Alessio Bechini

Appunti sulla rappresentazione dei numeri

Anno accademico 2007/2008

ultima modifica: 23/11/2007

Indice

Il materiale presentato in questi appunti non costituisce una trattazione rigorosa e formale del problema della rappresentazione dell'informazione in generale, e dei numeri in particolare; il suo scopo è semplicemente quello di illustrare i principali problemi e le soluzioni più comunemente adottate in questo ambito. Sebbene questi appunti siano necessariamente concisi, essi cercano di evidenziare soprattutto le idee che stanno alla base delle varie codifiche presentate, e di fornire esempi pratici di utilizzo di tali codifiche.

1 Rappresentare l'informazione

La funzione di un sistema di calcolo è la manipolazione delle informazioni. Affinché questa manipolazione possa avere luogo, occorre trovare un metodo per *rappresentare* le informazioni. Nelle applicazioni di tipo informatico, tale rappresentazione può essere strutturata come una sequenza di simboli, su cui poi il sistema di calcolo è in grado di operare.
In maniera estremamente intuitiva, possiamo pensare di definire un qualche meccanismo di corrispondenza tra le informazioni significative per il problema che dobbiamo trattare e un insieme di sequenze di simboli. A questo proposito, cerchiamo di fornire una definizione generale del concetto di *codice*, senza riferirci a nessun caso particolare; successivamente, andremo ad analizzare come tale definizione può essere applicata concretamente a problemi specifici.

Per fissare le idee, chiamiamo I l'insieme (con cardinalità finita o infinita), che contiene gli oggetti o le informazioni che vogliamo trattare. Inoltre, chiamiamo A l'*alfabeto* finito di simboli che intendiamo utilizzare; sia β la cardinalità di A. Con A^* indichiamo l'insieme di tutte le possibili sequenze ordinate (usualmente da sinistra a destra) di simboli dell'alfabeto A, di lunghezza finita e infinita (compresa la sequenza vuota). Ogni elemento di una sequenza in A^* è chiamato *cifra*.
Definiamo una funzione iniettiva di codifica $cod : I \rightarrow A^*$ ed una relativa funzione di decodifica $decod : A^* \rightarrow I \cup \{errore\}$ tali che per ogni elemento x appartenente ad I, $decod(cod(x)) = x$.

L'insieme I, l'alfabeto A, e le funzioni *cod* e *decod* formano un *codice* per la rappresentazione di elementi di I. Dato un elemento x in I, la sequenza di simboli *cod(x)* è chiamata *codifica di x* (talvolta anche *rappresentazione di x*) tramite il codice scelto. In generale, non tutte le sequenze di simboli in A^* hanno come controparte un elemento di I. La situazione è rappresentata schematicamente in Figura 1.

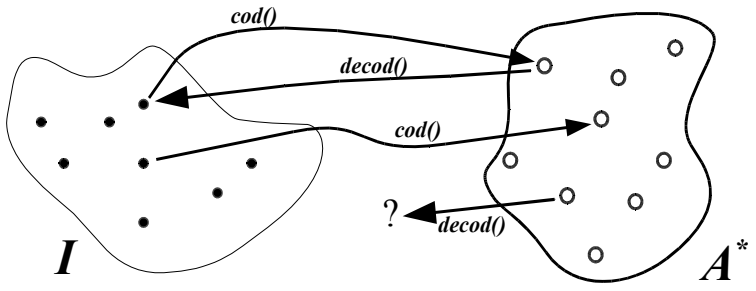


Figura 1: Corrispondenza fra elementi di I e sequenze di simboli

Dato che in un sistema di calcolo si può operare soltanto su un numero finito di cifre, utilizzando k cifre e un alfabeto composto da β simboli diversi, si potranno rappresentare al massimo β^k elementi distinti di I.

2 Rappresentazione di numeri naturali

Il problema della rappresentazione dei numeri naturali è un caso particolare del problema della rappresentazione degli elementi di un generico insieme I, come visto nella sezione precedente. In questo caso, $I \equiv \mathbb{N}$.
Se ci troviamo nella condizione di poter utilizzare soltanto un dato numero k di cifre, allora non tutti gli elementi di $\mathbb{N}$ possono essere rappresentati, ma tra essi dovremo selezionarne un sottoinsieme che indicheremo con $I_{\mathbb{N}}$; questa situazione è schematizzata in Figura 2.

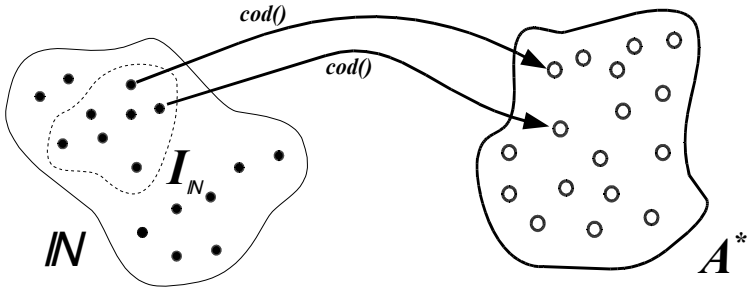


Figura 2: Corrispondenza fra naturali (o un sottoinsieme dei naturali) e sequenze di simboli

Dato che su $\mathbb{N}$ è definita una relazione di ordinamento totale <, è ragionevole occuparsi dei casi in cui $I_{\mathbb{N}}$ è un intervallo, solitamente della forma [0 ... n_{max}]. Una delle caratteristiche fondamentali di un codice è l'*intervallo di rappresentabilità* ad esso associato: ovvero, insieme alle regole che specificano la funzione di codifica, dobbiamo individuare qual'è l'intervallo di valori che viene coperto dalla rappresentazione. Se si utilizza un alfabeto A con cardinalità β, e un numero massimo di cifre prefissato pari a k, il numero di sequenze distinte che si possono generare è β^k : in genere, tali sequenze sono fatte corrispondere ai naturali in [0 ... β^k-1], che costituisce cioè l'intervallo di rappresentabilità.

Nella scelta di un codice per la rappresentazione dei naturali, si fa riferimento ad alcuni criteri generali: il codice deve essere generale, ovvero applicabile per fronteggiare esigenze diverse, e deve consentire l'utilizzo di procedure semplici e veloci per svolgere operazioni tra naturali.

2.1 Notazione posizionale

I codici che vengono usati più frequentemente per rappresentare i naturali, con le caratteristiche di generalità e praticità d'impiego richieste, presuppongono la scelta di un alfabeto A di cardinalità β e fanno riferimento a una forma particolare in cui può essere espresso ogni numero naturale[1]:

$$x = a_{k-1}\beta^{k-1} + \cdots + a_1\beta^1 + a_0 = \sum_{i=0}^{k-1} a_i\beta^i \qquad \text{dove, } \forall\, i, \quad a_i \in \mathbb{N} \quad \text{e} \quad 0 \le a_i < \beta \quad . \tag{1}$$

Da questa forma si ricava la cosiddetta *notazione posizionale*: essa prevede che per x si costruisca

1 Si omette la dimostrazione dell'esistenza e unicità dei coefficienti presenti in tale forma.

una sequenza di k cifre (ciascuna associata al corrispondente valore a_i), ordinate da sinistra a destra partendo da a_{k-1} fino ad arrivare ad a_0. Si dice anche che tale sequenza costituisce la *rappresentazione in base* β di x.
La cifra associata ad a_{k-1} (l'ultima a sinistra) viene chiamata *cifra più significativa*, mentre quella associata ad a_0 (l'ultima a destra) viene detta *cifra meno significativa*.

> *Nota*
>
> In pratica, i simboli dell'alfabeto A che vengono usati in una generica base β (minore o uguale a 36) sono i primi β della sequenza (0, 1, ... , 9, A, B, ... , Z). Secondo questa convenzione, il naturale associato al carattere "A" è 10, a "B" è 11, e così via.

Il termine *notazione posizionale* fa riferimento al fatto che ciascuna cifra della rappresentazione, nella determinazione del valore, assume un peso che dipende dalla posizione in cui essa è collocata.

La rappresentazione normalmente utilizzata per i naturali è quella posizionale in base 10.
Se, fornendo il valore di un numero, vogliamo specificare la base utilizzata, useremo la seguente notazione: $(134)_{10}$ indica un valore espresso in base dieci (per semplicità, si suppone che il numero del pedice sia sempre espresso in base dieci); $(5G3F)_{18}$ indica un valore in base diciotto; $(21011)_3$ indica un valore in base tre, e così via.
Qualora non venga esplicitata la base, si intende implicitamente che essa sia dieci.

La base impiegata più comunemente nelle rappresentazioni usate dai sistemi di calcolo è $\beta=2$; in questo caso, la codifica viene detta *binaria*, e le singole cifre prendono il nome di *bit*.
Altre basi che hanno rilevanza nel settore dell'informatica sono quelle corrispondenti a potenze di due, ovvero la 4, la 8 e la 16.

> *Esempi*
>
> Consideriamo ancora la rappresentazione $(21011)_3$: essa si riferisce al valore
> $2\cdot3^4 + 1\cdot3^3 + 0\cdot3^2 + 1\cdot3^1 + 1\cdot3^0$, ovvero $2\cdot81 + 1\cdot27 + 0\cdot9 + 1\cdot3 + 1\cdot1 = 193$
>
> Consideriamo la rappresentazione $(1B)_{12}$: essa si riferisce al valore
> $1\cdot12^1 + 11\cdot12^0$, ovvero $1\cdot12 + 11\cdot1 = 23$

Dal momento che sappiamo svolgere le operazioni aritmetiche in base dieci, la formula $x=\sum_{i=0}^{k-1} a_i \beta^i$ ci permette di ottenere il valore (in base dieci) di un numero rappresentato in qualsiasi altra base.
Se volessimo invece ottenere la rappresentazione di un generico naturale x in una base specificata, come possiamo procedere per ricavarne tutte le cifre (ovvero, i corrispondenti coefficienti a_i)? A tal proposito, basta osservare che, dividendo x per β , si può scrivere

$$x = a_0 + \beta \cdot \sum_{i=1}^{k-1} a_i \beta^{i-1} = a_0 + \beta \cdot q_1$$

con $0 \le a_0 < \beta$ e $q_1 > 0$; ovvero, a_0 è il resto della divisione (intera) x/β . Analogamente,

$$q_1 = a_1 + \beta \cdot \sum_{i=1}^{k-2} a_i \beta^{i-2} = a_1 + \beta \cdot q_2$$
$$q_2 = a_2 + \beta \cdot \sum_{i=1}^{k-3} a_i \beta^{i-3} = a_2 + \beta \cdot q_3$$
$$\vdots$$

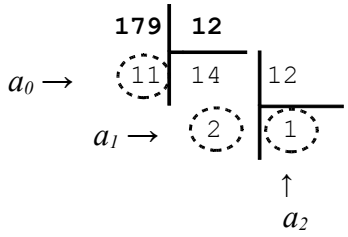
L'ultimo coefficiente (ovvero la cifra più significativa) si ottiene quando si arriva ad avere un quoziente $q_{k-1} < \beta$.

Esempio

Sappiamo che il valore di x , in base 10, è $(179)_{10}$. Qual'è la rappresentazione corrispondente in base 12?

