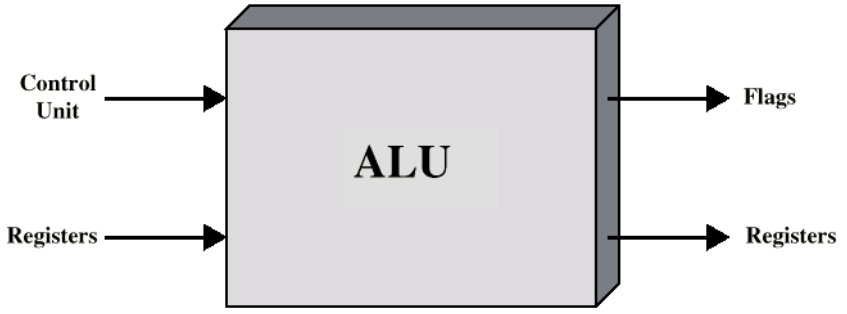
Aritmetica del calcolatore

Capitolo 9

Unità aritmetica e logica

- Esegue le operazioni aritmetiche e logiche
- Ogni altra componente nel calcolatore serve questa unità
- Gestisce gli interi
- Può gestire anche i numeri reali

Input e output della ALU



Rappresentazione degli interi

- Possiamo solo usare 0 e 1 per rappresentare tutto
- I numeri positivi sono scritti in binario come sappiamo
  - e.g. 41=00101001
- Non c'è bisogno del segno

Rappresentazione in modulo e segno

- Segno: bit più a sinistra
  - 0 significa positivo
  - 1 significa negativo
- Esempio:
  - +18 = 00010010
  - -18 = 10010010
- Problemi
  - Per eseguire operazioni aritmetiche bisogna considerare sia i moduli che i segni
  - Due rappresentazioni per lo 0: +0 and -0

Rappresentazione in complemento a due

- Segno nel bit più a sinistra
- Per n bit: possiamo rappresentare tutti i numeri da -2^{n-1} a $+2^{n-1} - 1$
- Per i numeri positivi, come per modulo e segno
  - n zeri rappresentano lo 0, poi 1, 2, … in binario per rappresentare 1, 2, … positivi
- Per i numeri negativi, da n uni per il -1, andando indietro

Rappresentazione in complemento a due

- +3 = 00000011
- +2 = 00000010
- +1 = 00000001
- +0 = 00000000
- -1 = 11111111
- -2 = 11111110
- -3 = 11111101

Complemento a due su 3 e 4 bit

a. Using patterns of length three

Bit pattern	Value represented
011	3
010	2
001	1
000	0
111	−1
110	−2
101	−3
100	−4

b. Using patterns of length four

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	−1
1110	−2
1101	−3
1100	−4
1011	−5
1010	−6
1001	−7
1000	−8

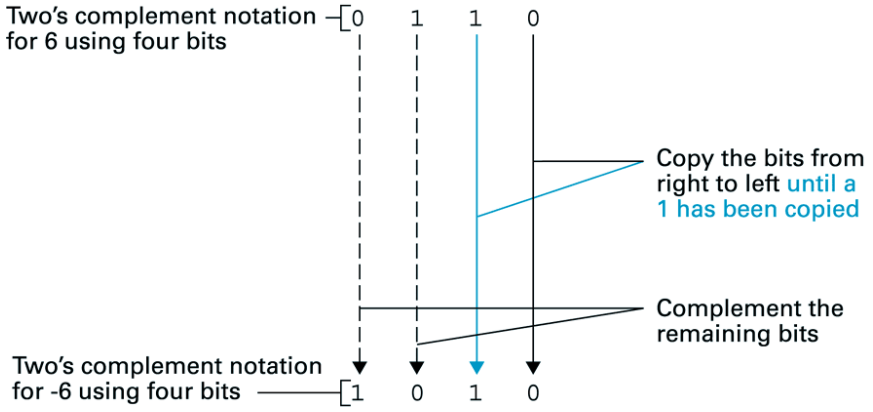
Complemento a due: numeri negativi

- Confrontiamo le rappresentazioni di k e –k
  - da destra a sinistra, uguali fino al primo 1 incluso
  - poi una il complemento dell'altra
- Esempio (su 4 bit): 2=0010, -2=1110

