CALCOLATORI

Aritmetica dei Calcolatori

Giovanni Iacca
giovanni.iacca@unitn.it

Lezione basata su materiale preparato dal Prof. Luca Abeni

UNIVERSITÀ DEGLI STUDI DI TRENTO
Dipartimento di Ingegneria
e Scienza dell'Informazione

Ci sono solo 10 tipi di persone: quelle che conoscono la numerazione binaria e quelle che non la capiscono.

[classica battuta “nerdissima”]

Informazione nei computer

- Un computer è un insieme di circuiti elettronici...
- ... In ogni circuito, la corrente può passare o non passare
  - Passa corrente: 1
  - Non passa corrente: 0
- $\Rightarrow$ un computer memorizza (e manipola) solo sequenze di 0 e 1
  - Sequenze di 0 e 1 devono poter rappresentare tutti i tipi di informazione che un computer gestisce: valori numerici, caratteri o simboli, immagini, suoni, programmi, ...
- Una singola cifra (0 o 1) è detta bit; sequenze di 8 bit sono chiamate byte

Codifica delle informazioni - 1

- Per far sí che una sequenza di bit (cifre 0 o 1) possa essere interpretata come informazione utile, deve essere stabilita una codifica (rappresentazione digitale)
- Tutto deve essere *codificato* opportunamente per rappresentarlo in questo modo
  - Numeri (interi con e senza segno, razionali, reali, ...)
  - Caratteri / testo
  - Programmi (sequenze di istruzioni macchina)
  - Immagini e suoni
- Sí, ma come codifichiamo tutto ció?

Codifica delle informazioni - 2

- Codifica: funzione che associa ad un oggetto / simbolo una sequenza di bit
- Codifica su n bit: associa una sequenza di n bit ad ogni entità da codificare
  - Permette di codificare 2^n entità / simboli distinti
- Esempio: per codificare i semi delle carte bastano 2 bit
  - Picche $\rightarrow 00$
  - Quadri $\rightarrow 01$
  - Fiori $\rightarrow 10$
  - Cuori $\rightarrow 11$

Codifica dei numeri

- Rappresentiamo un numero come sequenza di 0 e 1...
- ... cominciamo con cose semplici: numeri naturali (interi positivi) $n \in \mathcal{N}$
- Per capire come fare, consideriamo i numeri “che conosciamo” (base 10)
  - Se scrivo 501, intendo $5 * 100 + 0 * 10 + 1(*10^0)$
  - Sistema posizionale: il valore di ogni cifra dipende dalla sua posizione
  - Sistema decimale (base 10): ho 10 cifre $(0 \ldots 9)$

Basi di numerazione

- In generale, un numero naturale è una sequenza di cifre
- B possibili cifre (da 0 a $B-1$)
  - B è detto *base di numerazione*
  - Siamo abituati a 10 cifre (sistema decimale, o in base 10), ma nulla vieta di usarne di più o di meno...
- Il valore di una sequenza di cifre $c_i c_{i-1} \dots c_0$ si calcola moltiplicando ogni cifra per una potenza della base B, che dipende dalla posizione della cifra:

$$c_i c_{i-1} c_{i-2} \dots c_0 = c_i * B^i + c_{i-1} * B^{i-1} + \dots + c_0 * B^0 = \sum_{k=0}^{i} c_k B^k$$

- Una sequenza di cifre da interpretarsi come numero in base B è spesso indicata con pedice $_B$ (501_{10}, 753_8, 110110_2, $1AF_{16}$...)

Esempi con basi di numerazione diverse da 10

- Base 2 (cifre: 0 e 1): sistema binario
  - Esempio: $1101_2 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 13_{10}$
- Base 8: sistema ottale
  - Esempio: $170_8 = 1 * 8^2 + 7 * 8^1 + 0 * 8^0 = 120_{10}$
- Base 16: sistema esadecimale
  - Esempio: $1AF_{16} = 1 * 16^2 + 10 * 16^1 + 15 * 16^0 = 431_{10}$
- Le cifre da 10 a 15 sono rappresentate dalle lettere da A ad F ($A_{16} = 10_{10}$, $B_{16} = 11_{10}$, ... $F_{16} = 15_{10}$)

Hai detto... binario?

