


Floating pointer adder
ISA

Prof. Alberto Borghese
Dipartimento di Informatica
alberto.borghese@unimi.it

Università degli Studi di Milano

Riferimenti sul Patterson, 5a Ed.: 3.4, 3.5, 4.2


Sommario

- Somma in virgola mobile
- ISA
- Istruzioni aritmetico-logiche

Segno divisione e moltiplicazione

Moltiplicazione e divisione di numeri positivi -> viene aggiunto il segno alla fine.

Moltiplicazione: XOR dei bit di segno di moltiplicando e moltiplicatore

Se XOR positivo -> prodotto negativo

Divisione: a : b = q + r

XOR dei bit di segno di dividendo e divisore. Se XOR = 1 -> quoziente negativo.

E il resto?

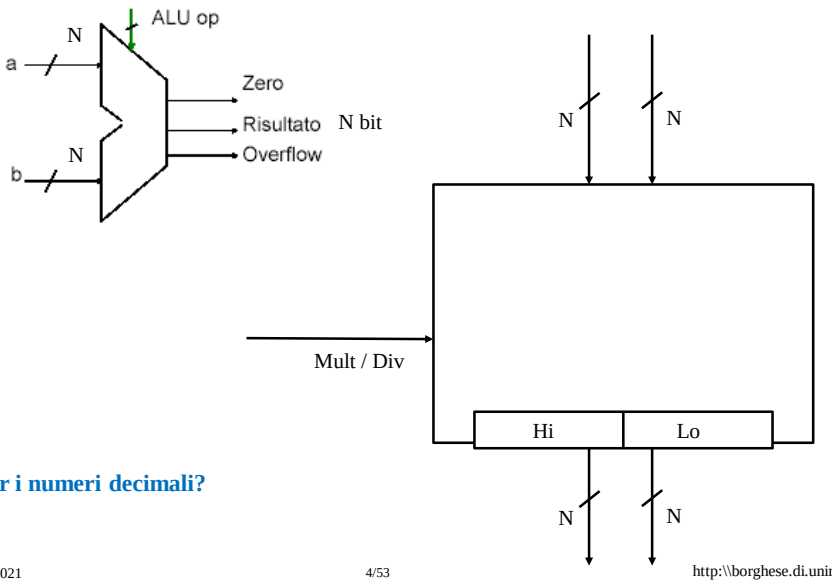
b*q = a – r

a > 0 b > 0 ➔ q > 0, r ≥ 0 7 : (+3) = +2 +1
a > 0 b < 0 ➔ q < 0, r ≥ 0 7 : (-3) = -2 +1

a < 0 b > 0 ➔ q < 0, r ≤ 0 (-7) : (+3) = -2 -1
a < 0 b < 0 ➔ q > 0, r ≤ 0 (-7) : (-3) = +2 -1

Il dividendo e il resto hanno sempre lo stesso segno

Circuiti operazioni tra numeri interi



E per i numeri decimali?

Codifica in virgola mobile
Standard IEEE 754 (1980)

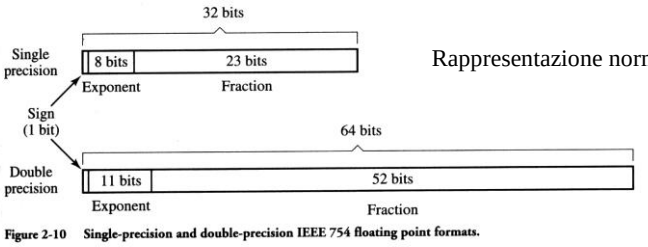


Figure 2-10 **Single-precision and double-precision IEEE 754 floating point formats.**

Rappresentazione normalizzata = 1,…. x 2^N

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 = 7,999 x 10^1 b = 1,61 x 10^{-1} a + b = ?

NB I numeri decimali sono normalizzati -> vanno riportati alla stessa base (incolonnati correttamente):

Una possibilità è:

79,99 + a = 7,999 x 10^1 = 79,99 x 10^0
0,161 = b = 1,61 x 10^{-1} = 0,161 x 10^0

80,151 x 10^0 = 80,151 = 8,0151 x 10^1 in forma normalizzata

Altre possibilità sono:

799,9 + 7,999 +
1,61 = 0,0161 =

801,51 x 10^{-1} = 8,0151 x 10^1 in forma normalizzata = 8,0151 x 10^1

Quale forma conviene utilizzare?

a = $7{,}999 \times 10^{1}$ b = $1{,}61 \times 10^{-1}$ a + b = ?

Supponiamo di avere 4 cifre in tutto per il risultato del prodotto: 1 per la parte intera e 3 per la parte decimale:

79,99 +	799,9 +	7,999 +
0,161 =	1,61 =	0,0161 =
---	---	---
~~8~~0,151 x 10^{0}	~~80~~1,51 x 10^{-1}	8,015~~1~~ x 10^{1}

La rappresentazione **migliore** è:

7,999 +
0,0161 =

8,015~~1~~ x 10^{1}

Risultato normalizzato

Con la quale posso scrivere: 1 cifra prima della virgola (8) e 3 cifre dopo la virgola (015), 1 va perso, ma è la cifra che pesa di meno.

Con la rappresentazione più a sinistra, perdo le decine, con quella in mezzo decine e centinaia commettendo un errore grande sulla rappresentazione.

Allineo al numero con esponente maggiore (perdo cifre di peso minore).

