Aritmetica dei calcolatori

Rappresentazione dei numeri naturali e relativi
Addizione a propagazione di riporto
Addizione veloce
Addizione con segno
Moltiplicazione con segno e algoritmo di Booth
Rappresentazione in virgola mobile e operazioni

versione del 22/10/03

La rappresentazione dei numeri

- Rappresentazione dei numeri: binaria
- Un numero binario è costituito da un *vettore di bit*

$$B = b_{n-1} \ldots b_1 b_0 \qquad b_i = \{0, 1\}$$

- Il valore di *B* e' dato da:

$$V(B) = b_{n-1} \times 2^{n-1} + \ldots + b_1 \times 2^1 + b_0 \times 2^0$$

- Un vettore di *n* bit consente di rappresentare i numeri naturali nell'intervallo da 0 a 2^n-1.
- Per rappresentare i numeri positivi e negativi si usano diverse codifiche

La rappresentazione dei numeri

- Codifiche per numeri relativi
  - Modulo e segno
  - Complemento a 1
  - Complemento a 2

B	V(B)		
$b_2b_1b_0$	Modulo e segno	Complemento a 1	Complemento a 2
000	+0	+0	+0
001	+1	+1	+1
010	+2	+2	+2
011	+3	+3	+3
100	-0	-3	-4
101	-1	-2	-3
110	-2	-1	-2
111	-3	-0	-1

La rappresentazione dei numeri

- Modulo e segno:
  - rappresentazione con *n* bit: il bit di segno è 1 per i numeri negativi
  - campo rappresentabile $-2^{n-1}-1 \leq N \leq +2^{n-1}-1$ (due rappresentazioni per lo 0)
  - è molto simile alla rappresentazione dei numeri decimali
- Complemento a 1
  - rappresentazione con *n* bit: i numeri negativi sono ottenuti invertendo bit a bit il corrispondente numero positivo
  - campo rappresentabile $-2^{n-1}-1 \leq N \leq +2^{n-1}-1$ (due rappresentazioni per lo 0)
  - è semplice
- Complemento a 2
  - rappresentazione con *n* bit: i numeri negativi sono ottenuti invertendo bit a bit il numero positivo corrispondente, quindi sommando il valore 1
  - campo rappresentabile $-2^{n-1} \leq N \leq +2^{n-1}-1$ (una rappresentazioni per lo 0)
  - consente di realizzare circuiti di addizione e sottrazione più semplici
  - è quella utilizzata nei dispositivi digitali per rappresentare numeri relativi

Addizione senza segno

- La somma di numeri positivi si esegue sommando coppie di bit parallele, partendo da destra.
- Si ha riporto quando si deve eseguire la somma 1+1.
- Le tabelle seguenti mostrano le regole per la somma.

```
     0          0          1          1
   + 0        + 1        + 0        + 1
  -----      -----      -----      -----
     0          1          1        1 0
```

Riporto in uscita

- Utilizzando queste regole in modo diretto è possibile
  - Realizzare sommatori modulari
  - Composti da blocchi elementari identici
  - Circuiti aritmetici di questo tipo sono detti bit-slice

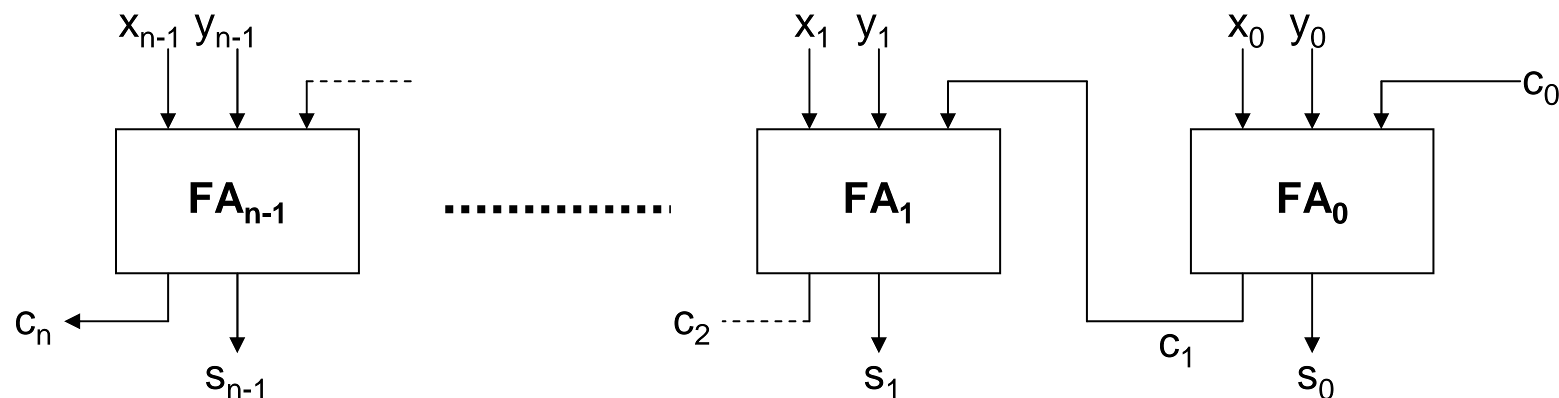
Addizione senza segno
bit-slice a propagazione di riporto