Calcoliamo a_0 : 179/12 = 14 con resto 11. Perciò a_0 = 11, ovvero la cifra "B".
Calcoliamo a_1 : 14/12 = 1 con resto 2. Perciò a_0 = 1, ovvero la cifra "1".
Calcoliamo a_2 : visto che il precedente quoziente è più piccolo di 12, corrisponderà ad esso. Perciò a_2 = 2, ovvero la cifra "2".

Le operazioni svolte possono essere visualizzate così:



In definitiva, $(179)_{10} = (12B)_{12}$

Nota

Un aspetto significativo della notazione posizionale riguarda la sua capacità di esprimere un valore in modo conciso. Per rendercene conto meglio, possiamo determinare qual'è il *numero minimo di cifre* su cui è rappresentabile un generico naturale x in base β.
Visto che vogliamo il numero *minimo* di cifre, questo significa che nella rappresentazione non ci sono zeri in testa; ovvero, con
$x = \sum_{i=0}^{k-1} a_i \beta^i$, $a_{k-1} \neq 0$ e dunque $\beta^{k-1} \leq x < \beta^k$; sotto queste condizioni, il nostro problema corrisponde a trovare il valore di k in funzione di x, una volta che sia noto β .

Se $\beta^{k-1} \leq x < \beta^k$ allora $\log_\beta(\beta^{k-1}) \leq \log_\beta(x) < \log_\beta(\beta^k)$ e quindi $k - 1 \leq \log_\beta(x) < k$.

Questa ultima catena di disuguaglianze (e il fatto che k è un naturale) ci indica che:

A) $k \le \log_\beta(x)+1$ e dunque $k \le \lfloor \log_\beta(x)+1 \rfloor$ [2];
B) $k > \log_\beta(x)$ e dunque $k \ge \lfloor \log_\beta(x)+1 \rfloor$

Dovendo valere entrambe le disuguaglianze A) e B), si ottiene

$k = \lfloor \log_\beta(x) +1 \rfloor$

Quindi, tra x e k c'è un legame di tipo logaritmico; ovviamente, a parità del valore x, con basi più grandi si hanno rappresentazioni più compatte. A titolo di esempio, vediamo che

$(10011101100)_2 = (20020)_5 = (1260)_{10} = (4EC)_{16}$

Ovvero, con basi più grandi bastano meno cifre per rappresentare lo stesso numero.

2.2 Operazioni aritmetiche con notazione posizionale

Probabilmente la notazione posizionale deve la sua larga diffusione alla capacità di supportare lo svolgimento di operazioni aritmetiche in modo molto semplice. In questa sezione mostreremo alcune semplici operazioni usando una base generica β, e ponendo poi particolare attenzione al caso della codifica binaria.

Moltiplicazione e divisione per una potenza della base

Come prima operazione, consideriamo la moltiplicazione di un naturale x per un numero che sia pari a β^g. Se chiamiamo y il risultato, avremo

$$y = x \cdot \beta^g = \beta^g \cdot (a_0\beta^0 + a_1\beta^1 + a_2\beta^2 + \cdots) = a_0\beta^g + a_1\beta^{g+1} + a_2\beta^{g+2} + \cdots$$

ovvero, la rappresentazione di y corrisponde alla rappresentazione di x seguita da g cifre pari a 0.

Esempi

Se la rappresentazione in base 13 di x è $(A1B)_{13}$, qual'è la rappresentazione (sempre in base 13) di $y = x \cdot 13$?

In accordo a quanto visto, avremo $y = (A1B0)_{13}$

Dato $x = (11101)_2$, quanto vale $y = 8x$ (sempre in binario)?

Visto che $8 = 2^3$, basterà aggiungere tre zeri in coda alla rappresentazione di x:

$y = (11101000)_2$

Bisogna precisare che, se abbiamo un numero prefissato k di cifre per la rappresentazione di x, è possibile che il risultato y non sia rappresentabile anch'esso su k cifre (condizione detta di

2 Il simbolo $\lfloor y \rfloor$ indica il più grande naturale che sia minore o uguale a y.

traboccamento od *overflow*); ci possiamo rendere conto di questo fatto andando a vedere se le g cifre più significative di x sono tutte uguali a zero (dunque, assenza di *overflow*) oppure no.

> *Esempi*
>
> Consideriamo rappresentazioni binarie *su cinque bit* di numeri naturali.
>
> Dato $x = (00101)_2$, è possibile ottenere $y = 4x$?
>
> Devo appurare che non ci sia overflow: visto che $4 = 2^2$, considero i due bit più significativi di x: entrambi sono 0, per cui non c'è overflow e $y = (10100)_2$.
>
> Dato $x = (10111)_2$, è possibile ottenere $y = 2x$?
>
> Visto che $2 = 2^1$, considero il bit più significativo di x: è 1, e dunque y non è rappresentabile su cinque bit (se eseguissi l'operazione usando la solita procedura, otterrei un risultato scorretto, essendo in condizione di overflow).

In modo del tutto analogo, se vogliamo dividere un naturale x per un numero che sia pari a β^g , basterà togliere le ultime g cifre dalla rappresentazione di x; se tali ultime g cifre sono pari a 0, allora la divisione sarà esatta, altrimenti esse corrisponderanno alla rappresentazione del resto.

> *Esempi*
>
> Se abbiamo $x = (13810)_{10}$, qual'è la rappresentazione di $y = x/1000$?
>
> In accordo a quanto visto, dato che $1000 = 10^3$, avremo $y = (13)_{10}$ con resto pari a $(810)_{10}$
>
> Dato $x = (100)_2$, quanto vale $y = x/2$ (sempre in binario)?
>
> Basta togliere il bit meno significativo: $y = (10)_2$
>
> Inoltre, la divisione è esatta (infatti il bit tolto vale 0).

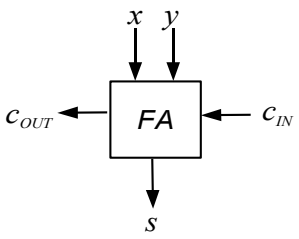
Nel caso della divisione appena visto, ovviamente non potranno mai verificarsi condizioni di overflow!

Somma

Consideriamo adesso l'operazione di somma tra due naturali x e y; chiamiamo s il risultato dell'operazione. Per fissare la idee, avremo

$$x = x_0\beta^0 + x_1\beta^1 + x_2\beta^2 + \cdots$$
$$y = y_0\beta^0 + y_1\beta^1 + y_2\beta^2 + \cdots$$
$$s = s_0\beta^0 + s_1\beta^1 + s_2\beta^2 + \cdots$$

Vogliamo identificare un meccanismo semplice che ci permetta di ottenere i coefficienti s_i a partire dagli x_i e y_i.
Notiamo che $0 \le x_0 + y_0 \le 2(\beta - 1)$ e dunque si può scrivere $x_0 + y_0 = s_0 + c_0 \cdot \beta$ con $0 \le s_0 < \beta$ e $0 \le c_0 \le 1$; c_0 è detto *riporto uscente* ("carry") di livello 0.
Analogamente, $0 \le x_1 + y_1 + c_0 \le 2\beta - 1$ e dunque si può scrivere $x_1 + y_1 + c_0 = s_1 + c_1 \cdot \beta$ con $0 \le s_1 < \beta$ e $0 \le c_1 \le 1$; c_0 è il riporto entrante, mentre c_1 è detto *riporto uscente* di livello 1. Procedendo in questo modo, si possono ricavare tutti i coefficienti s_i, con $x_i + y_i + c_{i-1} = s_i + c_i \cdot \beta$.
In pratica, per saper svolgere l'intera somma, basta saper sommare due singole cifre e un riporto entrante, ottenendo come risultato una singola cifra e un riporto uscente. Usando la codifica binaria, un sommatore elementare di questo tipo può essere specificato come in Figura 3: la tabella riportata indica quali sono i valori in uscita (risultati) in dipendenza dei valori forniti in ingresso (operandi).



c_{IN}	x	y	s	c_{OUT}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figura 3: Un sommatore binario elementare ("full adder")

Avendo a disposizione un sommatore binario elementare come quello in Figura 3 (detto *full adder*), possiamo agevolmente costruire sommatori per addendi con un numero dato qualsiasi n di bit, semplicemente concatenando n sommatori elementari[3]. Un esempio con $n = 3$ è mostrato in Figura 4.

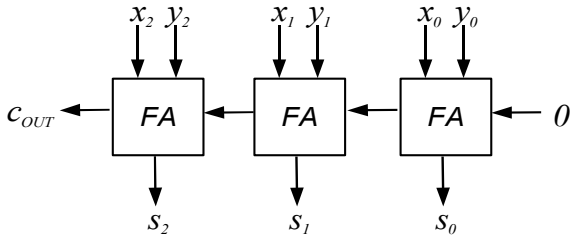


Figura 4: Un semplice sommatore con addendi naturali a tre bit

Notiamo che n bit possono non essere sufficienti per rappresentare la somma di due numeri naturali su n bit: se il valore di c_{OUT} è 1, allora significa che occorrerebbero $n + 1$ bit. Il valore di c_{OUT} può essere visto sia come l'indicazione della presenza di un riporto uscente (o *carry*), sia come una segnalazione del fatto che il risultato non è rappresentabile sul numero di bit a nostra disposizione (condizione di *overflow*).

3 Con il sistema indicato, il tempo impiegato per effettuare una somma è proporzionale al numero di bit considerati, in quanto occorre attendere che l'eventuale riporto sul bit meno significativo si propaghi fino al sommatore per il bit più significativo. E' possibile progettare sommatori più efficienti, ma la loro trattazione va oltre gli scopi di questi appunti.

2.3 Conversioni tra basi β e basi potenze di β

Qualora si debba procedere a convertire un valore espresso in base β in una rappresentazione in base $\gamma = \beta^g$, si possono fare alcune osservazioni per rendere più agevole questa operazione. Si può scrivere

$$x = a_0 \cdot \beta^0 + a_1 \cdot \beta^1 + a_2 \cdot \beta^2 + a_3 \cdot \beta^3 + a_4 \cdot \beta^4 + a_5 \cdot \beta^5 + \ldots$$ e, analogamente,

$$x = b_0 \cdot \gamma^0 + b_1 \cdot \gamma^1 + b_2 \cdot \gamma^2 + b_3 \cdot \gamma^3 + b_4 \cdot \gamma^4 + b_5 \cdot \gamma^5 + \ldots$$

Vogliamo esprimere i coefficienti b_i in funzione dei coefficienti a_i . Basta notare che, con $\gamma = \beta^g$,

$$\begin{aligned} x &= (a_0 \cdot \beta^0 + a_1 \cdot \beta^1 + \ldots a_{g-1} \cdot \beta^{g-1}) \cdot \gamma^0 + (a_g \cdot \beta^0 + a_{g+1} \cdot \beta^1 + \ldots a_{2g-1} \cdot \beta^{g-1}) \cdot \gamma^1 + \ldots \\ &= b_0 \cdot \gamma^0 + b_1 \cdot \gamma^1 + \ldots \end{aligned}$$

e dunque

$$\begin{aligned} b_0 &= a_0 \cdot \beta^0 + a_1 \cdot \beta^1 + \ldots a_{g-1} \cdot \beta^{g-1} \\ b_1 &= a_g \cdot \beta^0 + a_{g+1} \cdot \beta^1 + \ldots a_{2g-1} \cdot \beta^{g-1} \\ &\ldots \\ b_j &= \sum_{i=0}^{g-1} a_{j \cdot g + i} \cdot \beta^i = (a_{j \cdot g + g - 1} \; a_{j \cdot g + g - 2} \ldots a_{j \cdot g})_\beta \end{aligned}$$

In pratica, il valore delle cifre nella base γ si ottiene suddividendo la sequenza di cifre ($\ldots a_2 \, a_1 \, a_0$) in gruppi (partendo da destra) con g elementi ciascuno, e interpretando ogni gruppo come il valore (espersso in base β) della corrispondente cifra in base γ. Questa considerazione assume notevole importanza pratica quando occorra fare cambiamenti tra le basi 2, 4, 8 e 16.