Approssimazione

Interi -> risultato esatto (o overflow)

Numeri decimali -> Spesso occorrono delle approssimazioni

- Troncamento (floor): 8,0151 -> 8,015
- Arrotondamento alla cifra superiore (ceil): 8,0151 -> 8,016
- Arrotondamento alla cifra più vicina: (round) 8,0151 -> 8,015

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

Invece di approssimare gli operandi, consentono di approssimare il risultato.

Esempio: aritmetica in floating point accurata

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

Senza cifre di arrotondamento e utilizzando il troncamento, devo scrivere:

2,34 +
0,02 = ho troncato il secondo addendo per rimanere nella capacità

2,36

Con il bit di guardia e di arrotondamento posso scrivere:

2,3400 +
0,0256 =

2,3656

L'arrotondamento finale (round) viene effettuato **sul risultato** per rientrare in 3 cifre decimali fornisce: **2,37**

L'effetto perverso del troncamento

```
C = A + B
if (C > A) then
   (a)…
else
   (b)….
```

A = 7,999 x 10^1 B = 1,61 x 10^{-1} C = A + B = (7,999 + 0,0161) x 10^1 = 8,0151 x 10^1
*Passando alla codifica su 4 bit con **troncamento degli operandi** ottengo:*
A = 7,999 x 10^1 B = 1,61 x 10^{-1} C = A + B = (7,999 + 0,0161) x 10^1 = 8,015
=> C > A correttamente

A = 7,999 x 10^1 B = 1,61 **x 10^{-4}** C = A + B = 7,999161
*Passando alla codifica su 4 bit con **troncamento degli operandi** ottengo:*
A = 7,999 x 10^1 B = 1,61 x 10^{-4} C = A + B = (7,999 + 0,0000161) x 10^1 = 7,999
=> C = A errore sull'istruzione di test!!!

Questo è un errore molto comune quando si considera l'aritmetica con i numeri decimali

Non vale la proprietà associativa della somma

Z = A + (B + C)

A = -10^{38} B = 10^{38} C = 1

(B + C) = 10^{38} =➔ Z = 0 Risultato sbagliato

Z = (A + B) + C

(A + B) = 0 =➔ Z = 1 Risultato corretto

Risultati molto diversi.

Problemi di arrotondamento – IEEE 754

A = 4
B = 1,000000357627868652343 8 x 10^0

In IEEE754:
A = 1 x 2^2 = 4
B = 1, 00000 00000 00000 00000 011 x 2^0 parte frazionaria su 23 bit

A = 0 1000 0001 00000 00000 00000 00000 000 codifica IEEE754 su 32 bit
B = 0 1111 1111 00000 00000 00000 00000 011 codifica IEEE754 su 32 bit

Allineo B ad A: => B = 0, 01000 00000 00000 00000 000~~11~~ x 2^2 parte frazionaria su 23 bit
Segue che: => C = A + B = 1,01000 00000 00000 00000 000 x 2^2 su 23 bit

0,01000 00000 00000 00000 000~~11~~ + su 23 bit Base = 2^2
1,00000 00000 00000 00000 000 = su 23 bit

1,01000 00000 00000 00000 000 su 23 bit Base = 2^2
C = A+B = 5

Problemi di troncamento – IEEE 754

A = 4

B = 1,00000035762786865234 38 x 10^0

In IEEE754:

A = 1 x 2^2

B = 1, 00000 00000 00000 00000 011 x 2^0 parte frazionaria su 23 bit

A = 0 1000 0001 00000 00000 00000 00000 000 codifica IEEE754 su 32 bit

B = 0 1111 1111 00000 00000 00000 00000 011 codifica IEEE754 su 32 bit

Se aggiungo i bit di guardia e arrotondamento quando allineo B ad A ottengo:

0,01000 00000 00000 00000 00011 + su 25 bit Base = 2^2

1,00000 00000 00000 00000 00000 = su 25 bit

1,01000 00000 00000 00000 00011 su 25 bit Base = 2^2

Per rounding, C = 1,01000 00000 00000 00000 001 su 23 bit Base = 2^2

C = A+B = 5 + $2^{-23}2^2$ = 5+0.000000476837158203125

molto più vicino al valore vero della somma di 5!

Algoritmo di somma in virgola mobile - I

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

a = 9,12 x10^0 b= 8,99 x 10^{-1}

⇓ ⇓

a = 9,12 x10^0 b= 0,899 x 10^0

2) Effettuare la somma delle mantisse.

9,12 +

0,899 =

10,019 x 10^0

Se il numero risultante è normalizzato termino qui. Altrimenti:

3) Normalizzare il risultato.

10,019 x 10^0 ➔ 10019 x 10^1

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.