- Un sommatore *bit-slice ripple carry* è strutturato in modo che il modulo in posizione *i*-esima:
  - Riceve in ingresso i bit x_i e y_i degli operandi
  - Riceve in ingresso il riporto c_i del modulo precedente
  - Produce la somma $s_i = x_i'y_i'c_i + x_i'y_ic_i' + x_iy_i'c_i' + x_iy_ic_i$
  - Produce il riporto $c_{i+1} = x_iy_i + x_ic_i + y_ic_i$

- Il modulo in posizione 0 ha il bit di riporto $c_0=0$
- Il riporto c_0 può essere sfruttato per sommare il valore 1
  - Necessario per il calcolo del complemento a 2
- La somma di numero ad *n* bit richiede un tempo pari ad *n* volte circa quello richiesto da un modulo di somma

Addizione senza segno *ripple-carry*
Prestazioni: calcolo dei ritardi

- Il calcolo esatto del ritardo si effettua basandosi sulla seguente architettura
- Siano T_s e T_r i ritardi per il calcolo della somma e del riporto rispettivamente



- Prestazioni: il ritardo totale è dato dall'espressione:

$$T_{tot} = (n-1)T_r + T_s$$

- Il percorso critico è quindi quello del riporto

Addizione veloce (ad anticipazione di riporto)

Funzioni di generazione e di propagazione del riporto

Si basa sulle seguenti considerazioni

- Le espressioni di somma e riporto per lo stadio *i* sono:

  $s_i = x_i'y_i'c_i + x_i'y_ic_i' + x_iy_i'c_i' + x_iy_ic_i$

  $c_{i+1} = x_iy_i + x_ic_i + y_ic_i$

- L'espressione del riporto in uscita può essere riscritta come:

  $c_{i+1} = G_i + P_ic_i$ con $G_i = x_iy_i$ e $P_i = x_i + y_i$ (o anche $P_i = x_i \oplus y_i$)

- Le funzioni G_i e P_i
  - Sono dette funzioni di generazione e propagazione
    - G_i: se $x_i=y_i=1$, allora il riporto in uscita deve essere generato
    - P_i: se x_i *o* $y_i=1$ *e* $c_i=1$, allora il riporto in ingresso deve essere propagato in uscita
  - Possono essere calcolate in parallelo, per tutti gli stadi, rispetto alle rispettive somme.

Addizione veloce - *calcolo dei riporti in parallelo*

- L'espressione per il riporto $c_{i+1} = G_i + P_i c_i$ può essere calcolata in modo iterativo.
- Sostituendo $c_i = G_{i-1} + P_{i-1} c_{i-1}$ nell'espressione di c_{i+1} si ha:

$$c_{i+1} = G_i + P_i(G_{i-1} + P_{i-1} c_{i-1}) = G_i + P_i G_{i-1} + P_i P_{i-1} c_{i-1}$$

- Continuando con l'espansione fino a c_0 si ottiene:

$$\begin{aligned} c_{i+1} = G_i \quad & + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \\ & + \ldots + \\ & + P_i P_{i-1} \ldots P_1 G_0 + \\ & + P_i P_{i-1} \ldots P_1 P_0 c_0 \end{aligned}$$

- I riporti in uscita di ogni singolo stadio possono essere calcolati tutti in parallelo e con ritardo identico (realizzazione SOP) tramite:
  - le *i* funzioni di generazione G_i e le *i* funzioni di propagazione P_i
  - il riporto in ingresso allo stadio 0, c_0
- I sommatori che sfruttano il meccanismo della generazione dei riporti in anticipo sono detti Carry-Look-Ahead Adders o CLA Adders

