Architettura degli Elaboratori I

Corso di Laurea Triennale in Informatica

Università degli Studi di Milano

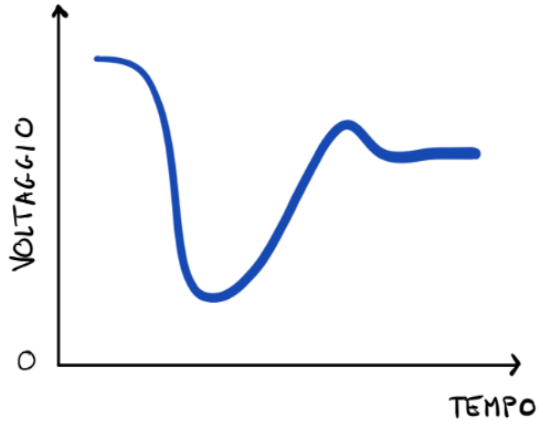
Dipartimento di Informatica “Giovanni Degli Antoni”

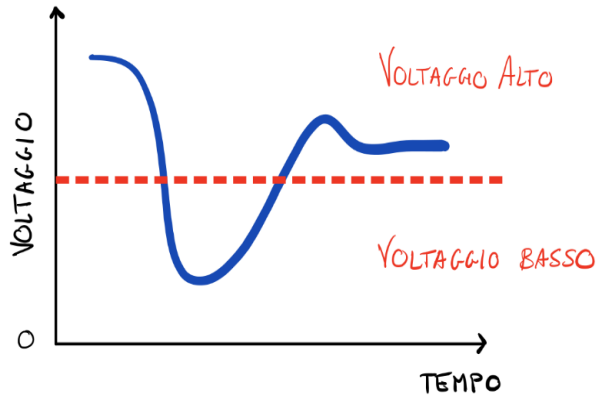
Edizione 2 (Cognomi H-Z), A.A. 2022-2023, Nicola.Basilico@unimi.it

Codifica dell’informazione

L’elaborazione digitale

- Per poter svolgere la sua funzione un elaboratore deve poter rappresentare l’informazione su cui lavora attraverso una grandezza fisica: **valori di tensione elettrica**
- Consideriamo un segnale di tensione che varia nel tempo
- Rappresenta un fenomeno fisico, ad esempio una misurazione di temperatura in una stanza, peso di un oggetto, vibrazione di una corda, …
- I valori che il segnale può assumere sono strettamente legati, o **in analogia**, con il fenomeno rappresentato. Per questo si chiama **segnale analogico o continuo**





- Mappiamo i valori del segnale su un insieme di n range di valori a cui associamo n simboli o cifre (le cifre sono simboli associati a valori numerici di base)
- Elaborazione digitale: rappresentare il fenomeno con un dato numero di cifre (in inglese *digits*)
- Si possono rappresentare sia misurazioni di fenomeni fisici (la temperatura) che altre informazioni più astratte (ad esempio, un punteggio)
- Esempio: $n = 2$, 2 range di valori, 2 simboli (alto/basso, $0/1$, vero/falso, A/B, …)

L’elaborazione digitale

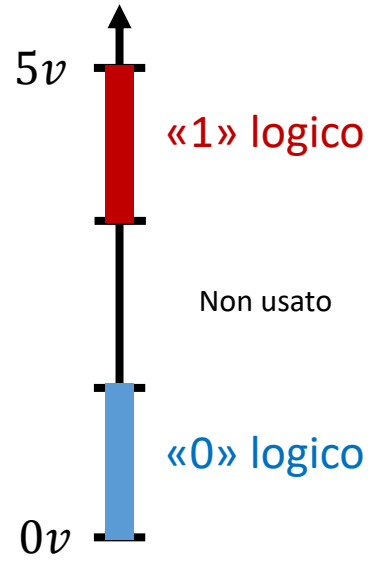
- Il tipo di informazione più importante trattata da un elaboratore è quella numerica

Supponiamo di dover rappresentare il valore diciotto

Analogico: il valore del segnale (voltaggio) rappresenta il valore 18

Digitale: il valore del segnale rappresenta una cifra del valore 18. Servono più segnali, uno per ogni cifra!

- I computer moderni utilizzano rappresentazioni binarie, con due simboli che, per convenzione, chiamiamo 0 e 1
- La memoria digitale è fatta da miliardi di componenti (basati su transistor) in grado di mantenere al loro interno un segnale che può rappresentare il simbolo 0 o il simbolo 1, le due cifre binarie (binary digits, **bit**)
- Come si costruisce la corrispondenza tra un valore (*diciotto*) e la sua rappresentazione digitale? Come si possono fare operazioni tra numeri rappresentati in quel modo?
- Le risposte stanno nella teoria dei **sistemi di numerazione**



Sistema di numerazione

```
There are only 10 types of people in the
world: those who understand binary,
and those who don't.
```

- Un sistema di numerazione è composto da due ingredienti fondamentali: la **base** e la **notazione**
- **Base:** insieme di simboli (cifre) che possiamo usare per rappresentare un numero; ogni simbolo è associato ad una quantità numerica elementare
- $B_{10} = \{0,1,2,3,4,5,6,7,8,9\}$ base 10, quelle che usiamo noi, sistema decimale
- $B_8 = \{0,1,2,3,4,5,6,7\}$, base 8, sistema ottale
- $B_{16} = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$, base 16, sistema esadecimale
- $B_2 = \{0,1\}$, base 2, sistema binario, quello usato dagli elaboratori
- Avendo a disposizione n cifre in base B ogni cifra può assumere B valori diversi e in totale si possono scrivere B^n stringhe diverse
- Esercizio: quanti numeri diversi posso scrivere con 4 cifre in base 16?
  - Risposta: $16^4 = 65.536$
- Esercizio: quante cifre della base B servono per poter scrivere almeno p stringe diverse?
  - Risposta: $\lceil \log_B p \rceil$

Sistema di numerazione: base

- Quali sono i valori di base associati ai simboli? Indico con $val(i)$ il valore associato al simbolo i della base
- Scegliamo dei simboli che ci aiutino a ricordare immediatamente il valore:
  - Di solito la cifra «i» è associata, in tutte le basi in cui compare al valore i in base 10. Ad esempio «9» rappresenta il valore 9 nelle basi decimale ed esadecimale
  - Nella base 16 le cifre non bastano: le lettere «A», «B», «C», «D», «E», «F» rappresentano i valori $10, 11, 12, 13, 14, 15$ (in base 10)
- **Attenzione**: conta solo il numero dei simboli e i loro valori associati, non i simboli scelti! Altre scelte possibili: $B_2 = \{\top, \bot\}$,$B_7 = \{I, V, X, L, C, D, M\}$, $B_3 = \{\triangle, \boxdot, \parallelogram\}$