- Sistema binario: 2 sole cifre (0 e 1)
  - Sembra fatto apposta per i computer!
- Un numero naturale è rappresentato in binario su k cifre binarie
  - Cifra binaria: binary digit — bit
  - Numero binario su k cifre: valori da 0 a $2^k - 1$
- Valori comuni per k: 8, 16, 32, 64
  - Le potenze di 2 rivestono un ruolo molto importante!
- Ricordate? Bit raggruppati in sequenze di 8, dette byte
  - Ogni byte può assumere 2^8 valori, da 0 ($= 00000000_2$) a 255 ($= 11111111_2$)

Altre basi di numerazioni utili

- La base 2 è la base “nativa” dei computer...
- ... ma i numeri in base 2 possono richiedere un gran numero di cifre!
  - Esempio: $1000_{10} = 1111101000_2$
- Spesso si usa la base 16 (numerazione esadecimale) per ridurre il numero di cifre
  - Ogni cifra esadecimale corrisponde a 4 cifre binarie
  - La conversione da binario a esadecimale (e viceversa) è molto semplice
  - Esempio: $(0011\ 1110\ 1000)_2 = 3E8_{16}$

Conversioni fra basi

- Base $2 \leftrightarrow 16$: come visto (si convertono 4 bit in una cifra esadecimale e viceversa)
- Base $B \rightarrow 10$: già visto anche questo (si moltiplica ogni cifra c_i per B^i, dipendentemente dalla sua posizione)
- Base $10 \rightarrow B$: si divide iterativamente il numero per B

1. Per convertire x_{10} in base B:
2. Divisione intera fra x e B
3. Il resto e' la cifra da inserire a sinistra nel numero convertito
4. Il quoziente viene assegnato ad x
5. Ritorna al punto 2

Esempio di conversione

$1000_{10} =?_2$

X	$X/2$	$X\%2$
1000	500	0
500	250	0
250	125	0
125	62	1
62	31	0
31	15	1
15	7	1
7	3	1
3	1	1
1	0	1

$1000_{10} = 1111101000_2$

$1000_{10} =?_{16}$

X	$X/16$	$X\%16$
1000	62	8
62	3	14 (E)
3	0	3

$1000_{10} = 3E8_{16}$

Somme e sottrazioni in base 2

- Un'introduzione, prima:

A	B	Carry	C	Carry
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

$$A + B + Carry = C + Carry$$

Somma di naturali

- Esempio

$$117_{10} + 97_{10} = 1110101_2 + 1100001_2 = ?$$

	1	1	1	0	1	0	1	+
	1	1	0	0	0	0	1	=
1	1	0	1	0	1	1	0	

$$11010110_2 = 214_{10}$$

Sottrazione di naturali

- Sottrazione di naturali: operazione di base con “prestito”.
  Come nella somma si va da destra a sinistra seguendo delle regole banali:

A	0	1	1	0
B	0	0	1	1
A-B	0	1	0	1

Nell’ultimo caso bisogna scorrere a sinistra finché non si trova un 1, invertendo il valore di tutte le cifre incontrate, 1 compreso.

$$10..0 \rightarrow 01..1$$

Sottrazione di naturali

- Esempio

$$214_{10} - 117_{10} = 11010110_2 - 1110101_2 = ?$$

1	1	0	1	0	1	1	0	-
	1	1	1	0	1	0	1	=
	1	1	0	0	0	0	1	

$$1100001_2 = 97_{10}$$

Moltiplicazione per potenze di 2

- Passiamo ora a moltiplicazione (e divisione) per potenze di due. Se dobbiamo moltiplicare un numero base 10 per 10 dobbiamo...

  .... aggiungere uno 0 a destra.

