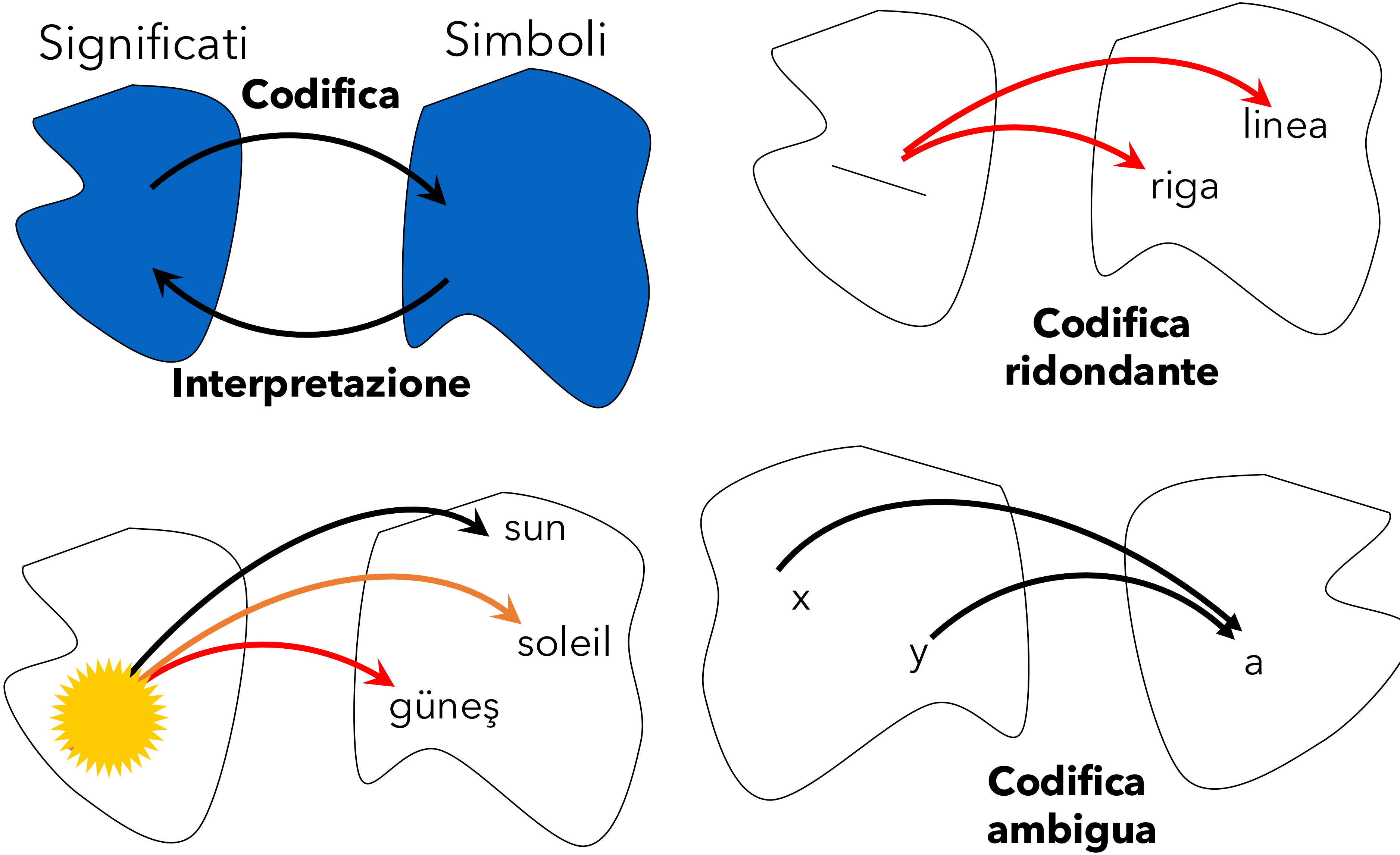
Codifica binaria dell’Informazione Aritmetica del Calcolatore

Credits to A. Fuggetta, A. Campi e P. Pinoli

Codificare e interpretare



Codifica dell'informazione

- Rappresentare (codificare) le informazioni
  - con un insieme limitato di simboli (detto *alfabeto A*)
  - in modo non ambiguo (algoritmi di traduzione tra codifiche)
- Esempio: numeri interi
  - Codifica decimale (**dec**, in base dieci)
  - A = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, |A| = dieci
    - "sette" : 7_{dec}
    - "ventitre" : 23_{dec}
    - "centotrentotto" : 138_{dec}
  - Notazione *posizionale*
    - dalla cifra più significativa a quella meno significativa
    - ogni cifra corrisponde a una diversa potenza di dieci
    - $138_{dec}=8*10^0 + 3*10^1 + 1*10^2$

Numeri naturali

- *Notazione posizionale*: permette di rappresentare un qualsiasi numero naturale (intero non negativo) nel modo seguente:

  la sequenza di **cifre c_i**:

  $$c_n \; c_{n-1} \; \ldots \; c_1$$

  rappresenta in **base B** ≥ 2 il valore:

  $$c_n \times B^{n-1} + c_{n-1} \times B^{n-2} + \ldots + c_1 \times B^0$$

  avendosi: $c_i \in \{0, 1, 2, \ldots, B-1\}$ per ogni $1 \leq i \leq n$
- La notazione decimale tradizionale è di tipo posizionale (ovviamente con B = dieci)
- Esistono notazioni non posizionali
  - Ad esempio i numeri romani: II IV VI XV XX ~~VV~~

Numeri naturali in varie basi

- Base generica: B
  - A = { ... }, con |A| = B, sequenze di *n* simboli (cifre)
  - $c_n\, c_{n-1} \ldots c_2\, c_1 = c_n \times B^{n-1} + \ldots + c_2 \times B^1 + c_1 \times B^0$
  - Con n cifre rappresentiamo B^n numeri: da 0 a B^n-1
- "ventinove" in varie basi
  - B = otto A = {0,1,2,3,4,5,6,7} $29_{10} = 35_8$
  - B = cinque A = {0,1,2,3,4} $29_{10} = 104_5$
  - B = tre A = {0,1,2} $29_{10} = 1002_3$
  - B = sedici A = {0,1,...,8,9,A,B,C,D,E,F} $29_{10} = 1D_{16}$
- Codifiche notevoli
  - Esadecimale (sedici), ottale (otto), binaria (due)

Codifica dei numeri naturali

- Considerato un sistema in base $\mathtt{B}$ in cui abbiamo N cifre per rappresentare un numero, quanti elementi (numeri) distinti possiamo rappresentare?