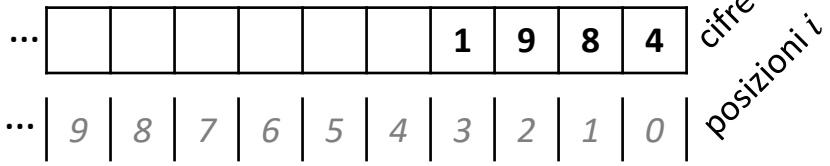
Esercizi

- Che aspetto ha un numero scritto in base 16?
- Risposta: $9F3A$ o in alternativa $0\text{x}9F3A$, $9F3A\ hex$
- Il numero 10011001 in quale base è scritto?
- Risposta: base 2, base 8, base 16 o base 10

Sistema di numerazione: notazione

- **Notazione**: regole con cui si calcola il valore rappresentato a partire dalla sequenza di simboli che lo rappresenta
- Una notazione fondamentale è quella **posizionale**: si fa la somma pesata dei valori associati ai simboli; i pesi dipendono dalla base utilizzata e dalla posizione del simbolo nella sequenza
- Le posizioni si indicano con valori interi ordinati, partendo da 0 e **da destra a sinistra**

Esempio con $B = 10, n = 4$: «1984»



$$val(4)B^0 + val(8)B^1 + val(9)B^2 + val(1)B^3$$
$$= 4 \times 10^0 + 8 \times 10^1 + 9 \times 10^2 + 1 \times 10^3$$
$$= 4 + 80 + 900 + 1000 = 1984$$

Valore rappresentato da una stringa di n cifre in base B

$$\sum_{i=0}^{n-1} val(d_i)B^i$$

- La cifra più a sinistra è detta Most Significant Digit (MSD), quella più a destra Least Significant Digit (LSD)

Esercizio: Quanto vale «10100» e quali sono il successivo e precedente nelle basi 10, 8 e 2?

- B_{10}: val. 10100, succ. «10101» (10101), prec. «10099» (10099)
- B_8: val. 4160, succ. «10101» (4161), prec. «10077» (4159)
- B_2: val. 20, succ. «10101» (21), prec. «10011» (19)

- A seconda del tipo di numeri che vogliamo rappresentare (naturali, interi, reali) useremo notazioni diverse
- La notazione impatta ovviamente anche su come vengono svolte le operazioni tra i numeri e quindi anche su come deve essere fatto l'hardware in grado di interpretare e manipolare i numeri rappresentati in un certo sistema di numerazione

Codifica dei Naturali $\mathbb{N}$

Numeri naturali $\mathbb{N} = \{1,2,3, \dots\}$

- I numeri naturali si scrivono in notazione posizionale pura (come nell'esempio precedente)
- **Da base B a base 10**: calcolo della somma pesata

$$(111011)_2 = \sum_{i=0}^{5} b_i 2^i = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 = (59)_{10}$$

- **Da base 10 a base B**: algoritmo iterativo delle divisioni

Dato $(N)_{10}$ da convertire nella base B:

1. dividere N per B (con una divisione intera);
2. il resto della divisione diventa la prima cifra meno significativa che resta da calcolare del numero in base B;
3. se il quoziente è 0 abbiamo finito;
4. se il quoziente è diverso da 0 si torna al passo 1 considerando il quoziente come dividendo N;

Numeri naturali $\mathbb{N} = \{1,2,3, \dots\}$

- **Esercizio**: convertire il numero $(113)_{10}$ nelle basi 2, 8 e 16

Base 2

$113 : 2 = 56 \; resto \; di \; 1$

$56 : 2 = 28 \; resto \; di \; 0$

$28 : 2 = 14 \; resto \; di \; 0$

$14 : 2 = 7 \; resto \; di \; 0$

$7 : 2 = 3 \; resto \; di \; 1$

$3 : 2 = 1 \; resto \; di \; 1$

$1 : 2 = 0 \; resto \; di \; 1$

Risultato: 1 1 1 0 0 0 1

Base 8

$113 : 8 = 14 \; resto \; di \; 1$

$14 : 8 = 1 \; resto \; di \; 6$

$1 : 8 = 0 \; resto \; di \; 1$

Risultato: 1 6 1

Base 16

$113 : 16 = 7 \; resto \; di \; 1$

$7 : 16 = 0 \; resto \; di \; 7$

Risultato: 7 1

Numeri naturali $\mathbb{N} = \{1,2,3, \dots\}$

Da base 2 a base 16 e *vice versa*

- È un caso particolare della conversione tra due basi B_1 e B_2 dove una è una potenza dell'altra, cioè $B_2 = B_1^m$ per un qualche m intero positivo
- In questo caso $B_1 = 2$, $B_2 = 16$ e quindi $m = 4$
- La cifra in posizione i di un numero in base B^m corrisponde all' i-esimo gruppo di m cifre del numero in base B
- Nel nostro caso abbiamo 16 possibili gruppi diversi di 4 cifre binarie, possiamo costruire una tabella che mette in corrispondenza ogni cifra esadecimale con una stringa di 4 bit

0	0000
1	0001
2	0010
3	0011

4	0100
5	0101
6	0110
7	0111

8	1000
9	1001
A	1010
B	1011

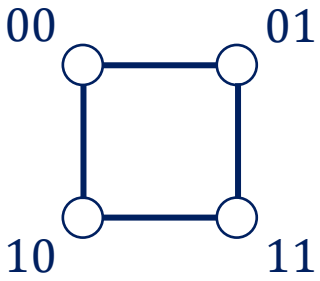
C	1100
D	1101
E	1110
F	1111

- Per le conversioni è sufficiente ispezionare la tabella e trovare le corrispondenze
  (nota: non dovrebbe servire impararla a memoria!)
- Esercizio: convertire $(A\ 8\ F\ B)_{16}$ in base 2
  Soluzione: Ispezionando, simbolo per simbolo, la tabella ottengo $(1010\ 1000\ 1111\ 1011)_2$

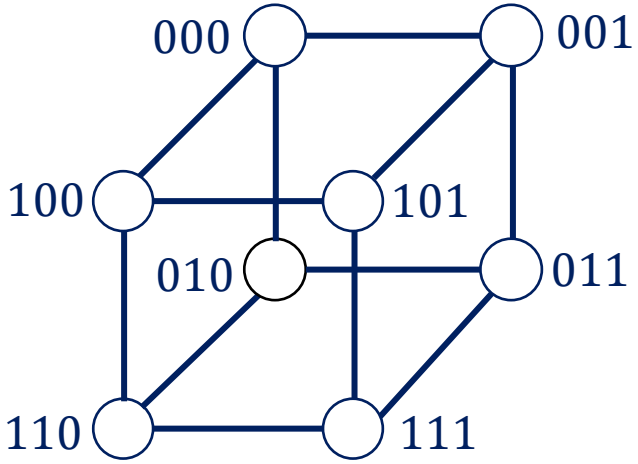
Rappresentazione grafica numeri binari

- Un numero binario su n bit può essere interpretato come un punto in uno spazio n-dimensionale