> *Esempio*
>
> Scrivere la rappresentazione in base otto del valore naturale $(1001100101111)_2$.
>
> In questo caso, $\beta = 2$. Dato che la base di "destinazione" è $\gamma = 8$, cioè 2^3, ovvero $g = 3$; suddividiamo la rappresentazione binaria in gruppi di tre bit ciascuno (partendo da destra):
>
> $(1 \;\; 001 \;\; 100 \;\; 101 \;\; 111)_2$ e adesso troviamo il valore associato a ciascun gruppo:
>
> $b_0 = (111)_2 = (7)_8$
> $b_1 = (101)_2 = (5)_8$
> $b_2 = (100)_2 = (4)_8$
> $b_3 = (001)_2 = (1)_8$
> $b_4 = (\;\; 1)_2 = (1)_8$
>
> e quindi $(1001100101111)_2 = (11457)_8$.

Esempio

A quale codifica binaria corrisponde la rappresentazione esadecimale $(FA5)_{HEX}$.?

Dato che la base di "partenza" è 16, e quella di "destinazione" è 2 e che $16=2^4$, possiamo utilizzare le osservazioni appena fatte in modo inverso rispetto a quanto fatto nell'esempio precedente: a ciascuna cifra esadecimale facciamo corrispondere una sequenza di quattro cifre binarie, e la codifica binaria dell'intero numero sarà data dalla concatenazione delle sequenze di bit ottenute. In pratica, abbiamo:

$b_0 = (5)_{16} = (0101)_2$
$b_1 = (A)_{16} = (10)_{10} = (1010)_2$
$b_2 = (F)_{16} = (15)_{10} = (1111)_2$

e quindi $(FA5)_{16} = (1111\ 1010\ 0101)_2$.

2.4 Codifica BCD

Oltre alla codifica dei naturali basata sulla notazione posizionale, in informatica e in elettronica viene usata talvolta anche un'altra codifica, detta *BCD* (binary-coded decimal). Questo codice presuppone l'utilizzo di un alfabeto binario, e si basa sulla forma decimale posizionale per la rappresentazione di un numero:

$$x=\sum_{i=0}^{k-1} a_i 10^i \quad \text{dove, } \forall\, i, \quad a_i \in \mathbb{N} \quad \text{e} \quad 0 \le a_i < 10 \quad .$$

La codifica BCD di x consiste nella concatenazione delle codifiche in binario delle cifre a_i. Dato che occorre rappresentare dieci simboli diversi, abbiamo bisogno almeno di quattro bit per ogni cifra. Se usiamo esattamente quattro bit, la citra "0" sarà codificata dalla sequenza (0000), e la cifra "9" dalla sequenza (1001).

I moderni calcolatori utilizzano locazioni di memorizzazione che contengono otto bit (ovvero un byte), e dunque si può scegliere se memorizzare su un singolo byte due cifre BCD successive, oppure una singola cifra BCD. La particolare codifica BCD che segue la prima convenzione (quattro bit per cifra) viene detta *packed*, mentre l'altra (otto bit per cifra) è chiamata *unpacked*.

Esempio

Determinare le rappresentazioni packed BCD e unpacked BCD del numero $(1653)_{10}$.

Caso "packed": $(1)_{10} = (0001)_2$; $(6)_{10} = (0110)_2$; $(5)_{10} = (0101)_2$; $(3)_{10} = (0011)_2$.
La rappresentazione packed sarà dunque $(0001\ 0110\ 0101\ 0011)_{packedBCD}$.

Per ottenere la rappresentazione unpacked, basta inserire quattro bit 0 in testa a ciascuna delle cifre in binario che abbiamo già calcolato:
$(0000\ 0001\ 0000\ 0110\ 0000\ 0101\ 0000\ 0011)_{unpackedBCD}$.

La codifica BCD è scarsamente efficiente per la memorizzazione dei numeri, in quanto comporta un notevole spreco di bit (infatti, in packed BCD delle possibili 2^4 stringhe di quattro bit se ne usano soltanto 10). Nonostante questo, il suo utilizzo può essere conveniente perché il BCD ha una semplice corrispondenza con il codice ASCII (utilizzato per codificare caratteri di testo): infatti basta anteporre il prefisso 0011 ai quattro bit meno significativi di ciascuna cifra BCD per ottenere il codice ASCII del carattere corrispondente.

> *Nota*
>
> In pratica, il packed BCD viene utilizzato per rappresentare numeri interi (quindi sia positivi che negativi). Per far questo, ogni codifica di un numero viene sempre terminata dalla codifica su quattro bit del segno: solitamente, si usano le sequenze 1100 per il + e 1101 per il − . Perciò, il numero +237 sarà rappresentato con 0010 0011 0111 1100, e −237 con 0010 0011 0111 1101.

3 Rappresentazione di numeri interi

Il problema della rappresentazione dei numeri interi (dunque con segno) può essere ricondotto a quello della rappresentazione dei naturali, a patto di riuscire a trovare un modo per far corrispondere ciascun numero intero a un naturale che sappiamo già rappresentare. In dipendenza dalla regola di corrispondenza che intendiamo applicare, otterremo codifiche diverse con diverse proprietà. Siamo interessati a trovare una codifica che ci permetta di manipolare facilmente i numeri interi e di svolgere velocemente le principali operazioni aritmetiche.
Innanzi tutto, visto che in pratica in un sistema di calcolo si impiega la base $\beta = 2$, e che inoltre possiamo usare solo sequenze di bit di lunghezza finita k, dobbiamo dunque restringere la nostra attenzione a un sottoinsieme degli interi con cardinalità 2^k. E' ragionevole scegliere a questo proposito un intervallo di valori interi $I_{\mathbb{Z}}$ centrato (per quanto possibile) sullo zero; in genere, si sceglie $I_{\mathbb{Z}}=[-2^{k-1}, 2^{k-1}-1]$. L'intervallo dei naturali $I_{\mathbb{N}}$ con cui vogliamo metterlo in corrispondenza sarà $I_{\mathbb{N}}=[0, 2^k-1]$. La corrispondenza viene espressa analiticamente con una funzione $R(x): I_{\mathbb{Z}} \rightarrow I_{\mathbb{N}}$; ci riferiremo a questa funzione con il termine *rappresentazione di x*, sottintendendo che *R(x)* verrà poi codificato in base due con notazione posizionale. La situazione descritta è schematizzata in Figura 5.

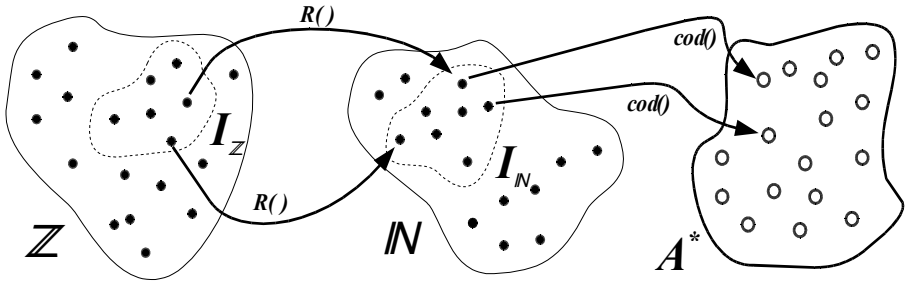


Figura 5: Corrispondenza fra un sottoinsieme degli interi, un sottoinsieme dei naturali e sequenze di simboli

Di seguito vengono presentate tre diverse codifiche, cercando di evidenziare l'idea che sta alla base di ciascuna di esse, e individuandone pregi e difetti.

3.1 Codifica in eccesso 2^{k-1}

Questa codifica si basa su una regola molto semplice per far corrispondere gli elementi di $I_{\mathbb{Z}}$ con quelli di $I_{\mathbb{N}}$: basta aggiungere una quantità fissa (detta *eccesso* o *polarizzazione*) ai primi, in modo da far "traslare" il primo intervallo esattamente sopra $I_{\mathbb{N}}$. L'*eccesso* cercato è ovviamente 2^{k-1}.
La forma analitica di *R(x)* in questa codifica è dunque

$$R(x)=x+2^{k-1} \tag{2}$$

L'intervallo di rappresentabilità per la codifica in eccesso 2^{k-1} corrisponde a tutto l'intervallo $I_{\mathbb{Z}}$.

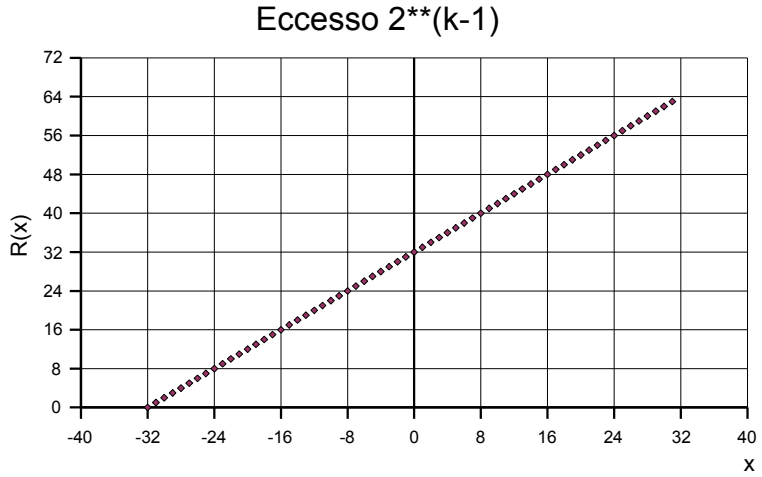


Figura 6: Grafico di R(x) per k = 6 bit

Questa codifica permette di confrontare agilmente due numeri interi, utilizzando le stesse regole del confronto tra le rappresentazioni dei naturali: infatti, se $x<y$, allora $R(x)<R(y)$ e viceversa.
Inoltre, ispezionando il bit più significativo di $R(x)$, si può individuare immediatamente il segno di x: vediamo perché.
Se $x<0$, allora $R(x) = x+2^{k-1} < 2^{k-1}$. Ma la rappresentazione binaria su k bit di 2^{k-1} è un 1 seguito da $k-1$ zeri. Dunque, se $R(x) < 2^{k-1}$, ha sicuramente il bit più significativo pari a 0.
Se $x \geq 0$, allora $R(x) = x+2^{k-1} \geq 2^{k-1}$. Ma la rappresentazione binaria su k bit di 2^{k-1} è un 1 seguito da $k-1$ zeri. Dunque, se $R(x) \geq 2^{k-1}$, ha sicuramente il bit più significativo pari a 1.