Addizione veloce - *Esempio: prestazioni per un CLA a 4 bit*

$P_3\ G_3P_2\ G_2P_1G_1P_0G_0$ — 1 — 2 ← c_0 — 3 — c_4

$P_2\ G_2P_1\ G_1P_0G_0$ — 1 — 2 ← c_0 — 3 — c_3

$P_1\ G_1\ P_0\ G_0$ — 1 — 2 ← c_0 — 3 — c_2

$P_0\ G_0$ — 1 — 2 ← c_0 — 3 — c_1

CLA logic

$x_3\ y_3$ — 2 — 5 — s_3

$x_2\ y_2$ — 2 — 5 — s_2

$x_1\ y_1$ — 2 — 5 — s_1

$x_0\ y_0$, c_0 — 2 — 2 — s_0

3 — c_4

Adder

- $c_1=G_0+P_0c_0$
- $c_2= G_1+P_1G_0+P_1P_0c_0$
- $c_3= G_2+P_2G_1+P_2P_1G_0+P_2P_1P_0c_0$
- $c_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0c_0$

$G_i=x_iy_i$

$P_i=x_i+y_i$

Sommatori Carry Look-Ahead

Carry Look-Ahead Logic

x_{n-1} y_{n-1} x_0 y_0

Carry Look-Ahead Logic

c_n c_0

x_{n-1} y_{n-1} c_{n-1} x_1 y_1 c_1 x_0 y_0

Full Adder Full Adder Full Adder

s_{n-1} s_1 s_0

Addizione veloce: *calcolo delle prestazioni*

- Il ritardo totale per ottenere tutte le somme ed il riporto più a sinistra c_{i+1} è dato dalla somma di:
  - Un ritardo di porta per il calcolo delle funzioni di generazione e di propagazione ($G_i = x_i y_i$ e $P_i = x_i + y_i$)
  - Due ritardi di porta logica per calcolare il riporto *i*-esimo (SOP)
  - Due ritardi di porta logica per calcolare la somma *i*-esima (SOP)
- Totale:
  - 5 ritardi di porta logica
- Il ritardo è costante e indipendente dalla lunghezza degli operandi
- Problema:
  - Realizzazione circuitale per operandi lunghi (ad esempio 32 bit) fa uso di porte con un *fan-in* molto elevato: non praticabile!!
  - Soluzione: addizionatore veloce a blocchi

Addizione veloce a blocchi

- Il sommatore completo a n bit è ottenuto utilizzando un insieme di blocchi costituiti da CLA a m bit e della logica CLA

- Esempio: blocco è costituito da un sommatore CLA a 4 bit (ragionevole)

- Struttura del blocco
  - Il riporto finale di questo sommatore ha la seguente espressione:

$$c_4 = \boxed{G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0} + \boxed{P_3P_2P_1P_0\,c_0}$$

  - che può essere riscritta come

$$c_{uscita} = G + Pc_0$$

  - con il tempo di ritardo per il calcolo di P e G:
    - P = attraversamento di 2 porte logiche (1 per calcolare P_3, P_2, P_1 e P_0, 1 per calcolare il prodotto)
    - G = attraversamento di 3 porte logiche (calcolo di P_i e G_i, calcolo dei prodotti, calcolo della somma)

Addizione veloce - *blocco CLA a 4 bit*

Esempio - *sommatore a 16 bit con CLA a 4 bit*

- Prestazioni: in questo caso ordine di n/2
- Che cosa succede se devo sommare due numeri da 32 o da 64 bit?
  - Le **prestazioni di un CLA adder a *n* bit** costituito da blocchi da *m* bit sono espresse come $log_m\ n$, a meno del fattore costante dato dal ritardo di un CLA a *m* bit.