- Nel caso dei numeri base 2, aggiungere uno 0 a destra significa moltiplicare il numero per 2, mentre eliminando la prima cifra a destra significa dividere per 2.

  Questa operazione é chiamata shifting, e viene utilizzata spesso dai compilatori per ottimizzare la moltiplicazione.

Moltiplicazione di due naturali

- Moltiplicare due numeri base 2 naturali altro non é che l'applicazione di shifting multipli e poi della somma sui risultati degli shifting.
  Alla fine questa operazione non é diversa dalla moltiplicazione in base 10, solo che é piú semplice perché o bisogna sommare uno dei due moltiplicatori shiftati, o nulla.
- Esempio

$$13_{10} \times 17_{10} = 1101_2 \times 10001_2 = ?$$

				1	1	0	1	x
			1	0	0	0	1	=
				1	1	0	1	+
						-		+
					-			+
				-				+
1	1	0	1					
1	1	0	1	1	1	0	1	

$$11011101_2 = 221_{10}$$

Codifica dei numeri interi

- k bit $\rightarrow$ codificano 2^k simboli/valori/numeri...
  - Si usa la base 2 per codificare i numeri
  - Numeri naturali $n \in \mathcal{N}$: valori da 0 a $2^k - 1$
- Come codificare numeri interi $z \in \mathcal{Z}$?
  - Problema: numeri negativi $z < 0 \Rightarrow z \notin \mathcal{N}$
  - Si codificano sempre 2^k valori...
  - ... ma quali?
- Varie possibilità:
  - modulo e segno
  - complemento a 1
  - complemento a 2

Codifica con modulo e segno

- Idea semplice: si usano $k-1$ bit per rappresentare il valore assoluto (modulo) del numero...
- ... ed un bit per codificare il segno!
  - bit più significativo a 0: numero positivo
  - bit più significativo a 1: numero negativo
- Valori codificati: da $-2^{k-1}+1$ a $2^{k-1}-1$
  - Perché? Due diverse sequenze di bit per codificare lo 0? ($+0$ e -0?)
  - $1\overbrace{000\ldots0}^{k-1}$ vs $0\overbrace{000\ldots0}^{k-1}$

Codifica in complemento a 1

- Altra idea semplice: i numeri negativi si rappresentano facendo il complemento a 1 del valore assoluto
  - Numero positivo: rappresento il valore assoluto
  - Numero negativo: rappresento il complemento a 1 del valore assoluto
    Operazione di complemento a 1 di un numero binario x:
    - Metodo 1: cambio tutti i bit di x, da 0 a 1, e da 1 a 0
    - Metodo 2: calcolo: $(2^k - 1)_2 - x = \overbrace{1 \ldots 1}^{k} - x$
  - Anche in questo caso:
    - bit più significativo a 0: numero positivo
    - bit più significativo a 1: numero negativo
- Ancora, due rappresentazioni dello 0 ($+0$ e -0)? $\overbrace{111 \ldots 1}^{k}$ vs $\overbrace{000 \ldots 0}^{k}$
- Somme e sottrazioni in modulo e segno $\rightarrow$ non facile!
- Somme e sottrazioni in complemento a $1 \rightarrow$ vedi prossima slide.

Somma e sottrazione di numeri in complemento a 1

- Somma di numeri interi codificati in complemento a 1:
  1. Sommare le rappresentazioni dei due numeri
  2. Riporto sul bit più significativo $\rightarrow$ sommarlo al risultato
  3. Se i riporti delle due cifre più significative di quest'ultima somma sono uguali, il risultato è attendibile
  4. Altrimenti, il risultato non è rappresentabile su k bit
- Esempio: $6 + -3$ (codifica su 5 bit)
  - $6 = 00110$; $-3 = \overline{00011} = 11100$

$$\begin{array}{ccccccc} & (1) & & & & & \\ & & 0 & 0 & 1 & 1 & 0 \\ & & 1 & 1 & 1 & 0 & 0 \\ \hline & & 0 & 0 & 0 & 1 & 0 \end{array}$$