1 bit, una dimensione



2 bit, due dimensioni



3 bit, tre dimensioni

- In tutti i casi, numeri adiacenti (collegati) differiscono solo di un bit
- Dati due numeri binari n_1 e n_2 il numero di posizioni in cui un bit ha valore diverso tra un numero e l'altro si chiama **distanza di Hamming** tra n_1 e n_2
- Misura la distanza tra due codifiche, la differenza tra due rappresentazioni di due numeri che è diversa, in generale, dalla differenza tra i due valori rappresentati
- **Esercizio**: quanto è la distanza di Hamming tra 10111010 e 10010111?
- Risposta: se evidenzio i bit diversi ho $10\mathbf{1}1\mathbf{0}1\mathbf{0}$ e $10\mathbf{0}1\mathbf{0}1\mathbf{1}1$ quindi la distanza di Hamming è pari a 4 (numero di bit che devo complementare per trasformare un numero nell'altro)

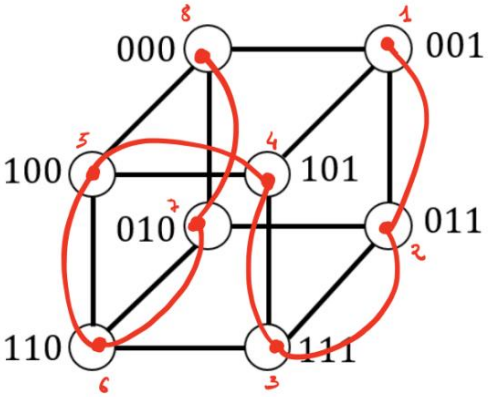
Codice di Grey

- Supponiamo di avere $n = 3$ bit
- Quanti numeri naturali possiamo scrivere in base $B = 2$? Risposta: B^n, in questo caso $2^3 = 8$

Numero decimale	Codifica binaria (posizionale)	Codifica binaria (Grey)
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

- Nella codifica posizionale, le distanze di Hamming tra codifiche di numeri successivi sono, nell'ordine, 1,2,1,3,1,2,1
- Nel codice di Grey le distanze sono **sempre pari a 1**!
- Svantaggio: più complicato calcolare il valore di un numero a partire dalla sua codifica
- Vantaggi? Esperimento degli interruttori…

- I codici che hanno questa proprietà sono detti codici a distanza unitaria. Come trovarli? Percorso Hamiltoniano sulla rappresentazione grafica (percorso che visita tutti i punti una volta sola percorrendo i collegamenti)



Operazioni aritmetiche

- Si usano le stesse regole della base 10 con somme e riporti (differenze e prestiti), che valgono indipendentemente dalla base utilizzata

	b_1	b_2	Somma	Riporto
	0	0	0	0
	0	1	1	0
	1	0	1	0
riporto	1	1	0	1
1	1	1	1	1

	b_1	b_2	Differenza	Prestito
	0	0	0	0
	0	1	1	1
	1	0	1	0
prestito	1	1	0	0
1	1	1	1	1

- Tutte le operazioni che producono un risultato a partire da uno o più operandi possono causare un **overflow**
- Vuol dire che il risultato dell'operazione è **troppo grande** per essere rappresentato su n bit, ne servono almeno $n + 1$
- Quale è il numero **naturale** più grande che posso rappresentare con n bit? Risposta: $2^n - 1$
- **Esercizio**: sommare 0101 e 1011 su 4 bit

$$\begin{array}{r} 1\,1\,1\,1 \\ 0\,1\,0\,1 + \\ 1\,0\,1\,1 = \\ \hline 1\,0\,0\,0\,0 \end{array}$$

- I due operandi valgono 5 e 11, la somma dovrebbe essere 16, ma se mi limito a lavorare con 4 bit leggo 0!
- Devo usare un bit in più! Sulla carta si può sempre fare, ma in hardware è impossibile: si lavora con n bit fissati, se non bastano si deve segnalare un errore!

Il problema dell'overflow

Base europea di Kourou (Guyana Francese)
Primo lancio dell'Ariane 5 (ESA), 4 Giugno 1996 (video)

- Dopo 39s dal lancio il sistema di navigazione inerziale misura un elevato ma normale valore di velocità orizzontale
- Il software converte il dato letto in binario su 16 bit, è un'eredità di Ariane 4, il razzo precedente **più lento**
- Il valore della velocità è troppo grande per essere rappresentato su 16 bit, si genera un **overflow**
- Il sistema inerziale va in crash, il controllo passa al sistema inerziale di backup che, dopo pochi millisecondi, va in crash per lo stesso motivo
- L'unità inerziale a questo punto è fuori controllo, trasmette dati completamente errati
- Il computer di bordo, interpreta le stringhe di bit come dati di volo e aziona una manovra correttiva non necessaria flettendo completamente gli ugelli del motore principale
- Il missile guadagna un angolo di attacco di 20 gradi e subisce un carico aerodinamico insostenibile: si disintegra completamente
(perdita stimata 370M USD)

Codifica degli Interi $\mathbb{Z}$

Numeri interi $\mathbb{Z} = \{\ldots -3, -2, 0, 1, 2, 3, \ldots\}$

- I numeri naturali sono sufficienti se dobbiamo solo contare (o ordinare), ma utilizzare solo quelli sarebbe fortemente limitante (ad esempio, non potremmo svolgere differenze arbitrarie tra numeri naturali)
- Consideriamo i numeri **interi** che presentano una caratteristica in più: **il segno**, che dobbiamo rappresentare in qualche modo
- Ci sono diverse soluzioni che presentano vantaggi e svantaggi per un elaboratore, ne vediamo due:
  - Modulo e segno (molto semplice, ma non molto vantaggiosa)
  - Complemento a 2 (più complicata, ma vantaggiosa e quindi molto usata)

Modulo e segno

- Supponiamo di avere a disposizione n bit

MSD indica il segno:
- 0 per positivo $(+)$
- 1 per negativo $(-)$

Restanti $n-1$ bit indicano il modulo con la stessa notazione dei numeri naturali (spesso chiamata *unsigned*)

- **Esempio**: il numero 10011, se interpretato come naturale vale 19, se invece lo interpretiamo come un intero in notazione modulo e segno vale -3
- Per un essere umano è il metodo più naturale: è una rappresentazione che ricalca il modo in cui noi pensiamo i numeri ma …
- … per un elaboratore è piuttosto inefficiente!
  - Ridondanza: lo zero ha due codifiche $+0$ e -0
  - Complessità: certe operazioni risultano laboriose, ad esempio la somma algebrica richiede di (a) controllare il segno degli operandi, (b) sottrarre il maggiore al minore se i segni sono diversi o (c) sommare i valori se i segni sono uguali e (d) calcolare il segno del risultato. Queste operazioni non possono essere svolte contemporaneamente, c'è una sequenzialità, e il circuito che le realizza diventa complesso

