La codifica dell'informazione

Cristiana Bolchini
http://home.dei.polimi.it/bolchini/docs/ri.pdf

14 settembre 2023

Questo documento raccoglie alcune nozioni relative alla rappresentazione dell'informazione utilizzabili come riferimento. Non ha in alcun modo la pretesa di essere esauriente nella trattazione di tali argomenti.

Indice

Elenco delle figure

1 | Codici, alfabeti e informazioni

Una *codifica* è un simbolo, o un gruppo di simboli, che rappresenta un'informazione, che può essere di natura numerica oppure no. L'obiettivo di questo capitolo è riflettere sul metodo seguito per arrivare a definire un codice di rappresentazione "conveniente" per un essere umano e cercare di capire in quale misura metodi analoghi ci possono portare a definire codici di rappresentazione "convenienti" per un sistema digitale di calcolo.
Il problema consiste nel voler rappresentare un insieme $\mathcal{X}$ di informazioni, avendo a disposizione un alfabeto $\mathcal{S}$ di simboli (i sette simboli utilizzati per la numerazione romana, le dieci cifre arabe, oppure le ventuno lettere dell'alfabeto italiano). Si tratta di trovare un legame univoco, o codice $\mathcal{C}$, tra un simbolo, o una sequenza di simboli, e un'informazione, che ci permetta di passare dall'uno all'altro, e viceversa.
Più formalmente, gli elementi in gioco sono i seguenti:

- l'alfabeto, o l'insieme non vuoto e finito di "simboli" tra loro distinguibili utilizzabili:

$$\mathcal{S} = \{\sigma_1, \sigma_2, \ldots, \sigma_b\}$$

- il codice, o insieme di "sequenze di simboli" – o insieme di "regole" per costruire le parole –:

$$\mathcal{C} = \{\pi_1, \pi_2, \ldots, \pi_m\}$$

  ogni π_i è costituito da una sequenza di σ_i di lunghezza finita.

- le informazioni da rappresentare, o l'insieme di dati che si desidera codificare:

$$\mathcal{X} = \{\omega_1, \omega_2, \ldots, \omega_n\}$$

Codici noti sono il codice di numerazione romana, il codice Morse, il codice di numerazione arabo, l'alfabeto italiano, e così via (Figura 1.1 ne riporta qualcuno). Ciascuno di essi ha un alfabeto finito di simboli su cui si basa ed un insieme di regole che specificano come si possono costruire sequenze valide di simboli. Per esempio, il codice Morse prevede sequenze di simboli di lunghezza variabile ma non superiore a 4 per quanto riguarda i caratteri alfabetici, senza alcuna distinzione tra caratteri minuscoli e maiuscoli; diversamente il codice di numerazione araba non pone alcun vincolo e qualsiasi sequenza di simboli è valida.

	Alfabeto $\mathcal{S}$	Sequenze di simboli valide $\mathcal{C}$
Romano	{ I, V, X, L, C, M, D }	{ I, II, IV, …, IC, … }
Morse	{ ·, – }	{ ·–, – · · ·, – – ·, … }
Arabo	{ 0, 1, 2, 3, 4, 5, 6,7, 8, 9 }	{ 11, 312, …, 9860, … }
Alfabeto min.	{ a, b, c, …, z }	{ alba, casa, … }

Figura 1.1: *Alcuni codici per la rappresentazione dell'informazione.*

Lo scopo dei codici è quello di rappresentare insiemi di informazioni, definendo quindi un legame tra la sequenza di simboli e l'informazione che si vuole rappresentare. Nell'ambito di questo documento, l'interesse

è rivolto alla codifica dell'informazione numerica e alfanumerica così come avviene all'interno di un sistema digitale di calcolo.
In prima istanza è possibile classificare i codici che verranno qui analizzati facendo una distinzione tra codici numerici e non: i primi codificano un valore numerico, più o meno complesso, i secondi vengono impiegati per codificare informazioni di natura diversa (caratteri, informazioni di controllo – come nel caso dei codici per la rilevazione d'errore –, ecc.). Naturalmente è possibile adottare diverse classificazioni, anche in relazione agli scopi che ci si prefigge; per quel che concerne questa trattazione, si farà riferimento a quella di seguito indicata.