Complemento a due: decodifica

- Se bit di segno =0 ➔ positivo, altrimenti negativo
- Se positivo, basta leggere gli altri bit
- Se negativo, scrivere gli stessi bit da destra a sinistra fino al primo 1, poi complementare, e poi leggere
- Es.: 1010 è negativo, rappresenta 0110 (6), quindi -6

Da k a -k



Complemento a due: altro metodo

- Data la rappresentazione di k (positivo), -k si può anche ottenere così:
  - Complemento bit a bit della rappresentazione di k
  - Somma di 1 al risultato
- Esempio:
  - 2=0010
  - Complemento: 1101
  - 1101 +1 = 1110
  - -2=1110

Complemento a due: in generale

- Positivi: da 0 (n zeri) a 2^{n-1} -1 (uno zero seguito da n-1 uni)
- Negativi:
  - Bit di segno a 1
  - I restanti n-1 bit possono assumere 2^{n-1} configurazioni diverse, quindi da -1 a -2^{n-1}
- Se sequenza di bit $a_{n-1}\ a_{n-2} \ldots a_1\ a_0$,

  $$\text{numero} = -2^{n-1} \times a_{n-1} + \sum_{(i=0,\ldots,n-2)} 2^i \times a_i$$
- Numeri positivi: $a_{n-1} = 0$
- Numeri negativi: positivo – 2^{n-1}

Benefici

- Una sola rappresentazione dello zero
- Le operazioni aritmetiche sono facili
- L'opposto è facile da calcolare
  - 3 = 00000011
  - Complemento Booleano 11111100
  - Somma di 1 11111101

Numeri rappresentabili

- Complemento a 2 su 8 bit
  - Numero più grande: +127 = 01111111 = 2^7 -1
  - Numero più piccolo: -128 = 10000000 = -2^7
- Complemento a 2 su 16 bit
  - +32767 = 011111111 11111111 = 2^{15} - 1
  - -32768 = 100000000 00000000 = -2^{15}

Conversione tra diverse lunghezze

- Da una rappresentazione su n bit ad una rappresentazione dello stesso numero su m bit (m > n)
- Modulo e segno: facile
  - Bit di segno nel bit più a sinistra
  - m-n zeri aggiunti a sinistra
  - Esempio (da 4 a 8 bit): 1001 ➔ 10000001

Conversione tra diverse lunghezze

- Complemento a 2: stessa cosa del modulo e segno per numeri positivi
- Per numeri negativi: replicare il bit di segno dalla posizione attuale alla nuova
- Esempi:
  - +18 (8 bit) = 00010010
  - +18 (16 bit) = 00000000 00010010
  - -18 (8 bit) = 11101110
  - -18 (16 bit) = 11111111 11101110

Opposto su numeri in complemento a 2

- Due passi:
  - Complemento
  - Somma 1

Opposto: caso speciale 1

- 0 = 00000000
- Complemento: 11111111
- Somma 1: +1
- Risultato: 1 00000000
- L'uno più a sinistra è un overflow, ed è ignorato. Quindi - 0 = 0

Opposto: caso speciale 2

- -128 = 10000000
- Complemento: 01111111
- Somma 1: +1
- Risultato: 10000000
- Quindi, -(-128) = -128 !
- 2^n stringhe su n bit, un numero positivo in più di quelli negativi: -2^n si può rappresentare, ma $+2^n$ no ➔ -2^n non può essere complementato