$$\mathtt{B}^{N}$$

- Al contrario, se dobbiamo rappresentare $\mathtt{M}$ numeri distinti, di quante cifre abbiamo bisogno?

$$\lceil log_{B}(M) \rceil$$

$\lceil\ \rceil$ = parte intera superiore

Codifica binaria

- Ogni dato o elemento di un dato è rappresentato come una sequenza di caratteri dell'alfabeto binario A= {0,1}.
- Le principali motivazioni sono:
  - All'inizio dell'informatica applicata, era più semplice costruire computer con due soli valori per ogni elemento (1 = acceso, 0 = spento)
  - Un sistema con solo due stati è molto affidabile
  - Le operazioni logiche sono facili da implementare nei circuiti
- Importante: solitamente i computer usano una sequenza di lunghezza fissa per rappresentare ciascun dato o componente

Codifica binaria

- B = 2, **A = { 0, 1 }**
- **BIT** (crasi di "BInary digIT"):
  - unità **elementare** di informazione
- *Numeri binari naturali:*

  la sequenza di **bit** $\mathbf{b_i}$ (cifre binarie):

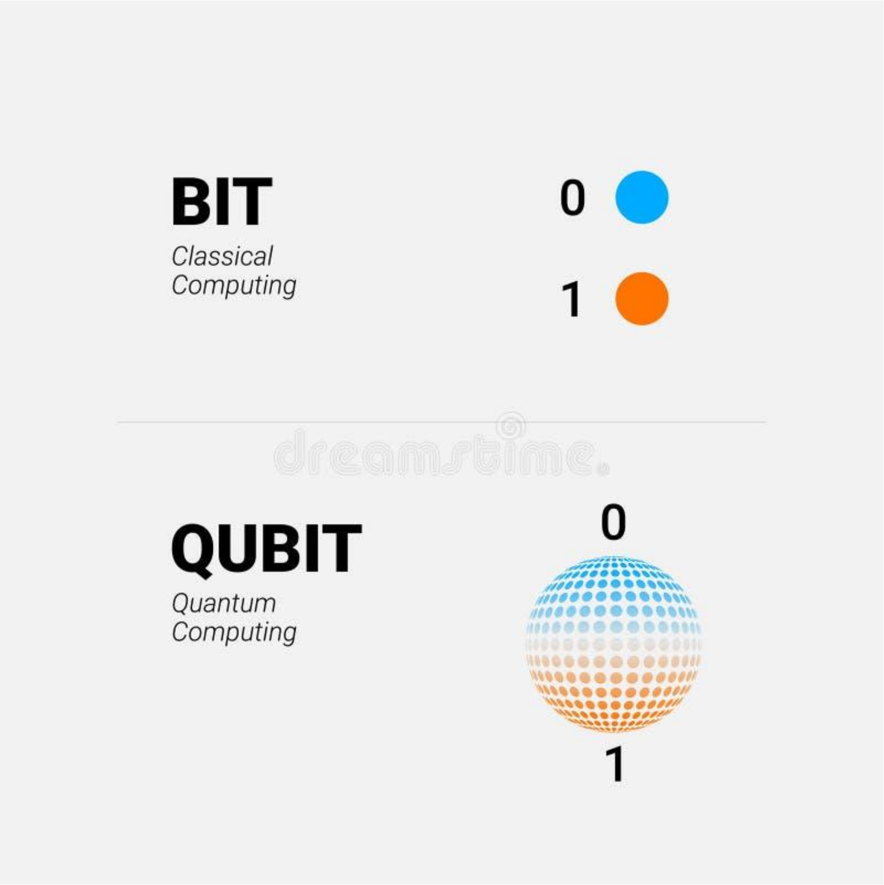
  $$b_n \; b_{n-1} \; \dots \; b_1 \qquad \text{con } b_i \in \{0, 1\}$$

  rappresenta in base 2 il valore:

  $$b_n \times 2^{n-1} + b_{n-1} \times 2^{n-2} + \dots + b_1 \times 2^0$$

Quantum bits (qubit)

- Un qubit è l'unità fondamentale di informazione nella computazione quantistica
- Può esistere in una sovrapposizione di entrambi gli stati 0 e 1 contemporaneamente
- Possono elaborare una quantità maggiore di informazioni
- Possono essere fisicamente realizzati tramite vari sistemi, come elettroni, fotoni o atomi



Numeri naturali binari (bin)

- Con n bit codifichiamo 2^n numeri: da 0 a 2^n-1
- Con 1 Byte (cioè una sequenza di 8 bit):
  - $00000000_{bin} = 0_{dec}$
  - $00001000_{bin} = 1 \times 2^3 = 8_{dec}$
  - $00101011_{bin} = 1 \times 2^5 + 1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = 43_{dec}$
  - $11111111_{bin} = \Sigma_{n = 1,2,3,4,5,6,7,8}\ 1 \times 2^{n-1} = 255_{dec}$
- Conversione bin $\rightarrow$ dec e dec $\rightarrow$ bin
  - bin$\rightarrow$dec: $11101_{bin} = \Sigma_i\, b_i 2^{i-1} = 2^4+2^3+2^2+2^0 = 29_{dec}$
  - dec$\rightarrow$bin: ***metodo dei resti***

Conversione dec → bin

Si calcolano i resti delle divisioni per due

In pratica basta:

1. Decidere se il numero è pari (resto 0) oppure dispari (resto 1), e annotare il resto
2. Dimezzare il numero (trascurando il resto)
3. Ripartire dal punto 1. fino a ottenere 0 come risultato della divisione

$19 : 2 \rightarrow 1$

$9 : 2 \rightarrow 1$

$4 : 2 \rightarrow 0$

$2 : 2 \rightarrow 0$

$1 : 2 \rightarrow 1$

si ottiene 1: fine

$\mathbf{19_{dec} = 10011_{bin}}$

Ecco un esempio, per quanto modesto, di **algoritmo**

Metodo dei resti

$$\begin{aligned}
\mathbf{29} : 2 &= 14 \quad (1)\\
14 : 2 &= 7 \quad (0)\\
7 : 2 &= 3 \quad (1)\\
3 : 2 &= 1 \quad (1)\\
1 : 2 &= \mathbf{0} \quad (1)
\end{aligned}$$

$29_{dec} = 11101_{bin}$

$$\begin{aligned}
\mathbf{76} : 2 &= 38 \quad (0)\\
38 : 2 &= 19 \quad (0)\\
19 : 2 &= 9 \quad (1)\\
9 : 2 &= 4 \quad (1)\\
4 : 2 &= 2 \quad (0)\\
2 : 2 &= 1 \quad (0)\\
1 : 2 &= \mathbf{0} \quad (1)
\end{aligned}$$

$76_{dec} = 1001100_{bin}$

Del resto 76 = 19x4 = **1001100**

Per raddoppiare, in base due, si aggiunge uno zero in coda, così come si fa in base dieci per decuplicare

N.B. Il metodo funziona con tutte le basi!