Esempio - *sommatore a 8 bit con CLA a 2 bit*

$G = G_{i+1}+P_{i+1}G_i$

$P = P_{i+1}P_i$

$c_{i+1} = G+Pc_i$

Addizione e sottrazione per valori rappresentati in complemento a 2

- Regole per la somma e sottrazione di due numeri in complemento a 2 su n bit
  - Per calcolare x+y
    - Fornire in ingresso ad un sommatore le codifiche binarie
    - Ignorare il bit di riporto in uscita
    - Il risultato è in complemento a due
  - Per calcolare x-y
    - Ricavare la rappresentazione di y in complemento a due
    - Sommare i valori così ottenuti come nella regola precedente
    - Il risultato è in complemento a due
- I risultati sono corretti se e solo se, disponendo di un sommatore ad *n* bit, il risultato sta nell'intervallo:

$$-2^{n-1} \leq x \pm y \leq 2^{n-1}-1$$

- In caso contrario si verifica overflow aritmetico

Addizione e sottrazione per valori rappresentati in complemento a 2

- Condizioni di overflow e di underflow per somme e sottrazioni in complemento a 2 su *n* bit

A+B			
A	B	Segno somma	Ov/Un
> 0	> 0	0	Si-Ov
> 0	< 0		no
< 0	> 0		no
< 0	< 0	1	Si-Un

A-B=A+(-B)				
A	B	-B = B_{CPL2}	Segno somma	Ov/Un
> 0	> 0	< 0		no
> 0	< 0	>0	0	Si-Ov
< 0	> 0	< 0	1	Si-Un
< 0	< 0	> 0		no

- overflow per somma = 0 0 1 (segno addendi e segno somma)
- underflow per somma = 1 1 0
- overflow per sottrazione = 0 1 1
- underflow per sottrazione = 1 0 0

Moltiplicazione interi senza segno

- La moltiplicazione di numeri senza segno si esegue con lo stesso metodo usato per la moltiplicazione decimale
- Il prodotto di due numeri binari di n e k bit è un numero binario di $n+k$ bit
- Ad esempio:

```
        1101  ×    →  Moltiplicando M = 13
        1011  =    →  Moltiplicatore Q = 11
    ----------
    00001101
    0001101
    000000
    01101
    ----------
    10001111       →  Prodotto P = 143
```

Moltiplicazione interi senza segno

- La moltiplicazione si effettua quindi
  - Sommando più volte il moltiplicando con opportuni allineamenti
- Ogni termine è il prodotto tra
  - Il moltiplicando *M*
  - Un bit q_i del moltiplicatore *Q*
- I prodotti parziali sono calcolabili in modo semplice:
- Se qi=1

  $PP_i = M \times q_i = M \times 1 = M$ cioè $(m_0, m_1, \ldots, m_n)$
- Se qi=0

  $PP_i = M \times q_i = M \times 0 = 0$ cioè $(0_0, 0_1, \ldots, 0_n)$

Moltiplicazione relativi in complemento a 2

- Moltiplicazione relativi in complemento a 2 di n bit
  - Si utilizza un metodo basato su quello usato per i numeri positivi appena visto (risultato su 2n-1 bit)
- Moltiplicatore *Q* positivo (moltiplicando positivo o negativo)
  - I prodotti parziali sono numeri con segno la cui rappresentazione va estesa fino alle dimensioni del risultato finale per poter essere sommati (estensione segno moltiplicando positivo 0, negativo 1)
- Moltiplicatore *Q* negativo (moltiplicando positivo o negativo)
  - Si calcola il complemento a due di *M* e di *Q* e *si* utilizza il metodo descritto sopra

Moltiplicatore	Moltiplicando	Prodotto
$Q > 0$	$M > 0$	$P = Q \times M$
$Q > 0$	$M < 0$	$P = Q \times M$
$Q < 0$	$M > 0$	$P = Q_{c2} \times M_{c2}$
$Q < 0$	$M < 0$	$P = Q_{c2} \times M_{c2}$

Esempio

- Si calcoli il prodotto -13 x 11
- Moltiplicando negativo

```
        10011  ×    →  Moltiplicando M = -13
        01011  =    →  Moltiplicatore Q = +11
  ---------------
    111110011
    11110011
    0000000
    110011
    00000
  ---------------
    101110001       →  Prodotto P = -143
```

Algoritmo di Booth

- Adatto per operandi con segno qualsiasi
- Se il moltiplicatore contiene sequenze di 1, l'algoritmo di Booth è più efficiente del metodo visto in precedenza (cioè devono essere generati molti meno prodotti parziali)
- Si consideri ad esempio la moltiplicazione per Q=30:

$$M \times 30 = M \times (32 - 2) = M \times 32 - M \times 2$$

- In rappresentazione binaria:

$$M \times 0011110 = M \times 0100000 - M \times 0000010$$
$$= M \times 0100000 + M_{C2} \times 0000010$$

- I moltiplicatori così ottenuti
  - Sono potenze del due
  - Sono sequenza di bit con un solo uno

