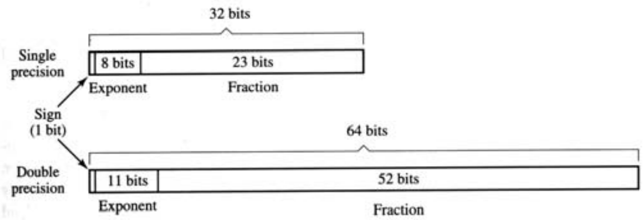


Figure 2-10 Single-precision and double-precision IEEE 754 floating point formats.

Rappresentazione polarizzata dell'esponente:

Polarizzazione pari a 127 per singola precisione =>
1 viene codificato come 1000 0000.

Polarizzazione pari a 1023 in doppia precisione.
1 viene codificato come 1000 0000 000.

Esempio di somma in virgola mobile

a = 9,999 x 10^1 b = 1,61 x 10^{-1} a + b = ?

NB I numeri decimali sono normalizzati.

Una possibilità è:

```
  99,99   +
   0,161 =
---------

100,151
```

100,51 → 1,0015 x 10^2 in forma normalizzata

Algoritmo di somma in virgola mobile - I

1) Trasformare i due numeri in modo che le due rappresentazioni abbiano la stessa base: allineamento della virgola. Quale si allinea?

2) Effettuare la somma delle mantisse.

Se il numero risultante è normalizzato termino qui. Altrimenti:

3) Normalizzare il risultato.

Algoritmo di somma in virgola mobile - II

1) Trasformare **uno dei due numeri** in modo che le due rappresentazioni abbiano la stessa base: allineamento della virgola. Si allinea all'esponente più alto (denormalizzo il numero più piccolo).

2) Effettuare la somma delle mantisse.

Se il numero risultante è normalizzato termino qui. Altrimenti:

3) Normalizzare il risultato.

Esempio di somma in virgola mobile - II

$a = 9{,}999 \times 10^1$ $b = 1{,}61 \times 10^{-1}$ $a + b = ?$

Supponiamo di avere a disposizione 4 cifre per la mantissa e due per l'esponente.

Devo trasformare uno dei numeri, una possibilità è:

$9{,}999 \times 10^1 +$
$0{,}016 \times 10^1 =$ Perdo un bit perchè non rientra nella capacità della mantissa
----
$10{,}015 \times 10^1$ Il risultato non è più normalizzato, anche se i due addendi sono normalizzati.

Normalizzazione per ottenere il risultato finale:

$10{,}015 \times 10^1 = 1{,}001 \times 10^2$ in forma normalizzata.

NB: In questa fase si può generare la necessità di rinormalizzare il numero (passo 3):
Esempio: $9{,}99999 \times 10^2$ => Arrotondo alla cifra più vicina => $10{,}00 \times 10^3$
=> Normalizzazione => $1{,}0 \times 10^4$

Approssimazione

Interi -> risultato esatto (o overflow)

Numeri decimali -> Spesso occorrono delle approssimazioni
- Troncamento (floor): $1{,}001 \times 10^2$ (cf. Slide precedente)
- Arrotondamento alla cifra superiore (ceil): $1{,}002 \times 10^2$
- Arrotondamento alla cifra più vicina: $1{,}002 \times 10^2$

IEEE754 prevede 2 bit aggiuntivi nei calcoli per mantenere l'accuratezza.
bit di guardia (guard)
bit di arrotondamento (round)

Esempio: aritmetica in floating point accurata

a = 2,34   b = 0,0256   a + b = ?   Codifica su 3 cifre decimali totali.
Approssimazione mediante arrotondamento.