Informazione numerica
- ◇ Numeri naturali
  - ▷ Rappresentazione posizionale
  - ▷ Codice BCD
- ◇ Numeri relativi
  - ▷ Rappresentazione in modulo e segno
  - ▷ Rappresentazione in complemento alla base diminuita b-1
  - ▷ Rappresentazione in complemento alla base b
- ◇ Numeri frazionari
  - ▷ Rappresentazione in virgola fissa
  - ▷ Rappresentazione in virgola mobile

Informazione alfanumerica
- ◇ ASCII
- ◇ UNICODE
- ◇ Morse

1.1 Lunghezza della parola di codice e rappresentabilità

Indipendentemente dal tipo di informazione che si desidera rappresentare, e dal suo significato, è importante definire degli aspetti importanti della codifica che si intende utilizzare. In primo luogo, è importante specificare la caratteristica di biunovicità del codice: una codifica è *biunivoca* o *non ambigua* se data una parola di codice ad essa corrisponde esattamente un'informazione e viceversa. In questo contesto ci riferiremo solamente a codici dotati di questa proprietà. L'altro aspetto caratteristico d'interesse è la lunghezza delle parole del codice; è possibile fare una distinzione tra codici a *lunghezza fissa* e codici a *lunghezza variabile*. Nel primo caso tutte le parole di codice hanno uguale numero di simboli, nel secondo invece ciò non avviene. Vale la pena approfondire ulteriormente questo aspetto di lunghezza delle parole di codice. In particolare, si vuole individuare il legame ben definito tra la cardinalità dell'insieme delle informazioni rappresentabili $|\mathcal{X}|$, la cardinalità dei simboli del codice $|\mathcal{S}|$ e la lunghezza delle parole di tale codice, ossia il numero di simboli di cui è composta ogni parola.

> Sia $n = |\mathcal{X}|$ la cardinalità dell'insieme di informazioni da rappresentare, $k = |\mathcal{S}|$ la cardinalità dell'insieme dei simboli del codice $\mathcal{C}$, la lunghezza minima della parole del codice è pari a:
>
> $$l = \lceil \log_k n \rceil \tag{1.1}$$
>
> in cui il simbolo $\lceil x \rceil$ indica l'intero superiore di x.

Per esempio, volendo codificare 100 informazioni utilizzando un codice il cui alfabeto è costituito da 4 simboli, ogni parola sarà costituito da $\lceil \log_4 100 \rceil = 4$ simboli; utilizzando invece un alfabeto di 6 simboli le parole avranno lunghezza 3. In realtà in entrambi i casi le sequenze rispettivamente di 4 simboli e di 3 simboli sono maggiori di quelle strettamente necessarie a rappresentare le 100 informazioni ($4^4 = 256$ e $6^3 = 216$). Nel caso in cui il numero di sequenze sia esattamente pari al numero di informazioni da rappresentare si parla di codice *a lunghezza minima*.

Si noti che la lunghezza fissa indicata definisce il numero di simboli strettamente necessari per rappresentare l'informazione; ciò non toglie che in determinati casi tale lunghezza non venga raggiunta. Per esempio, per rappresentare nel sistema decimale tutti i valori compresi tra zero e novantanove servono 2 cifre; quando però si vuole indicare il valore otto si utilizza comunemente una sola cifra, l'8 e non 08.
È intuitivo dedurre che se disponessimo di un alfabeto S dotato di un numero di simboli pari al numero di informazioni presenti in X la codifica di ogni informazione π_i avverrebbe mediante un unico simbolo, rendendo così poco "costosa" la memorizzazione delle informazioni (un simbolo per una informazione) ma molto complessa la loro manipolazione (ogni informazione è un caso speciale). L'altro caso limite è costituito da un alfabeto costituito da soli 2 simboli*, che porta a rappresentare le informazioni mediante sequenze di simboli (si vedano a titolo di esempio le figure 1.2-1.4).