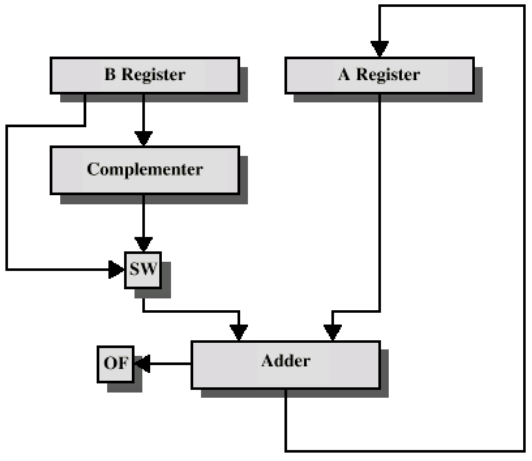
Somma e sottrazione

- Per la somma: normale somma binaria
  - Controllare il bit di segno per l'overflow
- Per la sottrazione: basta avere i circuiti per somma e complemento
  - Es. (4 bit): 7-5 = 7 +(-5) = 0111 + 1011 = 0010
  - 5 = 0101 ➔ -5 = 1011

Esempi di somme

Problem in base ten		Problem in two's complement		Answer in base ten
3 + 2	→	0011 + 0010 = 0101	→	5
−3 + −2	→	1101 + 1110 = 1011	→	−5
7 + −5	→	0111 + 1011 = 0010	→	2

Hardware per somma e sottrazione



OF = overflow bit
SW = Switch (select addition or subtraction)

Overflow

- Overflow: quando si sommano due numeri positivi tali che il risultato è maggiore del massimo numero positivo rappresentabile con i bit fissati (lo stesso per somma di due negativi)
- Se la somma dà overflow, il risultato non è corretto
- Come si riconosce? Basta guardare il bit di segno della risposta: se 0 (1) e i numeri sono entrambi negativi (positivi) ➔ overflow

Esempi di somme

- -4 (1100) + 4 (0100) = 10000 (0)
  - Riporto ma non overflow
- -4 (1100) - 1 (1111): 11011 (-5)
  - Riporto ma non overflow
- -7 (1001) -6 (1010) = 10011 (non è -13, ma 3)
  - Overflow
- + 7 (0111) + 7 (0111) = 1110 (non è 14, ma -2)
  - Overflow

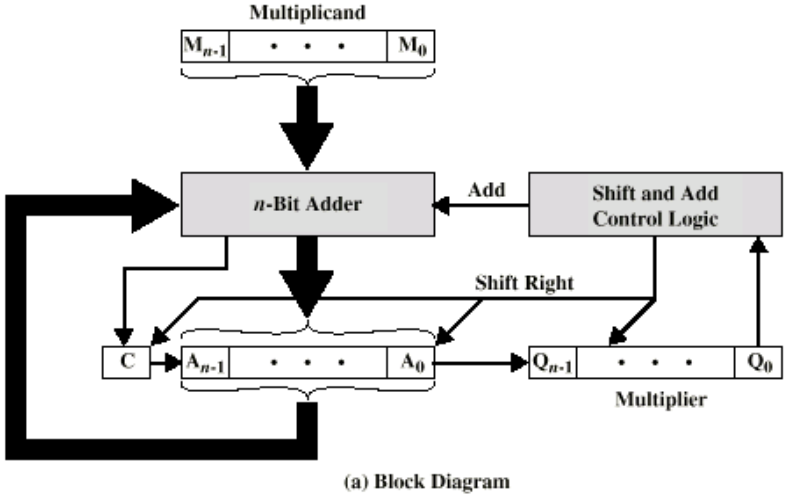
Moltiplicazione

- Più complessa
- Calcolare il prodotto parziale per ogni cifra
- Sommare i prodotti parziali

Esempio di moltiplicazione

- 1011 Moltiplicando (11 decimale)
- x 1101 Moltiplicatore (13 decimale)
- 1011 Prodotto parziale 1
- 0000 Prodotto parziale 2
- 1011 Prodotto parziale 3
- 1011 Prodotto parziale 4
- 10001111 Prodotto (143 decimale)
- Nota: da due numeri di n bit potremmo generare un numero di 2n bit