Una delle caratteristiche negative di questa codifica è quella di non permettere l'utilizzo di procedure semplici e veloci (quali quelle usate per i naturali) per svolgere operazioni aritmetiche; per esempio, in generale **non** possiamo affermare che $R(x+y)=R(x)+R(y)$.

> *Esempio*
>
> Determiniamo la codifica in eccesso 2^{k-1} di $(-18)_{10}$ su 6 bit.
>
> Sarà $R(x) = x + 2^5$ perciò $R(x) = (-18)_{10} + (32)_{10} = (+14)_{10} = (001110)_2$

3.2 Modulo e segno

Questa codifica si basa su una regola molto semplice per far corrispondere gli elementi di $I_{\mathbb{Z}}$ con quelli di $I_{\mathbb{N}}$: si pone x nella forma $x = sgn(x) \cdot |x|$, si codifica il segno sul bit più significativo, e si usano i rimanenti $k-1$ bit per codificare il modulo. Procedendo in questo però si hanno due inconvenienti:

- Non è possibile rappresentare l'estremo inferiore di $I_{\mathbb{Z}}$ (cioè -2^{k-1}), in quanto il suo modulo non è rappresentabile su $k-1$ bit. Dunque, l'intervallo di rappresentabilità per questa codifica è $[-(2^{k-1}-1),\ +(2^{k-1}-1)]$.

- Lo zero ha una doppia rappresentazione: +0 e -0 !

La forma analitica di *R(x)* in questa codifica è dunque

$$R(x)=\begin{cases} x & se\ x \geqslant 0 \\ -x+2^{k-1} & se\ x \leqslant 0 \end{cases} \tag{3}$$

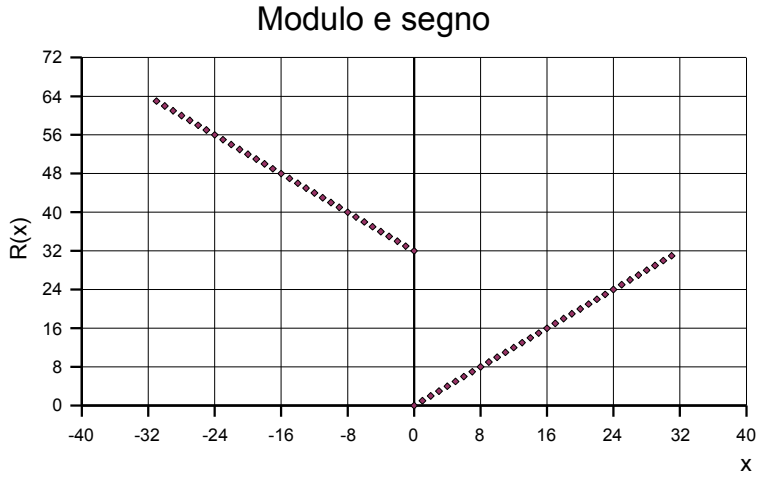


Figura 7: Grafico di R(x) per k = 6 bit

Considerando la regola con cui è definita questa codifica, ci rendiamo immediatamente conto che con questa rappresentazione è banale confrontare il valore assoluto o il segno di due numeri. Come nel caso della codifica in eccesso 2^{k-1}, anche questa non consente l'impiego di procedure semplici e veloci (quali quelle usate per i naturali) per svolgere operazioni aritmetiche; per esempio, in generale **non** possiamo affermare che $R(x+y)=R(x)+R(y)$.

> *Esempio*
>
> Conosciamo la codifica modulo e segno di un intero su 5 bit: $R(x) = (11101)_2$
> Qual'è il valore di *x* (in base 10)?
>
> Si nota che il bit più significativo è 1, e dunque siamo di fronte a un numero negativo.
>
> $|x| = (1101)_2 = (13)_{10}$
>
> Dunque, $x = -13$

3.3 Eccesso 2^k (o complemento a due)

Le codifiche per gli interi viste in precedenza hanno in comune la difficoltà a supportare procedure semplici e veloci per svolgere operazioni aritmetiche: la codifica che analizziamo adesso è stata studiata per ovviare a questo inconveniente. Si tratta ancora di una codifica in eccesso, che comporta però una traslazione di $+2^k$ per $I_{\mathbb{Z}}$. Questo significa che i valori interi nell'intervallo $[-2^{k-1}, -1]$ vengono mappati sui naturali dell'intervallo $[+2^{k-1}, +(2^k-1)]$. La traslazione degli interi in

$[0, +(2^{k-1}-1)]$ presenta però un problema: tale intervallo verrebbe mappato fuori da $I_{\mathbb{N}}$, su $[+2^k, +(3\cdot 2^{k-1}-1)]$, e per codificare questi naturali occorrerebbe un bit in più rispetto ai k che abbiamo a disposizione. Però si può osservare che, se avessimo tale bit (il $k+1$-esimo), il suo valore sarebbe sempre a 1, e la sua rimozione corrisponderebbe a togliere 2^k al valore rappresentato. Se adottiamo questo "trucco", non facciamo altro che mappare gli *interi* in $[0, +(2^{k-1}-1)]$ sui *naturali* in $[0, +(2^{k-1}-1)]$: dunque, se vogliamo rappresentare un intero positivo, basterà interpretarlo come un naturale e prenderne la corrispondente rappresentazione!
In definitiva, la forma analitica di $R(x)$ può essere espressa come

$$R(x)=\begin{cases} x & se\ x\geq 0 \\ x+2^k & se\ x<0 \end{cases} \tag{4}$$

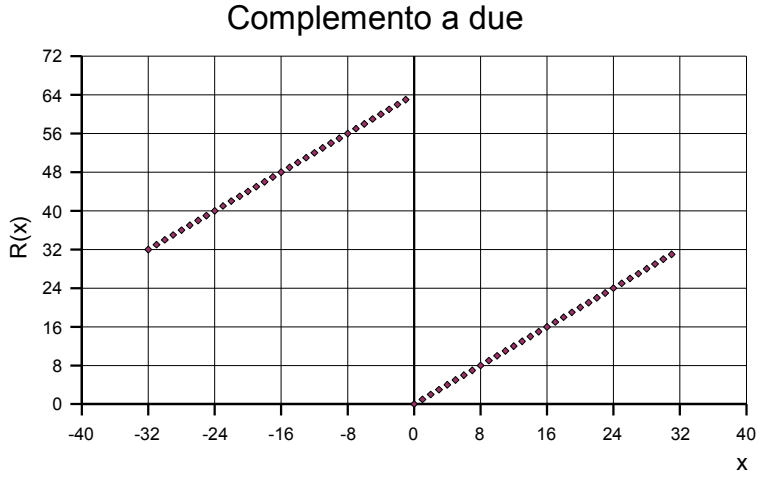


Figura 8: Grafico di R(x) per k = 6 bit

La rappresentazione in complemento a due è sicuramente quella più usata per la codifica degli interi sui moderni calcolari elettronici.

> *Nota*
>
> Il nome "complemento a due" di questa codifica deriva da una possibile interpretazione del meccanismo adottato per la rappresentazione dei numeri negativi. In generale, se si opera su k bit, il numero intero $\tilde{x}$ tale che $x+\tilde{x}=2^k$ viene detto *complemento a due* di x. Nel caso in cui x sia negativo, potremmo pensare di rappresentarlo come il complemento a due del suo modulo: dunque $R(x)=\widetilde{|x|}=2^k-|x|$, ma, dato che $x<0$, allora $|x| = -x$, e otteniamo $R(x)=\widetilde{|x|}=2^k+x$, che coincide con quanto già indicato.

Come per le precedenti codifiche, il segno di x si può individuare ispezionando il bit più significativo di $R(x)$, cioè il bit che supporta il coefficiente di 2^{k-1}.
Se $x \geq 0$, il bit più significativo di $R(x)$ sarà 0; infatti, se $x \geq 0$, $R(x) = x$, e dovendo x appartenere all'intervallo di rappresentabilità, sarà comunque strettamente minore di $+2^{k-1}$, e dunque il bit più significativo sarà 0.
Se $x<0$, il bit più significativo di $R(x)$ sarà 1; infatti, se $x<0$, allora $R(x) = x+2^k$, e dovendo x appartenere all'intervallo di rappresentabilità, sarà comunque maggiore o uguale a -2^{k-1}, dunque

$R(x) \geq 2^{k-1}$, ovvero il bit più significativo sarà 1.

In ogni caso, l'aspetto più interessante della codifica in complemento a due consiste nel fatto di consentire lo svolgimento delle operazioni aritmetiche con le procedure (semplici e veloci) usate per i naturali. Per esempio, qualora il numero $x+y$ ricada nell'intervallo di rappresentabilità, $R(x+y)$ può essere ottenuta in modo semplicissimo a partire da $R(x)+R(y)$.

Per ottenere x a partire dalla sua rappresentazione (o viceversa), si può sempre applicare la definizione di $R(x)$. Per motivi di praticità, nel caso in cui si abbia a che fare con numeri negativi, è utile eventualmente ricorrere a una regoletta che ci permetta di esprimere $R(x)$ in termini di $|x|$. Per ricavare questa regola, notiamo innanzitutto che , se $x<0$, allora $|x| = -x$.
Dato un generico numero n, indichiamo con $\overline{n}$ il numero che si ottiene prendendo la sua rappresentazione binaria e complementandola bit a bit (cioè, mettendo a 0 tutti i bit che erano a 1, e a 1 tutti i bit che sono a 0). Si ha che $n+\overline{n}=2^k-1$, ovvero una rappresentazione con tutti i k bit a 1.

$$R(x)=x+2^k=-|x|+(|x|+\overline{|x|}+1)=\overline{|x|}+1$$

In generale, R(x) può essere riscritta come

$$R(x)=\begin{cases} x & se\ x\geq 0 \\ \overline{|x|}+1 & se\ x<0 \end{cases} \qquad (5)$$

In pratica, se devo rappresentare in complemento a due un intero negativo x su k bit, posso operare in tre passi:

- trovo la rappresentazione binaria su k bit del modulo di x
- complemento bit a bit la rappresentazione trovata
- aggiungo 1.

In genere, è conveniente applicare questa regoletta quando si trattano numeri negativi piccoli in modulo su un numero elevato di bit.

Esempio

Sia $x = (-27)_{10}$. Qual'è la sua rappresentazione in complemento a due su 6 bit?

Preliminarmente, notiamo che x appartiene all'intervallo di rappresentabilità, che è $[-2^{6-1}, +(2^{6-1}-1)]$ ovvero $[-2^5, +(2^5-1)]$, ovvero $[-(32)_{10}, +(31)_{10}]$.

Sapendo che $R(x)=x+2^k$, ottengo $-(27)_{10}+(64)_{10}=(37)_{10}=(100101)_2$

Alternativamente, potevo procedere utilizzando la regoletta del complemento bit a bit:

$|(-27)_{10}| = (27)_{10} =(011011)_2$
$\overline{(011011)_2}=(100100)_2$
$R(x) = (100100)_2 + 1 = (100101)_2$

Esempio

Conosciamo la codifica in complemento a due di un intero su 9 bit:
$R(x) = (111100101)_2$

Qual'è il valore di x (in base 10)?

Si nota che il bit più significativo è 1, e dunque siamo di fronte a un numero negativo.