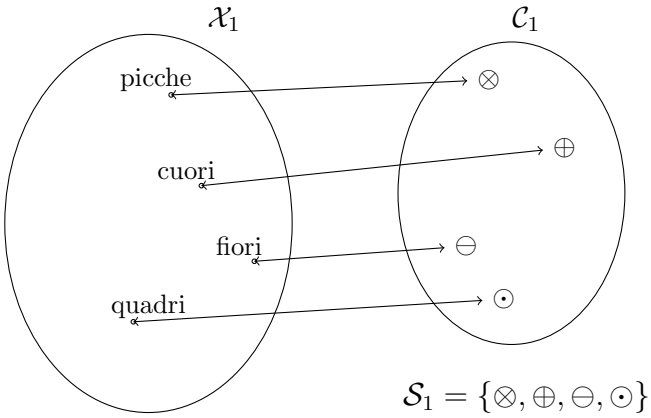


Figura 1.2: *Codice con alfabeto di 4 simboli per la codifica di 4 informazioni: codifica biunivoca, a lunghezza minima.*

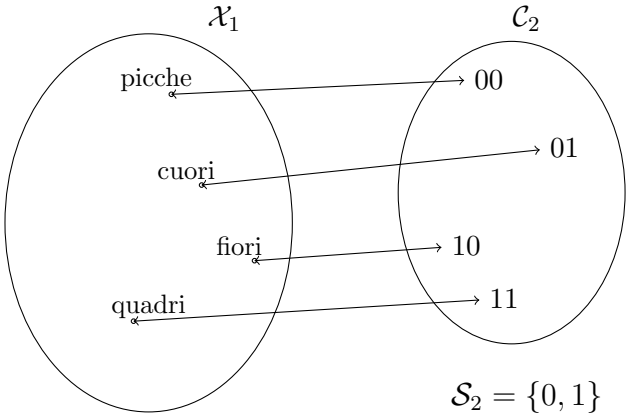


Figura 1.3: *Codice con alfabeto di 2 simboli per la codifica di 4 informazioni: codifica biunivoca, a lunghezza minima.*

La relazione (1.1) stabilisce la lunghezza minima delle parole di codice per rappresentare un insieme di informazioni. Nel caso in cui tale lunghezza sia fissata a priori ci si troverà nella condizione di poter rappresentare un insieme limitato di dati, e in tale situazione può avvenire che non esista una parola di codice per l'informazione da rappresentare, problema a cui non esiste soluzione. La cosa importante sarà in tali casi rilevare l'insorgere di una tale situazione.
Un esempio di applicazione di una rappresentazione a lunghezza fissa e prefissata è il contachilometri analogico presente su vetture un po' datate, in cui è definito il numero di cifre† utilizzate per visualizzare il numero di chilometri percorsi. Il valore iniziale è 00000 e via via che si percorrono chilometri il valore è sempre rappresentato su 5 cifre (per esempio 00135). D'altra parte, quando si superano i 99999 chilometri,

*L'alfabeto basato su un unico simbolo baserebbe la sua codifica sulla lunghezza della sequenza di simboli.

†Nell'ambito della rappresentazione dell'informazione, quando questa è di natura numerica, i simboli dell'alfabeto del codice adottato prendono il nome di *cifre*.