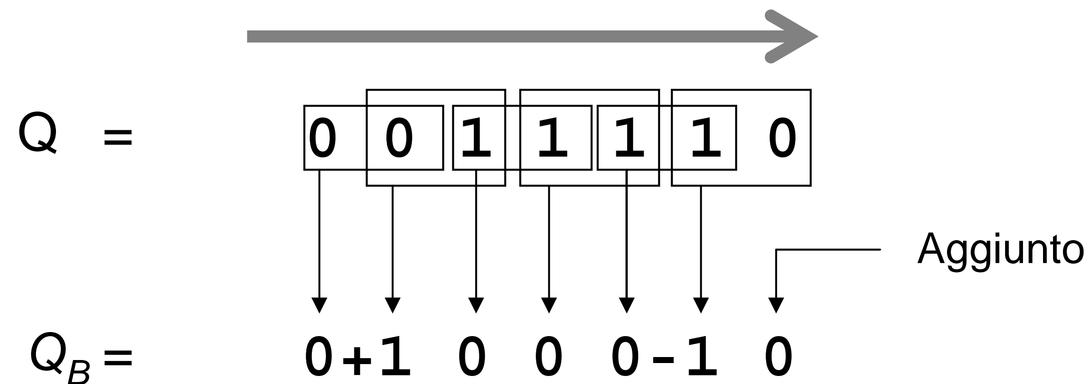
Algoritmo di Booth: codifica del moltiplicatore

- L'algoritmo si basa sulla scomposizione appena vista
- Tale scomposizione è rappresentata come una codifica del moltiplicatore basata sulle seguenti regole

- Si consideri un moltiplicatore *Q* di lunghezza *n*
  - Si scorre il moltiplicatore da sinistra verso destra
  - Il moltiplicatore codificato Q_B si ottiene:
    - Scrivendo il simbolo +1 quando si passa da 0 ad 1
    - Scrivendo il simbolo -1 quando si passa da 1 a 0
    - Scrivendo il simbolo 0 quando due bit successivi sono uguali
    - Se *Q* termina con 0 aggiungo 0 a Q_B altrimenti aggiungo -1

Algoritmo di Booth: esempio di codifica

- Ad esempio $Q = 30$ è codificato come $Q_B = 0{+}1000{-}10$



- Utilizzando tale codifica, i prodotti parziali saranno:
  - *0* con estensione del segno, quando $q_{B,i} = 0$
  - M_{C2} con estensione del segno, quando $q_{B,i} = -1$
  - *M* con estensione del segno, quando $q_{B,i} = +1$

Algoritmo di Booth: codifica del moltiplicatore

- Le regole esposte per l'algoritmo di Booth possono essere riassunte nella tabella seguente:

Moltiplicatore		**Codifica**	PP_i
q_i	q_{i-1}		
0	0	0	$0\times M = 0$
0	1	+1	$+1\times M = M$
1	0	-1	$-1\times M = M_{c2}$
1	1	0	$0\times \mathrm{M} = 0$

- E inoltre, se:
  - $q_0 = 0$, la codifica del bit aggiunto è 0 e quindi il prodotto parziale è 0
  - $q_0 = 1$, la codifica del bit aggiunto è -1 e quindi il prodotto parziale è M_{C2}

Esempio

- Moltiplicare 13 x -9, usando l'algoritmo di Booth su 5 bit
- I valori binari da usare sono:
  - *13 = 01101* $\qquad$ 13_{C2} = *10011*
  - *-9 = 10111* $\qquad$ -9_B = *-1+100-1*
- Il prodotto si esegue quindi nel modo seguente:

					0	1	1	0	1	×	→ Moltiplicando $M = 13$
					-1	+1	0	0	-1	=	→ Moltiplicatore $Q = -9$
M_{C2}	1	1	1	1	1	0	0	1	1		
0	0	0	0	0	0	0	0	0			
0	0	0	0	0	0	0	0				
M	0	0	1	1	0	1					
M_{C2}	1	0	0	1	1						
	1	1	0	0	0	1	0	1	1		→ Prodotto $P = -117$

Numeri in virgola fissa

- Fino a questo punto abbiamo assunto che
  - Un vettore di bit rappresentasse sempre un numero intero
  - Eventualmente con segno
- Tutte le considerazioni fatte fino ad ora e tutti i metodi esposti continuano a valere se si attribuisce ai vettori di bit il significato di numeri in *virgola fissa*
- Un sistema di numerazione in virgola fissa è quello in cui:
  - La posizione della virgola decimale è implicita
  - La posizione della virgola decimale uguale in tutti i numeri
- La posizione della virgola equivale alla interpretazione del valore intero moltiplicato per un fattore di scala