Sapendo che $x=R(x)-2^9$, calcoliamolo: $x = 2^8 + 2^7\ 2^6 + 2^5 + 2^2 + 2^0 - 2^9 = -(27)_{10}$

Il conto per arrivare al risultato è un po' laborioso; allora, in alternativa, visto che già sappiamo che si tratta di un negativo, ci basterebbe trovare il modulo.
Posso osservare che, essendo $R(x)+\overline{R(x)}=2^k-1$,

$$|x|=-x=-(R(x)-2^k)=-[R(x)-(R(x)+\overline{R(x)}+1)]=\overline{R(x)}+1$$

dunque, $|x|=\overline{(111100101)_2}+1=(000011010)_2+1=(000011011)_2=(27)_{10}$

$x = -(27)_{10}$

3.4 Operazioni aritmetiche in complemento a due

Svolgere operazioni aritmetiche con operandi interi rappresentati in complemento a due richiede l'applicazione di procedure analoghe a quelle utilizzate per i numeri naturali. In questa sezione analizziamo alcune di esse, allo scopo di evidenziarne la semplicità. Sottolineiamo comunque che, a differenza di quanto presupposto per la rappresentazione binaria dei naturali, nel caso dei numeri interi si stabilisce sempre un numero k di bit da usare per la rappresentazione.

Moltiplicazione e divisione per due

Come prima operazione, consideriamo la moltiplicazione di un intero x (su k bit) per due.
Se x è positivo e $y = 2x$ ricade nell'intervallo di rappresentabilità, allora $R(y) = 2x$: ovvero, si può applicare la procedura vista per i numeri naturali: si trasla $R(x)$ di un bit a sinistra, mettendo poi a zero il bit meno significativo.
Se invece x è negativo, anche $y = 2x$ è negativo; ricordiamo inoltre che, in questo caso, $R(x) \geq 2^{k-1}$, ovvero il bit più significativo sarà 1. Se y ricade nell'intervallo di rappresentabilità, allora $R(y) = 2x + 2^k$. Dato che $R(x) = x + 2^k$, abbiamo $R(y) = 2 \cdot R(x) - 2^k$. Visto che $R(x) \geq 2^{k-1}$ allora $2 \cdot R(x) \geq 2^k$, e quindi fare $2 \cdot R(x) - 2^k$ corrisponde a traslare $R(x)$ di una posizione a sinistra, ignorando il bit a 1 che esce a sinistra, e mettendo uno zero nel bit meno significativo.
Riassumendo, *indipendentemente dal segno di x*, per ottenere $R(2x)$, si trasla *sempre* $R(x)$ di un bit a sinistra, mettendo poi a zero il bit meno significativo.

Ci possiamo chiedere se è possibile stabilire, analizzando il valore di x, se $2x$ è ancora rappresentabile sullo stesso numero k di bit; in altri termini, vogliamo capire come individure una indicazione di *overflow* per l'operazione di moltiplicazione per due. La cosa risulta abbastanza semplice: se $2x$ cade nell'intervallo di rappresentabilità, abbiamo $-2^{k-1} \leq 2x \leq (2^{k-1}-1)$ e dunque $-2^{k-2} \leq x \leq (2^{k-2}-1)$; in pratica, questo significa che i due bit più significativi devono essere uguali.

Siamo dunque giunti alla conclusione che l'operazione di moltiplicazione per due su una rappresentazione binaria corrisponde alla stessa procedura sia per i naturali che per gli interi in complemento a due; ciò che cambia è soltanto il modo per ottenere l'indicazione di overflow.

> *Esempi*
>
> Consideriamo rappresentazioni in complemento a due *su sei bit*.
>
> Dato $x = (111011)_2$, qual'è la rappresentazione di $y = 2x$?
>
> I due bit più a sinistra sono entrambi a 1, per cui non avrò overflow. Allora banalmente si ha: $y = (110110)_2$.
>
> Dato $x = (010010)_2$, qual'è la rappresentazione di $y = 2x$?
>
> I due bit più a sinistra sono diversi tra loro: la procedura di traslazione determinerebbe una condizione di overflow.

Possiamo a questo punto affrontare il problema della divisione intera per due. Indipendentemente dal segno di x, sappiamo che $y = x/2$ ricade sicuramente nell'intervallo di rappresentabilità.
Se $x \geq 0$, possiamo applicare la procedura vista per i numeri naturali: si trasla $R(x)$ di un bit a destra, mettendo poi a zero il bit più significativo.
Se invece $x < 0$, anche y lo è. Dato che $R(x) = x + 2^k$, dunque[4] $R(x)/2 = (x + 2^k)/2 = x/2 + 2^{k-1}$.
Ma $R(y) = x/2 + 2^k = R(x)/2 - 2^{k-1} + 2^k = R(x)/2 + 2^{k-1}$. Questo corrisponde a traslare $R(x)$ di una posizione a destra, mettendo inoltre a uno il bit più significativo.
Riassumendo, *indipendentemente dal segno di x*, per ottenere $R(x/2)$, si trasla *sempre* $R(x)$ di un bit a destra, mantenendo sul bit più significativo il valore da esso assunto prima della traslazione.
Questo tipo particolare di procedura è nota come traslazione a destra *con mantenimento del segno*.

> *Esempi*
>
> Consideriamo rappresentazioni in complemento a due *su otto bit*.
>
> Dato $x = (11100100)_2$, qual'è la rappresentazione di $y = x/4$?
>
> Occorre dividere due volte per due, ovvero fare una traslazione a destra di due posizioni con mantenimento del segno. Otteniamo $y = (11111001)_2$.
>
> Dato $x = (01111111)_2$, qual'è la rappresentazione di $y = x/2$?
>
> Bisogna fare una traslazione a destra di un bit con mantenimento del segno. Otteniamo $y = (00111111)_2$. La divisione non è esatta: infatti il bit meno significativo di x è 1, e dunque il resto vale 1.

4 Per comprendere il seguente passaggio, ricordiamo che stiamo considerando la divisione intera (quindi che può essere non esatta), e che una potenza di due è sempre pari.

Somma

Abbiamo già anticipato che, per la codifica in complemento a due, $R(x+y)$ può essere ottenuta in modo banale a partire da $R(x)+R(y)$, ovviamente qualora $x+y$ ricada nell'intervallo di rappresentabilità.
Andiamo adesso ad approfondire questo punto. Notiamo preliminarmente che, nell'eseguire la somma $x + y$ su k bit, la possibilità di avere overflow esiste soltanto quando i due addendi hanno segno concorde; se hanno segno diverso, allora varrà la disuguaglianza $| x+y | \leq MAX(| x |, | y |)$ e il risultato sarà sempre rappresentabile. Per fissare le idee, indichiamo la somma delle rappresentazioni degli addendi con $S_R \equiv R(x)+R(y)$: dobbiamo scoprire come ottenere $R(x+y)$ a partire da S_R . Occorre capire come procedere nei vari casi che si possono presentare, che sono i seguenti quattro:

Caso 1) $x \geq 0$ e $y \geq 0$. La situazione è analoga a quella che si presenta per la somma di naturali, perciò $R(x+y) = S_R$, limitatamente ai primi k bit.
Si ha overflow quando il bit più significativo della rappresentazione su k bit della somma viene ad assumere il valore 1 (e dunque, in complemento a due, andrebbe interpretata come un numero negativo: l'overflow determina quindi un "cambio di segno" anomalo). Il valore di tale bit può perciò essere preso come indicatore di overflow.

Caso 2) $x < 0$ e $y < 0$. In questo caso, $(x+y) < 0$, per cui $R(x+y) = (x+y) + 2^k$.
Si ha $R(x) = x + 2^k$ e $R(y) = y + 2^k$; ambedue hanno il bit più significativo a 1, ovvero $S_R \geq 2^k$, ovvero S_R richiederà sicuramente $k+1$ bit, e il suo bit più significativo (quello del termine $+2^k$, interpretabile come riporto uscente) sarà sicuramente 1.
$S_R = x + 2^k + y + 2^k$ e dunque $R(x+y) = S_R - 2^k$: questo significa che per avere $R(x+y)$ basta prendere i primi k bit di S_R.
L'overflow si ha quando il bit più significativo della rappresentazione su k bit della somma viene ad assumere il valore 0 (e dunque, in complemento a due, andrebbe interpretata come un numero positivo: l'overflow determina quindi un "cambio di segno" anomalo). Il valore di tale bit può perciò essere preso come indicatore di overflow.

Caso 3) $x > 0$ e $y < 0$, con $| x | > | y |$. Abbiamo dunque $(x+y) > 0$, per cui $R(x+y) = (x+y)$.
Si ha $R(x) = x$ e $R(y) = y + 2^k$, e dunque $S_R = x + y + 2^k$. Ma, dato che $(x+y) > 0$, la presenza del termine $+2^k$ implica che la rappresentazione di S_R debba sicuramente richiedere $k+1$ bit, e il suo bit più significativo (quello del termine $+2^k$, interpretabile come riporto uscente) sia sicuramente 1. In definitiva, $R(x+y) = S_R$, limitatamente ai primi k bit.
Come già precisato, in questo caso non si può avere overflow.

Caso 4) $x > 0$ e $y < 0$, con $| x | < | y |$. Abbiamo $(x+y) < 0$, e dunque $R(x+y) = (x+y) + 2^k$.
Come nel caso precedente, si ha $R(x) = x$ e $R(y) = y + 2^k$, e dunque $S_R = x + y + 2^k$; inoltre, visto che $(x+y) < 0$, la rappresentazione di S_R richiede soltanto k bit: in definitiva, abbiamo $R(x+y) = S_R$.
Come già precisato, anche in questo caso non si può avere overflow.

Possiamo adesso concludere che la somma tra interi in complemento a due può essere fatta utilizzando le stesse procedure (e quindi gli stessi componenti circuitali) usati per la somma di naturali; ciò che invece cambia è il modo di ottenere l'indicazione di overflow, in quanto il riporto uscente non è più significativo per questo scopo. Da quanto precisato nei vari casi analizzati, deduciamo che si ha overflow ogni volta che sono verificate *entrambe* le seguenti condizioni:

- i bit più significativi dei due addendi sono uguali
- il bit più significativo del risultato (su *k* bit) è diverso dal bit più significativo del primo addendo (o, indifferentemente, del secondo addendo, dato che deve sussistere anche la codizione precedente)

Un esempio di sommatore per interi in complemento a due su tre bit è presentato in Figura 9; esso comprende un apposito modulo per la generazione dell'indicazione di overflow.