Moltiplicazione di interi senza segno



(a) Block Diagram

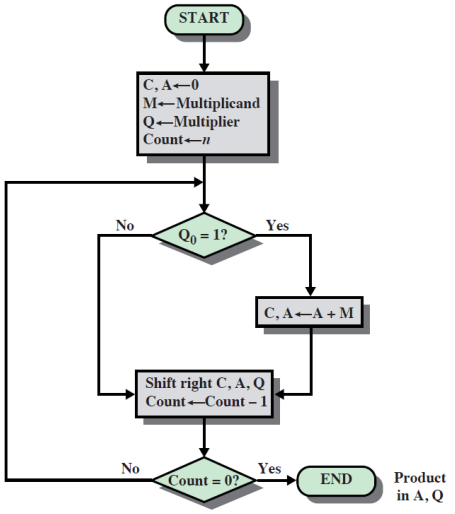
Implementazione

- Se $Q_0 = 0$, traslazione di C, A e Q
- Se $Q_0 = 1$, somma di A e M in A, overflow in C, poi traslazione di C, A, e Q
- Ripetere per ciascun bit di Q
- Prodotto (2n bit) in A e Q

Un esempio

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

Diagramma di flusso per la moltiplicazione senza segno



Moltiplicare numeri in complemento a 2

- Per la somma, i numeri in complemento a 2 possono essere considerati come numeri senza segno
- Esempio:
  - 1001 + 0011 = 1100
  - Interi senza segno: 9+3=12
  - Complemento a 2: -7+3=-4

Moltiplicare numeri in complemento a 2

- Per la moltiplicazione, questo non funziona!
- Esempio: 11 (1011) x 13 (1101)
  - Interi senza segno: 143 (10001111)
  - Se interpretiamo come complemento a 2: -5 (1011) x -3 (1101) dovrebbe essere 15, invece otteniamo 10001111 (-113)
- Non funziona se almeno uno dei due numeri è negativo

Moltiplicare numeri in complemento a 2

- Bisogna usare la rappresentazione in complemento a due per i prodotti parziali:

```
     1001  (9)                    1001  (−7)
   ×0011  (3)                   ×0011  (3)
00001001  1001 × 2^0        11111001  (−7) × 2^0 = (−7)
00010010  1001 × 2^1        11110010  (−7) × 2^1 = (−14)
00011011  (27)              11101011  (−21)
```

(a) Unsigned integers (b) Twos complement integers

- Problemi anche se moltiplicatore negativo
- Una possibile soluzione:
  1. convertire I fattori negativi in numeri positivi
  2. effettuare la moltiplicazione
  3. se necessario (-+ o +-), cambiare di segno il risultato
- Soluzione utilizzata: algoritmo di Booth (più veloce)

Divisione

- Più complessa della moltiplicazione
- Basata sugli stessi principi generali
- Utilizza traslazioni, somme e sottrazioni ripetute

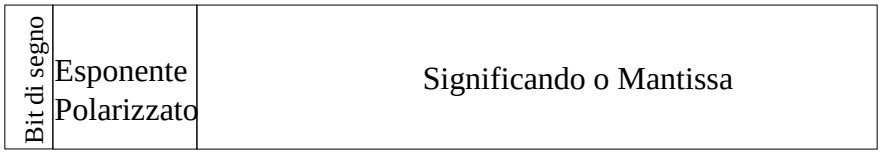
Numeri reali

- Numeri con frazioni
- Posso essere rappresentati anche in binario
  - Es.: $1001.1010 = 2^4 + 2^0 + 2^{-1} + 2^{-3} = 9.625$
- Quante cifre dopo la virgola?
- Se numero fisso, molto limitato
- Se mobile, dobbiamo saper specificare dove si trova la virgola