1) Esprimo entrambi i numeri con la base 10^1
$1{,}61 \times 10^{-1} = 0{,}0161 \times 10^{1}$

2) Somma delle mantisse:

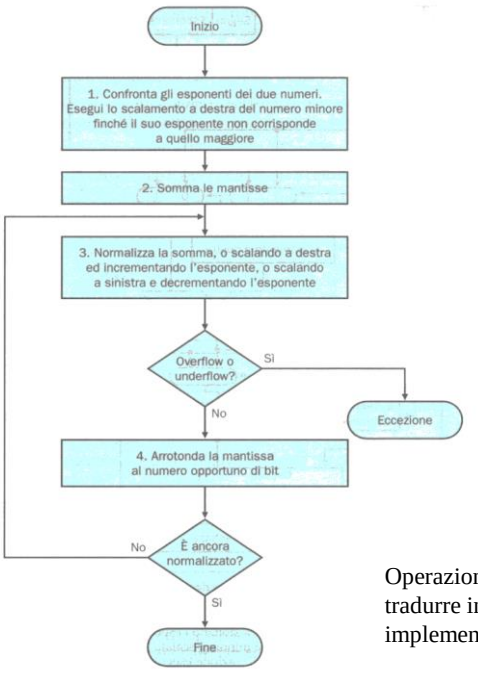
9,999 +
0,016 = Perdo una cifra perchè non rientra nella capacità della mantissa (troncamento)

$10{,}015 \times 10^{1}$ Il risultato non è più normalizzato, anche se i due addendi sono normalizzati.

NB: In questa fase si può generare la necessità di rinormalizzare il numero (passo 3):

$10{,}015 \times 10^{1} = 1{,}001 \times 10^{2}$ in forma normalizzata (per una cifra per effetto del troncamento)

Algoritmo risultante

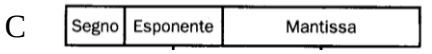


Operazioni complesse da tradurre in operazioni implementabili dall'hardware

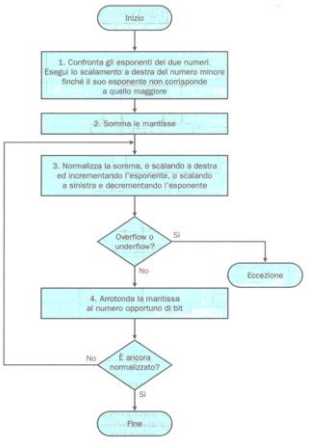
Il circuito della somma floating point: gli attori

A + B

C



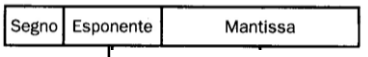
Rappresentazione normalizzata IEEE754

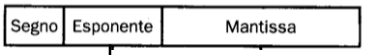


Determinazione dell'esponente

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

A





B

1 −1

exp_A exp_B

Mux ALU 9

sub

Δexp = 2

Sign(Δexp)

Controllo

Confronto degli esponenti
(sottrazione: 8 bit di campo + 1 segno)

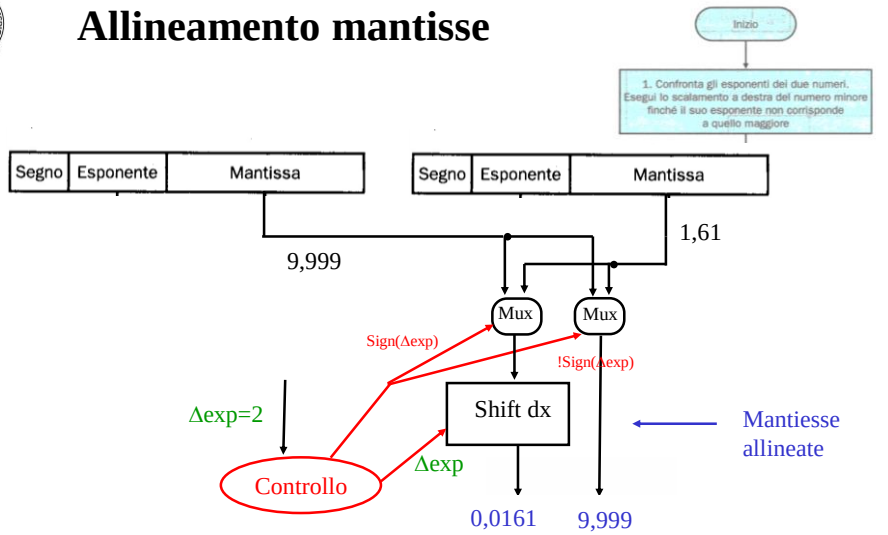
If (Δexp ≥ 0) Exp somma = exp_A
If (Δexp < 0) Exp somma = exp_B