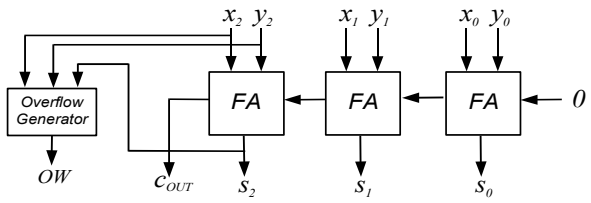


Figura 9: Un sommatore con addendi interi a tre bit (in complemento a due)

4 Rappresentazione dei numeri razionali e reali

All'interno di un sistema di calcolo si può ovviamente dedicare soltanto un numero *finito* di bit per la rappresentazione di un numero. Qualora si desideri rapprentare numeri razionali o reali, in dipendenza dallo specifico valore e/o dal tipo di rappresentazione scelta, può capitare che non si riesca a rappresentare *esattamente* il valore desiderato, ma ci si debba accontentare di far riferimento a un altro valore (necessariamente razionale) "molto vicino" ad esso. Per questo motivo, da un punto di vista pratico la rappresentazione dei razionali e quella dei reali si riducono a una stessa problematica. Di seguito parleremo esclusivamente di "rappresentazione dei razionali", intendendo con ciò evidenziare come, volendo trattare i reali, a un valore reale si debba sempre associare un valore razionale che lo approssimi al meglio.
Per la rappresentazione dei numeri razionali possiamo pensare di procedere analogamente a quanto fatto per gli interi, ovvero individuare un modo per far corrispondere numeri razionali a interi o a naturali che sappiamo già rappresentare. Se individuiamo un intervallo di valori razionali $I_{\mathbb{Q}}$ che siamo interessati a rappresentare, in generale utilizzando un numero dato di cifre k non siamo ovviamente in grado di rappresentare *tutti* i numeri razionali dell'intervallo, ma soltanto 2^k tra di essi. Dunque, ogni volta che si desideri rappresentare un razionale il cui valore non coincida con uno di quei 2^k prescelti, andremo necessariamente incontro a un *errore di rappresentazione*. Per questo motivo, quando si definisce una qualche rappresentazione dei razionali, occorre comprendere anche quale *precisione* è in grado di fornire; per *precisione* si intende l'errore massimo che si può commettere nel rappresentare il valore di un razionale. Questa precisione dipende da due fattori:

- la distribuzione su $I_{\mathbb{Q}}$ dei valori razionali con rappresentazione esatta;
- il meccanismo di arrotondamento adottato per far corrispondere un razionale (non rappresentabile in modo esatto) a uno dei due valori razionali "esatti" ad esso più vicini.

Le regole di corrispondenza tra razionali e naturali normalmente utilizzate, definiscono implicitamente anche la modalità di distribuzione su $I_{\mathbb{Q}}$ dei razionali con rappresentazione esatta. Tali regole fanno riferimento ad alcune "forme" specifiche in cui può essere posto un razionale; la codifica del numero razionale è ottenuta come concatenemanto delle codifiche dei vari elementi presenti nella forma scelta. Quanto detto risulterà sicuramente più chiaro affrontando direttamente i due tipi di codifica praticamente utilizzati per i razionali: quella in *virgola fissa*, e quella in *virgola mobile*. Nella presentazione di queste due codifiche, partiremo trattando il caso più generale di rappresentazioni in una base qualunque β, per poi concretizzare quanto introdotto nel caso pratico della codifica binaria.

4.1 Rappresentazione in virgola fissa

Consideriamo un generico numero razionale positivo[5] x : possiamo esprimerlo nella forma

$$x = x_I + x_F \tag{6}$$

dove x_I è la *parte intera* (ovvero $|x_I| \geq 1$), e x_F è la *parte frazionaria* ($0 \leq |x_F| < 1$).

Per la rappresentazione di x_I possiamo sfruttare la notazione posizionale utilizzata per i numeri naturali, e la determinazione dei coefficienti a_i può essere fatta con il metodo della divisione.

Per esprimere x_F possiamo ugualmente fare riferimento alla notazione posizionale, esprimendolo nella forma

$$x_F = a_{-1}\beta^{-1} + a_{-2}\beta^{-2} + a_{-3}\beta^{-3} + \cdots$$

Occorre notare che, in dipendenza dagli specifici valori assunti da x_F e da β, questa sommatoria potrebbe non avere un numero finito di termini. Quindi, se abbiamo a disposizione f cifre per la memorizzazione dei coefficienti a_i , la rappresentazione che se ne ottiene si riferisce a un'approssimazione di x_F ogni volta che, per esprimere x_F nella forma indicata, abbiamo bisogno di almeno un termine con un β elevato a una potenza minore di $-f$. Nei casi in cui la rappresentazione non sia esatta, l'errore commesso è comunque minore di β^{-f}.

Per determinare i coefficienti della rappresentazione posizionale di x_F, basta osservare che

$$\beta x_F = \beta(a_{-1}\beta^{-1} + a_{-2}\beta^{-2} + a_{-3}\beta^{-3} + \ldots) = a_{-1} + a_{-2}\beta^{-1} + a_{-3}\beta^{-2} + \ldots = a_{-1} + p_{-1}$$

con $0 \leq a_{-1} < \beta$ e $p_{-1} < 1$; ovvero, il coefficiente a_{-1} si ottiene come parte intera di βx_F .

Con procedimento analogo si possono ricavare i restanti coefficienti:

$$\beta p_{-1} = \beta(a_{-2}\beta^{-1} + a_{-3}\beta^{-2} + \ldots) = a_{-2} + a_{-3}\beta^{-1} + \ldots = a_{-2} + p_{-2}$$
$$\beta p_{-2} = \beta(a_{-3}\beta^{-1} + a_{-4}\beta^{-2} + \ldots) = a_{-3} + a_{-4}\beta^{-1} + \ldots = a_{-3} + p_{-3}$$
…

Esempio

$(6.625)_{10} = (?)_4$
$x_I = (6)_{10}$ $x_F = (0.625)_{10}$
Utilizzando il metodo della divisione, otteniamo $x_I = (6)_{10} = (12)_4$.
Cerchiamo a_{-1} : $\beta x_F = (4)_{10} \cdot (0.625)_{10} = (2.5)_{10}$ e dunque $a_{-1} = (2)_4$, $p_{-1} = (0.5)_{10}$
Continuando il procedimento, troviamo $a_{-2} = (2)_4$, $p_{-2} = (0)_{10}$ e dunque ci fermiamo.
Possiamo concludere che:
$(6.625)_{10} = (12.22)_4$

5 Trattiamo inizialmente solo il caso dei positivi; i negativi verranno presi in considerazione quando ci occuperemo esplicitamente di rappresentazioni binarie con un numero dato di cifre per la parte frazionaria.

In un sistema di calcolo, si adottano rappresentazioni con un numero finito k di cifre. Nel caso della rappresentazione in virgola fissa, si stabilisce che le f cifre più a destra della sequenza vengano dedicate alla parte frazionaria, e le restanti (le $k-f$ più a sinistra) alla parte intera. Si parla in questo caso di *rappresentazione in virgola fissa*, in quanto la scelta di f corrisponde a fissare una virgola "immaginaria" in una determinata posizione della sequenza di bit.

Esempio

Sia $x = (3.1)_{10}$. Qual'è la rappresentazione in virgola fissa di x in base due, su 7 bit, di cui 4 riservati alla parte frazionaria?

$x_I = (3)_{10} = (11)_2$

$x_F = (0.1)_{10} = (\ ?\ )_2$

$\beta\, x_F = (2)_{10} \cdot (0.1)_{10} = (0.2)_{10}$ e dunque $a_{-1} = (0)_2$ e $p_{-1} = (0.2)_{10}$
$\beta\, p_{-1} = (2)_{10} \cdot (0.2)_{10} = (0.4)_{10}$ e dunque $a_{-2} = (0)_2$ e $p_{-2} = (0.4)_{10}$
$\beta\, p_{-2} = (2)_{10} \cdot (0.4)_{10} = (0.8)_{10}$ e dunque $a_{-3} = (0)_2$ e $p_{-3} = (0.8)_{10}$
$\beta\, p_{-2} = (2)_{10} \cdot (0.8)_{10} = (1.6)_{10}$ e dunque $a_{-4} = (1)_2$ e $p_{-4} = (0.6)_{10}$

Siamo arrivati a determinare i quattro coefficienti che ci occorrono. Notiamo però che $p_{-4} \neq 0$: questo significa che la rappresentazione ottenuta è approssimata.

Il risultato è:

0	1	1	0	0	0	1
$R(x_I)$			$R(x_F)$			

Ciò che accade in pratica quando si rappresenta in virgola fissa un razionale x su k cifre di cui f per la parte frazionaria, corrisponde all'operazione di codifica del numero intero $x \cdot 2^f$ (preso come intero) su k cifre. Se operiamo in base 2, possiamo pensare di codificare $x \cdot 2^f$ in complemento a due: in questo modo siamo in grado di estendere quanto discusso finora anche ai razionali negativi.
Se scegliamo di procedere in questo modo, potremo esprimere $R(x)$ così:

$$R(x) = \begin{cases} x \cdot 2^f & se\ x \geq 0 \\ x \cdot 2^f + 2^k & se\ x < 0 \end{cases} \quad \text{oppure} \quad R(x) = \begin{cases} x \cdot 2^f & se\ x \geq 0 \\ \overline{|x \cdot 2^f|} + 1 & se\ x < 0 \end{cases} \tag{7}$$

Esempio

Vogliamo la rappresentazione binaria in virgola fissa di $(5.26)_{10}$ su 8 bit, di cui 3 per la parte frazionaria.

$(5.26)_{10} \times (2^3)_{10} = (42.08)_{10}$ non è un intero: la rappresentazione sarà approssimata.

Andremo allora a rappresentare $(42)_{10}$ come intero su 8 bit: $(42)_{10} = (00101010)_2$

Esempio

Vogliamo la rappresentazione binaria in virgola fissa di $(-10.25)_{10}$ su 8 bit, di cui 2 per la parte frazionaria.
Si tratta di un numero negativo, perciò

$|(-10.25)_{10} \times (2^2)_{10}| = (41)_{10}$ è un intero: la rappresentazione sarà esatta.

Andremo allora a rappresentare $(41)_{10}$ come intero su 8 bit: $(41)_{10} = (00101001)_2$

$\overline{(00101001)_2} = (11010110)_2$ per cui

$R(x) = (11010110)_2 + 1 = (11010111)_2$

Vale la pena osservare che, con questa codifica, l'intervallo di rappresentabilità è dunque $[-(2^{k-1})\cdot 2^{-f}, +(2^{k-1}-1)\cdot 2^{-f}]$; all'interno di questo intervallo, i valori rappresentabili sono equispaziati, e due valori consecutivi differiscono di 2^{-f} . La precisione della codifica non dipende dunque dal valore rappresentato, e si mantiene costante lungo tutto l'intervallo di rappresentabilità: questo può essere facilmente notato anche nell'esempio riportato in Figura 10.

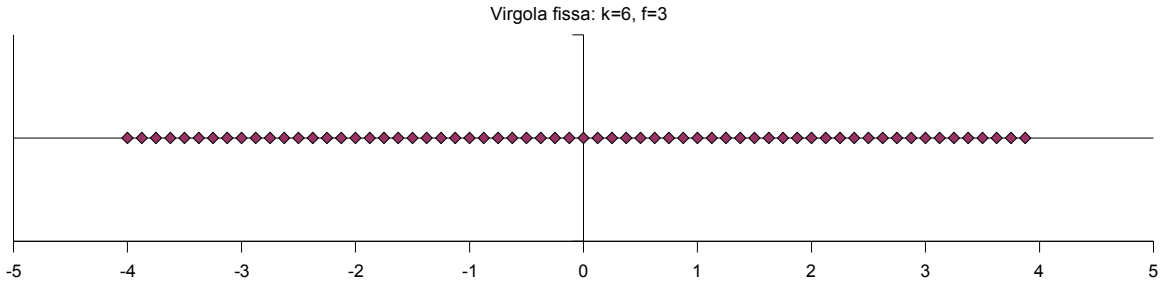


Figura 10: Valori razionali rappresentabili in modo esatto in virgola fissa su 6 bit, di cui tre per la parte frazionaria

4.2 Rappresentazione in virgola mobile

La rappresentazione dei numeri razionali in virgola fissa può rivelarsi poco pratica nel momento in cui si vogliano trattare numeri che coprono un intervallo di valori molto ampio.
Si osserva che, spesso, non è molto sensato usare la stessa precisione per numeri molto piccoli e numeri molto grandi. Per esempio, se il valore x_1 è dell'ordine di 10^2, una precisione di 10^{-5} può essere considerata accettabile; se il valore x_2 è dell'ordine di 10^{15}, una precisione di 10^{-5} quasi sicuramente è eccessiva. In molti casi pratici, è più significativo considerare la *precisione relativa*[6], piuttosto che la precisione "semplice". Lo scopo per cui è stata introdotta la rappresentazione descritta in questa sezione è duplice: dato il numero di cifre a nostra disposizione, allargare il più possibile l'intervallo di rappresentabilità, mantenedo la stessa *precisione relativa* su tutto l'intervallo.

