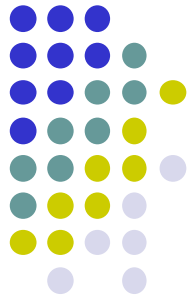
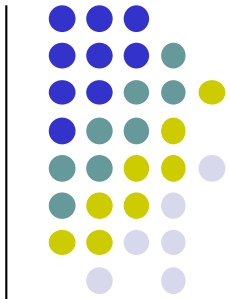
Codifica numeri interi (positivi e negativi)



- Esistono diverse codifiche
- Tra le più note:
  - rappresentazione in modulo e segno
    - più semplice e diretta
  - rappresentazione in complemento a *2*
    - ha il pregio di poter effettuare somme algebriche, ossia la sottrazione *a-b* equivale ad effettuare la somma *a+(-b)*
    - *a* livello di circuteria elettronica ciò consente di poter effettuare somme e sottrazioni in modo unificato tramite un unico dispositivo sommatore che opera sulle codifiche di *a* e *-b*
  - *… (tra un po' di slides ci ritorniamo)…*



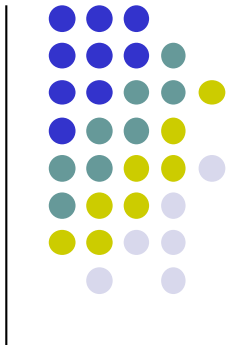
- Rappresentazione in modulo e segno a *k* bit:
  - *1* bit per il segno, solitamente il più significativo posto ad *0* per indicare il segno + e ad *1* per indicare il segno –
  - *k-1* bit per il modulo o valore assoluto, secondo la codifica dei numeri naturali

- L'intervallo dei numeri rappresentabili quindi è

$$[-(2^{k-1}-1),\ 2^{k-1}-1]$$

- Si noti che esistono due codifiche possibili per il numero *0*, ossia *00…0* (corrispondente a *+0*) e *10…0* (corrispondente a *-0*)

Esempio



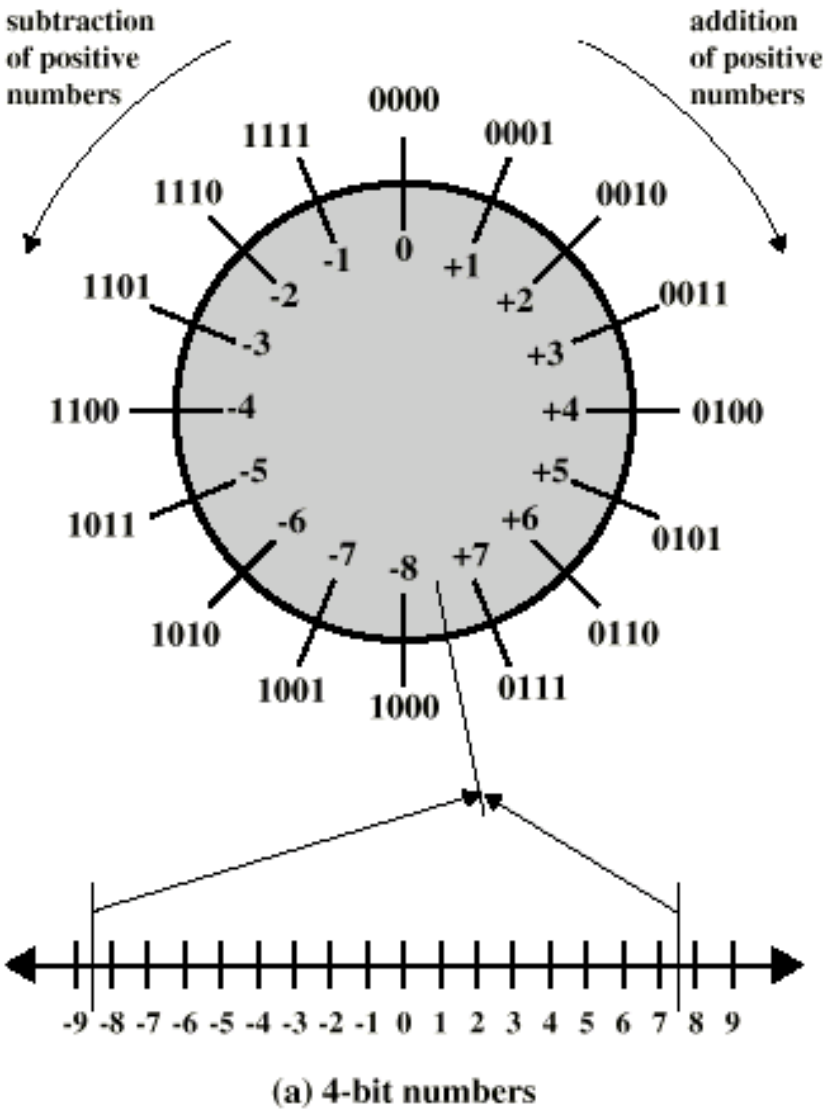
Rappresentazione in modulo e segno a *k=8* bit dei numeri *26* e *-26*