Exp somma = exp_A = +1

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

1a) $1{,}61 \times 10^{-1}$ => $0{,}0161 \times 10^{1}$ => Δexp = +2

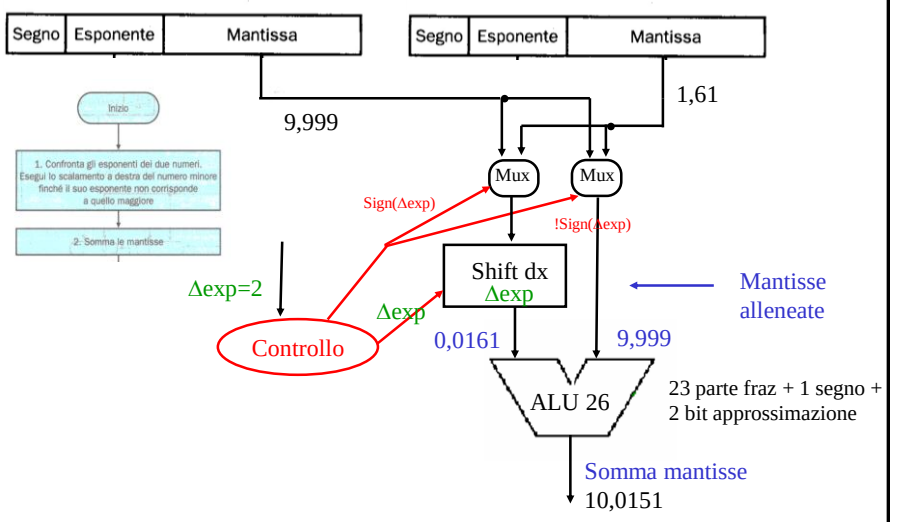
Allineamento mantisse



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

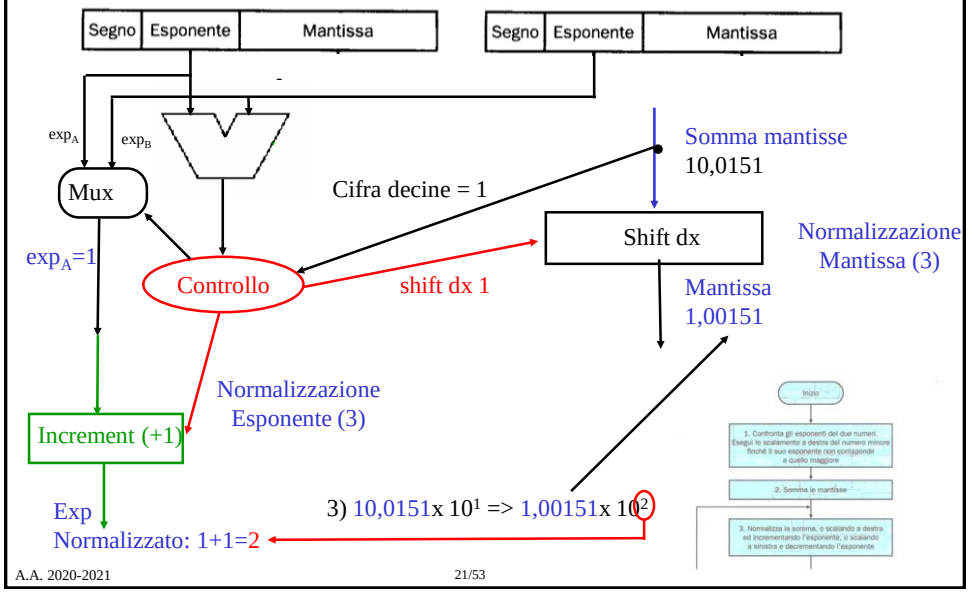
1b) $1{,}61 \times 10^{-1} \Rightarrow 0{,}0161 \times 10^{1} \Rightarrow \Delta exp = +2$

Somma delle mantisse

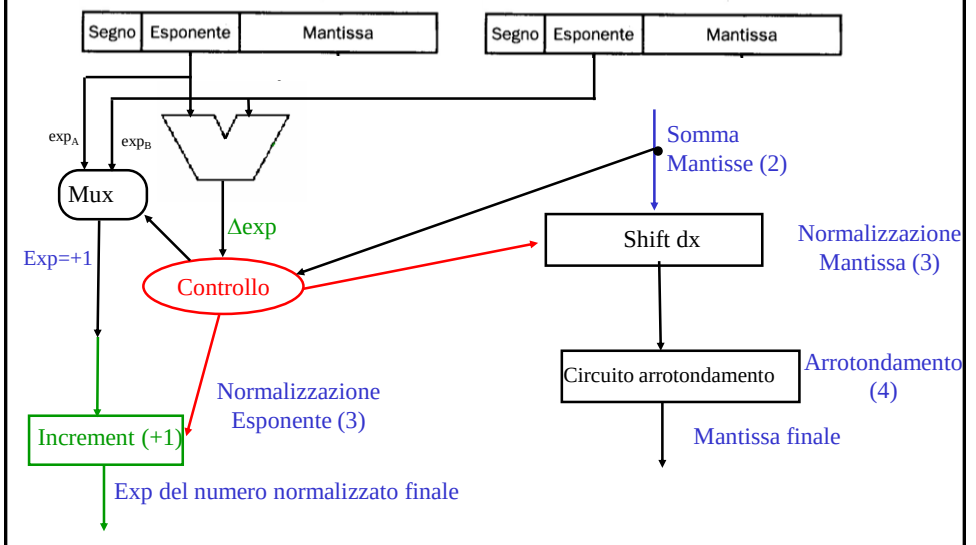


2) Somma delle mantisse: $0{,}0161 \times 10^{1} + 9{,}999 \times 10^{1} = 10{,}0151 \times 10^{1}$