$29_{10}=45_6=32_9=27_{11}=21_{14}=10_{29}$

Conversioni rapide bin → dec

- In binario si definisce una *notazione abbreviata*, sulla falsariga del sistema metrico-decimale:

K	$= 2^{10}$	$= 1.024 \approx 10^3$	(Kilo)
M	$= 2^{20}$	$= 1.048.576 \approx 10^6$	(Mega)
G	$= 2^{30}$	$= 1.073.741.824 \approx 10^9$	(Giga)
T	$= 2^{40}$	$= 1.099.511.627.776 \approx 10^{12}$	(Tera)

- È curioso (benché *non* sia casuale) come K, M, G e T in base 2 abbiano valori molto prossimi ai corrispondenti simboli del sistema metrico decimale, tipico delle scienze fisiche e dell'ingegneria
- L'errore risulta $< 10\ \%$ (infatti la seconda cifra è sempre 0)

Ma allora…

- Diventa molto facile e quindi *rapido* calcolare il valore *decimale approssimato* delle *potenze di 2*, anche se hanno esponente grande
- Infatti basta:
  - *Tenere a mente* l'elenco dei valori esatti delle prime dieci potenze di 2 **[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]**
  - *Scomporre* in modo *additivo* l'esponente in contributi di valore 10, 20, 30 o 40, "leggendoli" come successioni di simboli K, M, G oppure T

Tenendo presente che:

$2^0=1$, $2^1=2$, $2^2=4$, $2^3=8$, $2^4=16$, $2^5=32$,

$2^6=64$, $2^7=128$, $2^8=256$, $2^9=512$, $2^{10}=1024$

Tenendo presente che:

$2^0=1$, $2^1=2$, $2^2=4$, $2^3=8$, $2^4=16$, $2^5=32$, $2^6=64$, $2^7=128$, $2^8=256$, $2^9=512$, $2^{10}=1024$

Aumento e riduzione dei bit in bin

- ***Aumento*** dei bit
  - premettendo in modo progressivo un bit 0 a sinistra, il valore del numero non muta

$4_{dec} = 100_{bin} = 0100_{bin} = 00100_{bin} = \dots 000000000100_{bin}$

$5_{dec} = 101_{bin} = 0101_{bin} = 00101_{bin} = \dots 000000000101_{bin}$

- ***Riduzione*** dei bit
  - cancellando in modo progressivo un bit 0 a sinistra, il valore del numero non muta, *ma bisogna arrestarsi quando si trova un bit* 1!

$7_{dec} = 00111_{bin} = 0111_{bin} = 111_{bin}$ STOP !

$2_{dec} = 00010_{bin} = 0010_{bin} = 010_{bin} = 10_{bin}$ STOP !

Numeri interi in modulo e segno (m&s)

- *Numeri binari interi* (positivi e negativi) *in modulo e segno* (**m&s**)
  - il primo bit a sinistra rappresenta il segno del numero (*bit di segno*), i bit rimanenti rappresentano il valore
    - 0 per il segno positivo
    - 1 per il segno negativo
- Esempi con n = 9 (8 bit + un bit per il segno)
  - $000000000_{m\&s} = +\ 0$
  - $000001000_{m\&s} = +\ 1 \times 2^3 = 8_{dec}$
  - $100001000_{m\&s} = -\ 1 \times 2^3 = -8_{dec}$
  - … e così via …

Osservazioni sul m&s

- Il bit di segno è *applicato* al numero rappresentato, ma non fa propriamente *parte* del numero in quanto tale
  - il bit di segno non ha significato numerico
- *Distaccando* il bit di segno, i bit rimanenti rappresentano il **valore assoluto** del numero
  - che è intrinsecamente positivo

Quiz

Intervallo

Fonte: PlaygroundAI

Il complemento a 2 (C_2)

- *Numeri interi in complemento a* 2: il C_2 è *un* sistema binario, ma il primo bit (quello a sinistra, il più significativo **MSB**) ha *peso negativo*, mentre tutti gli altri bit hanno peso positivo
- La sequenza di bit:

  $$b_n\, b_{n-1} \ldots b_1$$

  rappresenta in C_2 il valore:

  $$-b_n \times 2^{n-1} + b_{n-1} \times 2^{n-2} + \ldots + b_1 \times 2^0$$

Il bit più a sinistra è ancora chiamato *bit di segno*

Numeri a tre bit in C_2

- $000_{C2} = -0\times2^2 + 0\times2^1 + 0\times2^0 = \ 0_{dec}$
- $001_{C2} = -0\times2^2 + 0\times2^1 + 1\times2^0 = \ 1_{dec}$
- $010_{C2} = -0\times2^2 + 1\times2^1 + 0\times2^0 = \ 2_{dec}$
- $011_{C2} = -0\times2^2 + 1\times2^1 + 1\times2^0 = \ 2+1 = 3_{dec}$
- $100_{C2} = -1\times2^2 + 0\times2^1 + 0\times2^0 = -4_{dec}$
- $101_{C2} = -1\times2^2 + 0\times2^1 + 1\times2^0 = -4+1 = -3_{dec}$
- $110_{C2} = -1\times2^2 + 1\times2^1 + 0\times2^0 = -4+2 = -2_{dec}$
- $111_{C2} = -1\times2^2 + 1\times2^1 + 1\times2^0 = -4+2+1 = -1_{dec}$

N.B.: in base al bit di segno lo zero è considerato positivo

Interi relativi in m&s e C_2

Se usiamo 1 Byte (8-bit): da –128 a 127

dec.	m&s	C_2
127	**0**1111111	**0**1111111
126	**0**1111110	**0**1111110
…	…	…
2	**0**0000010	**0**0000010
1	**0**0000001	**0**0000001
+0	**0**0000000	**0**0000000
-0	**1**0000000	**-**
-1	**1**0000001	**1**1111111
-2	**1**0000010	**1**1111110
…	…	…
-126	**1**1111110	**1**0000010
-127	**1**1111111	**1**0000001
-128	**-**	**1**0000000

Il complemento a 2 (C_2): perché?