6 Per *precisione relativa* si intende l'errore massimo (in valore assoluto) di rappresentazione di un numero, diviso il valore assoluto del numero rappresentato.

Una rappresentazione di un numero razionale, diversa da quella vista in precedenza, può essere basata sulla seguente forma:

$$x = sgn(x) \cdot m \cdot \beta^e \tag{8}$$

In questa forma (indicata spesso come "notazione scientifica"), $sgn(x)$ indica il segno di x, m è detta mantissa, β base ed e esponente. Si nota che, fissato il valore della base, questa forma non è univoca, nel senso che, dato il valore di x, esistono infinite coppie (m, e) che soddisfano la precedente uguaglianza. Per esempio, possiamo osservare che le seguenti forme (espresse con valori in base 10) per x = 3.1415 sono tutte "valide":

$$+3.1415 = +3.1415 \cdot 10^1 = +31.415 \cdot 10^{-1} = +0.31415 \cdot 10^2 = +0.031415 \cdot 10^3$$

Se vogliamo rendere univoca la forma da adottare (in modo da usarla per definire una rappresentazione del numero), occorre stabilire un preciso intervallo entro cui deve cadere il valore della mantissa; questa procedura è nota come *normalizzazione*.
Una volta fissata la base e la condizione di normalizzazione, la rappresentazione di x viene dunque ricondotta alle rappresentazioni del segno, dell'esponente e della mantissa. Da qui in avanti, ci occuperemo della rappresentazione di x in base due, utilizzando un numero totale k di bit, di cui uno per il segno, k_e per l'esponente e k_m per la mantissa; in definitiva, $k_e + k_m = k$.
Una rappresentazione del numero che consista nella concatenazione su k bit della rappresentazione del segno, dell'esponete e della mantissa (solitamente proprio in quest'ordine) è nota come *rappresentazione in virgola mobile* (floating point).

Per la codifica del segno, si usa un solo bit; normalmente 0 indica il seno positivo, 1 indica il negativo.

Per la scelta della codifica dell'esponente, occorre fare qualche considerazione preliminare. L'esponente deve poter assumere sia valori positivi che negativi, in un'intervallo centrato sullo 0; per poter confrontare facilmente due esponenti, sarebbe opportuno che si potesse ridurre tale operazione a un confronto tra naturali. Si adotta a questo scopo una rappresentazione in eccesso:

$$R(e) = e + n_{bias} \tag{9}$$

Rimane da determinare il valore da assegnare a n_{bias}, detto talvolta *polarizzazione*. In genere, se si opera su k_e bit, viene fatta la scelta $n_{bias} = 2^{ke-1} - 1$. Per esempio, se $k_e = 8 \rightarrow n_{bias} = 2^7 - 1 = 127$; in questo caso, il valore più piccolo per l'esponente è -127, e il più grande è +128.

La scelta della codifica della mantissa è influenzata dalla condizione di normalizzazione che si intende adottare, ovvero alla definizione dell'intervallo di valori entro cui deve ricadere. Nel caso di rappresentazioni binarie della mantissa, la condizione di normalizzazione che viene generalmente adottata è

$$(1.0)_2 \le m < (10.0)_2 \quad \text{, ovvero} \quad (1.0)_{10} \le m < (2.0)_{10} \tag{10}$$

In pratica, questo vincolo impone che la mantissa debba essere nella forma $(1.\ldots)_2$. Poichè, per

la condizione di normalizzazione, il primo bit prima della virgola è sempre 1, possiamo limitarci a rappresentare solo i bit che seguono. Dunque, con k_m bit riservati per mantissa, possiamo rappresentarne valori a k_m+1 bit! In altre parole, adottare questa forma di normalizzazione corrisponde a utilizzare la seguente forma per un generico razionale *x*:

$$x = sgn(x)\cdot(1+\tilde{m})\cdot 2^e \quad \text{con} \quad (0)_{10} \le \tilde{m} < (1)_{10} \tag{11}$$

Sebbene questa particolare condizione di normalizzazione determini rappresentazioni vantaggiose, ha anche la spiacevole conseguenza di rendere non rappresentabile lo zero. Per questo motivo, in pratica si utilizzano sequenze di bit particolari da interpretare come "zero".

Esempio

Dobbiamo rappresentare il numero binario $1.1010001\times 2^{10100}$ su 16 bit, di cui 7 per l'esponente e 8 per la mantissa.. Il numero si trova già in forma normalizzata.
Bit di segno: 0
La polarizzazione si intende pari a $2^6 - 1 = (0111111)_2$, perciò la rappresentazione dell'esponente sarà
$(0010100)_2 + (0111111)_2 = (1010011)_2$
Della mantissa si memorizzano i bit a partire dalla seconda cifra della parte frazionaria.
Il risultato è:

0	1	0	1	0	0	1	1	1	0	1	0	0	0	1	0
+	esponente							mantissa							

Esempio

Dobbiamo rappresentare in virgola mobile il numero $(+13.75)_{10}$ su 16 bit, di cui 7 per l'esponente e 8 per la mantissa.

Il risultato può essere ottenuto seguendo i seguenti passi.

1) Bit di segno: 0
2) Conversione in binario della parte intera: $(13)_{10} = (1101)_2$
3) Conversione in binario della parte frazionaria: $(0.75)_{10} = (0.11)_2$
4) Normalizzazione: $(1101.11)_2\times 2^0 = (1.10111)_2\times 2^3$
5) Calcolo esponente polarizzato: $3 + 63 = 66 = (1000010)_2$
6) Della mantissa si memorizzano i bit dopo la virgola.

Il risultato è:

0	1	0	0	0	0	1	0	1	0	1	1	1	0	0	0
+	esponente							mantissa							

Vale la pena osservare che, con questa codifica, l'intervallo di rappresentabilità è dunque $[-(2-2^{-km})\cdot 2^{ke-1},\ +(2-2^{-km})\cdot 2^{ke-1}]$. All'interno di questo intervallo, i valori rappresentabili *non sono equispaziati*, ma, come precedentemente detto, la precisione della codifica dipende direttamente dal modulo del valore rappresentato, in modo da mantenere costante la precisione relativa sull'intervallo di rappresentabilità[7]; la precisione relativa[8] è dell'ordine di 2^{-km}. La Figura 11 permette di visualizzare, in un esempio concreto, quanto appena discusso.

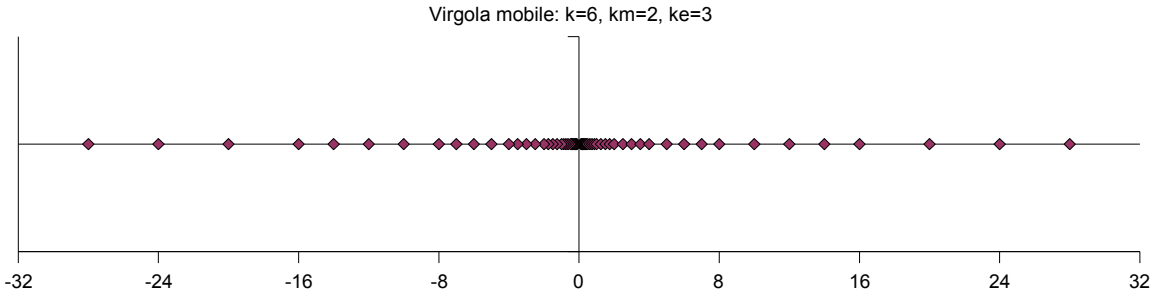


Figura 11: Valori razionali rappresentabili in modo esatto in virgola mobile (in accordo alle convenzioni adottate nel testo) su 6 bit, di cui due per la mantissa, tre per l'esponente e uno per il segno.

4.3 Lo standard IEEE 754

Per la rappresentazione in virgola mobile dei razionali nei moderni sistemi di calcolo, è stato definito uno standard che stabilisce le regole da adottare. Tale standard è noto come IEEE 754, e prevede l'utilizzo di due formati distinti: uno su 32 bit (detto "in precisione singola") e uno su 64 bit (detto "in precisione doppia").
Lo standard adotta la condizione di normalizzazione della mantissa che abbiamo descritto nella sezione precedente, e per i due formati stabilisce il numero di bit riservati a segno, esponente e mantissa, come sintetizzato nella seguente tabella:

	segno	***k_e (esponente)***	***k_m (mantissa)***	***n_{bias} (polarizz.)***
Precisione singola (32 bit)	1	8	23	$2^7-1 = 127$
Precisione doppia (64 bit)	1	11	52	$2^{11}-1 = 1023$

Per quanto riguarda l'esponente, le configurazioni composte da tutti 0 e tutti 1 non sono ammesse per la usuale rappresentazione, in quanto utilizzate per codifiche "notevoli". Dunque l'esponente può normalmente variare nell'intervallo[-126, +127] con precisione singola, e su [-1022, +1023] in precisione doppia.

7 Occorre precisare che, se indichiamo con x_m il più piccolo modulo rappresentabile in modo esatto (ovvero 2^{ke-1}), la precisione relativa non si mantiene costante nell'intorno dello zero ($-x_m$, $+x_m$); per questa ragione, talvolta si afferma impropriamente che in virgola mobile non si riescono a rappresentare i valori in ($-x_m$, $+x_m$). Se il risultato di un'operazione tra razionali in virgola mobile va a cadere in tale intervallo, si dice che si è verificata una condizione di *underflow*; se invece ricade all'esterno dell'intervallo di rappresentabilità, si parla di *overflow*.

8 A questo proposito, non si fa alcuna ipotesi sulle modalità di arrotondamento.

La seguente tabella riassume le codifiche notevoli previste dallo standard.

Significato	***Segno***	***Valore esponente***	***Valore mantissa***
0	0/1	0	0
$\pm \infty$	0/1	Tutti 1	0
NaN (not a number)	0/1	Tutti 1	$\neq 0$
Numeri denormalizzati	0/1	0	$\neq 0$

I valori "speciali" $+\infty$, $-\infty$ e *NaN* sono usati per rappresentare il risultato di operazioni con operandi non validi[9]. Un valore ∞ viene ottenuto a seguito di una condizione di overflow o di una divisione per zero. Un *NaN* si ottiene nelle divisioni 0/0, $\pm\infty/\pm\infty$, nelle moltiplicazioni $0 \cdot \pm\infty$, nelle somme $+\infty + (-\infty)$ e $-\infty + \infty$, e come risultato di funzioni in cui l'argomento sia al di fuori del dominio della specifica funzione (p.es. radice quadrata di un numero negativo, logaritmo di un numero minore o uguale a zero, arcoseno di un numero con modulo strettamente maggiore di 1, ecc.).
L'ultimo caso speciale è finalizzato a rappresentare ulteriori valori nell'intorno dello zero: la convenzione adottata è che il valore corrispondente per l'esponente sia -126 (su precisione singola) o -1022 (su precisione doppia), e che la mantissa sia della forma 0.xxxx (invece che 1.xxxx, come nel caso usuale).