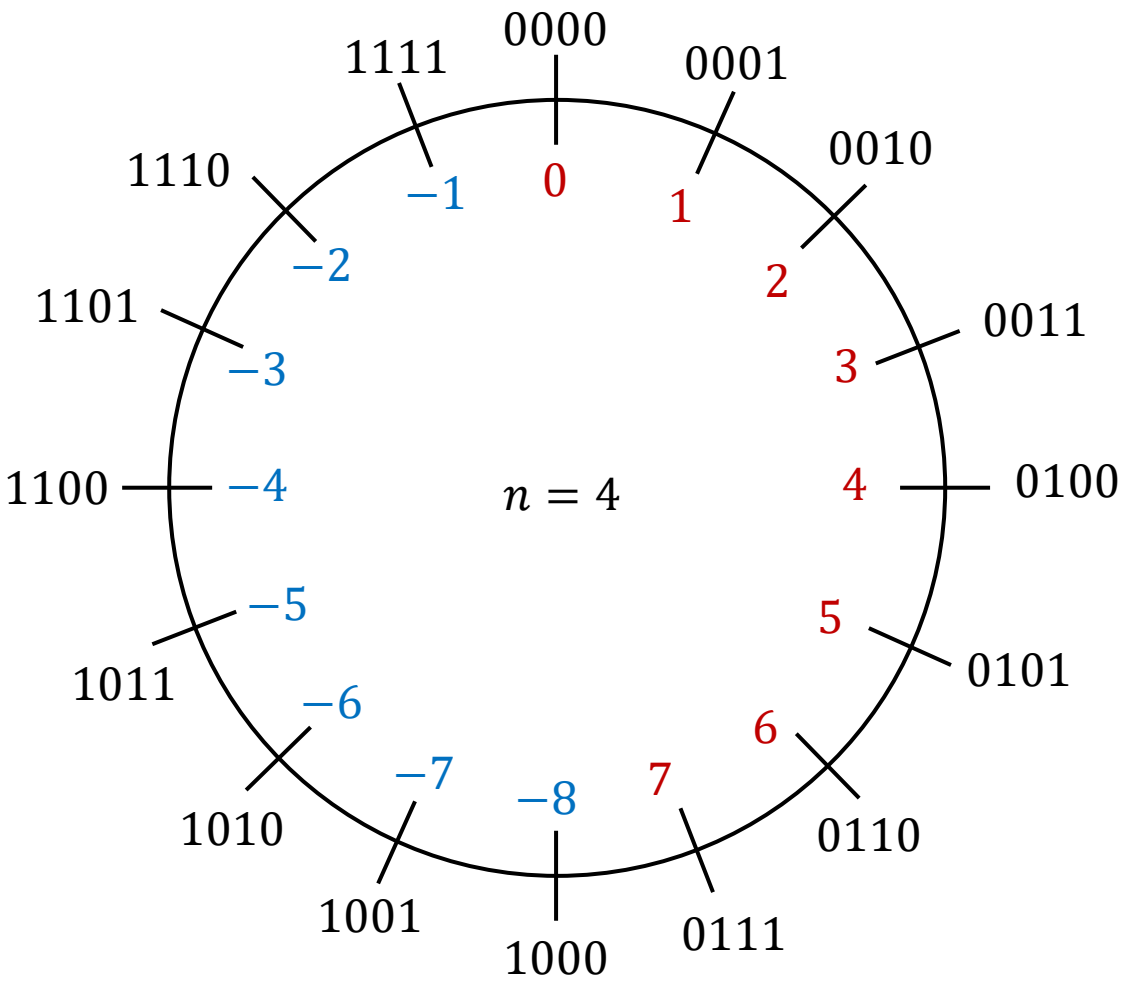
Complemento a 2

- Esiste un modo di rappresentare i numeri interi che ci permetta di fare la somma algebrica in modo semplice, analogamente a quanto succedeva con i naturali (così da avere, come per $\mathbb{N}$, hardware semplice)? Sì, il **complemento a 2** (C2)
- Supponiamo, come prima, di avere a disposizione n bit e un numero intero N da codificare
- Se N è positivo o nullo lo codifico come il naturale di valore N (come prima!) su $n-1$ bit e pongo MSD a 0
- Se N è negativo lo codifico come il naturale $2^n - |N|$ su n bit cioè il valore che mancherebbe a $|N|$ per arrivare a 2^n (il suo complemento a 2^n)
- **Attenzione!** In entrambi i casi devo stare molto attento al **problema della rappresentabilità**!

Complemento a 2

- **Esercizi**: suppongo di avere $n = 4$ bit
- $N = 5$ → converto 5 su 3 bit (metodo delle divisioni iterative) e aggiungo uno 0 a sinistra → 0101
- $N = -5$ → converto $2^4 - 5 = 16 - 5 = 11$ su 4 bit → 1011
- $N = 8$ → converto 8 su 3 bit → **overflow**!
  I 4 bit non bastano ne servono almeno 5
- $N = -1$ → converto $2^4 - 1 = 15$ su 4 bit → 1111
- $N = -8$ → converto $2^4 - 8 = 8$ su 4 bit → 1000
- $N = -11$ → converto $2^4 - 11 = 5$ su 4 bit → $\mathbf{0101}$
  ma quindi è 5 o -11? Attenzione all'intervallo rappresentabilità!

Complemento a 2



- Con n bit posso scrivere 2^n stringhe binarie, le rappresento su una ruota in ordine orario crescente secondo il valore che avrebbero se fossero naturali
- Per i positivi devo usare $n-1$ bit e aggiungere 0 a sinistra:
  1. l'N più grande è $2^{n-1}-1$
  2. Tutti i positivi e lo zero iniziano con 0
- Le stringhe restanti sono assegnate ai negativi che si dispongono «all'inverso» seguendo la regola del complemento: il numero $-N$ è assegnato alla stringa che avrebbe valore $2^n - N$ nei naturali
  1. l'N più piccolo è -2^{n-1}
  2. Tutti i negativi iniziano con 1

- Chi sta fuori dall'intervallo $[-2^{n-1}, 2^{n-1}-1]$ **non può essere rappresentato su n bit in complemento a 2!**

- Prima buona proprietà: lo zero ha una sola codifica!

Complemento a 2

- Metodo alternativo per convertire $-N$ in C2 su n bit:
  1. Verifico la rappresentabilità su n bit (va sempre fatto!)
  2. Converto N in binario
  3. Faccio il complemento a 1 (inverto tutti i bit)
  4. Sommo 1 in binario (regole dei naturali)

Cambio di segno!