Notazione scientifica (decimale)

- 976.000.000.000.000 viene rappresentato come 9,76 x 10^{14}
- 0,0000000000000976 viene rappresentato come 9,76 x 10^{-14}
- Vantaggio: numeri molto grandi e molto piccoli con poche cifre
- Lo stesso per numeri binari: +/- S x $B^{+/-E}$
  - S = significando o mantissa (come 976)
  - Si assume la virgola dopo una cifra della mantissa
  - B = base

Floating Point



- Numero rappresentato:

  +/- 1.mantissa x $2^{esponente}$
- Esponente polarizzato: una valore fisso viene sottratto per ottenere il vero esponente

  k bit per esponente polarizzato ➔ 2^{k-1} -1

  e = ep – (2^{k-1} -1)

Es.: 8 bit ➔ valori tra 0 e 255 ➔ 2^7-1=127 ➔ esponente da -127 a +128

Normalizzazione

- I numeri in virgola mobile di solito sono normalizzati
- L'esponente è aggiustato in modo che il bit più significativo della mantissa sia 1
- Dato che è sempre 1 non c'è bisogno di specificarlo
- Numero: +/- 1.mantissa x $2^{esponente}$
- L'1 non viene rappresentato nei bit a disposizione ➔ se 23 bit per la mantissa, posso rappresentare numeri in [1,2)
- Se non normalizzato, aggiusto l'esponente
  - Es.: 0,1 x 2^0 = 1,0 x 2^{-1}

Esempi

(a) Format

$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.638125 \times 2^{20}$
$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.638125 \times 2^{20}$
$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.638125 \times 2^{-20}$
$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.638125 \times 2^{-20}$

(b) Examples

Numeri rappresentabili (32 bit)

- Complemento a due: da -2^{31} a $+ 2^{31} - 1$
- Virgola mobile (con 8 bit per esponente):

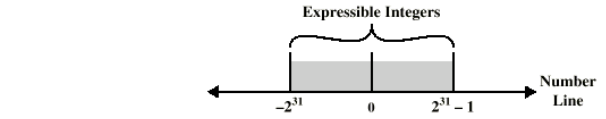
  Esponente: da -127 (tutti 0) a 128 (tutti 1)

  Mantissa: 1.0 (tutti 0) a $2-2^{-23}$ (tutti 1, cioè 1. 1...1, cioè $1 + 2^{-1} + 2^{-2} + ... + 2^{-23} = 2-2^{-23}$)

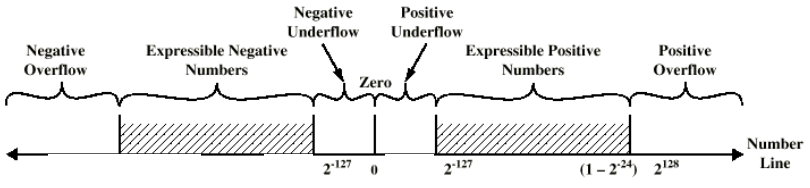
  Negativi: Da $-2^{128} \times (2-2^{-23})$ a -2^{-127}

  Positivi: da 2^{-127} a $2^{128} \times (2-2^{-23})$

Numeri rappresentabili (32 bit)



(a) Twos Complement Integers

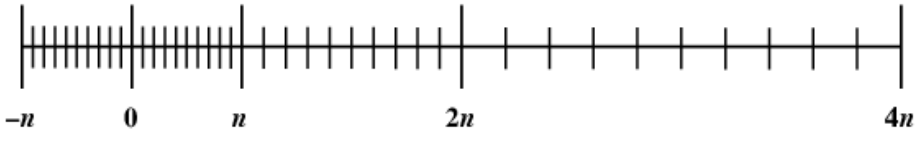


(b) Floating-Point Numbers

Numeri non rappresentabili (32 bit)