  - $00010 + 1 = 00011\ (= 3)$

Codifica in complemento a 2

Operazione di complemento a 2 di un numero binario x:

- Metodo 1:
  - Scorro i bit di x a partire dalla destra (bit meno significativo)
  - Fino al primo bit che vale 1 (compreso), lascio invariato
  - A partire dal bit successivo, faccio il complemento a 1
- Metodo 2: faccio il complemento a 1 di x, e poi sommo 1
- Metodo 3: calcolo $(2^k)_2 - x = 1\overbrace{0\ldots0}^{k} - x$
  (utile nelle somme, vedi prossima slide)

Esempio: complemento in base 2 di 87 $(= 1010111)$

- Metodo 1: 1 più a destra invariato, poi inverto : 0101001
- Metodo 2: complemento a 1, e sommo $+1$: $0101000 + 1 = 101001$
- Metodo 3: $(2^7)_2 - 1010111 = 10000000 - 1010111 = 101001$

Codifica in complemento a 2

- Numeri positivi: rappresento il valore assoluto
- Numeri negativi: rappresento il complemento a 2 del valore assoluto
  - Anche in questo caso:
    - bit più significativo a 0: numero positivo
    - bit più significativo a 1: numero negativo
  - Tuttavia, la codifica dello 0 è unica
  - Codifica i numeri da -2^{k-1} a $2^{k-1} - 1$
- Inoltre, anche la somma è piú semplice...
  - $x + (-y) = x + (2^k - y) = 2^k + x - y$...
  - ... su k bit, questo è equivalente a $x - y$
- Operazione inversa (dato un numero binario in complemento a 2, per ottenere il numero decimale corrispondente):
  - Se il primo bit a sinistra (bit piú significativo) é 0, converto semplicemente il binario in numero decimale
  - Se il primo bit a sinistra é 1, faccio il complemento a 2 e sommo $+1$, e converto in numero decimale

Esempi di codifica

decimale	modulo e segno	complemento a 1	complemento a 2
-4			100
-3	111	100	101
-2	110	101	110
-1	101	110	111
0	100 / 000	111 / 000	000
1	001	001	001
2	010	010	010
3	011	011	011

Nota: Usando il complemento a 2 su k bit, il minimo valore rappresentabile é sempre -2^{k-1}. Perché?
Es. complemento a 2 di -4: $\overline{100} + 1 = 011 + 1 = 100$

Somma e sottrazione di numeri in complemento a 2

- Se si usa il complemento a 2, somme e sottrazioni si applicano facilmente anche con numeri negativi...
  - Ma occhio agli overflow!
  - Overflow? Numeri su k bit, risultato non rappresentabile su k bit
  - Si usa la matematica “dell’orologio” (vedi prossime slide)

Facciamo un primo esempio (senza overflow):

- $k = 6, 5 + 12$
  - $5 = 0000101$; $12 = 001100$

$$\begin{array}{cccccc} 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 1 \end{array}$$

  - $010001 = 17$

Esempi senza overflow

- $k = 7, 5 - 8$
  - $5 = 0000101; -8 = \overline{0001000} + 1 = 1110111 + 1 = 1111000$

$$\begin{array}{cccccccc} & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ \hline & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{array}$$

  - $1111101 =$... $-(\overline{1111101} + 1) = -0000011 = -3$
- $k = 7, -5 + 8$
  - $-5 = \overline{0000101} + 1 = 1111010 + 1 = 1111011; 8 = 0001000$

(1)

$$\begin{array}{cccccccc} & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}$$

  - $0000011 = 3$

Esempi con overflow

- $k = 7, -64 - 8$
  - $64 = 1000000$; $-64 = \overline{1000000} + 1 = 0111111 + 1 = 1000000$
  - $8 = 0001000$; $-8 = \overline{0001000} + 1 = 1110111 + 1 = 1111000$

    (1)

1	0	0	0	0	0	0
1	1	1	1	0	0	0
0	1	1	1	0	0	0

  - $0111000 = 56$...
    Sommando 2 numeri negativi si ottiene un numero positivo?
  - Numeri rappresentabili ($k = 7$): da $-2^6 = -64$ a $2^6 - 1 = 63$
  - $-64 - 8 = -72$ non è rappresentabile!