26:2 = 13 con resto 0
13:2 = 6 con resto 1
6:2 = 3 con resto 0
3:2 = 1 con resto 1
*1:2 = **0** con resto 1*

Quindi $26 = 11010_2$, per cui

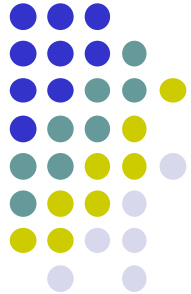
1. rappresentazione di *26*: *00011010*
2. rappresentazione di *-26*: *10011010*

- Rappresentazione in complemento a 2 a *k* bit:



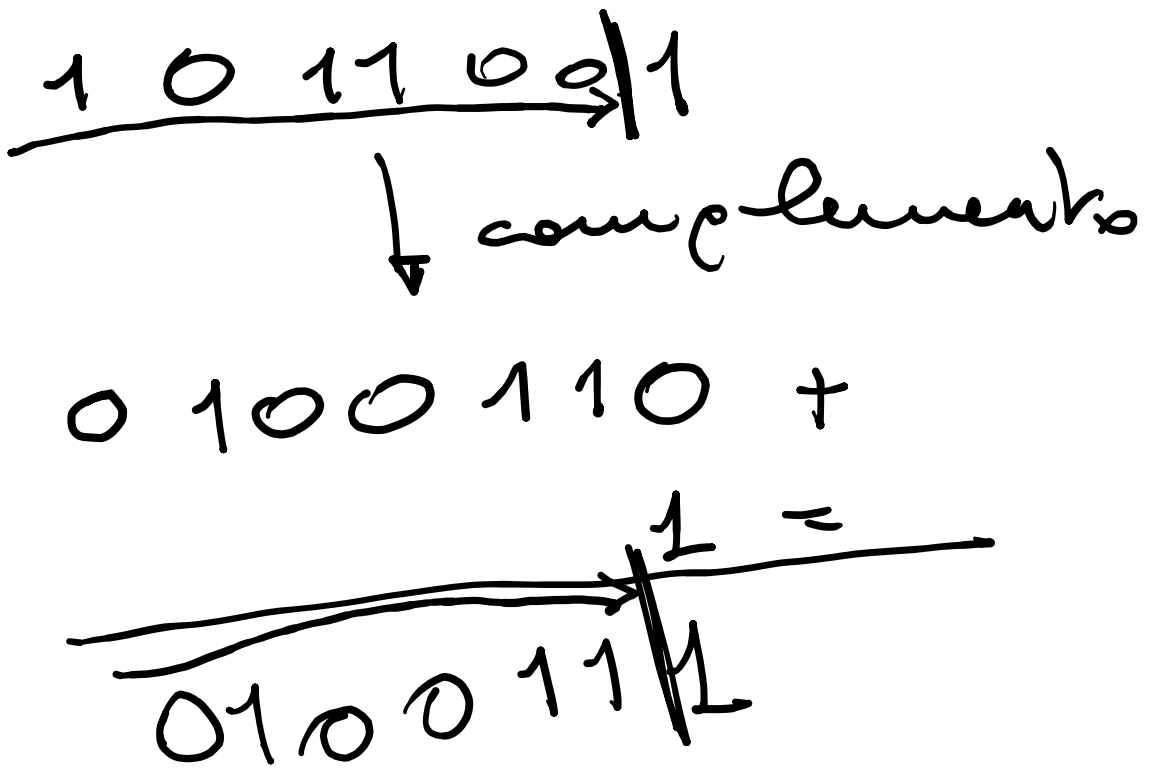
(a) 4-bit numbers

Valore binario $b_3b_2b_1b_0$	Notazione modulo e segno	Notazione complemento a 2
0000	+ 0	0
0001	+ 1	+ 1
0010	+ 2	+ 2
0011	+ 3	+ 3
0100	+ 4	+ 4
0101	+ 5	+ 5
0110	+ 6	+ 6
0111	+ 7	+ 7
1000	– 0	– 8
1001	– 1	– 7
1010	– 2	– 6
1011	– 3	– 5
1100	– 4	– 4
1101	– 5	– 3
1110	– 6	– 2
1111	– 7	– 1

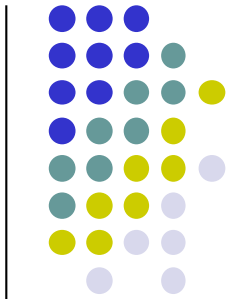


- Poiché esiste un'unica codifica del numero *0* (*-0* non viene rappresentato), l'intervallo dei numeri rappresentabili è
  $[-2^{k-1}, 2^{k-1}-1]$
- I numeri non negativi coincidono con la rappresentazione in modulo e segno
- Infatti la rappresentazione di un numero non negativo si ottiene semplicemente convertendolo in binario
- La rappresentazione di un numero negativo $-N$ si ottiene facendo la conversione in binario a k bit del numero 2^k-N
- Una semplice regola di conversione:
  - si converte in binario a k bit il numero N
  - si complementano tutti i bit
  - si somma *1*