- Negativi minori di $-2^{128} \times (2-2^{-23})$ [overflow negativo]
- Negativi maggiori di -2^{-127} [underflow negativo]
- Positivi minori di 2^{-127} [underflow positivo]
- Positivi maggiori di $2^{128} \times (2-2^{-23})$ [overflow positivo]
- Non c'è una rappresentazione per lo 0
- I numeri positivi e negativi molto piccoli (valore assoluto minore di 2^{-127}) possono essere approssimati con lo 0
- Non rappresentiamo più numeri di 2^{32}, ma li abbiamo divisi in modo diverso tra positivi e negativi
- I numeri rappresentati non sono equidistanti tra loro: più densi vicino allo 0 (errori di arrotondamento)

Densità dei numeri in virgola mobile



Precisione e densità

- Nell'esempio, 8 bit per esponente e 23 bit per mantissa
- Se più bit per l'esponente (e meno per la mantissa), espandiamo l'intervallo rappresentabile, ma i numeri sono più distanti tra loro ➔ minore precisione
- La precisione aumenta solo aumentando il numero dei bit
- Di solito, precisione singola (32 bit) o doppia (64 bit)

Densità

- Numero (positivo) = $1,m \times 2^e$
- Fissato e, i numeri rappresentabili sono tra 2^e (mantissa tutta a 0) e $2^e \times (2-2^{-23})$
  - Quanti numeri? 2^{23}
- Consideriamo adesso e+1 ➔ i numeri rappresentabili sono tra $2x2^e$ e $2x2^e \times (2-2^{-23})$
  - Intervallo grande il doppio
  - Quanti numeri? Sempre 2^{23}

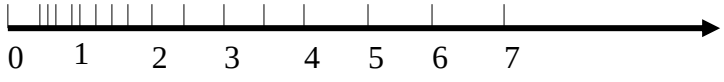
Esempio con 4 bit (+ segno)

fe (due bit per esponente), ab (due bit per mantissa)

Numero= $2^{fx2+e-1}x(1+ax2^{-1}+bx2^{-2})$

f	e	a	b		numero	
0	0	0	0		0.5	
0	0	0	1		0.625	
0	0	1	0		0.75	
0	0	1	1		0.875	
0	1	0	0		1	
0	1	0	1		1.25	
0	1	1	0		1.5	
0	1	1	1		1.75	
1	0	0	0		2	
1	0	0	1		2.5	
1	0	1	0		3	
1	0	0	1		3.5	
1	1	0	0		4	
1	1	0	1		5	
1	1	1	0		6	
1	1	1	1		7	

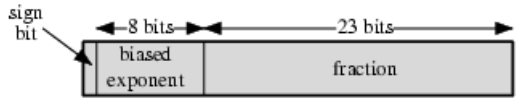
- Numeri positivi rappresentati con 2 bit esponente e 2 bit mantissa



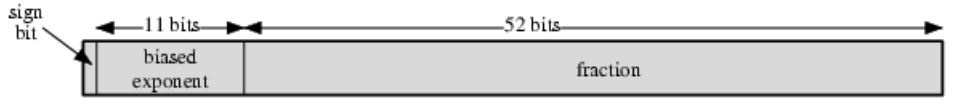
Standard IEEE 754

- Standard per numeri in virgola mobile
- Formato singolo a 32 bit e doppio a 64 bit
- Esponente con 8 e 11 bit
- 1 implicito a sinistra della virgola
- Formati estesi (più bit per mantissa ed esponente) per risultati intermedi
  - Più precisi ➔ minore possibilità di risultato finale con eccessivo arrotondamento

Formati IEEE 754



(a) Single format



(b) Double format

Numeri rappresentati (formato singolo)