Normalizzazione



Il circuito della somma floating point: arrotondamento e normalizzazione



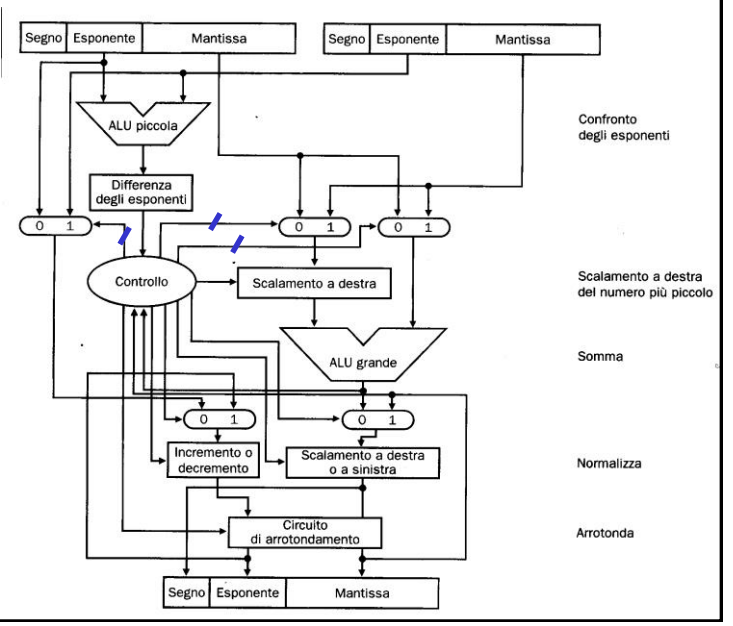
Circuito della somma floating point

Le tre linee in blu contengono lo stesso segnale di controllo (funzione di Δexp).

Gestisce anche la rinormalizzazione: $9{,}99999 \times 10^2 = 10{,}00 \times 10^1$

Gestisce anche i numeri negativi.

Problemi?



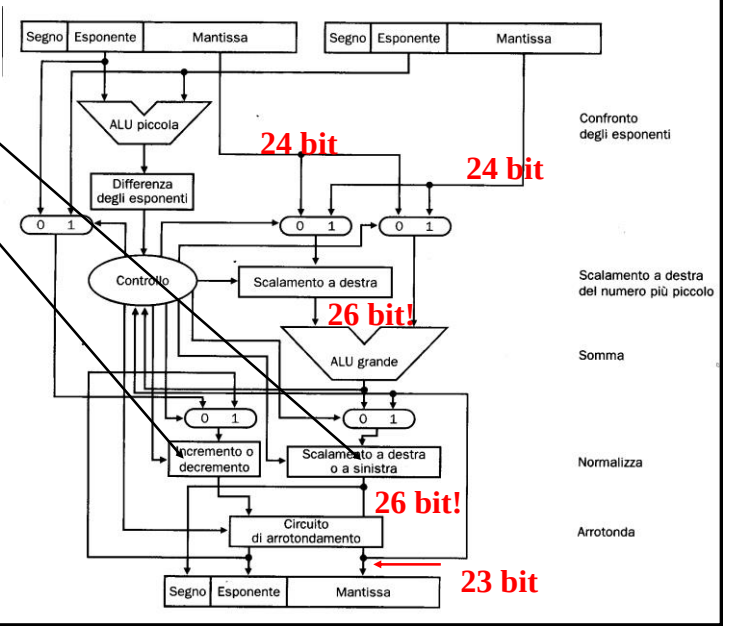
Circuito della somma floating point con bit di arrotondamento

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.

Mantissa 1,…



Prodotto e divisione in virgola mobile

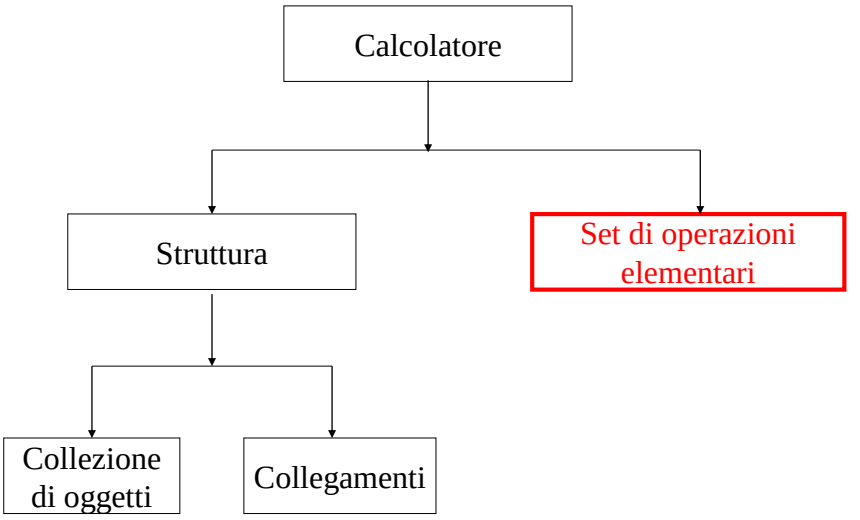
- Prodotto delle mantisse
- Somma degli esponenti
- Normalizzazione

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

Sommario

- Somma in virgola mobile
- ISA
- Istruzioni aritmetico-logiche

Descrizione di un elaboratore



Definizione di un'ISA (Instruction Set Architecture)

ISA: definisce le istruzioni messe a disposizione dalla macchina (in linguaggio macchina).

*Definizione del **funzionamento**: insieme delle istruzioni (interfaccia verso i linguaggi ad alto livello).*

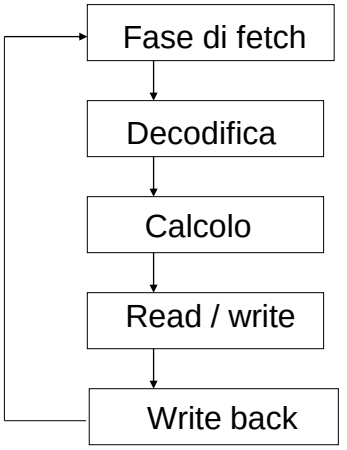
- Tipologia di istruzioni.
- Meccanismo di funzionamento delle istruzioni.