- **Esercizio:** convertire -6 su 3 bit
  1. Il numero più piccolo su 3 bit è $-2^2 = -4$, non si può fare!
- **Esercizio:** convertire -6 su 4 bit
  1. Il numero più piccolo su 4 bit è $-2^3 = -8$, ok!
  2. 6 in binario su 4 bit è 0110
  3. Complemento a 1: 1001
  4. Sommo 1:

$$\begin{array}{r} {\scriptstyle 1} \\ 1\,0\,0\,1\,+ \\ 0\,0\,0\,1\,= \\ \hline 1\,0\,1\,0 \end{array} \longleftarrow \textbf{risultato}$$

- Per convertire **da C2 a base 10** basta usare la regola posizionale dando al bit più significativo un peso negativo

$$(111011)_{C2} = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$
$$= -1 \times 2^5 + 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4$$
$$= (-5)_{10}$$

- Il bit più significativo è un bit di segno a tutti gli effetti: può essere esteso senza modificare il valore assoluto: $(1111111111111111011)_{C2}$ è sempre $(-5)_{10}$
- Altra buona proprietà: le somme algebriche si fanno con lo stesso procedimento dei naturali ignorando l'ultimo riporto

Esercizio: eseguire $4 - 5$ (su 4 bit)

$(4)_{10} = (0100), (-5)_{10} = (1011)$, faccio $4 + (-5)$ ⟶

$$\begin{array}{r} 0\,1\,0\,0\,+ \\ 1\,0\,1\,1\,= \\ \hline 1\,1\,1\,1 \end{array}$$

Esercizio: eseguire $7 - 1$ (su 4 bit)

$(7)_{10} = (0111)_{C2}, (-1)_{10} = (1111)_{C2}$, faccio $7 + (-1)$ ⟶

$$\begin{array}{r} {\scriptstyle 1\,1\,1\,1} \\ 0\,1\,1\,1\,+ \\ 1\,1\,1\,1\,= \\ \hline (1)\,0\,1\,1\,0 \end{array}$$

Complemento a 2

- Il problema dell'overflow si ripropone anche per le somme in C2
- Il risultato di una somma di due numeri in C2 su n bit potrebbe cadere fuori dall'intervallo di rappresentabilità $[-2^{n-1}, 2^{n-1}-1]$

$$\begin{array}{r} {\scriptstyle 111} \\ 00110010 + \\ 01010000 = \\ \hline 10000010 \end{array} \begin{array}{l} (80)_{10} \\ (50)_{10} \end{array}$$

Mi aspetto $(130)_{10}$ ma il risultato in C2 rappresenta $(-126)_{10}$

Il numero più grande su 8 bit è 127!

$$\begin{array}{r} {\scriptstyle 1} \\ 10000000 + \\ \hline 11111111 = \end{array} \begin{array}{l} (-128)_{10} \\ (-1)_{10} \end{array}$$

$(1)\ 01111111$

Mi aspetto $(-129)_{10}$ ma il risultato in C2 rappresenta $(127)_{10}$

Il numero più piccolo su 8 bit è -128!

- Può succedere solo quando si sommano numeri dello stesso segno
- Si riconosce facilmente:
  1. Sommo due numeri positivi (bit di segno 0) e ho un risultato negativo (bit di segno 1)
  2. Sommo due numeri negativi (bit di segno 1) e ho un risultato positivo (bit di segno 0)
- Alternativa: controllare gli ultimi due riporti generati, se sono diversi c'è stato overflow

Codifica dei Reali $\mathbb{R}$

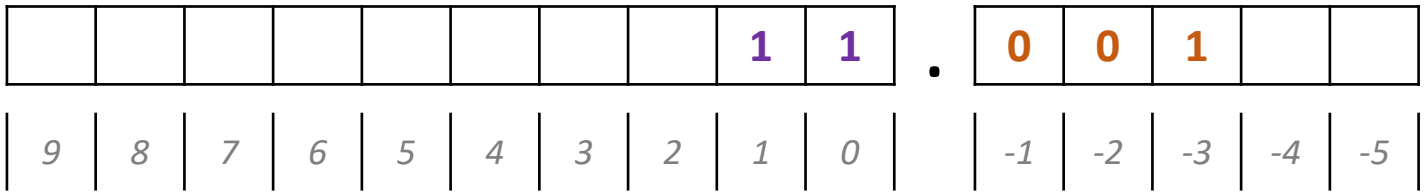
Rappresentazione dei numeri reali $\mathbb{R}$

- Sono i numeri «veri e propri» quelli che descrivono fenomeni del mondo, esempi: $0.45, \pi, 2.718281828459045235 36, 6.626 \times 10^{-34}$
- Come li rappresentiamo su n bit?
- Presentano due fondamentali differenze con i naturali e gli interi:
  - Non li usiamo per contare ma per «misurare», un processo che di solito richiede di poter rappresentare numeri piccolissimi (vicini allo zero, e.g., la massa atomica dell'ossigeno $\cong 2.6 \times 10^{-23}$g) e numeri molto grandi (e.g., diametro dell'universo osservabile $\cong 8.8 \times 10^{23}$km)
  - A differenza di naturali e degli interi non sono enumerabili, non si possono contare: tra due reali qualsiasi ci sono infiniti reali
- **Non possiamo rappresentare i numeri reali**, possiamo solo approssimarli con dei numeri **razionali** a precisione finita, quindi anch'essi approssimazioni di $\mathbb{Q} \subset \mathbb{R}$
- I numeri razionali sono il risultato di una divisione tra interi, lo sviluppo decimale è infinito ma periodico (e.g., $0.5000 \ldots, 1.3333 \ldots, 0.285714285714 \ldots$)

Possiamo rappresentare solo questi!

Rappresentazione dei numeri reali $\mathbb{R}$

- Primo problema: come estendiamo la notazione posizionale per poter rappresentare una frazione di un numero?
- Introduco la virgola e associo alle posizioni alla sua destra indici negativi, la formula della somma pesata si estende naturalmente



$$(11.001)_2 = \sum_{i=0}^{2} b_i 2^i + \sum_{i=-3}^{-1} b_i 2^i = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = (3.125)_{10}$$

- Con questo procedimento posso convertire **da base 2 a base 10** un numero con parte frazionaria
- Assumiamo per semplicità che le parti intere siano sempre naturali, anche se possiamo facilmente generalizzare agli interi pensandole codificate in C2

Rappresentazione dei numeri reali $\mathbb{R}$

- Da base 10 a base 2: algoritmo iterativo delle moltiplicazioni (il «duale» di quello delle divisioni)

Dato $(I.F)_{10}$ da convertire nella base $B = 2$:

1. I si converte in binario naturale;
2. Moltiplicare $.F$ per 2
3. La parte intera del risultato diventa la prima cifra più significativa che resta da calcolare;
4. Tornare a 2 considerando la parte frazionaria del risultato al posto di $.F$;

Si termina quando:

1. la parte frazionaria del risultato è 0
   - In questo caso il numero frazionario può essere rappresentato con un numero finito di cifre senza perdita di approssimazione! Succede solo ai numeri del tipo $p(2^{-q})$ con $p, q \in \mathbb{N}$, ad esempio $7(2^{-2}) = \frac{7}{4} = 1.75$
2. Abbiamo finito i bit: troncamento o arrotondamento