- Alcune combinazioni (es.: valori estremi dell'esponente) sono interpretate in modo speciale
- Esponente polarizzato da 1 a 254 (cioè esponente da -126 a +127): numeri normalizzati non nulli in virgola mobile ➔ $+/- 2^{e-127} \times 1.f$
- Esponente 0, mantissa (frazione) 0: rappresenta 0 positivo e negativo
- Esponente con tutti 1, mantissa 0: infinito positivo e negativo
  - Overflow può essere errore o dare il valore infinito come risultato
- Esponente 0, mantissa non nulla: numero denormalizzato
  - Bit a sinistra della virgola: 0, vero esponente: -126
  - Positivo o negativo
  - numero: $2^{-126} \times 0.f$
- Esponente tutti 1, mantissa non nulla: errore (Not A Number)

Aritmetica in virgola mobile

- Allineare gli operandi aggiustando gli esponenti (per somma e sottrazione)
- Possibili eccezioni del risultato:
  - Overflow dell'esponente: esponente positivo che è più grande del massimo
  - Underflow dell'esponente: esponente negativo minore del minimo valore (numero troppo piccolo)
  - Underflow della mantissa: mantissa 0 (allineando, gli 1 sono usciti fuori)
  - Overflow della mantissa: riporto del bit più significativo

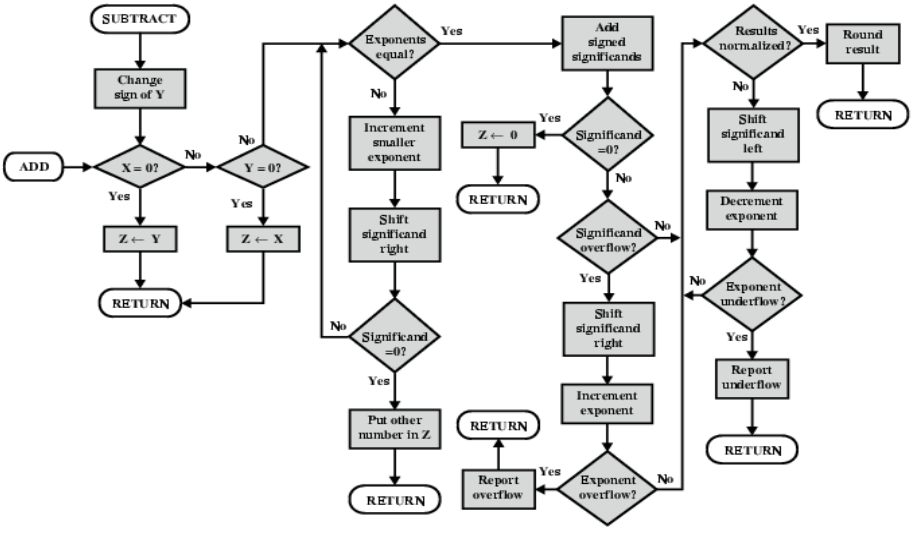
Somma e sottrazione

- Quattro fasi:
  - Controllo dello zero
    - Se uno dei due è 0, il risultato è l'altro numero
  - Allineamento delle mantisse
    - Rendere uguali gli esponenti
  - Somma o sottrazione delle mantisse
  - Normalizzazione del risultato
    - Traslare a sinistra finché la cifra più significativa è diversa da 0

Allineamento delle mantisse

- Esempio (in base 10):
- $(123 \times 10^{0}) + (456 \times 10^{-2})$
- 123 x 4,56 ➔ Non possiamo semplicemente sommare 123 a 456: il 4 deve essere allineato sotto il 3
- Nuova rappresentazione: $(123 \times 10^{0}) + (4{,}56 \times 10^{0})$
- Adesso posso sommare le mantisse (123 + 4,56 = 127,56)
- Risultato: $127{,}56 \times 10^{0}$