- Utilizzato anche da Von Neumann (1945)
- Esiste anche il complemento a 1
  - I numeri positivi si rappresentano normalmente
  - I numeri negativi si ottengono invertendo i bit della versione positiva
  - Problema del «doppio zero»
- I **circuiti di addizione e sottrazione** non guardano il segno ma usano un solo circuito (**sommatore**)
- Tecnologie più semplici e con maggiore precisione

Invertire un numero in C_2

- L'*inverso additivo* (o *opposto*) –N di un numero N rappresentato in C_2 si ottiene:
  - Invertendo (negando) ogni bit del numero
  - Sommando 1 alla posizione meno significativa e ignorando l'overflow
- Esempio:
  - $01011_{C2} = 1\times2^3+1\times2^1+1\times2^0 = 8+2+1 = 11_{dec}$
  - $10100 + 1 = 10101_{C2} = -1\times2^4+1\times2^2+1\times2^0 = -16+4+1 = -11_{dec}$

Invertire un numero in C_2

Esercizio

1. Invertire $11011_{C2} = -5_{dec}$
2. Si verifichi che con due applicazioni dell'algoritmo si riottiene il numero iniziale [$-(-N) = N$] e che lo zero in C2 è (correttamente) opposto di se stesso [$-0 = 0$]

Conversione dec $\rightarrow C_2$

- Se $D_{dec} \geq 0$:
  - Converti D_{dec} in binario naturale.
  - Premetti il bit 0 alla sequenza di bit ottenuta.
  - Esempio: $154_{dec} \Rightarrow 10011010_{bin} \Rightarrow 010011010_{C2}$
- Se $D_{dec} < 0$:
  - Trascura il segno e converti D_{dec} in binario naturale
  - Premetti il bit 0 alla sequenza di bit ottenuta
  - Calcola l'opposto del numero così ottenuto, secondo la procedura di inversione in C_2
  - Esempio: $-154_{dec} \Rightarrow 154_{dec} \Rightarrow 10011010_{bin} \Rightarrow$
    $\Rightarrow 010011010_{bin} \Rightarrow 101100101 + 1 \Rightarrow 101100110_{C2}$
- Occorrono 9 bit sia per 154_{dec} che per -154_{dec}

Aumento e riduzione dei bit in C_2

- ***Estensione*** del segno:
  - *replicando* in modo progressivo il bit di segno a sinistra, il valore del numero non muta

$4 = 0100 = 00100 = 00000100 = \ldots$ (indefinitamente)

$-5 = 1011 = 11011 = 11111011 = \ldots$ (indefinitamente)

$-1\times2^3+1\times2^1+1\times2^0$

$-1\times2^4+1\times2^3+1\times2^1+1\times2^0$

- ***Contrazione*** del segno:
  - *cancellando* in modo progressivo il bit di segno a sinistra, il valore del numero non muta
  - *purché il bit di segno non abbia a invertirsi* !

$7 = 000111 = 00111 = 0111$ **STOP**! (111 è < 0)

$-3 = 111101 = 11101 = 1101 = 101$ **STOP**! (01 è > 0)

Osservazioni sul C_2

- Il segno è *incorporato* nel numero rappresentato in C_2, non è semplicemente *applicato* (come in m&s)
- Il bit più significativo *rivela* il segno: 0 per numero positivo, 1 per numero negativo (il numero zero è considerato positivo), ma…
- **NON** si può *distaccare* il bit più significativo e dire che i bit rimanenti rappresentano il valore assoluto del numero
    - questo è ancora vero, però, <u>se il numero è positivo</u>

Intervalli di rappresentazione

- Binario naturale a $n \geq 1$ bit: $[0, 2^{n}-1]$
- Modulo e segno a $n \geq 2$ bit: $[-2^{n-1}-1, 2^{n-1}-1]$
- C_2 a $n \geq 2$ bit: $[-2^{n-1}, 2^{n-1}-1]$
  - In modulo e segno, il numero zero ha due rappresentazioni *equivalenti* (00..0, 10..0)
  - L'intervallo del C_2 è *asimmetrico* (-2^{n-1} è compreso, 2^{n-1} è escluso); poco male …

Operazioni – <u>numeri binari naturali</u>

Algorimo di “addizione a propagazione dei riporti”

È l’algoritmo decimale elementare, adattato alla base 2

Pesi	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
Riporto			1	1	1					
Addendo 1	0	1	0	0	1	1	0	1	+	77_{dec}
Addendo 2	1	0	0	1	1	1	0	0	=	156_{dec}
Somma	1	1	1	0	1	0	0	1		233_{dec}

addizione naturale (a 8 bit)

Operazioni – numeri binari naturali

overflow (o *trabocco*)

Pesi		*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
Riporto	1	1	1	1	1	1					
Addendo 1		0	1	1	1	1	1	0	1	+	125_{dec}
Addendo 2		1	0	0	1	1	1	0	0	=	156_{dec}
Somma		0	0	0	1	1	0	0	1		25_{dec} !

Riporto “perduto”

overflow

risultato errato!

addizione **naturale** con overflow

Riporto e overflow (addizione naturali)

- Si ha ***overflow*** quando il risultato corretto dell'addizione eccede il potere di rappresentazione dei bit a disposizione
  - 8 bit nell'esempio precedente
- Nell'addizione tra numeri binari naturali si ha overflow **ogni volta** che si genera un riporto addizionando i bit della colonna più significativa (riporto "perduto")

Operazioni – numeri C_2

Pesi	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
Riporto			1	1	1					
Addendo 1	0	1	0	0	1	1	0	1	+	77_{dec}
Addendo 2	1	0	0	1	1	1	0	0	=	-100_{dec}
Somma	1	1	1	0	1	0	0	1		-23_{dec}

addizione ***algebrica*** (a 8 bit)

L’algoritmo è ***identico*** a quello naturale

(come se il primo bit non avesse peso negativo)

Operazioni – numeri C_2

ancora overflow

Pesi	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*		
Riporto	1		1	1	1					
Addendo 1	0	1	0	0	1	1	0	1	+	77_{dec}
Addendo 2	0	1	0	1	1	1	0	0	=	92_{dec}
Somma	1	0	1	0	1	0	0	1		-87_{dec} !

nessun riporto "perduto"

Overflow: risultato negativo!

risultato errato!

addizione **algebrica** con overflow

Riporto e overflow in C_2 (addizione algebrica)

- Si ha ***overflow*** quando il risultato corretto dell'addizione eccede il potere di rappresentazione dei bit a disposizione
  - La definizione di overflow non cambia
- Si può avere overflow senza "riporto perduto"
  - Capita quando da due addendi positivi otteniamo un risultato negativo, come nell'esempio precedente
- Si può avere un "riporto perduto" senza overflow
  - Può essere un innocuo effetto collaterale
  - Capita quando due addendi discordi generano un risultato positivo

Esercizio

1. Si può avere un “riporto perduto” senza overflow
   - **Si provi a sommare +12 e -7**

Quando non si verifica **mai** l’overflow?