Esercizio convertire in binario $(4.4375)_{10}$
Parte intera 100, parte frazionaria:

$.4375 \times 2 = .875$ parte intera 0
$.875 \times 2 = 1.75$ parte intera 1
$.75 \times 2 = 1.5$ parte intera 1
$.5 \times 2 = 1.0$ parte intera 1

Risultato: $(100.0111)_2$

Esercizio convertire in binario $(10.76)_{10}$
parte intera 1010, parte frazionaria:

$.76 \times 2 = 1.52$ parte intera 1
$.52 \times 2 = 1.04$ parte intera 1
$.04 \times 2 = .08$ parte intera 0
$.08 \times 2 = .16$ parte intera 0
$.16 \times 2 = .32$ parte intera 0
⋮

Risultato: $(1010.11000\ldots)_2$

Troncamento e arrotondamento

- Alcuni numeri frazionari richiedono un numero di cifre dopo la virgola molto alto, in certi casi anche infinito. Noi però abbiamo sempre a disposizione un numero finito e limitato di bit
- **Troncamento**: genero cifre fino a quando esaurisco i bit e lascio così
- Esempio di prima: $(10.76)_{10}$ → $(1010.11000010100\ldots)_2$ se in totale avessi 6 bit → 1010.11~~$000010100\ldots$~~
- Esempio in base 10: $\pi = 3.14159265358979323846264338327950288419716939\ldots$ → 3.1415
- Il troncamento è sempre un arrotondamento **verso lo zero** (per difetto sui positivi, per eccesso sui negativi)

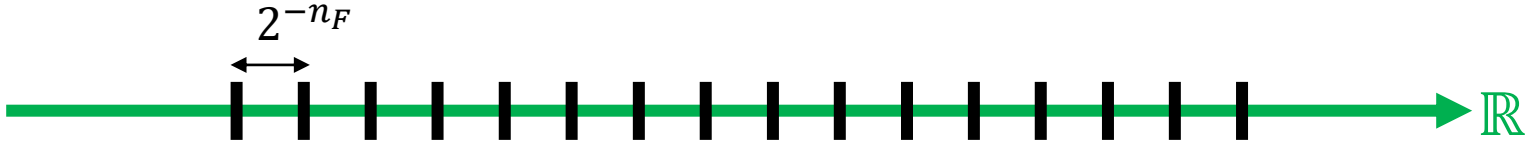
- **Arrotondamento:** scarto le cifre come nel troncamento, ma scelgo se arrotondare per eccesso o per difetto cercando di minimizzare l'errore di approssimazione
- **Esercizio** arrotondare $(101.110\mathbf{1}1010100)_2$: per eccesso $101.110 + 000.001 = 101.111$
- **Esercizio** arrotondare $(0.00101\mathbf{0}011010100)_2$: per difetto 0.00101 (come troncamento)

Rappresentazione dei numeri reali $\mathbb{R}$

- Abbiamo visto come funziona la notazione posizionale in base 2 per i numeri con parte frazionaria, ma per avere un sistema di numerazione non basta: dobbiamo prendere una decisione fondamentale su come organizzare gli n bit di cui disponiamo
- Come ripartisco gli n bit tra parte intera e frazionaria? ($n = n_I + n_F$)
- La risposta a questa domanda ha implicazioni forti sulla rappresentazione dei reali e sta alla base dei due metodi principali che vediamo:
  - Rappresentazione in **virgola fissa**
  - Rappresentazione in **virgola mobile**

Virgola fissa

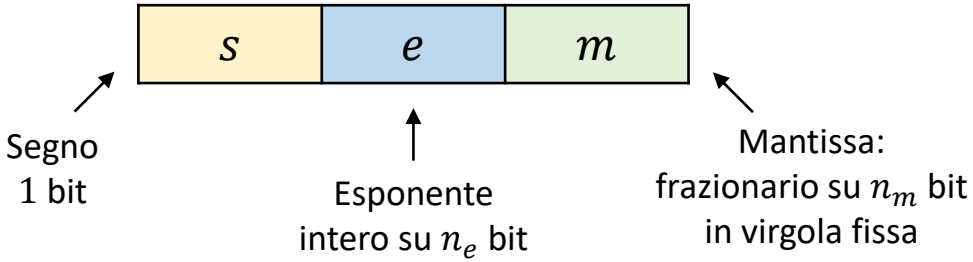
- Assegno un numero di bit n_I alla parte intera, i restanti $n - n_I = n_F$ a quella frazionaria e mantengo questa ripartizione per sempre
- Nel nostro primo esempio su $n = 5$ bit con il numero 11.001 abbiamo usato $n_I = 2$ bit per la parte intera e $n_F = 3$ bit per la parte frazionaria
- La virgola non cambia mai posizione. Ha sempre n_I cifre alla sua sinistra e n_F alla sua destra, essendo implicita può essere omessa (non uso bit per indicare la sua posizione nel numero)
- **Domanda**: quale è il numero massimo rappresentabile? Risposta: $2^{n_I} - 1 + \sim 1 \cong 2^{n_I}$
  nell'esempio sopra è $(11.111)_{C2} = (3.875)_{10}$, cioè quasi 4 ($\cong 4$)
- **Domanda**: quale è il numero più vicino allo 0 rappresentabile?
- Risposta: 2^{-n_F}
- 2^{-n_F} è il contributo più piccolo possibile dato da un bit nella nostra codifica, è anche la differenza di valore tra due rappresentazioni numeriche successive (due tacche): viene anche chiamato **precisione**



- Più cifre dedichiamo alla parte frazionaria, più piccola è la precisione e più fitte sono le tacche: otteniamo una approssimazione migliore

Virgola mobile

- Nella rappresentazione in virgola mobile la virgola **non ha un posizione prefissata**, ci permette di rappresentare nella stessa codifica i numeri: 11.011 e -0.0001111; queste due scritture non possono coesistere nello stesso codice a virgola fissa!
- La posizione della virgola non è più implicita, dobbiamo «consumare» bit per dire dove sta
- Dati n bit, la notazione del numero è suddivisa in tre campi:
  - 1 bit per il segno s del numero (0 per dire positivo, 1 per dire negativo)
  - n_e bit che codificano un numero intero con segno detto esponente e
  - n_m bit che codificano un numero frazionario positivo in virgola fissa detto mantissa m



In totale $1 + n_e + n_m = n$

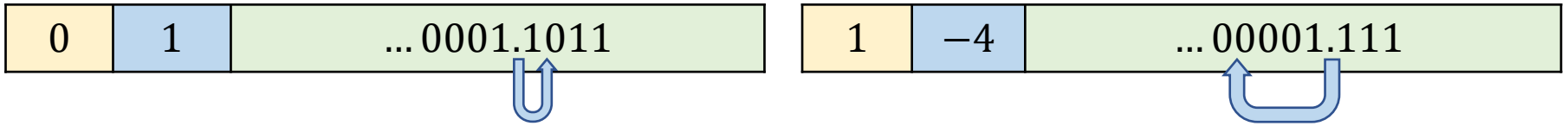
Come si calcola il valore che questi tre campi rappresentano?