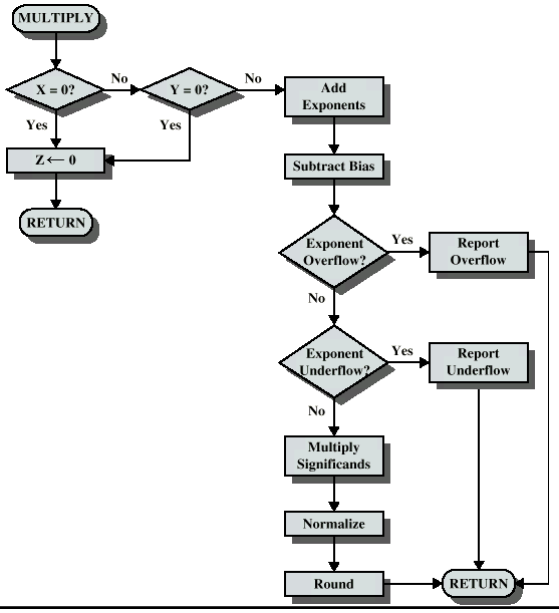
Somma e sottrazione in virgola mobile



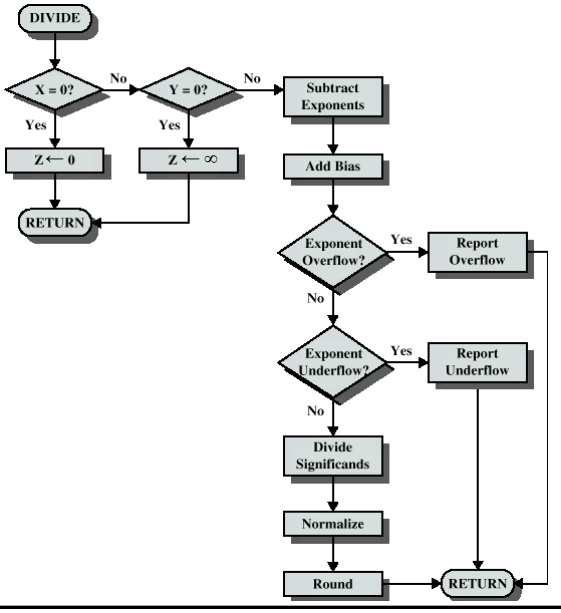
Moltiplicazione e divisione

- Controllo dello zero
- Somma degli esponenti
- Sottrazione polarizzazione
- Moltiplicazione/divisione operandi
- Normalizzazione
- Arrotondamento

Moltiplicazione in virgola mobile



Divisione in virgola mobile



Precisione del risultato: bit di guardia

- Di solito operandi nei registri della ALU, che hanno più bit di quelli necessari per la mantissa +1 ➔ i bit più a destra sono messi a 0 e permettono di non perdere bit se i numeri vengono shiftati a destra
- Es.: X-Y, con $Y=1,11...11 \times 2^0$ e $X=1,00...00 \times 2^1$
- Y va shiftato a destra di un bit, cioè diventa
  $0,111...11 \times 2^1$ ➔ un 1 viene perso senza i bit di guardia
- Risultato:
  - Senza bit di guardia:
    $(1,0...0 - 0,1...1) \times 2 = 0,0...01 \times 2 = 1,0...0 \times 2^{-22}$
  - Con bit di guardia:
    $(1,0...0 - 0,1...11) \times 2 = 0,0...001 \times 2 = 1,0...0 \times 2^{-23}$

Precisione del risultato: arrotondamento

- Se il risultato è in un registro più lungo, quando lo si riporta nel formato in virgola mobile, bisogna arrotondarlo
- Quattro approcci:
  - Arrotondamento al più vicino (default)
    - Bit aggiuntivi che iniziano con 1 ➔ sommo 1
      - Bit aggiuntivi 10...0 ➔ sommo 1 se l'ultimo bit è 1, altrimenti 0
    - Bit aggiuntivi che iniziano con 0 → elimino
  - Arrotondamento (per eccesso) a $+\infty$ e arrotondamento (per difetto) a $-\infty$
    - Usati nell'aritmetica degli intervalli
  - Arrotondamento a 0 (cioè troncamento dei bit in più)