Rilevare l’overflow in C_2

- Se gli addendi sono tra loro ***discordi*** (di segno diverso) non si verifica mai
- Se gli addendi sono tra loro ***concordi***, si verifica se e solo se il risultato è discorde
  - addendi positivi ma risultato negativo
  - addendi negativi ma risultato positivo
- Criterio di controllo facile da applicare!

Quiz

Intervallo

Fonte: PlaygroundAI

Rappresentazione ottale e esadecimale

- *Ottale* o in base otto (oct):
  - Si usano solo le cifre 0-7

    $534_{oct} = 5_{oct}\times 8_{dec}^{2} + 3_{oct}\times 8_{dec}^{1} + 4_{oct}\times 8_{dec}^{0} = 348_{dec}$
- *Esadecimale* o in base sedici (hex):
  - Si usano le cifre 0-9 e le lettere A-F per i valori 10-15

    $B7F_{hex} = B_{hex}\times 16_{dec}^{2} + 7_{hex}\times 16_{dec}^{1} + F_{hex}\times 16_{dec}^{0} =$

    $= 11_{dec}\times 16_{dec}^{2} + 7_{dec}\times 16_{dec}^{1} + 15_{dec}\times 16_{dec}^{0} = 2943_{dec}$
- Entrambe queste basi sono facili da convertire in binario, e viceversa

Conversioni bin → hex e hex → bin

- Converti: $10011110101011011_{bin} =$

  $0001_{bin}\ 0011_{bin}\ 1101_{bin}\ 0101_{bin}\ 1011_{bin} =$

  $= 1_{dec}\quad 3_{dec}\quad 13_{dec}\quad 5_{dec}\quad 11_{dec} =$

  $= 1_{hex}\quad 3_{hex}\quad D_{hex}\quad 5_{hex}\quad B_{hex} =$

  $= 13D5B_{hex}$

- Converti: $A7B40C_{hex}$

  $A_{hex}\quad 7_{hex}\quad B_{hex}\quad 4_{hex}\quad 0_{hex}\quad C_{hex} =$

  $= 10_{dec}\quad 7_{dec}\quad 11_{dec}\quad 4_{dec}\quad 0_{dec}\quad 12_{dec} =$

  $= 1010_{bin}\quad 0111_{bin}\quad 1011_{bin}\quad 0100_{bin}\quad 0000_{bin}\quad 1100_{bin} =$

  $= 101001111011010000001100_{bin}$

- Provate a convertire anche
  - oct → bin, dec → hex, dec → oct

Numeri frazionari in virgola fissa

- $0{,}1011_{bin}$ (in binario)

  $0{,}1011_{bin} = 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3} + 1\times2^{-4} = 1/2 + 1/8 + 1/16 =$
  $= 0{,}5 + 0{,}125 + 0{,}0625 = 0{,}6875_{dec}$

- Si può rappresentare un numero frazionario in *virgola fissa* (o *fixed point*) nel modo seguente:

  $19{,}6875_{dec} = 10011{,}1011_{\text{virgola fissa}}$

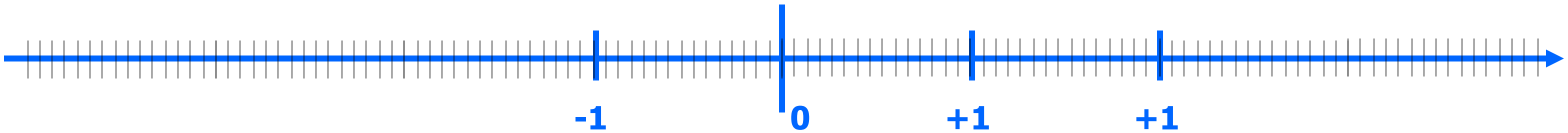
  poiché si ha:

  $19_{dec} = 10011_{bin}$ e $0{,}6875_{dec} = 0{,}1011_{bin}$

proporzione fissa:

5 bit per la parte intera, 4 bit per quella frazionaria

- Avremo 2^9 diversi valori codificati, e avremo 2^4 valori tra 0 e 1, 2^4 valori tra 1 e 2, … e così via, con tutti i valori distribuiti su un asse a distanze regolari



Numeri frazionari in virgola fissa

- La sequenza di bit rappresentante un numero frazionario consta di due parti di lunghezza prefissata
- Il numero di bit a sinistra e a destra della virgola è stabilito a priori, anche se alcuni bit restassero nulli
- È un sistema di rappresentazione semplice, ma poco flessibile, e può condurre a sprechi di bit
- Per rappresentare in virgola fissa numeri molto grandi (o molto precisi) occorrono molti bit
- La precisione nell'intorno dell'origine e lontano dall'origine è la stessa
- Anche se su numeri molto grandi in valore assoluto la parte frazionaria può non essere particolarmente significativa

Numeri frazionari in virgola mobile

- La rappresentazione in *virgola mobile* (o *floating point*) è usata spesso in base 10 (si chiama allora *notazione scientifica*):

  $0{,}137 \times 10^8$ $_{\text{notazione scientifica}}$ per intendere $13.700.000_{dec}$

- La rappresentazione si basa sulla relazione

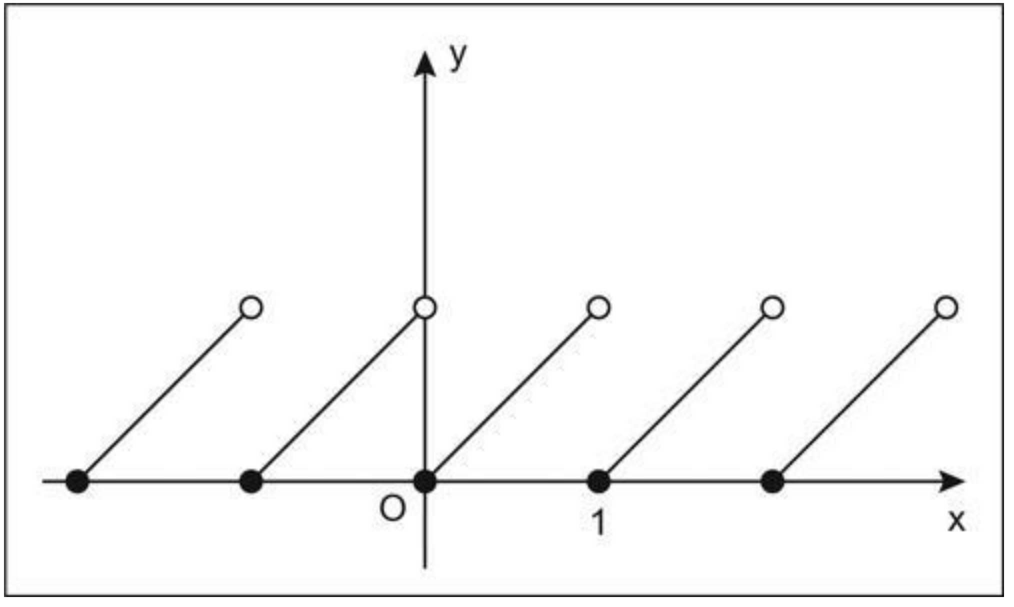
  $$\mathbf{R_{virgola\ mobile} = M \times B^E}$$ [attenzione: ***non*** $(M \mathrm{x} B)^E$]