- Oppure: si complementano tutti i bit da sinistra verso destra, fino all'ultimo bit pari ad 1 escluso

Complemento BIT A BIT



Esempio



Rappresentazione in *k=8* bit dei numeri *26* e *-26*

26:2 = 13 con resto 0
13:2 = 6 con resto 1
6:2 = 3 con resto 0
3:2 = 1 con resto 1
*1:2 = **0** con resto 1*

Quindi $26 = 11010_2$, per cui

		modulo e segno	complemento a 2
1.	rappresentazione di *26*:	*00011010*	*00011010*
2.	rappresentazione di *-26*:	*10011010*	*11100110*

-6 in modulo e segno (6bit)

$8 - 6 \equiv 8 + (-6)$ modulo e segno

$(8) \rightarrow 001000 +$

$(-6) \rightarrow 100110 =$

$2 \leftarrow 101110 \quad -14$

p_{-6} in complemento a 2 (6 bit)

1 1 1
001000 +

111010 =

000010 → 2

6 in 6 bit

000110

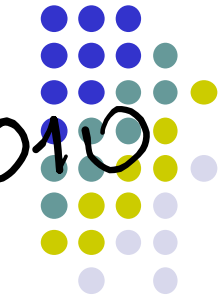
↓

-6 in C2

111010

2 CONTROLLI:

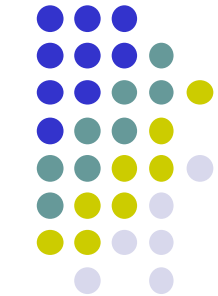
* SE c'è riporto, il risultato è positivo e il rip. si scarta
* SE NON c'è riporto, il risultato è negativo



* SE c'è riporto, il risultato è positivo e il rip. si scarta

* SE NO RIPORTO
  - il risultato è negativo
  - il suo valore assoluto si ottiene facendo il complemento a 2

6 − 8 in compl. a 2 (6 bit)



000110 +

111000 =

---

(1)111110

(8) 001000

↓ C2

(−8) 111000

è un num. negativo

C2 → 000010 = 2

−2

-5 e -28 in C2 (6 bit)