Senza cifre di arrotondamento devo scrivere:

2,34 +
0,02 =

2,36

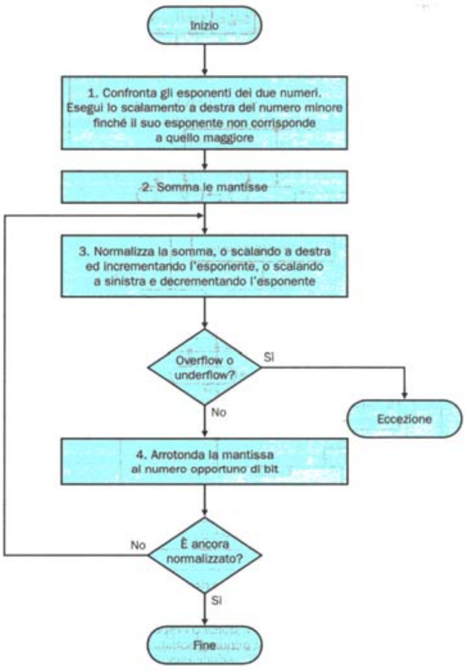
Con il bit di guardia e di arrotondamento posso scrivere:

2,3400 +
0,0256 =

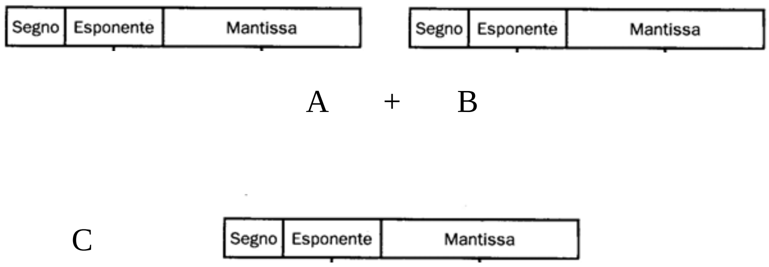
2,3656

L'arrotondamento finale fornisce per rintrare in 3 cifre decimali fornisce: **2,37**