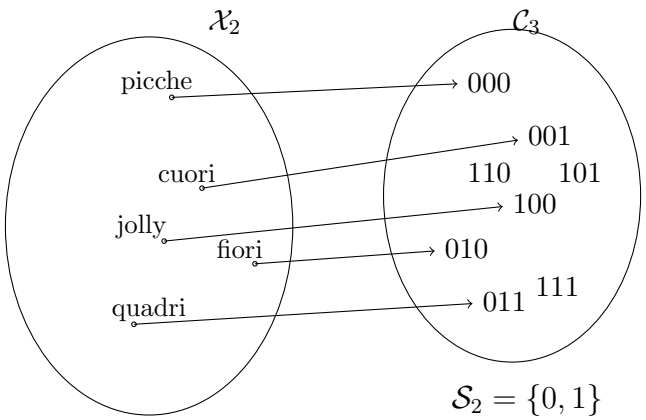


Figura 1.4: *Codice con alfabeto di 2 simboli per la codifica di 5 informazioni: codifica non a lunghezza minima e non biunivoca.*

non c'è modo di visualizzare 100000 e quindi il conteggio riparte da 00000. Si noti, come verrà ulteriormente discusso parlando di aritmetica (Sezione 3), che il valore 00000 che fa seguito a 99999 è corretto in relazione alla rappresentazione su 5 cifre. L'interpretazione che noi diamo al valore che risulta sommando ai 99999 chilometri un ulteriore chilometro (ovvero 00000) non può essere corretta, e questo errore è causato dal limite dei valori rappresentabili.

Con questa premessa possiamo quindi dire che nel momento in cui si è scelto/adottato un alfabeto di simboli è, ad esempio quello costituito dai simboli $\mathcal{S} = \{0, 1\}$, il significato di una parola che rispetta la codifica, quale ad esempio 10010011 dipende dal codice utilizzato. In seguito vedremo come tale significato varia in base a che insieme di valori si sta rappresentando.

2 | Numeri naturali

L'insieme dei numeri naturali $\mathbb{N}$ costituisce il sottoinsieme più semplice dei numeri reali $\mathbb{R}$ ed è costituito da tutti i valori interi positivi o nulli*: esso verrà analizzato per primo con riferimento a diversi sistemi di codifica e verrà utilizzato per l'introduzione dell'aritmetica.
Il sistema di numerazione considerato, e più comunemente adottato, per la rappresentazione dei numeri naturali è quello "posizionale". In questo sistema, un arbitrario valore N non negativo è rappresentato dalla sequenza di n simboli $\alpha_i \in \mathcal{S} = \{\sigma_1, \sigma_2, \ldots, \sigma_b\}$, dove $\mathcal{S}$ è l'alfabeto di simboli a disposizione.

$$N = \alpha_{n-1}\alpha_{n-2}\alpha_{n-3}\ldots\alpha_0$$

La cardinalità dell'alfabeto $b = |\mathcal{S}|$ prende il nome di *base* e costituisce un elemento importante per poter interpretare le codifiche ed il loro significato. È infatti fondamentale specificare quale alfabeto (o base) si sta utilizzando per rappresentare l'informazione; la sola codifica non sempre è sufficiente: 101 è una codifica valida in qualsiasi base, rappresenta però valori diversi nei diversi alfabeti. Di conseguenza, si associa comunemente ad una codifica anche l'indicazione del codice utilizzato per rappresentarla: in questa trattazione l'indicazione viene fatta mettendo un pedice alla codifica in modo tale che sia possibile interpretarne correttamente il significato; per esempio, 2304_8 è una codifica in base 8, 1011_2 è una codifica in base 2, e infine $A123_{16}$ è una codifica in base 16.
Ogni posizione nella rappresentazione ha un peso b_i, da cui l'attributo "posizionale", che fa assumere allo stesso simbolo in posizioni diverse valore diverso. Nei codici numerici posizionali, l'associazione codifica–valore (indicato con $v(N)$) è data dalla seguente relazione:

$$v(N) = \alpha_{n-1}b^{n-1} + \alpha_{n-2}b^{n-2} + \ldots + \alpha_0 b^0 = \sum_{i=0}^{n-1} \alpha_i b^i \tag{2.1}$$

da cui si deduce che il valore di N $v(N)$ è pari alla somma dei prodotti dei simboli α_i e del loro relativo peso b_i che nell'ambito dei numeri naturali è calcolato come potenza della base utilizzata per rappresentare l'informazione, ovvero $b_i = b^i$.
Nella rappresentazione posizionale, la cifra più a sinistra (α_{n-1}) prende il nome di *cifra più significativa* in quanto ha peso massimo, quella più a destra (α_0) di *cifra meno significativa*, in inglese rispettivamente *Most Significant Digit – MSD* e *Least Significant Digit – LSD*.
Mediante la relazione (2.1) è possibile calcolare il valore numerico corrispondente alle diverse codifiche della parola di codice 111 in relazione a varie basi, ottenendo così i seguenti valori:

111_2 $\quad valore = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \quad$ = sette
111_8 $\quad valore = 1 \cdot 8^2 + 1 \cdot 8^1 + 1 \cdot 8^0 \quad$ = settantatre
111_{10} $\quad valore = 1 \cdot 10^2 + 1 \cdot 10^1 + 1 \cdot 10^0 \quad$ = centoundici
111_{16} $\quad valore = 1 \cdot 16^2 + 1 \cdot 16^1 + 1 \cdot 16^0 \quad$ = duecentosettantatre

Come ulteriore esempio si consideri la codifica $N = 351_{10}$. Il valore del numero rappresentato è pari a:

$$N = 3 \cdot 10^2 + 5 \cdot 10^1 + 1 \cdot 10^0 \tag{2.2}$$

che corrisponde al valore numerico trecentocinquantuno.

*In alcune trattazioni l'insieme dei numeri naturali esclude lo zero.

Interpretando la stessa codifica di N come se fosse espressa nel codice base 8 ($b = 8 \rightarrow N = 351_8$) il valore numerico rappresentato in tal caso sarebbe:

$$N = 3 \cdot 8^2 + 5 \cdot 8^1 + 1 \cdot 8^0 \tag{2.3}$$

pari al valore duecentotrentatre.
Un'altra base utilizzata frequentemente è la base esadecimale ($b = 16$), il cui alfabeto di simboli è $a_i \in \mathcal{S} = \{0, 1, \ldots 9, A, B, C, D, E, F\}$. Sempre con riferimento a $N = 351$ il valore rappresentato quando interpretato come codifica in codice base 16 è pari a:

$$N = 3 \cdot 16^2 + 5 \cdot 16^1 + 1 \cdot 16^0 \tag{2.4}$$

che corrisponde al valore numerico ottocentoquarantanove.
Come ultimo esempio si consideri il sistema binario, caratterizzato da $b = 2$ e $\alpha_i \in \mathcal{S} = \{0, 1\}$. È chiaro che in questo codice la codifica $N = 351$ non è ammissibile, in quanto i simboli 3 e 5 non appartengono all'alfabeto S.
Come si può intuire, la rappresentazione basata sul sistema binario è quella meno efficiente in termini di "lunghezza" della codifica, a causa del limitato numero di simboli presenti nell'alfabeto. D'altra parte, come anche si vedrà più avanti, la convenienza di una rappresentazione rispetto ad un'altra non è legata esclusivamente alla facilità di codifica, quanto soprattutto all'efficienza nella manipolazione dell'informazione.
La Figura 2.1 riassume i dati relativi ai codici più comunemente utilizzati nell'ambito informatico in relazione alla base di rappresentazione.

Base	Codice	Alfabeto
2	Binario	$\mathcal{S} = \{0, 1\}$
8	Ottale	$\mathcal{S} = \{0, 1, 2, 3, 4, 5, 6, 7\}$
10	Decimale	$\mathcal{S} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
16	Esadecimale[†]	$\mathcal{S} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}\}$

Figura 2.1: *Basi comunemente utilizzate per la rappresentazione posizionale dell'informazione.*

La Figura 2.2 riporta i primi 16 valori numerici e la corrispondente codifica nelle basi introdotte.