- In binario, si utilizzano **m** $\geq$ 1 bit per la ***mantissa M*** e **n** $\geq$ 1 bit per **l'*esponente E***
  - mantissa: un numero frazionario (tra -1 e +1)
  - la base B non è rappresentata (è implicita)
  - in totale si usano m + n bit

Mantissa (o parte frazionaria)

- Funzione reale di variabile reale, indicata con `mant(x)`, definita come differenza tra `x` e la sua parte intera `[x]`:

$$\mathrm{mant}(x) = x - [x]$$



Fonte: Treccani

Numeri frazionari in virgola mobile

- Esempio
  - Supponiamo B=2, m=3 bit, n=3 bit, M ed E in binario naturale
    $M = 011_2$ ed $E = 010_2$
    $R_{\text{virgola mobile}} = 0{,}011 \times 2^{010} = (1/4 + 1/8) \times 2^2 = 3/8 \times 4 = 3/2 = 1{,}5_{dec}$
- M ed E possono anche essere negativi
  - Normalmente infatti si usa il *modulo e segno* per M, mentre per E si usa la rappresentazione cosiddetta *in eccesso*
- **Vantaggi della virgola mobile**
  - si possono rappresentare con pochi bit numeri molto grandi **oppure** molto precisi (cioè con molti decimali)
  - Sull'asse dei valori i numeri rappresentabili si affollano nell'intorno dello zero, e sono sempre più sparsi al crescere del valore assoluto

ATTENZIONE (i pericoli dei *floating points*)

- **Approssimazione**
  - $0{,}375 \times 10^7 + 0{,}241 \times 10^3 = 0{,}3750241 \times 10^7 \approx 0{,}375 \times 10^7$
- Ma, in virgola mobile, *se disponiamo di poche cifre per la mantissa:*
  - $0{,}375 \times 10^7 + 0{,}241 \times 10^3 = 0{,}375 \times 10^7$
  - del resto sarebbe sbagliato approssimare a $0{,}374 \times 10^7$ o $0{,}376 \times 10^7$
- Definiamo un ciclo che ripete la somma un milione di volte…
  - Inizia con $\mathbf{X = 0{,}375 \times 10^7}$
  - Ripeti 1.000.000 di volte $\mathbf{X = X + 0{,}241 \times 10^3}$ (*incremento non intero*)
  - Alla fine **dovrebbe essere** $X = 0{,}375 \times 10^7 + (0{,}241 \times 10^3 \times 10^6) \approx 0{,}245 \times 10^9$
- **Ma**, in virgola mobile…
  - Il contributo delle singole somme (una alla volta) si perde del tutto!
  - Il risultato **resta** $\mathbf{0{,}375 \times 10^7}$, sbagliato di due ordini di grandezza (*underflow*)
  - Scrivendo programmi che trattano valori rappresentati in virgola mobile è **necessario** essere consapevoli dei limiti di rappresentazione
  - Lo stesso è vero con gli interi (rischio di overflow)

Aritmetica standard

- Quasi tutti i calcolatori oggi adottano lo *standard aritmetico* **IEEE 754**, che definisce:
  - I *formati di rappresentazione* binario naturale, C_2 e virgola mobile
  - Gli *algoritmi* di somma, sottrazione, prodotto, ecc, per tutti i formati previsti
  - I metodi di *arrotondamento* per numeri frazionari
  - Come trattare gli *errori* (overflow, divisione per 0, radice quadrata di numeri negativi, ...)
- Grazie a IEEE 754, i programmi sono *trasportabili* tra calcolatori diversi senza che cambino né i *risultati* né la *precisione* dei calcoli svolti dal programma stesso

Standard IEEE 754-1985

L'idea e' di rappresentare un numero con virgola nella sua notazione scientifica:

$$13{,}564 = 0{,}\boxed{13564}*10\wedge\boxed{(2)}$$

$$0{,}00343 = 0{,}\boxed{343}*10\wedge\boxed{(-2)}$$

ESPONENTE

MANTISSA

Nello standard IEEE-754 **a precisione singola** abbiamo:
1 bit per il segno, 8 bit per l'esponente e 23 bit per la matita => totale 32 bit

Gradi di precisione

Previsti tre possibili gradi di precisione: singola, doppia, quadrupla

Campo	Precisione singola	Precisione doppia	Precisione quadrupla
ampiezza totale in bit di cui	32	64	128
Segno	1	1	1
Esponente	8	11	15
Mantissa	23	52	111
massimo E	255	2047	32767
minimo E	0	0	0
K	127	1023	16383

Il valore rappresentato vale quindi $X = (-1)^S \times 2^{E-K} \times 1.M$

Convertire un numero in virgola mobile con precisione singola

Conversione del numero **23,75**

Parte intera:
- Divido per 2
- Prendo il resto
- Fino a che ottengo 0

23 / 2 = 11 resto 1
11 / 2 = 5 resto 1
5 / 2 = 2 resto 1
2 / 2 = 1 resto 0
1 / 2 = 0 resto 1

Risultato 10111

Parte decimale:
- Moltiplico per 2
- Prendo la parte intera
- Fino a che ottengo 1

0,75 * 2 = 1,5 I = 1
0,5*2 = 1 I = 1

Risultato 11

Concateno:
10111,11
Sposto la virgola:
1,011111*2^4

Ottengo:
- **mantissa:** 011111
- **esponente**: 100

Convertire un numero in virgola mobile con precisione singola

Concateno:
10111,11
Sposto la virgola:
1,011111*2^4

Ottengo:
- **mantissa:** 011111
- **esponente**: 100

Il primo bit e' il segno: siccome il numero e' positivo il primo bit sara' **0**

Poi rappresento l'esponente in eccesso K, dove:

$K = 2^{\text{bit esponente -1}} - 1 \rightarrow 2^7 - 1 = 127$

Quindi ottengo 127+4 = 131 -> **10000011**

La mantissa la aggiungo semplicemente e completo gli 0 a destra

$\boxed{0}\boxed{10000011}\boxed{01111100000000000000000}$

Esempio

Esempio di rappresentazione in precisione singola

$X = 42.6875_{10} = 101010.1011_2 = 1.010101011 \times 2^5$

Si ha

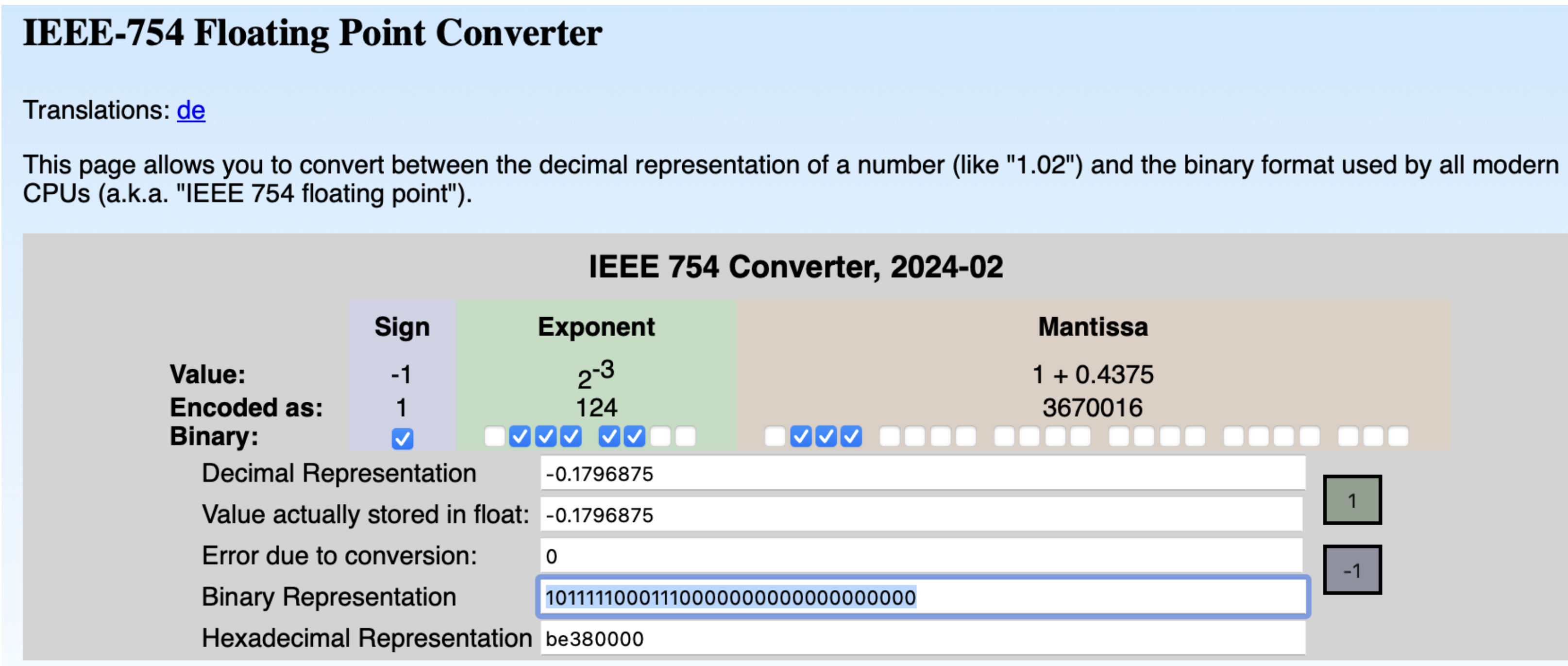
$S = 0$ (1 bit)

$E = 5 + K = 5_{10} + 127_{10} = 132 = 10000100_2$ (8 bit)

$M = 01010101100000000000000$ (23 bit)

Risorsa utile

- https://www.h-schmidt.net/FloatConverter/IEEE754.html



Proprietà fondamentali

- I circa 4 miliardi di configurazioni dei 32 bit usati consentono di coprire un campo di valori molto ampio grazie alla distribuzione non uniforme.
- Per numeri piccoli in valore assoluto valori rappresentati sono «fitti»,
- Per numeri grandi in valore assoluto valori rappresentati sono «diradati»

- Approssimativamente gli intervalli tra **valori contigui** sono
  - per valori di 10000 l'intervallo è di un millesimo
  - per valori di 10 milioni l'intervallo è di un'unità
  - per valori di 10 miliardi l'intervallo è di mille

Non solo numeri – codifica dei caratteri

- Nei calcolatori i caratteri vengono *codificati* mediante *sequenze* di n ≥ 1 bit, ognuna rappresentante un carattere distinto
  - Corrispondenza biunivoca tra numeri e caratteri
- Codice ASCII (*American Standard Computer Interchange Interface*): utilizza n=7 bit per 128 caratteri
- Il codice ASCII a 7 bit è pensato per la lingua inglese. Si può estendere a 8 bit per rappresentare il doppio dei caratteri
  - Si aggiungono così, ad esempio, le lettere con i vari gradi di accento (come À, Á, Â, Ã, Ä, Å, ecc), necessarie in molte lingue europee, e altri simboli speciali ancora

Alcuni simboli del codice ASCII

# (in base 10)	Codifica (7 bit)	Carattere (o simbolo)
0	0000000	<terminator>
9	0001001	<tabulation>
10	0001010	<carriage return>
12	0001100	<sound bell>
13	0001101	<end of file>
32	0100000	blank space
33	0100001	!
49	0110001	1
50	0110010	2
64	1000000	@
65	1000001	A
66	1000010	B
97	1100000	a
98	1100001	b
126	1111110	~
127	1111111	·

Tabella ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Rilevare gli errori

- Spesso, quando il codice ASCII a 7 bit è usato in un calcolatore avente parole di memoria da un Byte (o suoi multipli), l'ottavo bit del Byte memorizzante il carattere funziona come *bit di parità*
- Il bit di parità serve per *rilevare* eventuali *errori* che potrebbero avere alterato la sequenza di bit, purché siano errori di tipo abbastanza semplice

Bit di parità (*parity bit*)

- Si aggiunge un bit extra, in modo che il numero di bit uguali a 1 sia sempre *pari*:

  1100101 (quattro bit 1) ⇒ 11001010 (quattro bit 1)

  0110111 (cinque bit 1) ⇒ 01101111 (sei bit 1)

- Se per errore un (solo) bit si *inverte*, il conteggio dei bit uguali a 1 dà valore *dispari*!
- Così si può rilevare l'*esistenza* di un errore da un bit (ma non localizzarne la posizione)
- Aggiungendo *più* bit extra (secondo schemi opportuni) si può anche *localizzare* l'errore.
- Il bit di parità *non rileva* gli errori da due bit; ma sono meno frequenti di quelli da un bit