Esempi con overflow

- $k = 7, 63 + 1$
  - $63 = 0111111$
  - $1 = 00000001$

$$\begin{array}{ccccccc} 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

  - $1000000 = -(\overline{1000000} + 1) = -(0111111 + 1) = -1000000 = -64$... Sommando 2 numeri positivi si ottiene un numero negativo?
- Ancora, il risultato corretto ($63 + 1 = 64$) non è rappresentabile!

Esempi ed esercizi

- Numeri su $k = 7$ bit, codificati in complemento a 2
- Verificare se c'è overflow
  - $0110110 + 0000011$
    - Somma di due numeri positivi: mi aspetto risultato positivo...
  - $1001010 + 1111101$
    - Somma di due numeri negativi: mi aspetto risultato negativo...
  - $1001010 + 1100000$
    - Somma di due numeri negativi: mi aspetto risultato negativo...
  - $0100000 + 1111010$
    - Somma di un numero positivo e uno negativo: verifichiamo...
  - $1100000 + 0000110$
    - Somma di un numero positivo e uno negativo: verifichiamo...

Ancora su overflow

- Abbiamo visto overflow in somme e sottrazione di interi
  - Se $a + b > 2^{k-1} - 1$, overflow!
  - Se $a + b < -2^{k-1}$, overflow!
- Nota: per definizione, $-2^{k-1} \leq a \leq 2^{k-1} - 1$ e $-2^{k-1} \leq b \leq 2^{k-1} - 1$
  - Overflow solo se a e b hanno lo stesso segno!

La matematica dell'orologio

- Aritmetica modulare: considero solo n numeri interi
  - Numeri da 0 a $n-1$...
  - ... oppure numeri da $-n/2$ a $n/2-1$...
- Arrivato al numero più grande, riparto dal più piccolo
  - Se i numeri vanno da 0 a $n-1$, $(n-1)+1=0$ (contando dopo $n-1$ riparto da 0)
  - Se i numeri vanno da $-n/2$ a $n/2-1$, $n/2-1+1=-n/2$
  - Numeri sul quadrante di un orologio!
- Definizione più formale: classi di equivalenza modulo n
  - Relazione di equivalenza fra due numeri: $a \equiv b \Leftrightarrow a\%n = b\%n$

Cosa succede in caso di overflow?

- Operazioni su numeri di k bit $\rightarrow$ aritmetica modulo 2^k
  - Calcolando $a + b$ si ottiene in realtà $(a + b)\%2^k$
  - Se $a + b < 2^k$, allora $(a + b)\%2^k = a + b \rightarrow$ no overflow!
  - Altrimenti, risultato "strano"... cosa significa $(a + b)\%2^k$?
- Usando la rappresentazione in complemento a 2 dei numeri interi, il discorso è analogo:
  - Se $a + b$ è compreso fra -2^{k-1} e $2^{k-1} - 1$, allora no overflow (risultato corretto)!
  - Se $a + b < -2^{k-1}$, ottengo $a + b + 2^k$. Positivo! Overflow!
  - Se $a + b > 2^{k-1} - 1$, ottengo $a + b - 2^k$. Negativo! Overflow!