*Definizione del **formato**: codifica delle istruzioni (interfaccia verso l'HW).*

- Formato delle istruzioni.
- Suddivisione in gruppi omogenei dei bit che costituiscono l'istruzione.
- Formato dei dati.

Le istruzioni devono contenere tutte le informazioni necessarie ad eseguire il ciclo di esecuzione dell'istruzione: registri, comandi, ...

Caratteristiche di un'ISA



*Definizione del **funzionamento**: insieme delle istruzioni (interfaccia verso i linguaggi ad alto livello).*

- Tipologia di istruzioni.
- Meccanismo di funzionamento delle istruzioni.

*Definizione del **formato**: codifica delle istruzioni (interfaccia verso l'HW).*

- Formato delle istruzioni.
- Suddivisione in gruppi omogenei dei bit che costituiscono l'istruzione.
- Formato dei dati.

ISA - IPR

ARM (**Advanced RISC Machine** and originally **Acorn RISC Machine**) è una famiglia di architetture di istruzioni.

Acorn, poi diventata RISC Limited, vende le licenze sull' ISA a società che poi realizzano i loro processori RISC. Tra i più diffusi i processori Cortex, alcuni realizzati come SoC – Systems on Chip), FPGA che comprendono memorie, interfacce, radio, ecc..

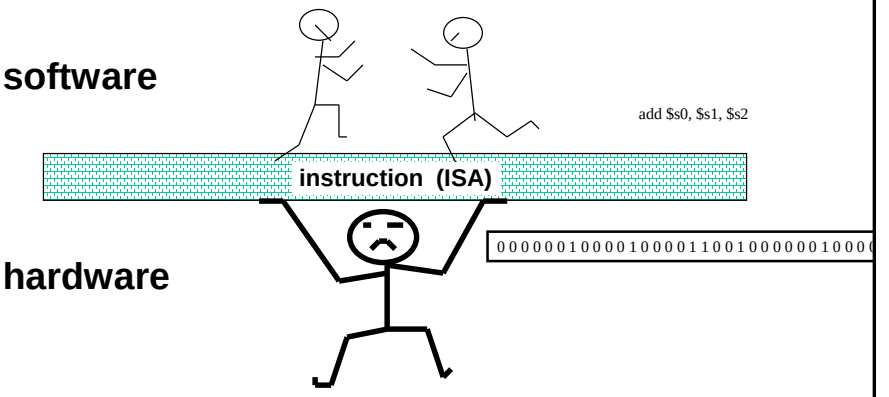
IPR – Intellectual Property Rights (proprietà intellettuale).

Nel 2019. RISC concede la licenza per lo sviluppo con pagamento delle royalities solo a partire dal primo prototipo delle CPU.

Non solo insieme di istruzioni elementari messe a disposizione dalla macchina (in linguaggio macchina).

L'architettura delle istruzioni, specifica come devono essere strutturate le istruzioni in modo tale che siano comprensibili alla macchina (in linguaggio macchina).

Insieme delle istruzioni



Quale è più facile modificare?

Tipi di istruzioni

Il linguaggio macchina (di una calcolatore) è costituito da un insieme di istruzioni che possono essere impartite alla macchina.

- Le istruzioni comprese nel linguaggio macchina di ogni calcolatore possono essere classificate nelle seguenti quattro categorie:
  - Istruzioni aritmetico-logiche;
  - Istruzioni di trasferimento da/verso la memoria *(load/store);*
  - Istruzioni di salto condizionato e non condizionato per il controllo del flusso di programma;
  - Istruzioni di trasferimento in ingresso/uscita (I/O).
- Le istruzioni e la loro codifica costituiscono l'ISA di un calcolatore.

Tipi di istruzioni

```
for (i=0; i<N; i++)          // Istruzioni di controllo
{   elem = i*N + j;          // Istruzioni aritmetico-logiche
    s = v[elem];             // Istruzioni di accesso a memoria
   z[elem] = s;              // Istruzioni di accesso a memoria
}
```


Le istruzioni in linguaggio macchina

- Linguaggio di programmazione direttamente comprensibile dalla macchina
  - Le parole di memoria sono interpretate come *istruzioni*
  - Vocabolario è *l'insieme delle istruzioni (instruction set)*

Codice in linguaggio ad alto livello (C)

```
a = a + c
b = b + a
var = m [a]
```

Codice in linguaggio macchina

```
011100010101010
000110101000111
000010000010000
001000100010000
```

Linguaggio Assembler

- Le istruzioni assembler sono una **rappresentazione simbolica** del **linguaggio macchina** comprensibile dall'HW.

- Rappresentazione simbolica del linguaggio macchina
  - Più comprensibile del linguaggio macchina in quanto utilizza simboli invece che sequenze di bit

- Rispetto ai linguaggi ad alto livello, l'assembler fornisce limitate forme di controllo del flusso e non prevede articolate strutture dati

- Linguaggio usato come linguaggio target nella fase di compilazione di un programma scritto in un linguaggio ad alto livello (es: C, Pascal,ecc.)

- Vero e proprio linguaggio di programmazione che fornisce la visibilità diretta sull'hardware.

Codice in linguaggio ad alto livello (C)

```
a = a + c
b = b + a
var = m[a]
```

Codice Assembler

```
add $t0,$s0,$s1
add $s2,$s2,$t0
muli $t1,$s4,4
add $t2,$s5,$t1
lw $s3,0($t2)
```

Codice in linguaggio macchina

```
011100010101010
000110101000111
000010000010000
001000100010000
```


I registri

- Un registro è un insieme di celle di memora che vengono lette / scritte in parallelo.