Valore	Base 2	Base 8	Base 10	Base 16
zero	0000	00	00	0
uno	0001	01	01	1
due	0010	02	02	2
tre	0011	03	03	3
quattro	0100	04	04	4
cinque	0101	05	05	5
sei	0110	06	06	6
sette	0111	07	07	7
otto	1000	10	08	8
nove	1001	11	09	9
dieci	1010	12	10	A
undici	1011	13	11	B
dodici	1100	14	12	C
tredici	1101	15	13	D
quattordici	1110	16	14	E
quindici	1111	17	15	F

Figura 2.2: *La rappresentazione dei primi sedici numeri naturali nelle basi 2, 8, 10 e 16.*

A proposito della "lunghezza" della codifica, è utile vedere come sia possibile determinare il numero n di simboli necessari per rappresentare un valore numerico N mediante un alfabeto S caratterizzato da b

simboli $\{\sigma_1, \sigma_2, \ldots \sigma_b\}$. L'equazione che permette di determinare il valore di n è riconducibile alla (1.1) ed è la seguente:

$$n = \lceil ln_b(N+1) \rceil \tag{2.5}$$

dove l'operatore $\lceil x \rceil$ restituisce l'intero superiore di x.
Il motivo del $+1$ è da ricercarsi nel fatto che rappresentare il numero naturale N significa poter rappresentare tutti i valori compresi tra 0 e N, estremi inclusi, $[0, N]$, ovvero poter rappresentare $N+1$ informazioni diverse utilizzando sequenze di simboli.
Riprendendo gli esempi prima visti, si verifichi come la lunghezza delle codifiche rispetti la formula (2.5). Si consideri il valore numerico 18: la rappresentazione in base 2 richiede $n = \lceil ln_2(19) \rceil = 5$ simboli.
In modo analogo, la rappresentazione del valore numerico 849 in base 16 richiede $n = \lceil ln_{16}(850) \rceil = 3$ simboli.
Nell'ambito di questo documento l'attenzione verrà focalizzata sulla rappresentazione dell'informazione, numerica e non, utilizzando il sistema di codifica binario, in cui ogni informazione è rappresentata mediante una sequenza di cifre che assumono valore 0 oppure 1[‡].
La prossima sezione illustra come è possibile passare da un codice ad un altro, rappresentando lo stesso valore con basi b diverse.

2.1 Conversione di base

L'operazione di transcodifica, o passaggio da un codice ad un altro, quando si riferisce alla notazione posizionale prende il nome di "conversione di base", che consente di partire da un valore N rappresentato in una base b_i allo stesso valore rappresentato in una base b_j. Si consideri, a questo proposito, il seguente problema:

dato $N = 26_{10}$ *rappresentato in base 10, lo si rappresenti il base 2*

La relazione (2.1) consente di scrivere la seguente equazione, ove $\alpha_i \in \mathcal{S}_i$ per la base b_i e $\beta_i \in \mathcal{S}_j$ per la base b_j. Si tratta quindi di individuare i coefficienti β_i.

$$v(N) = \alpha_{n-1} b_i^{n-1} + \ldots + \alpha_0 b_i^0 = \beta_{m-1} b_j^{m-1} + \ldots + \beta_0 b_j^0 \tag{2.6}$$

Ci sono diversi procedimenti per effettuare questo cambio di base, tra cui due sono quelli abitualmente utilizzati: il metodo delle divisioni ripetute e il metodo della somma di prodotti, entrambi riconducibili alla relazione (2.1) e quindi alla relazione (2.6). Il primo procedimento può essere informalmente descritto come segue:

> *Il valore iniziale N viene diviso per la base b_j in cui si vuole rappresentare il valore: il resto R costituisce un elemento della codifica, mentre il quoziente della divisione intera Q viene preso come nuovo valore di partenza. Il procedimento viene ripetuto fino a quando $Q = 0$. Tutti i resti R_l ottenuti, presi in ordine inverso rispetto a quello in cui sono stati ottenuti e rappresentati nella base b_j costituiscono la codifica cercata.*

Tale procedimento deriva dalla relazione (2.1) e può essere adottato per convertire da una base b_i ad una base b_j qualsivoglia, tuttavia è comunemente adottato quando $b_i = 10$.
Con riferimento all'esempio, $N = 26_{10}$ viene diviso ripetutamente per 2 (la base in cui si desidera rappresentare il valore espressa nella base di partenza): i resti prodotti (che possono essere esclusivamente 0 o 1 e che quindi coincidono con gli analoghi valori rappresentati in base 2) presi in ordine inverso costituiscono la codifica cercata.