$$[-2^5, 2^5 - 1]$$

$5 = 000101$

$28 = 011100$

$-5 = 111011$

$-28 = 100100$

con 7 bit

$$\begin{array}{r} 0\,111011 \\ +\;0\,100100 \\ \hline 0\,1111111 \end{array}$$

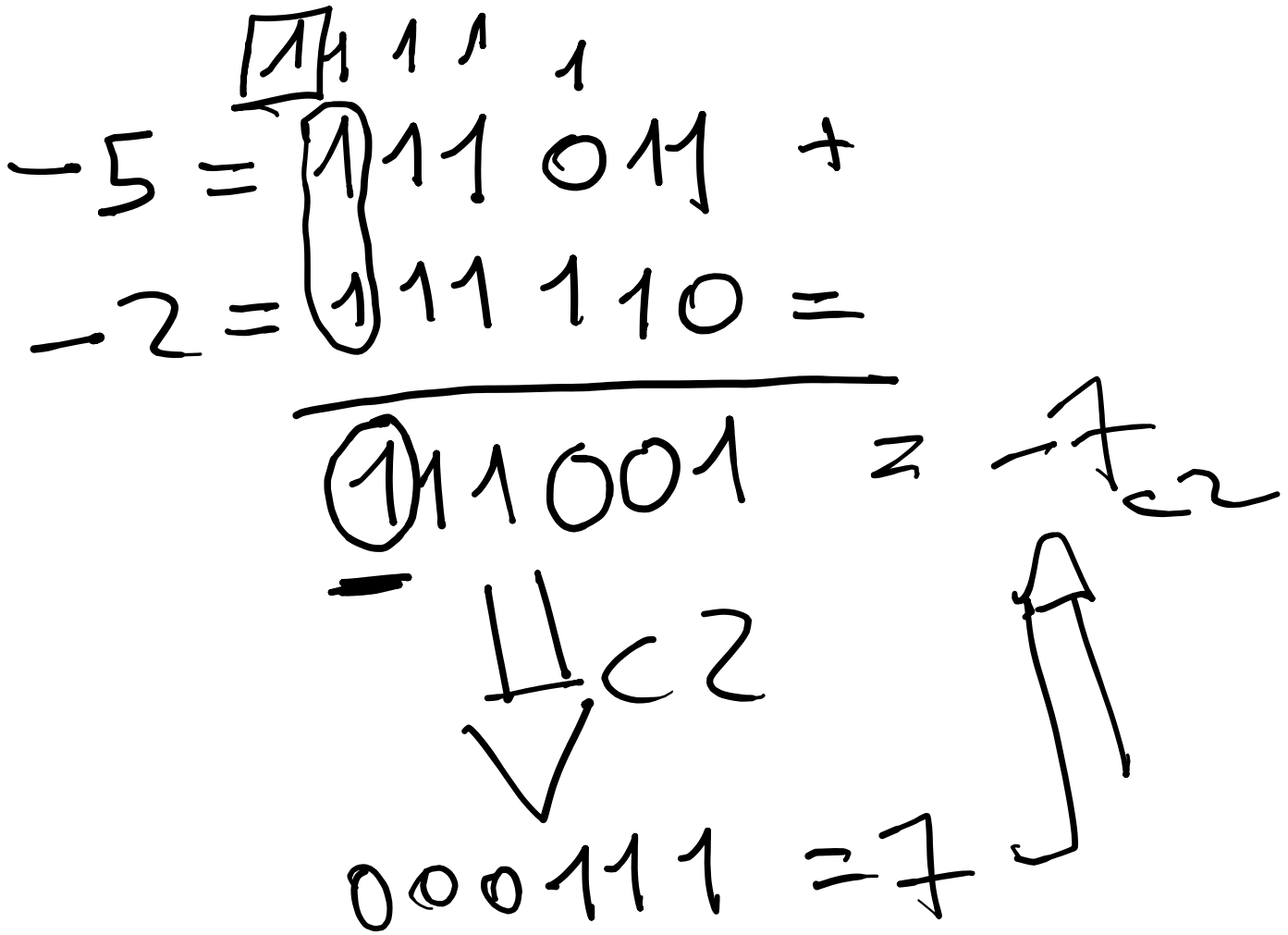
$1\,011111 = -33$

C2 (0)

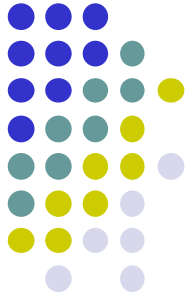
$0100001 = 33$

(3)

−5 e −2 in C2 (6 bit)



Codifica numeri frazionari (reali compresi tra *0* ed *1*)