- I registri sono associati alle variabili di un programma dal compilatore. Contengono i **dati**.

- Un processore possiede un numero limitato di registri: ad esempio il processore MIPS possiede **32 registri composti da 32-bit (word), register file.**

- I registri possono essere direttamente indirizzati mediante il loro numero progressivo (0, …, 31) preceduto da $: ad es.
  `$0, $1, …, $31`

- Per convenzione di utilizzo, sono stati introdotti nomi simbolici significativi. Sono preceduti da $, ad esempio:

`$s0,$s1,...,$s7($s8)`	Per indicare variabili in C
`$t0, $t1, ... $t9`	Per indicare variabili temporanee

I registri del register file

Nome	Numero	Utilizzo
$zero	0	costante zero
$at	1	riservato per l'assemblatore
$v0-$v1	2-3	valori di ritorno di una procedura
$a0-$a3	4-7	argomenti di una procedura
$t0-$t7	8-15	registri temporanei (non salvati)
$s0-$s7	16-23	registri salvati
$t8-$t9	24-25	registri temporanei (non salvati)
$k0-$k1	26-27	gestione delle eccezioni
$gp	28	puntatore alla global area (dati)
$sp	29	stack pointer
$s8	30	registro salvato (fp)
$ra	31	indirizzo di ritorno

I registri per le operazioni floating point

- Esistono 32 registri utilizzati per l'esecuzione delle istruzioni.
- Esistono **32** registri per le operazioni floating point (virgola mobile) indicati come

  `$f0, …, $f31`

  – Per le operazioni in doppia precisione si usano i registri contigui

  `$f0, $f2, $f4, …`

Linguaggio C: somma dei primi 100 numeri al quadrato

```
main()
{
        int i;
        int sum = 0;
        for (i = 0; i <= 100; i = i + 1)
                sum = sum + i*i;
        printf("La somma da 0 a 100 è
  %d\n",sum);
}
```


Linguaggio Assembler: somma dei primi 100 numeri al quadrato

```
.text
.align 2
.globl main
main:
        add $t6, $zero, $zero     // $t6 = 0  ind ciclo
        add $s0, $zero, $zero     // $s0 variabile da aggiornare
        add $s1, $a0, $zero       // $s1 Indice di fine ciclo, $sa proviene dall'esterno
loop:   beq $t6, $s1, exit        // termine ciclo quando $t6 = $s1
        mult $t4, $t6, $t6        // $t4 = indice*indice
        addu $s0, $s0, $t4        // sommo $t4 a $s0 – $s0 è risultato parziale
        addu $t6, $t6, 1          // incremento ind ciclo
        j loop                    // vai all'inizio del ciclo
                                  .....
```

Assembler come linguaggio di programmazione

- Principali *svantaggi* della programmazione in linguaggio assembler:
  - Mancanza di portabilità dei programmi su macchine diverse
  - Maggiore lunghezza, difficoltà di comprensione, facilità d'errore rispetto ai programmi scritti in un linguaggio ad alto livello
- Principali *vantaggi* della programmazione in linguaggio assembler:
  - Ottimizzazione delle prestazioni.
  - Massimo sfruttamento delle potenzialità dell'hardware sottostante.

- Le strutture di controllo hanno forme limitate
- Non esistono tipi di dati all'infuori di interi, virgola mobile e caratteri.
- La gestione delle strutture dati e delle chiamate a procedura deve essere fatta in modo esplicito dal programmatore.

Assembler come linguaggio di programmazione

- Alcune applicazioni richiedono un approccio *ibrido* nel quale le parti più critiche del programma sono scritte in assembly (per massimizzare le prestazioni) e le altre parti sono scritte in un linguaggio ad alto livello (le prestazioni dipendono dalle capacità di ottimizzazione del compilatore).

  Esempio: Sistemi embedded o dedicati

Sistemi "automatici" di traduzione da linguaggio ad alto livello (linguaggio C) ad Assembly e codice binario ed implementazione circuitale (e.g. sistemi di sviluppo per FPGA).

Sommario

- Somma in virgola mobile
- ISA
- Istruzioni aritmetico-logiche

Tipi di istruzioni

- Le istruzioni comprese nel linguaggio macchina di ogni calcolatore possono essere classificate nelle seguenti quattro categorie:
  - Istruzioni aritmetico-logiche;
  - Istruzioni di trasferimento da/verso la memoria *(load/store);*
  - Istruzioni di salto condizionato e non condizionato per il controllo del flusso di programma;
  - Istruzioni di trasferimento in ingresso/uscita (I/O).

Istruzioni aritmetico-logiche

- In MIPS, un'istruzione aritmetico-logica possiede in generale *tre* operandi: i due registri contenenti i valori da elaborare *(registri sorgente)* e il registro contenente il risultato *(registro destinazione).*

- L'ordine degli operandi è **fisso:** prima il registro contenente il r**isultato** dell'operazione e poi i due operandi.

```
OPCODE  DEST,  SORG1, SORG2
```

- La codifica prevede il codice operativo e tre campi relativi ai tre operandi:

Le operazioni vengono eseguite esclusivamente su dati presenti nella CPU, non su dati residenti nella memoria.