IEEE 754 è adottato praticamente nella totalità dei calcolatori moderni, ad eccezione di alcuni modelli di supercomputer IBM e Cray. Al momento attuale (2007), questo standard è sottoposto a un processo di revisione che porterà, tra le altre novità, all'introduzione di ulteriori due formati, come sotto specificato.

	segno	***k_e (esponente)***	***k_m (mantissa)***	***n_{bias} (polarizz.)***
Half Precision (16 bit)	1	5	10	$2^4-1 = 15$
Quad Precision (128 bit)	1	15	112	$2^{14}-1 = 16383$

9 Si sottolinea che in alcuni applicativi software per elaborazioni matematiche, quali p.es. MATLAB, il valore *NaN* ha un significato diverso da quello qui presentato (ovvero, indica la presenza di un *valore mancante*).

5 Memorizzazione delle rappresentazioni

Tra le varie rappresentazioni proposte per i numeri naturali, interi, razionali o reali, quelle che assumono particolare importanza nel campo dell'informatica adottano tutte la base $\beta = 2$; la codifica viene comunque detta *binaria*, e le singole cifre prendono il nome di *bit*. Dunque, la rappresentazione di un numero corrisponderà sempre a una *sequenza ordinata* di bit. Per poter operare su numeri, i calcolatori devono accedere alle loro rappresentazioni, che saranno mantenute in appositi moduli dette "memorie". Le operazioni fondamentali che si possono fare nei confronti di una porzione di memoria sono due: *lettura* e *scrittura*.
L'elemento fondamentale di una memoria è una cella elementare con la capacità di un solo bit; da un punto di vista logico, la memoria può essere vista come una sequenza ordinata di bit. Allo scopo di facilitare la gestione dei contenuti della memoria, la sequenza dei bit viene suddivisa in gruppi di otto bit ciascuno: tali raggruppamenti sono detti *byte*. Detto questo, la memoria può essere considerata come una sequenza ordinata di *locazioni*, ciascuna delle quali contiene un byte, e all'interno di ciascun byte è presente una sequenza ordinata di otto bit. I byte sono considerati come l'elemento *basilare* di memorizzazione: ovvero, in qualsiasi calcolatore moderno, ogni singola operazione di lettura o scrittura prevede sempre di accedere ad almeno un byte, e mai a un numero di bit contigui inferiore a otto. Per poter riferire uno specifico byte in memoria, si ricorre alla numerazione progressiva delle locazioni di memoria, partendo da zero. Il numero che esprime la posizione della locazione viene detto *indirizzo* della cella[10].

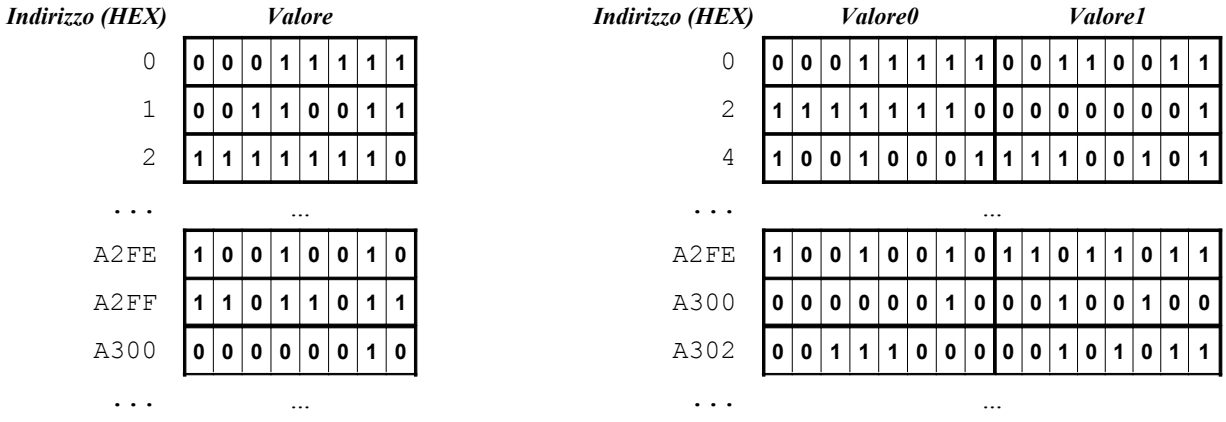


Figura 12: Memoria organizzata come: (a) sequenza di byte;(b) sequenza di coppie di byte.

Alcuni calcolatori sono in grado di accedere contemporaneamente a due, quattro, otto byte; pertanto in questi casi può essere utile immaginare la memoria organizzata a gruppi di due, quattro, otto byte.
Il raggruppamento di bit/byte che un calcolatore può maneggiare all'interno di una sola operazione è

10 Considerando che la dimensione di una memoria può essere molto grande, per consuetudine gli indirizzi vengono espessi in esadecimale, essendo tale rappresentazione più concisa di quella decimale.

detto comunemente *word*[11]. In ogni modo, i valori degli indirizzi si riferiscono sempre ai singoli byte di memoria, e mai a raggruppamenti di byte. In Figura 12 si possono vedere due esempi di semplici organizzazioni logiche della memoria: come sequenza di singoli byte (word di otto bit), o come sequenza di coppie di byte (word di sedici bit). Si precisa che, nel caso di calcolatori con word di lunghezza superiore al byte, è in genere opportuno memorizzare le rappresentazioni a partire da indirizzi multipli della dimensione in byte della word (ovvero, collocarla "su una riga"): in tal caso, si dice che il dato è "allineato sulla word".

La dimensione (capienza) della memoria di un calcolatore viene dunque espressa in numero di byte. In analogia a quanto viene fatto nel Sistema Internazionale di misura per quantità espesse con notazione decimale, tradizionalmente si usa definire i seguenti multipli del byte *secondo opportune potenze di due*:

1 *KILO*byte (kB)	$= 2^{10}$ byte	= 1024 byte
1 *MEGA*byte (MB)	$= 2^{20}$ byte	= 1024 kB = 1048576 byte
1 *GIGA*byte (GB)	$= 2^{30}$ byte	= 1024 MB = 1048576 kB = 1024 byte
1 *TERA*byte (TB)	$= 2^{40}$ byte	= 1024 GB = 1048576 MB = 1073741824 kB = ...
1 *PENTA*byte (PB)	$= 2^{50}$ byte	= 1024 TB = 1048576 GB = 1073741824 MB = ...

Vale la pena notare che nel Sistema Internazionale di misura si usano gli stessi prefissi *kilo-*, *mega-*, ecc. per riferirsi ai fattori moltiplicativi 10^3, 10^6, ecc.. In molti casi i prefissi sono usati in modo ambiguo, visto che $2^{10} \cong 10^3$, $2^{20} \cong 10^6$, ecc., per cui a rigore occorrerebbe ogni volta specificare a quale dei due fattori moltiplicativi ci si riferisce. Recentemente è stato stabilito di utilizzare il simbolo "kiB", contrazione di "kilo binary Byte" (e analogamente MiB, GiB, ...) per indicare in modo inequivocabile fattori moltiplicativi potenze di due; tale convenzione a tutt'oggi (2007) non ha però ancora preso piede.

In un calcolatore, le rappresentazioni dei numeri dovranno essere collocate all'interno di memorie che hanno la struttura logica appena descritta. La memorizzazione deve essere fatta tenendo conto di gruppi di otto bit: perciò, nel caso in cui la rappresentazione occupi più di un byte, occorre suddividerla in gruppi di otto bit ciascuno, partendo sempre dal bit meno significativo. I byte della rappresentazione così ottenuti potranno poi essere collocati in celle di memoria adiacenti.
Tra i byte che compongono una rappresentazione, specialmente nel caso di rappresentazioni di interi, si indica come MSB (most significant byte) quello più "a sinistra", ovvero quello con i bit più significativi, e si indica come LSB (most significant byte) quello più "a destra", ovvero quello con i bit meno significativi.
Volendo memorizzare la rappresentazione di un numero a partire da una certa locazione in poi, occorre scegliere se inserire i vari byte partendo da MSB oppure da LSB: si tratta semplicemente di due diverse convezioni, entrambe potenzialmente valide. La prima convenzione (che parte da MSB) è nota come *Big-endian*, mentre l'altra (che parte da LSB) è detta *Little-endian*.

11 Il termine "word" è usato anche comunemente (in modo improprio) per indicare gruppi di 16 bit; con "double word" (o Dword) si intendono gruppi di 32 bit, e con "Quad.word" (o Qword) gruppi di 64 bit.

Esempio

Prendiamo la rappresentazione in complemento a due su 32 bit del valore $(- 267242410)_{10}$: essa è
(1111 0000 0001 0010 0011 0100 0101 0110).

Visualizziamola come sequenza di byte, da MSB a LSB:
(1111 0000) (0001 0010) (0011 0100) (0101 0110)

Esprimiamo, per semplicità, il contenuto di ciascun byte in esadecimale:
$(F0)_{HEX}$ $(12)_{HEX}$ $(34)_{HEX}$ $(56)_{HEX}$

Collocando tale rappresentazione a partire dall'indirizzo $(A2FE)_{HEX}$ con la convenzione big-endian, si ha la seguente situazione in memoria (visualizzando sia l'organizzazione su singolo byte, sia quella con parola di 16 bit) :

Indirizzo	*Valore*
...	...
A2FE	**F0**
A2FF	**12**
A300	**34**
A301	**56**
...	...

Indirizzo	*Valore0*	*Valore1*
...	...	
A2FE	**F0**	**12**
A300	**34**	**56**
A302	**-**	**-**
	-	**-**
...	...	

Facendo la medesima collocazione, ma con la convenzione big-endian, si ha:

Indirizzo	*Valore*
...	...
A2FE	**56**
A2FF	**34**
A300	**12**
A301	**F0**
...	...

Indirizzo	*Valore0*	*Valore1*
...	...	
A2FE	**56**	**34**
A300	**12**	**F0**
A302	**-**	**-**
	-	**-**
...	...	

6 Nota Bibliografica

Gli argomenti presentati in questi appunti vengono trattati, in modi diversi e con diversi gradi di approfondimento, nella maggior parte dei testi sui fondamenti dell'Informatica.

Tra tutti gli altri, vale la pena menzionare il seguente:

D. A. Patterson, J.L. Hennessy: “Struttura, organizzazione e progetto dei calcolatori: indipendenza tra hardware e software” - Jackson Libri

Si tratta di un testo fondamentale e molto approfondito sull'architettura dei calcolatori. Al suo interno si possono trovare, oltre a un'esposizione dei concetti fondamentali, anche tutti i dettagli relativi alla struttura di sistemi per svolgere operazioni con le varie tipologie di rappresentazione di numeri.