Rappresentazione dei numeri reali

- Rappresentare esattamente un numero reale $x \in \mathcal{R}$ non è sempre possibile
  - E' il caso dei numeri irrazionali (es. π), ovvero numeri reali con un numero infinito di cifre "dopo la virgola" non periodiche...
  - ... non rappresentabili con un numero finito di bit!
- In alcuni casi possiamo quindi rappresentare solo un'approssimazione dei numeri reali
  - Alcuni numeri reali sono rappresentabili correttamente...
  - ... altri no
- Rappresentazioni in "virgola fissa" vs "virgola mobile" (rispettivamente dette "fixed point" e "floating point")

Virgola fissa

- Numero reale su k cifre: si possono dedicare $k - f$ cifre alla parte intera e f cifre alla parte frazionaria: $\overbrace{\underbrace{c_{k-1}c_{k-2}\cdots c_f}_{k-f} \cdot \underbrace{c_{f-1}\cdots c_0}_{f}}^{k}$
  - $x_{10} = \sum_{i=0}^{k-1} c_i B^{i-f} = \sum_{i=0}^{k-1} c_i B^i \cdot B^{-f} = \left(\sum_{i=0}^{k-1} c_i B^i\right) \cdot B^{-f}$
  - In altre parole, convertiamo il numero in base 10 come se non avesse virgola e poi moltiplichiamo per B^{-f}.
  - Un grosso vantaggio é che le operazioni si fanno in maniera semplice. E' sufficiente effettuare somme, sottrazioni, moltiplicazioni come se fossero numeri interi e riscalare il risultato.
  - Questo semplifica i requisiti per realizzare una CPU.
  - Inoltre se *i risultati sono rappresentabili*, le operazioni non introducono errori di approssimazione.
  - Gli svantaggi emergono quando vogliamo trattare nella stessa applicazione sia numeri grandi che numeri piccoli (cioé quando la nostra applicazione abbraccia piú ordini di grandezza).

Virgola fissa: esempio di conversione

- Prendiamo in considerazione un paio di esempi di codifica a virgola fissa a 6 bit, 3 per la parte intera e 3 per la parte decimale
- Limitiamoci per ora a due casi di numeri reali *rappresentabili* con questa codifica
- Conversione da binario a decimale di 101.100
  $101.100 = \left(1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0\right) + \left(1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3}\right) = 5.5$
- Conversione da decimale a binario di 6.125
  - Parte intera $\rightarrow$ converto direttamente: $6 = 110$
  - Parte decimale $\rightarrow$ moltiplico la parte decimale ricorsivamente per 2, tante volte quanti sono i bit della parte decimale, e considero nell'ordine le parti intere della moltiplicazione:
    1) $0.125 * 2 = 0.25$ - parte intera: 0
    2) $0.25 * 2 = 0.5$ - parte intera: 0
    3) $0.5 * 2 = 1$ - parte intera: 1
  - Mettendo insieme $\rightarrow$ $6.125 = 110.001$

Virgola mobile

- Un numero reale x può essere riscritto come:
  - $x = M \cdot B^E$
  - M: mantissa
  - E: esponente
- x puó essere quindi rappresentato in virgola mobile su k bit:
  - m bit per la mantissa M
  - $e = k - m$ bit per l'esponente E
- L'esponente E è quello che fa "muovere la virgola"
- Nei computer, $B = 2$ (rappresentazione binaria)
- Mantissa ed esponente: positivi o negativi
  - Segno mantissa: segno di x
  - Esponente negativo: x "piccolo"
  - Esponente positivo: x "grande"
- Rappresentati in complemento a 2, o con altre tecniche di codifica...

... vedi prossima slide.

Numeri reali in C (Standard IEEE 754)

- Esponente $-2^{e-1}+1 < E \leq 2^{e-1}-1$ rappresentato come:

$$E' = E + 2^{e-1} - 1$$

  Nota: per definizione, $E' \geq 0$ (vedi limiti di E)
- Se $E' > 0 \rightarrow$ numeri "normalizzati":
  - Mantissa $1.M$ ($M \geq 0$ e bit di segno s)
  - Esponente $E = E' - (2^{e-1}-1)$
- Se $E' = 0 \rightarrow$ numeri "denormalizzati":
  - Mantissa $0.M$ ($M > 0$ e bit di segno s)
  - Esponente $E = -(2^{e-1}-2)$

$$E' > 0 : x = \begin{cases} 1.M \cdot 2^{E'-(2^{e-1}-1)} & \text{se } s = 0 \\ -1.M \cdot 2^{E'-(2^{e-1}-1)} & \text{se } s = 1 \end{cases}$$

$$E' = 0 : x = \begin{cases} 0.M \cdot 2^{-(2^{e-1}-2)} & \text{se } s = 0 \\ -0.M \cdot 2^{-(2^{e-1}-2)} & \text{se } s = 1 \end{cases}$$