[‡]Ciascuna di queste cifre prende il nome di *bit*, *bi*nary digi*t*, cifra binaria.

			resti	
base	$\longrightarrow$	2	$\downarrow$	
	26		0	$(0_{10} = 0_2)$
	13		1	$(1_{10} = 1_2)$
	6		0	
	3		1	$\Uparrow$
	1		1	
	0	$\longleftarrow$	fine	

I resti presi in ordine inverso rispetto a quello con cui vengono ottenuti danno la seguente codifica: $26_{10} = 11010_2$.
Un ulteriore esempio d'applicazione del procedimento è il seguente:

dato $N = 79_{10}$ *rappresentato in base 10, lo si rappresenti in base 2*

Il primo passo consiste nel determinare la lunghezza attesa della codifica: $n = \lceil ln_2(80) \rceil = 7$ bit.

	2
79	1
39	1
19	1
9	1
4	0
2	0
1	1
0	

I resti presi in ordine inverso rispetto a quello con cui vengono ottenuti danno la seguente codifica: $26_{10} = 1001111_2$.
Ancora un esempio:

si converta $N = 143_{10}$ *rappresentato in base 10 nella base 16*

La lunghezza della codifica è pari a $n = \lceil ln_{16}(144) \rceil = 2$ cifre ed il risultato è $143_{10} = 8F_{16}$, determinato come segue.

	16	
143	15	$(15_{10} = F_{16})$
8	8	$(8_{10} = 8_{16})$
0		

È possibile fare due considerazioni che consentono di effettuare una rapida analisi sulla correttezza o meno del risultato a cui si perviene quando si lavora in base 2:

1. il bit più significativo è sempre 1;
2. i numeri dispari rappresentati in base 2 hanno sempre bit meno significativo a 1.

La prima considerazione è legata al fatto che viene utilizzato sempre il numero minimo di bit necessario a rappresentare il valore in esame. Per generalizzare al caso di base generica, è possibile dire che il bit più significativo è sempre diverso da 0.
La seconda considerazione è altrettanto intuitiva: la primissima divisione che dà luogo al primo resto e al bit meno significativo nel caso di quantità dispari ha resto 1, in caso di quantità pari dà resto 0.
Questi rapidi controlli, oltre a quello legato alla lunghezza della codifica, consentono di rilevare eventuali errori nello svolgimento delle operazioni elementari.
Si consideri ora la conversione il seguente problema, "apparentemente" inverso rispetto alle conversioni sino ad ora considerate:

dato $N = 1011_2$ rappresentato in base 2, lo si rappresenti in base 10

Per effettuare questa conversione si utilizzerà un procedimento diverso, ma sempre derivante dalla relazione (2.1), che può essere espresso in modo informale come segue:

> *Si convertono i coefficienti α_h rappresentati nella base b_i nella base b_j e li si moltiplica per base b_i (rappresentata nella base b_j) elevata alla potenza h determinata appunto dalla posizione del coefficiente nella codifica. La somma di tutti i contributi è il valore cercato.*

Di fatto questa procedura non fa altro che interpretare il significato della notazione posizionale, convertendo i singoli elementi in gioco (coefficienti e basi) nella base in cui si vuole rappresentare il valore ed effettuando prodotti e somme per calcolare il valore finale.
Con riferimento al valore in esame $N = 1011_2$ e la sua conversione in base 10 si ha:

$$v(N) = 1_{10} \cdot 2_{10}^3 + 0_{10} \cdot 2_{10}^2 + 1_{10} \cdot 2_{10}^1 + 1_{10} \cdot 2_{10}^0 = 11_{10}$$