Con questa formula: $(-1^{s}) \times \boldsymbol{m} \times 2^{e}$

L'esponente fa muovere la virgola!

Questa notazione è anche detta «scientifica»

I due esempi precedenti (esponente scritto in base 10):



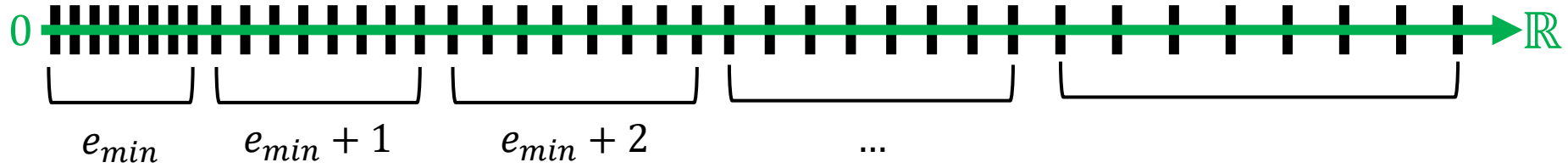
Virgola mobile

- Per semplicità e convenienza di notazione introduciamo la **forma normalizzata**
- Il numero in virgola mobile è normalizzato se **la parte intera della mantissa ha una sola cifra significativa**
- In base 2 implica che la mantissa è sempre fatta così $1.10110\ldots$, significa anche che $1.$ è implicito
- Rende più semplici alcune operazioni, come i confronti:
- **Domanda**: chi è il maggiore tra 1101×2^{-1} e 10.11×2^{1} (non normalizzati)
- **Domanda** riformulata: chi è il maggiore tra 1.101×2^{2} e 1.011×2^{2} (normalizzati)
- Nel secondo caso è più facile rispondere! Basta confrontare gli esponenti (ordini di grandezza) e, se sono uguali, si confrontano le mantisse

Virgola mobile

- Facciamo un confronto tra virgola fissa e mobile, per semplicità consideriamo la base 10 (lo stesso vale per una base B generica)
- Supponiamo di avere a disposizione $n = 6$ cifre decimali e un bit di segno che trascuriamo
- Per la virgola fissa assegniamo 3 cifre alla parte intera e 3 a quella frazionaria es: 123.447, 003.012, etc. …
- Per la virgola mobile assegniamo 4 cifre alla mantissa (di cui una per la parte intera) e 2 all'esponente con segno, es 1.122×10^{-21}, 0.043×10^{04}, etc. …
- **Numero massimo?** Virgola fissa: $999.999 \cong 10^3$, virgola mobile $9.999 \times 10^{99} \cong 10^{100}$, oltre: **overflow**
- **Numero più vicino allo zero?** Virgola fissa: $000.001 = 10^{-3}$, virgola mobile $0.001 \times 10^{-99} = 10^{-102}$, più vicino allo zero: **underflow**
- La rappresentazione in virgola mobile, a differenza di quella in virgola fissa, ci consente di scrivere numeri molto grandi e molto piccoli! Ma come è possibile se abbiamo solo 6 cifre e quindi il numero di numeri diversi (le tacche) è lo stesso*?
- La differenza sta in **come le tacche sono distribuite** sulla linea!
- Virgola fissa: precisione costante

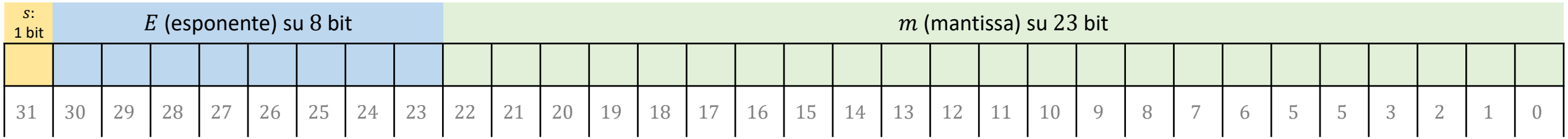
- Virgola mobile: precisione variabile (con l'esponente)



Lo standard IEEE 754

- L'implementazione della rappresentazione in virgola mobile dentro i calcolatori moderni è regolata da uno standard: l'**IEEE-SA Standard n. 754 for Floating-Point Arithmetic** (in breve, IEEE 754)
- Formato a precisione singola (detto «float») su 32 bit (quello che vediamo)
- Formato a precisione doppia (detto «double») su 64 bit

- **Regola**: in precisione singola un numero in virgola mobile è fatto così



- Questi 32 bit possono rappresentare:
  1. Numeri in virgola mobile normalizzati
  2. Numeri in virgola mobile non normalizzati (detti sub-normalizzati o de-normalizzati)
  3. Codici speciali

Lo standard IEEE 754: Numeri normalizzati

- Se $0 < E < 255$ (binario naturale) allora i 32 bit stanno codificando un numero normalizzato. Di conseguenza dobbiamo interpretare i 3 campi in questo modo:
- s è il segno del numero (0 per dire positivo, 1 per dire negativo)
- m è la parte frazionaria del numero assunto in forma normalizzata: $(1.m)_2$, dove 1. è implicito (ricordiamoci che siamo in base 2!)
- E è il valore dell'esponente **a cui è stato sommato 127**, si dice «in eccesso» 127, quindi il vero esponente è $e = E - 127$
- Il numero rappresentato è $(-1)^s \times 1.m \times 2^e$

- **Esercizio**: cosa rappresenta il seguente numero in formato IEEE 754 a precisione singola?

s: 1 bit	E (esponente) su 8 bit								m (mantissa) su 23 bit																					
1	1	0	0	0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	5	3	2	1	0

- Risposta:
  - $E = 1 \times 2^7 + 1 \times 2^0 = 128 + 1 = 129$ quindi siamo di fronte ad un numero normalizzato
  - Esponente $e = E - 127 = 2$, mantissa 1.01100
  - Spostamento della virgola $1.01100 \times 2^2 = 101.1$, in base 10: $1 \times 2^2 + 1 \times 2^0 + 1 \times 2^{-1} = 4 + 1 + \frac{1}{2} = \mathbf{5.5}$

Lo standard IEEE 754: Numeri normalizzati

- **Esercizio**: rappresentare il valore 3.25 in formato IEEE 754 a precisione singola
- Converto in binario la parte intera 11 e la parte frazionaria $.01$
- Normalizzo $11.01 = 1.101 \times 2^1$
- $s = 0, E = 1 + 127 = 128 = (10000000)_2, m = 101000$ …