Esempi: istruzioni `add` e `sub`

Codice C:

```
R = A + B;
```

Codice assembler MIPS:

```
add $s6,  $s7, $s8
add rd,   rs,   rt
```

mette la somma del contenuto di rs e rt in rd:

```
add rd, rs, rt        # rd  ← rs +  rt
add $s6, $s7, $s8     # $s6 ← $s7 + $s8
```

Nella traduzione da linguaggio ad alto livello a linguaggio assembler, le variabili sono associate ai registri dal compilatore

`sub` serve per sottrarre il contenuto di due registri sorgente rs e rt:

```
sub rd rs rt
```

e mettere la differenza del contenuto di rs e rt in rd

```
sub rd,   rs,  rt   # rd  ← rs  - rt
sub $s6, $s7, $s8   # $s6 ← $s7 - $s8
```

Istruzioni aritmetico-logiche in sequenza

Il fatto che ogni istruzione aritmetica ha tre operandi sempre nella stessa posizione consente di semplificare l'hw, ma complica alcune cose…

Codice C: `Z = A – (B + C + D); =>`
`E = B + C + D; Z = A – E;`

Suppongo che le variabili siano contenute nei seguenti registri:
A -> $s0 B -> $s1 C -> $s2 D -> $s3 Z -> $s5

Occcorre spezzare la catena di operazioni in tante operazioni su 2 operandi. Codice MIPS:

```
add $t0, $s1, $s2
add $t1, $t0, $s3
sub $s5, $s0, $t1
```


Istruzioni aritmetico-logiche

- Operazioni con un numero di operandi maggiore di tre possono essere effettuate scomponendole in operazioni più semplici.
- Ad esempio, per eseguire la somma e sottrazione delle variabili `A.. D` nella variabile `Z` servono tre istruzioni che eseguono le operazioni in sequenza da sinistra a destra:

Codice C: `Z = A + B – C + D;`

Codice MIPS:

```
add $t0, $s0, $s1
sub $t1, $t0, $s2
add $s5, $t1, $s3
```

Suppongo che le variabili siano contenute nei seguenti registri:
A -> $s0 B -> $s1 C -> $s2 D -> $s3 Z -> $s5

Implementazione alternativa

- Operazioni con un numero di operandi maggiore di tre possono essere effettuate scomponendole in operazioni più semplici.

- Ad esempio, per eseguire la somma e sottrazione delle variabili A.. D nella variabile Z servono tre istruzioni :

  Codice C: **Z = A + B - C + D;**

  Può essere riscritta con il seguente codice C: **Z = (A + B) – (C – D);**
  Suppongo che le variabili siano contenute nei seguenti registri:
  A -> $s0 B -> $s1 C -> $s2 D -> $s3 Z -> $s5

Codice MIPS:

```
add $t0, $s0, $s1      add $t0, $s0, $s1
add $t1, $s2, $s3      sub $t1, $t0, $s2
sub $s5, $t0, $t1      add $s5, $t1, $s3
```

Sono implementazioni equivalenti. Quale implementazione è la migliore? Sceglierà il compilatore il quale cerca di massimizzare la parallelizzazione del codice.

add: varianti

- **addi $s1, $s2, 100 #add immediate: $s1 = $s2+100**
  - Somma una costante: il valore del secondo operando è presente nell'istruzione come costante e sommata estesa in segno.
    rt ← rs + costante

- **addiu $s0, $s1, 100 #add immediate unsigned: $s0 = $s1+100**
  - Somma una costante ed evita overflow.

- **addu $s0, $s1, $s2 #add unsigned: $s0 = $s1+$s2**

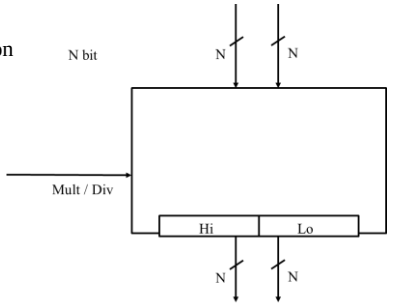
  - Evita overflow: la somma viene eseguita considerando gli addendi sempre positivi. Il bit più significativo è parte del numero e non è bit di segno.

  Non esiste un'istruzione di subi. Perchè?

- **addi $s1, $s2, -20 #add immediate: $s1 = $s2-20**

Moltiplicazione

- Due istruzioni:
  - `mult rs rt` `mult $s0, $s1`
  - `multu rs rt` `multu $s0, $s1` `# unsigned`
- Il registro destinazione è ***implicito.*** Il risultato della moltiplicazione viene posto sempre in due registri dedicati di una parola (special purpose) denominati ***Hi (High order word)*** e ***Lo (Low order word)***
- La moltiplicazione di due numeri rappresentabili con 32 bit può dare come risultato un numero non rappresentabile in 32 bit.



Moltiplicazione

- Il risultato della moltiplicazione si può elaborare prelevando il contenuto del registro **Hi** e del registro **Lo** utilizzando le due istruzioni:
  - `mfhi rd1` `mfhi $t0` `# move from Hi`
    - Sposta il contenuto del registro **hi** nel registro **rd**
  - `mflo rd2` `mfhi $t1` `# move from Lo`
    - Sposta il contenuto del registro **lo** nel registro **rd**

Test sull'overflow

Risultato del prodotto

Sommario

- Somma in virgola mobile
- ISA
- Istruzioni aritmetico-logiche