Numeri in virgola fissa: fattore di scala

- Si consideri ad esempio il vettore di $k+n$ bit (k bit per rappresentare la parte intera e n bit per rappresntare la parte frazionaria):

$$B = b_{k-1} \ldots b_0,b_{-1} \ldots B_{-n}$$

- Il suo valore è dato da

$$V(B) = b_{k-1}x2^{k-1} + \ldots + b_0x2^0 + b_{-1}x2^{-1} + \ldots + b_{-n}x2^{-n}$$

- Il fattore di scala che consente di passare dalla rappresentazione intera a quella a virgola fissa è pari a

$$S_n = 2^{-n} = 1 / 2^n$$

- Detti V_I il valore intero e V_{VF} il valore in virgola fissa di B:

$$V_{VF}(B) = V_I(B) \text{ x } S_n = V_I(B) \text{ x } 2^{-n}$$

Esempio

- Si consideri il vettore binario:

  $B = 010.10110$

- Il suo valore in virgola fissa è:

  $V_{VF}(B) = 2^{1} + 2^{-1} + 2^{-3} + 2^{-4} = 2 + {}^{1}/_{2} + {}^{1}/_{8} + {}^{1}/_{16}$

  $= {}^{43}/_{16} = 2.6875$

- Il fattore di scala da utilizzare per la conversione è:

  $S_5 = 2^{-5} = {}^{1}/_{32} = 0.03125$

- Il valore di *B*, considerandolo intero è:

  $V_I(B) = 2^6 + 2^4 + 2^2 + 2^1 = 64 + 16 + 4 + 2 = 86$

- Da cui, moltiplicando per il fattore di scala, si ha:

  $V_{VF}(B) = V_I(B) \times S_5 = 83 \times 0.03125 = 2.6875$

Virgola fissa vs. virgola mobile

Intervallo di variazione di un numero binario di 32 bit

- Codifica intera

  $0 \leq |V_I(B)| \leq +2^{31} \approx 2.15 \times 10^9$

- Codifica a virgola fissa

  $+4.65 \times 10^{-10} \approx +2^{-31} \leq |V_{VF}(B)| \leq +1$

- A pari numero di bit disponibili
  - con la rappresentazione intera o in virgola fissa, i valori rappresentati sono distribuiti uniformemente nel campo di rappresentabilità
  - con la rappresentazione in virgola mobile, i valori rappresentati sono distribuiti non uniformemente nel campo di rappresentabilità
    - sono "più fitti" vicino allo 0 e "più radi" per valori assoluti grandi
- Nella rappresentazione in virgola mobile (floating point) la posizione della virgola è mobile ed è indicata dal valore di un fattore moltiplicativo

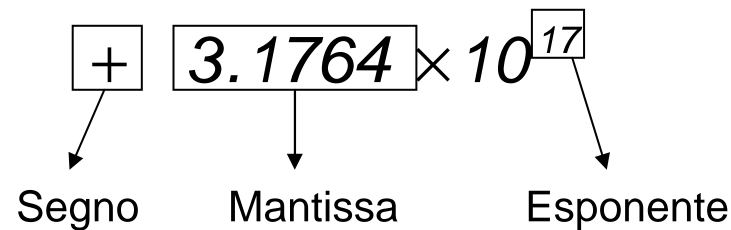
Errore di quantizzazione: virgola fissa vs. virgola mobile

- Virgola fissa (con n bit per la parte frazionaria)
- $E_{Ass} = Val_{Vero} - Val_{Rappr}$ = costante con $(-1/2)2^{-n} < E_{Ass} < (+1/2)2^{-n}$
- $E_{Rel} = E_{Ass} / Val_{Vero}$ (e cioè $E_{Rel}\, Val_{Vero}$= costante)
- tanto più piccolo è il valore vero da rappresentare tanto maggiore è l'errore relativo che si commette nel rappresentarlo
- tanto più grande è il valore vero da rappresentare tanto minore è l'errore relativo che si commette nel rappresentarlo

- Virgola mobile
- E_{Rel} = costante (= $2^{-\#\text{bit della M}}$)
- E_{Ass} = aumenta all'aumentare del valore valore vero da rappresentare