Tipi di floating point in C

- `float`: precisione singola
  - $k = 32$
  - $e = 8$, $m = 23$
- `double`: precisione doppia
  - $k = 64$
  - $e = 11$, $m = 52$
- `long double`: precisione estesa o quadrupla
  - $k = 80$ o $k = 128$
  - $e = 15$, $m = 64$ o $m = 112$

Precisione singola

- Approssimazioni di numeri reali rappresentati su $k = 32$ bit
- Esponente rappresentato su $e = 8$ bit
- Mantissa rappresentata su $m = 23$ bit (e bit di segno s)
  - Se $E' > 0$, esponente $E = E' - (2^7 - 1) = E' - 127$
  - Se $E' = 0$, esponente $E = -(2^7 - 2) = -126$

$$E' > 0 \Rightarrow x = (-1)^s \cdot 1.M \cdot 2^{E'-127}$$

$$E' = 0 \Rightarrow x = (-1)^s \cdot 0.M \cdot 2^{-126}$$

Precisione singola: valori massimo e minimo

- Massimo valore normalizzato rappresentabile:
  - Massima mantissa: $M = \overbrace{1 \cdots 1}^{23}$
  - Massimo esponente: $E = 254 - 127 = 127$
    (nota: il massimo valore di E' é 254 e non 255, perché 255 é riservato per casi particolari, vedi dopo)
  - $x = 1.\overbrace{1 \cdots 1}^{23} \cdot 2^{127} = (2 - 2^{-23}) \cdot 2^{127} \simeq 3.4 \cdot 10^{38}$
    $(1.1 \cdots 11 + 0.0 \cdots 01 = 2 \rightarrow 1.1 \cdots 11 = 2 - 0.0 \cdots 01 = 2 - 2^{-23})$
- Minimo valore normalizzato rappresentabile: $\simeq -3.4 \cdot 10^{38}$

Precisione singola: valori massimo e minimo

- Minimo valore normalizzato (positivo) rappresentabile:
  - Minima mantissa: $M = 0$
  - Minimo esponente: $E = 1 - 127 = -126$
    (nota: il minimo valore di E' é 1 e non 0, perché 0 é riservato ai numeri denormalizzati, vedi dopo)
  - $x = 1.\overbrace{0 \cdots 0}^{23} \cdot 2^{-126} = 2^{-126} \simeq 1.755 \cdot 10^{-38}$
- Minimo valore denormalizzato (positivo) rappresentabile:
  - Minima mantissa: $M = \overbrace{0 \cdots 01}^{23}$
  - Esponente: $E' = 0 \Rightarrow E = -126$
  - $x = 0.\overbrace{0 \cdots 01}^{23} \cdot 2^{-126} = 2^{-23} \cdot 2^{-126} \simeq 1.4 \cdot 10^{-45}$

Precisione singola: riepilogo - 1

Lo standard comprende quindi due categorie di numeri:

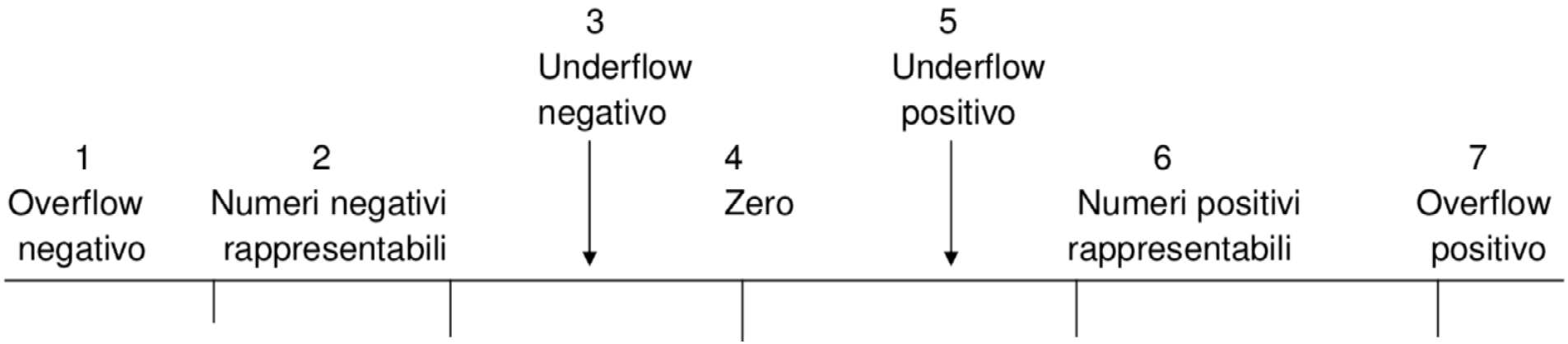
- Numeri normalizzati $\rightarrow 1.M$, $M \geq 0$, $E' > 0$.
- Numeri denormalizzati (o subnormalizzati) $\rightarrow 0.M$, $M > 0$, $E' = 0$. Questi sono numeri compresi nell'intervallo tra zero ed il più piccolo numero normalizzato rappresentabile. Si verificano in caso di "underflow" (con una perdita graduale di precisione man mano che ci si allontana dallo zero). Rappresentati con esponente pari a 0 e mantissa pari ad una qualunque sequenza di bit diversa da 0.

Oltre a questi, lo standard prevede i seguenti casi particolari:

- Due zeri (± 0), uno positivo e l'altro negativo, determinati dal bit di segno. Entrambi rappresentati con esponente 0 e mantissa 0.
- Due valori di "infinito" ($\pm\infty$), uno positivo e l'altro negativo, determinati dal bit di segno. Entrambi rappresentati con esponente pari 255 (non permesso per i numeri normalizzati) e mantissa 0.
- Un formato speciale, chiamato "NaN" (Not a Number). Rappresentato con esponente pari a 255 (non permesso per i numeri normalizzati) e mantissa pari ad una qualunque sequenza di bit diversa da 0.

Precisione singola: riepilogo - 2

Categoria	E'	M	s
Numeri normalizzati	1-254	qualunque	0/1
Numeri denormalizzati	0	non zero	0/1
$\pm$ Zero	0	0	0/1
$\pm$ Infinito	255	0	0/1
NaN (Not a Number)	255	non zero	0/1



Precisione singola: esempio di conversione

Convertire in virgola mobile il numero 132.125

1. Converto parte intera e parte decimale in binario:
   132 = 10000100; 0.125 = 001 $\Rightarrow$ 10000100.001
2. Sposto la virgola a sinistra, lasciando solo un 1 a sinistra (rappresentazione normalizzata, equivale a dividere per 2^k dove k é la lunghezza in bit della parte intera)
   10000100.001 = $1.0000100001 \cdot 2^7 = (-1)^s \cdot 1.M \cdot 2^{E'-127}$
3. Mantissa $M = 0000100001[00000000000]$ (sottintendo l'1 a sinistra della virgola, e aggiungo tanti 0 a destra quanti sono i bit della parte decimale, 23 nel caso di precisione singola)
4. Esponente: $E = E' - 127 = 7 \rightarrow E' = 7 + 127 = 134 = 10000110$ (notazione "eccesso 127")
5. Segno: $s = 0$ ($132.125 > 0$)
6. Mettendo insieme: $\underbrace{0}_{\text{segno}}\ \underbrace{10000110}_{\text{esponente}}\ \underbrace{00001000010000000000000}_{\text{mantissa}}$