Algoritmo risultante

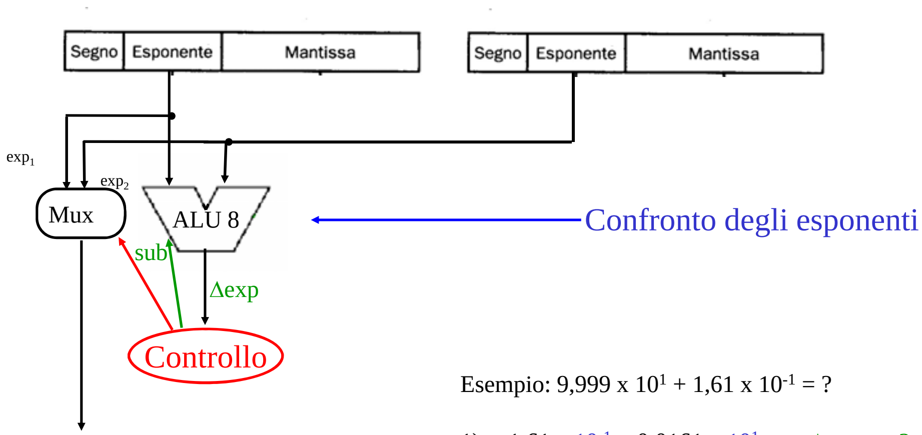


Il circuito della somma floating point: gli attori



Rappresentazione normalizzata IEEE754

Il circuito della somma floating point: determinazione dell'esponente comune

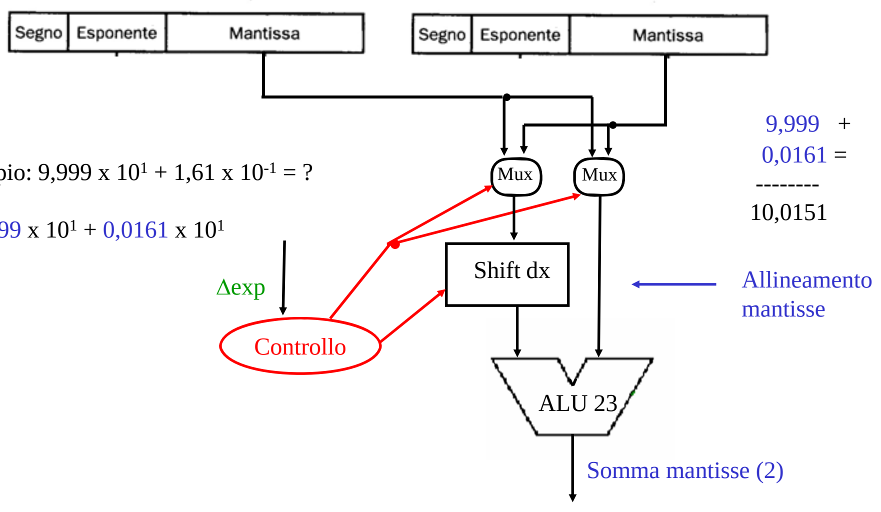


Exp somma = exp1 = 1

Esempio: $9{,}999 \times 10^1 + 1{,}61 \times 10^{-1} = ?$

1) $1{,}61 \times 10^{-1} = 0{,}0161 \times 10^1 \Rightarrow \Delta exp = +2$

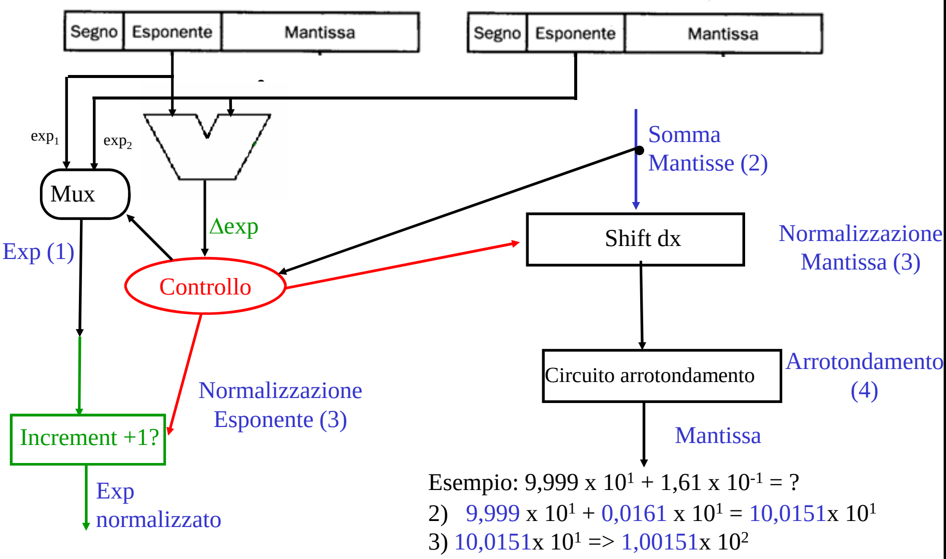
Il circuito della somma floating point: allineamento delle mantisse e somma