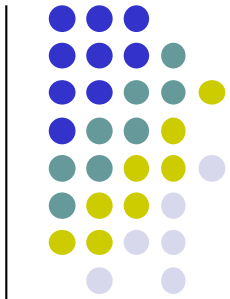


- Estendendo quanto detto per i sistemi di numerazione alla parte del numero dopo la virgola si ha che

$$0,b_{-1}b_{-2}\cdots b_{-n} = b_{-1}\cdot 10^{-1} + b_{-2}\cdot 10^{-2} + \cdots + b_{-n}\cdot 10^{-n} = \sum_{i=-1}^{-n} b_i \cdot 10^i$$

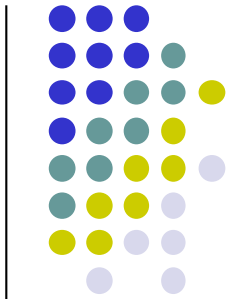
- Per una generica base *B*

$$(0,b_{-1}\cdots b_{-n})_B = b_{-1}\cdot B^{-1} + \cdots + b_{-n}\cdot B^{-n} = \sum_{i=-1}^{-n} b_i \cdot B^i \qquad (2)$$



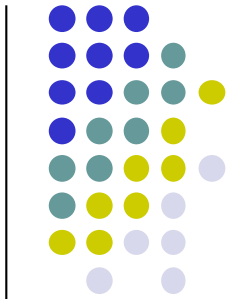
- Come convertire un numero frazionario in base *B* nel suo equivalente in una base *B*′ ≠ *B*?

- Di nuovo abbiamo due regole generali che effettuano le operazioni aritmetiche coinvolte rispettivamente nella base partenza *B* e nella base di arrivo *B*′



Regola 1:

- svolge le operazioni nella base di arrivo *B'*, per cui è molto adatta al caso in cui *B'=10*
- consiste nell ' applicare in modo diretto la sommatoria *(2)* nel seguente modo:
  1. si esprimono le cifre b_i e la base *B* nella base *B'* (solitamente banale)
  2. si calcola la sommatoria *(2)*

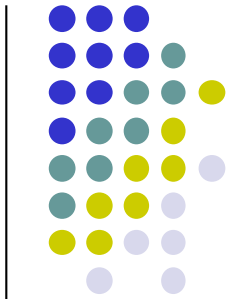


Esempi

1. $0,1101_2 = 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} =$
   $= 1/2 + 1/4 + 1/16 = 0.8125$

2. $0,452_8 = 4 \cdot 8^{-1} + 5 \cdot 8^{-2} + 2 \cdot 8^{-3} =$
   $= 4/8 + 5/64 + 2/512 = 0,582031252$

3. $0,1A8_{16} = 1 \cdot 16^{-1} + 10 \cdot 16^{-2} + 8 \cdot 16^{-3} =$
   $= 1/16 + 10/256 + 8/4096 = 0,103515625$



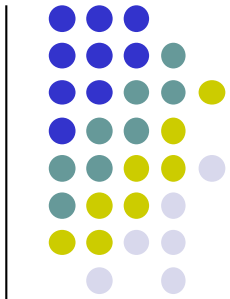
Regola 2 (delle moltiplicazioni successive):

- svolge le operazione nella base di partenza *B,* per cui è molto adatta al caso in cui *B=10*

- si basa sull'osservazione che, moltiplicando il numero per *B´:*
  - la parte intera corrisponde alla prima cifra dopo la virgola del numero nella base B´
  - la parte frazionaria al numero ottenuto cancellando la prima cifra dopo la virgola dal numero di partenza espresso in base *B´*
  - le cifre successive alla prima dopo la virgola possono essere quindi determinate riapplicando ricorsivamente lo stesso metodo alla parte frazionaria

Esempio

0.623·10 = *6.23*, ossia parte intera *6* e frazionaria *0.23*
0.23·10 = *2.3*, ossia parte intera *2* e frazionaria *0.3*
0.3·10 = *3*, ossia parte intera *3* e frazionaria 0



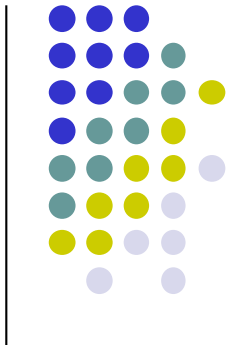
Regola 2:

1. si determinano la parte intera e frazionaria della moltiplicazione del numero per *B'*
2. si prosegue come al passo *1.* considerando di volta in volta come numero di partenza la parte frazionaria della moltiplicazione effettuata nel passo precedente, finché non si determina una parte frazionaria nulla
3. si scrivono tutte le parti intere nell'ordine in cui sono state ottenute, esprimendole nella base *B'* (solitamente banale)

NB:

- la prima parte intera ottenuta (al passo *1.*) corrisponde alla prima cifra dopo la virgola, mentre l'ultima a quella più lontana
- se *B'*<*B* tutte le parti intere ottenute sono già espresse nella base *B'*

Esempi



1) 0.375 (B=10 e B'=2)

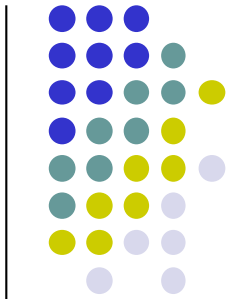
$$
\begin{array}{lll}
0.375 \cdot 2 = & 0 + 0.75 \\
0.75 \cdot 2 = & 1 + 0.5 \\
0.5 \cdot 2 = & 1 + \mathbf{0}
\end{array}
$$

Quindi $0.375 = 0{,}011_2$

2) 0.84375 (B=10 e B'=16)

$$
\begin{array}{lll}
0.84375 \cdot 16 = & 13\ (D) + 0.5 \\
0.5 \cdot 16 = & 8\ (8) + \mathbf{0}
\end{array}
$$

Quindi $0.84375 = 0{,}D8_{16}$

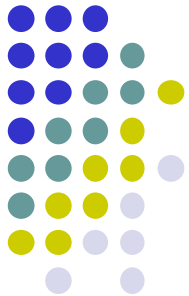


- Per codificare i numeri frazionari di nuovo viene utilizzato il sistema di numerazione posizionale in base *2*

- Fissato il numero di bit *k* da utilizzare nella rappresentazione
  1. si converte il numero di partenza nella base *2*
  2. si considerano solo i primi *k* bit dopo la virgola, eventualmente tagliando i rimanenti o aggiungendo bit uguali a *0* alla parte finale fino ad ottenere complessivamente esattamente *k* bit

NB: non è necessario rappresentare lo *0* iniziale e la virgola, perché sono comuni a tutti i numeri frazionari

Esempi *(k=4)*

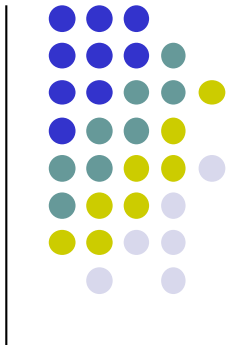


1) $0.375 = 0{,}011_2$

0110

2) $0.84375 = 0{,}11011_2$

1101

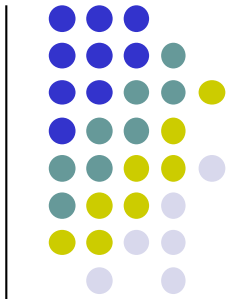


Osservazioni:

- l'eventuale aggiunta di bit pari a *0* nel passo 2. è necessaria perché nella memorizzazione del numero bisogna specificare per ogni bit (anche per gli ultimi dopo la virgola che convenzionalmente non indichiamo quando pari a *0*) lo stato del relativo dispositivo bistabile; in caso contrario si potrebbero avere errori di rappresentazione
- l'eventuale taglio di bit finali (dopo il bit finale) causa un errore di rappresentazione; in particolare, se n è il numero frazionario da rappresentare e $r(n)$ il numero rappresentato al posto di n tagliando dopo il bit k, l'errore assoluto commesso è

$$err.ass. = |n - r(n)| \leq 2^{-k}$$

Codifica numeri reali



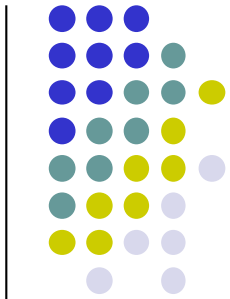
- Come fare per rappresentare **numeri reali R**?