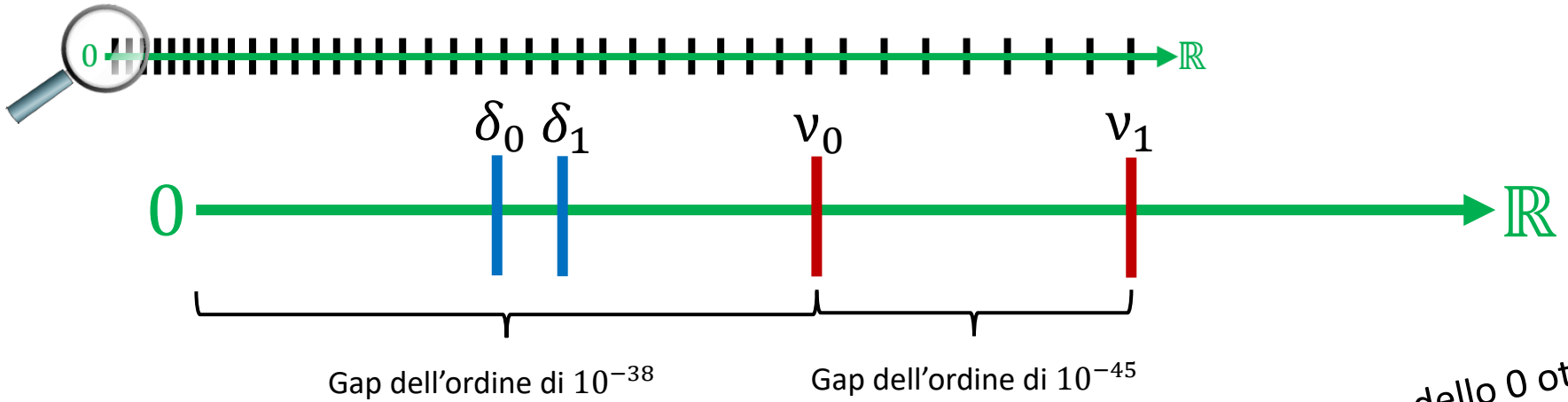
s: 1 bit	E (esponente) su 8 bit								m (mantissa) su 23 bit																						
0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	5	3	2	1	0

Lo standard IEEE 754: Numeri de-normalizzati

- Se $E = 0$ e $m \neq 0$ allora i 32 bit stanno codificando un numero **sub-normalizzato**. Di conseguenza dobbiamo interpretare i 3 campi in questo modo:
- s è il segno del numero (0 per dire positivo, 1 per dire negativo)
- m è la parte frazionaria del numero la cui parte intera è assunta essere 0: $(0.m)_2$, dove $0.$ è implicito
- E **viene scartato** e si assume $e = -126$ (**attenzione!** Non -127)
- Il numero rappresentato è $(-1)^s \times 0.m \times 2^{-126}$
- I numeri de-normalizzati ci aiutano a infoltire la rappresentazione nelle vicinanze dello 0, ma non solo …

Lo standard IEEE 754: risoluzione

- **Esercizio**: numero normalizzato più vicino allo 0?
- Risposta: $\nu_0 = 1.000\ldots \times 2^{-126} = 2^{-126} \cong 1.17 \times 10^{-38}$
- **Esercizio**: chi è il successivo?
- Risposta: $\nu_1 = 1.00\ldots001 \times 2^{-126} = (1 + 2^{-23}) \times 2^{-126} = 2^{-126} + 2^{-149} \cong 1.17 \times 10^{-38} + 1.4 \times 10^{-45}$



- **Esercizio**: numero de-normalizzato più vicino allo 0?
- Risposta: $\delta_0 = 0.000\ldots1 \times 2^{-126} = 2^{-149} \cong 1.4 \times 10^{-45}$
- **Esercizio**: chi è il successivo?
- Risposta: $\delta_1 = 0.00\ldots010 \times 2^{-126} = 2 \times \delta_0 = 2^{-148} \cong 2.8 \times 10^{-45}$

Nelle vicinanze dello 0 otteniamo una risoluzione maggiore e localmente costante!

Lo standard IEEE 754: codici speciali

Valore di E	Valore di m	Cosa rappresentano i 32 bit in IEEE 754?
$0 < E < 255$	qualsiasi	Numero normalizzato
$E = 0$	$m \neq 0$	Numero de-normalizzato

$E = 0$	$m = 0$	± 0 (a seconda di s)
$E = 255$	$m = 0$	$\pm \infty$ (a seconda di s)
$E = 255$	$m \neq 0$	NaN («Not a Number»)

- Due codifiche per lo 0! (ma con i reali potrebbe non essere ridonante…)
- NaN è un simbolo che indica il risultato di una operazione non permessa, ad esempio $\frac{12.45}{0}, \sqrt{-9}, \infty - \infty, \log(-5), \frac{0}{0} \ldots$

Lo standard IEEE 754: considerazioni

- Perché quel formato? Perché sommare 127 al vero valore dell'esponente?
- Riduce la complessità dell'hardware che deve manipolare questi numeri, facilitando alcune operazioni frequenti
- **Esempio**: confronto fra due numeri in IEEE 754, chi è il maggiore?
  - Se i segni sono diversi è immediato (il maggiore è quello con $s = 0$)
  - Se i segni sono uguali basta confrontare i restanti bit come si faceva con i naturali! L'esponente, in posizione più alta, domina i bit della mantissa e il segno non va gestito perché non è codificato esplicitamente (eccesso 127)
- Per un numero generico di bit n, l'eccesso si calcola come $K = 2^{n-1} - 1$, l'eccesso-*K* è un altro modo di rappresentare numeri interi
- In generale le operazioni in virgola mobile richiedono hardware più complesso delle corrispettive svolte su numeri naturali e interi. Sono però anche le più frequenti che ci serve svolgere, visto che molti fenomeni del mondo sono descritti da numeri reali (che noi approssimiamo)
- Negli elaboratori di solito c'è una unità dedicata a queste operazioni (floating-point unit, o co-processore in virgola mobile) e il numero di operazioni in virgola mobile che una CPU può svolgere in un intervallo di tempo è una metrica della sua **potenza di calcolo**: **FLOPS** (Floating-point Operations per Second)

Considerazioni finali

- Abbiamo visto modi di rappresentare l'informazione numeri utilizzando simboli binari, 1 e 0, che gli elaboratori sanno rappresentare e manipolare in hardware
- In diversi linguaggi di programmazione, il programmatore ha accesso alla scelta della rappresentazione da usare mediante la specifica del tipo.
- Ad es. in C `int` significa di solito 32 bit in C2, `unsigned int` 32 bit in naturale, `float` IEEE 754 precisione singola, `double` IEEE 754 precisione doppia. In Go è analogo, in JavaScript i «Number» sono IEEE 754 in doppia precisione