Numeri in virgola mobile

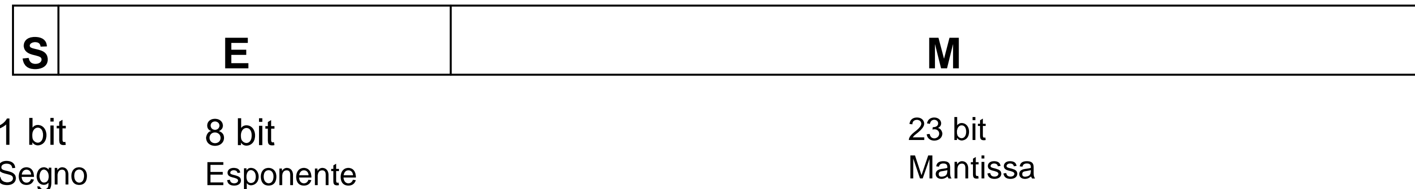
- Codifica in virgola mobile per i numeri in base 10
- Un numero in virgola mobile è composto da diverse parti:
- Si dice normalizzato un numero in cui $1 \leq M < 10$



- Facilmente estendibile al sistema di numerazione binario
- In un numero binario in virgola mobile e normalizzato
  - La prima cifra della mantissa è sempre 1 ($1 \leq M < 2$)
  - Tale cifra non viene rappresentata esplicitamente

Numeri in virgola mobile

- IEEE standard: Numeri floating-point in singola precisione



- L'esponente utilizza la codifica in *eccesso 127*, e cioè il valore effettivo dell'esponente è pari a (E-127)
  - *E = 0* e *M = 0* Rappresenta lo zero (pos/neg)
  - *E = 255* e *M = 0* Rappresenta infinito (pos/neg)
  - *E = 255* e *M !=0* *NotANumber*
  - 0<E<255 $(-1)^s \times 2^{(E-127)} \times (1,M)$
    (127≤E≤254 esp.positivi, 126≤E≤1esp.negativi)
  - *E = 0* e *M !=0* $(-1)^s \times 2^{-126} \times (0,M)$ non normalizzati

- Standard IEEE 32 bit: intervallo rappresentato $-1.M \times 10^{-38} \leq x \leq +1.M \times 10^{38}$
- La precisione consentita è di circa 7 cifre decimali

Operazioni in virgola mobile

- Le operazioni che si possono compiere su numeri in virgola mobile sono:
  - Somma
  - Sottrazione
  - Moltiplicazione
  - Divisione
  - Elevamento a potenza
  - Estrazione di radice

- Inoltre sono definite le operazioni di:
  - Calcolo del resto della divisione intera
  - Normalizzazione
  - Troncamento

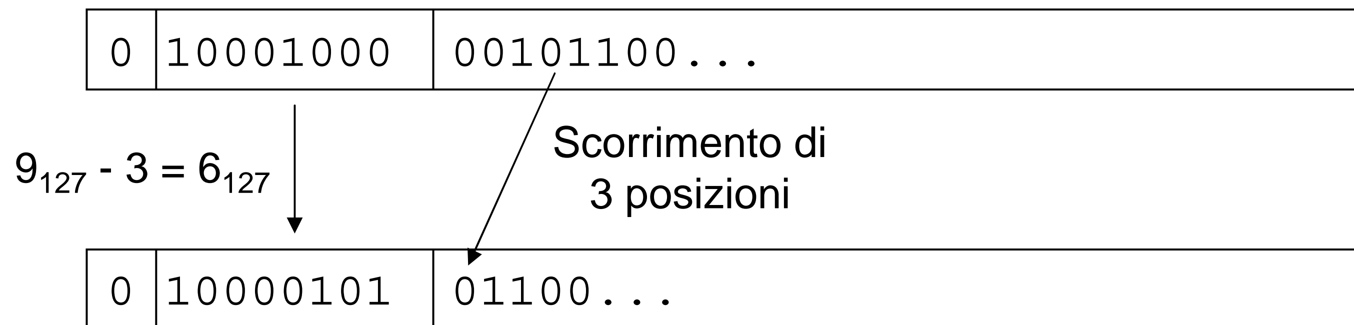
Operazioni in virgola mobile

- L'esecuzione di una operazione in virgola mobile può provocare una *eccezione*
- Una *eccezione* è il risultato di una operazione anomala, quale, ad esempio:
  - Divisione per zero
  - Estrazione della radice quadrata di un numero negativo
- Le eccezioni che vengono generate dalle unità aritmetiche in virgola mobile sono:
  - Operazione non valida
  - Divisione per zero
  - Overflow
  - Underflow