- **Problema:** insieme **continuo**, **non numerabile**, ossia
  - per R **non esiste una corrispondenza biunivoca** con i numeri naturali (a differenza di Q, ovvero dei numeri razionali)
  - un numero razionale è un reale sempre esprimibile come frazione di due numeri interi (il secondo sempre diverso da 0)

- Unica soluzione:
  - approssimare i numeri reali con numeri razionali con la maggiore precisione possibile

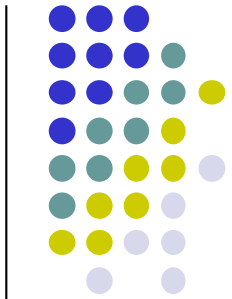
- Esistono due principali **strategie** di codifica
  - **Rappresentazione in virgola fissa**
  - **Rappresentazione in virgola mobile**

Codifica numeri reali

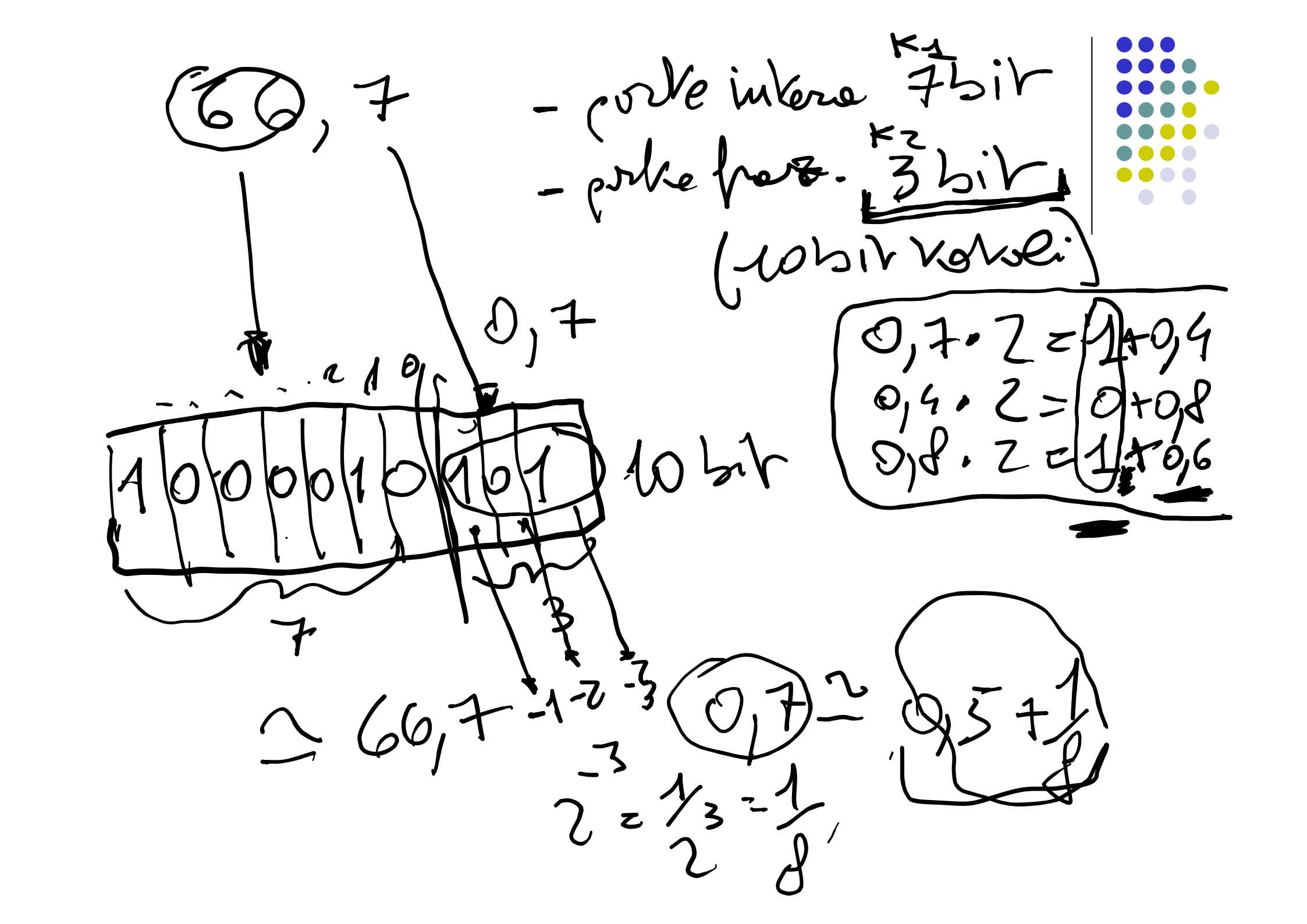


- Rappresentazione in virgola fissa
  - più semplice e diretta
  - k_1 bit per la parte intera secondo la codifica dei numeri interi
  - k_2 bit per la parte frazionaria secondo la codifica de numeri frazionari
  - NB: una volta stabilito che i primi k_1 bit sono per la parte intera e i rimanenti k_2 bit per la parte frazionaria, non è chiaramente necessario codificare la virgola
- Rappresentazione in virgola mobile (floating point)
  - si basa sull'osservazione che l'errore (assoluto) per la rappresentazione deve essere **piccolo** per numeri **piccoli** e può esser **grande** per numeri **grandi**
  - in altre parole, l'errore che bisogna limitare è l'**errore relativo (o percentuale)**, ossia l'errore assoluto rapportato alla grandezza del numero
  - Ciò ha il pregio di poter rappresentare da un lato numeri reali grandi, dall'altro numeri reali molto piccoli con precisione molto grande

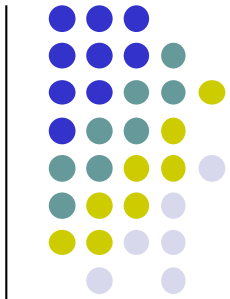
Problema Virgola Fissa



- **PROBLEMA:** e se volessi manipolare numeri con diversi ordini di grandezza?
- Avrei bisogno di codifiche diverse con differente numero di cifre