Altre codifiche alfanumeriche

- Codifica **ASCII** esteso a 8 bit (256 parole di codice). È la più usata.
- Codifica **FIELDATA** (6 bit, 64 parole codificate) Semplice ma compatta, storica
- Codifica **EBDC** (8 bit, 256 parole codificate) Usata per esempio nei nastri magnetici
- Codifiche **ISO-X** (rappresentano i sistemi di scrittura internazionali). P. es.: ISO-LATIN

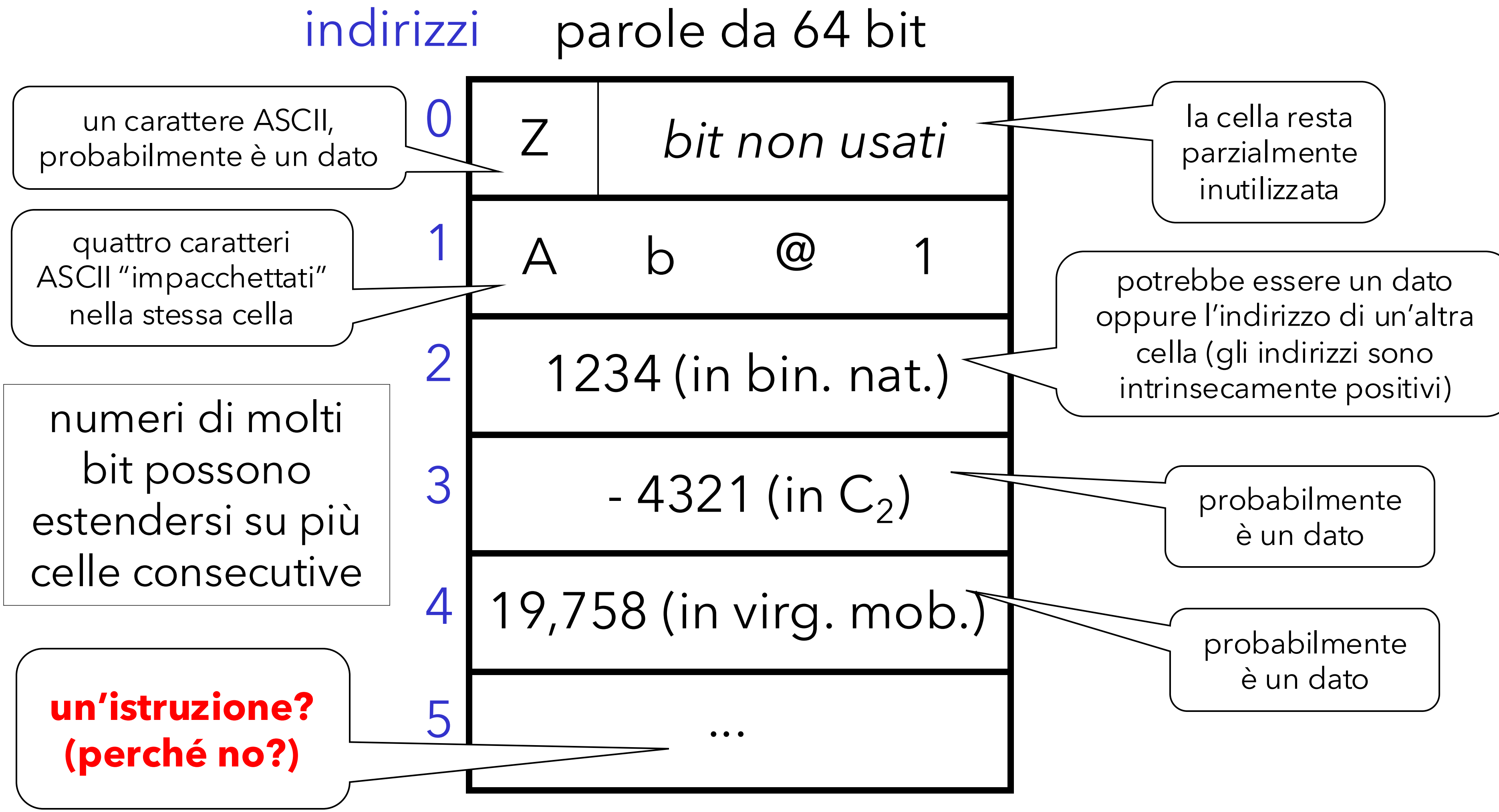
Codifica di testi, immagini, suoni

- Caratteri: sequenze di bit
  - Codice ASCII: utilizza 7(8) bit: 128(256) caratteri
  - 1 Byte (l’8° bit può essere usato per la *parità*)
- Testi: sequenze di caratteri (cioè di bit)
- Immagini: sequenze di bit
  - bitmap: sequenze di pixel (n bit, 2^n colori)
  - jpeg, gif, pcx, tiff, …
- Suoni (musica): sequenze di bit
  - wav, mid, mp3, ra, …
- Filmati: immagini + suoni
  - sequenze di …? … "rivoluzione" digitale

Dentro al calcolatore…informazione e memoria

- Una *parola di memoria* è in grado di contenere una *sequenza* di n ≥ 1 bit
- Di solito si ha: n = 8, 16, 32 o 64 bit
- Una parola di memoria può dunque contenere gli *elementi d'informazione* seguenti:
  - Un carattere (o anche più di uno)
  - Un numero intero in binario naturale o in C_2
  - Un numero frazionario in virgola mobile
  - Alcuni bit della parola possono essere non usati
- Lo stesso può dirsi dei registri della CPU

Esempio


indirizzi
parole da 64 bit
un carattere ASCII, probabilmente è un dato
0
Z
bit non usati
la cella resta parzialmente inutilizzata
quattro caratteri ASCII “impacchettati” nella stessa cella
1
A
b
@
1
potrebbe essere un dato oppure l’indirizzo di un’altra cella (gli indirizzi sono intrinsecamente positivi)
2
1234 (in bin. nat.)
numeri di molti bit possono estendersi su più celle consecutive
3
- 4321 (in C_2)
probabilmente è un dato
4
19,758 (in virg. mob.)
probabilmente è un dato
un’istruzione? (perché no?)
5
...

Recap

- Conversione decimale $\rightarrow$ binario = regola dei resti
- Conversione binario $\rightarrow$ decimale = metodo potenze
- M&S e Complemento a 2 (C_2) per rappresentare numeri negativi
- Numeri frazionari in virgola fissa e mobile
- Codifica dei caratteri (tabella ASCII)