Operazioni in virgola mobile: *normalizzazione*

- Tutte le operazioni descritte nel seguito operano su numeri normalizzati (1 implicito prima della virgola)
- Se l'1 implicito manca, la normalizzazione di un numero con mantissa *M* ed esponente *n*, si esegue come segue:
  - Si fa scorrere verso sinistra la mantissa *M* fino al primo uno, compreso; sia *k* il numero di posizioni di tale scorrimento
  - Si sottrae *k* all'esponente *n*

- Ad esempio:



Operazioni in virgola mobile:
somma e sottrazione

- La somma o sottrazione tra numeri in virgola mobile viene eseguita secondo i seguenti passi:
  - Si sceglie il numero con esponente minore
  - Si fa scorrere la sua mantissa a destra un numero di bit pari alla differenza dei due esponenti
  - Si assegna all'esponente del risultato il maggiore tra gli esponenti degli operandi
  - Si esegue l'operazione di somma (algebrica) tra le mantisse per determinare il valore ed il segno del risultato
  - Si normalizza il risultato così ottenuto
    - Non sempre quest'ultima operazione è necessaria
    - Attenzione!!! Il riporto si può propagare anche dopo la posizione della virgola

Operazioni in virgola mobile : *moltiplicazione*

- La moltiplicazione tra numeri in virgola mobile viene eseguita secondo i seguenti passi:
  - Si sommano gli esponenti e si sottrae 127
  - Si calcola il risultato della moltiplicazione delle mantisse
  - Si determina il segno del risultato
  - Si normalizza il risultato così ottenuto
    - Non sempre quest'ultima operazione è necessaria

- La sottrazione di 127 dalla somma degli esponenti è necessaria in quanto sono rappresentati in eccesso 127

$$E_{a,127} = E_a + 127$$

$$E_{b,127} = E_b + 127$$

$$E_{axb,127} = E_{axb} + 127 = (E_a + 127) + (E_b + 127) - 127$$

Operazioni in virgola mobile : *divisione*

- La divisione tra numeri in virgola mobile viene eseguita secondo i seguenti passi:
  - Si sottraggono gli esponenti e si somma 127
  - Si calcola il risultato della divisione delle mantisse
  - Si determina il segno del risultato
  - Si normalizza il risultato così ottenuto
    - Non sempre quest'ultima operazione è necessaria

- La somma di 127 alla differenza degli esponenti è necessaria in quanto sono rappresentati in eccesso 127

$$E_{a,127} = E_a + 127$$
$$E_{b,127} = E_b + 127$$
$$E_{a/b,127} = E_{a/b} + 127 = (E_a + 127) - (E_b + 127) + 127$$

Operazioni in virgola mobile: *troncamento*

- Spesso accade di rappresentare i risultati intermedi di una operazione con una precisione maggiore di quella degli operandi e del risultato
- Al termine dell'operazione è necessario effettuare una operazione di *troncamento*
- Il troncamento serve a rimuovere un certo numero di bit per ottenere una rappresentazione approssimata del risultato
- Si consideri il valore numerico rappresentato dal vettore:

$$B = 0.b_{-1} \ldots b_{-(k-1)} b_{-k} b_{-(k+1)} \ldots b_{-n}$$

- Si voglia effettuare troncamento al bit k-esimo

Operazioni in virgola mobile: *troncamento*

- Chopping
  - Consiste nell'ignorare i bit dal *k*-esimo all'*n*-esimo
  - Questo metodo è *polarizzato* o *biased*
  - L'errore è sempre positivo e varia nell'intervallo:

    $0 < \varepsilon < +(2^{-k+1} - 2^{-n})$

- Rounding
  - Se il bit k-esimo vale 0, lasciare invariato il bit in posizione (k-1) e ignorare i bit dal *k*-esimo all'*n*-esimo
  - Se il bit k-esimo vale 1, sommare 1 in posizione (k-1) e ignorare i bit dal *k*-esimo all'*n*-esimo
  - Questo metodo è *simmetrico* o *unbiased*
  - L'errore è centrato sullo zero e vale:

    $-(2^{-k+1} - 2^{-n}) < \varepsilon < +(2^{-k+1} - 2^{-n})$