- Come fare per rappresentare numeri **molto grandi** o **molto piccoli** con lo stesso numero di cifre?
  - Necessità di manipolare numeri di ordini di grandezza diversi con lo stesso numero di cifre
  - Una rappresentazione stabile (con numero di cifre fissato) si rende necessaria per implementazione dei circuiti elettronici aritmetici
  - Sacrifico un po' di precisione => **Virgola mobile** (floating point)


66,7
- celé číslo k1 7bit
- desetinná k2 3bit
(10bit celkem)
0,7
2 1 0
1000010101
10 bit
7
3
0,7 · 2 = 1 + 0,4
0,4 · 2 = 0 + 0,8
0,8 · 2 = 1 + 0,6
≈ 66,7
-1 -2 -3
0,7 ≈
2^-3 = 1/3 = 1/8
0,5 + 1/8

Rappr. In Virgola Mobile

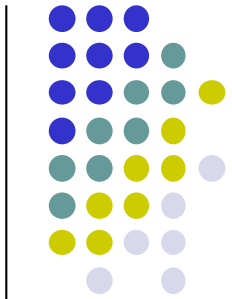


- Obiettivo**:** limitare l'errore percentuale, ossia l'errore assoluto rapportato alla grandezza del numero

- Si basa sull'osservazione che l'errore (assoluto) per la rappresentazione deve essere piccolo per numeri piccoli e può essere grande per numeri grandi

- Vantaggio**:** è possibile rappresentare da un lato numeri reali molto **grandi con minor precisione**, dall'altro numeri reali **molto piccoli con precisione più alta**

Rappr. In Virgola Mobile



- Si parte dal concetto di notazione scientifica
- Un generico reale n viene espresso nella forma:
  - **n =** $s \cdot m \cdot B^e$
    - s: è il **segno** di n, cioe indica se e positivo o negativo
    - m: è un numero frazionario (m compreso fra 0 e 1 o fra 1 e 2, vedremo) chiamato **mantissa**
    - B: è la **base** della rappresentazione
    - e: è un numero intero chiamato **esponente** (o **caratteristica**)
- Dato un certo numero di cifre per ciascuna parte della rappresentazione, il resto non deve essere codificato, poiché la struttura rimane invariata nei diversi numeri
- La base B è fissata dalla rappresentazione e non deve essere codificata, per cui non è soggetta a vincoli tecnologici e può anche essere diversa da 2