Esempio: $9,999 \times 10^1 + 1,61 \times 10^{-1} = ?$

2) $9,999 \times 10^1 + 0,0161 \times 10^1$

Il circuito della somma floating point: arrotondamento e normalizzazione



Esempio: $9,999 \times 10^1 + 1,61 \times 10^{-1} = ?$

2) $9,999 \times 10^1 + 0,0161 \times 10^1 = 10,0151 \times 10^1$

3) $10,0151 \times 10^1 \Rightarrow 1,00151 \times 10^2$

4) $1,00151 \times 10^2 \Rightarrow 1,001 \times 10^2$

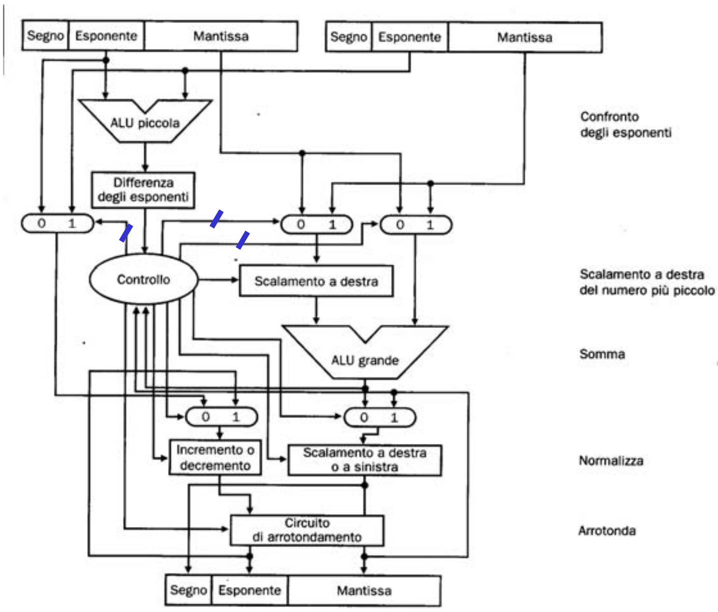
Circuito della somma floating point

Le tre linee in blu contengono lo stesso segnale di controllo.

Gestisce anche la rinormalizzazione: $9,99999 \times 10^2 = 10,00 \times 10^3$

Gestisce anche i numeri negativi.

Problemi?



Esempio

P = 0,5 – 0,4375   somma binaria con precisione di 4 bit.

$A = 1,000 \times 2^{-1}$
$B = -1,110 \times 2^{-2}$
Espressione in forma normalizzata.

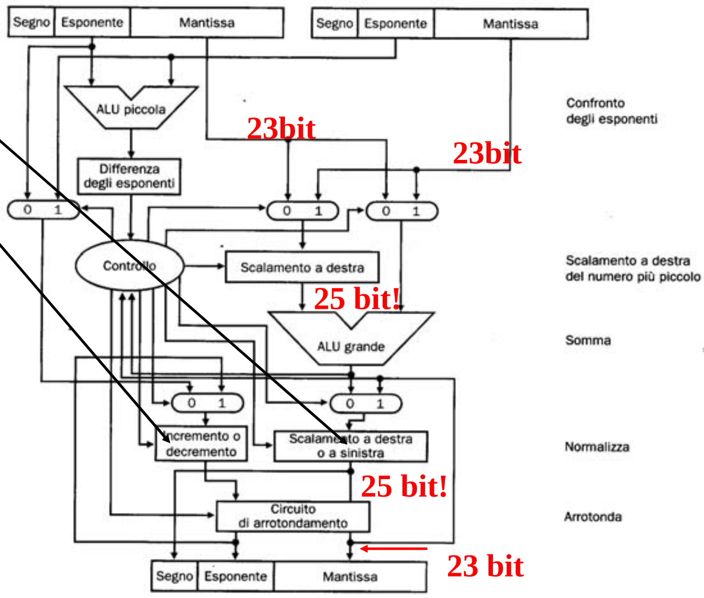
1) Allineamento di A e B. Trasformo B: $-1,110 \times 2^{-2} = -0,1110 \times 2^{-1}$
2) Somma delle mantisse: $1,000 + (-0,111) = 0,001 \times 2^{-1}$
3) Normalizzazione della somma: $0,001 \times 2^{-1} = 1,000 \times 2^{-4}$
   Controllo di overflow o underflow: $-126 \le -4 \le 127$.
4) Arrotondamento della somma: 1,000 non lo richiede.

$1,000 \times 2^{-4} = 1/2^4 = 1/16 = 0,0625$ c.v.d.

Circuito della somma floating point con bit di arrotondamento

In quale caso la mantissa viene scalata a sx?

In quale caso l'esponente viene decrementato?



La rappresentazione interna, secondo IEEE 754, prevede 2 bit aggiuntivi: bit di guardia e bit di arrotondamento.

Prodotto e divisione in virgola mobile

- Prodotto delle mantisse
- Somma degli esponenti
- Normalizzazione

- Divisione in virgola mobile = Prodotto di un numero per il suo inverso.

Sommario

- Divisione intera
- Somma in virgola mobile