I simboli (e i valori) α_h della base $b_i = 2$ coincidono con i simboli (e i valori) β_k della base $b_j = 10$, e la conversione risulta immediata.
Come ulteriore esempio si consideri la conversione dalla base 8 alla base 10 della codifica 103_8:

$$v(N) = 1_{10} \cdot 8_{10}^2 + 0_{10} \cdot 8_{10}^1 + 3_{10} \cdot 8_{10}^0 = 67_{10}$$

Questo procedimento risulta immediato per l'operazione di conversione da una qualsiasi base b_i alla base $b_j = 10$, ma può essere utilizzato qualsiasi sia la base b_j. Più precisamente, entrambi i procedimenti hanno validità generale e possono essere utilizzati sempre, qualsiasi siano le base b_i e b_j; concretamente però, data la nostra consuetudine a lavorare in base 10, si utilizza tipicamente il primo procedimento per conversioni da una base $b_i = 10$ ad una base b_j generica, e il secondo per conversioni da una base b_i qualsiasi alla base $b_j = 10$. Nel caso in cui la base 10 non sia coinvolta, come per esempio nella conversione di una codifica dalla base $b_i = 5$ alla base $b_j = 3$, è abitudine comune effettuare un doppio passaggio: dalla base $b_i = 5$ alla base $b_k = 10$ e poi dalla base $b_k = 10$ alla base $b_j = 3$.
Questo passaggio intermedio attraverso la base 10 può essere omesso, quando si effettua una conversione di base tra due basi tali che una è una potenza dell'altra: in questo caso è anche possibile effettuare una conversione diretta. Si consideri infatti l'esempio appena terminato di passaggio dalla base 2 alla base 8. Poiché $2^3 = 8$ la conversione può essere effettuata mediante una corrispondenza biunivoca tra valore rappresentato in base 2 e valore in base 8. Questa corrispondenza è messa in evidenza dalla Figura 2.2, da cui si vede che c'è una corrispondenza biunivoca tra la rappresentazione in base 8 e quella in base 2: ai simboli della base 8 $0, 1, \ldots, 7$ corrispondono le codifiche $000, 001, \ldots, 111$.
Grazie a questa corrispondenza biunivoca, è possibile passare da una rappresentazione all'altra: per passare dalla base 8 alla base 2 è sufficiente sostituire ad ogni cifra della codifica in base 8 la corrispondente sequenza di bit della base 2, eliminando infine tutti gli 0 in posizione più significativa prima dell'1 più significativo (in quanto inutili)[§].

$32_8 \rightarrow$

3	2
↓	↓
011	010

$\rightarrow 11010_2$

Il passaggio opposto è analogo: si raggruppano i bit della codifica in base 2 a sequenze di tre bit, a partire dal bit meno significativo e si converte ciascuna tripletta nella corrispondente cifra in base 8. Per la tripletta in posizione più significativa, nel caso abbia una dimensione inferiore ai tre bit è possibile anteporre degli 0 ("padding") che in posizione più significativa non cambiano il valore del numero naturale rappresentato.

$10011_2 \rightarrow$

010	011
↓	↓
2	3

$\rightarrow 23_8$

[§]Si ricordi che gli unici 0 che sono eliminabili sono quelli in posizione più significativa, non quelli delle singole cifre convertite.

In modo analogo è possibile effettuare un cambiamento di base diretto tra base 2 e base 16 (infatti $2^4 = 16$), pur di considerare la corrispondenza tra una cifra della base 16 ed una sequenza di quattro bit in base 2. Sempre considerando $N = 10011_2$, la conversione diretta in base 16 dà $N = 13_{16}$.

$$10011_2 \rightarrow \begin{array}{|c|c|}\hline \mathbf{000}1 & 0011 \\ \hline \downarrow & \downarrow \\ \hline 1 & 3 \\ \hline \end{array} \rightarrow 13_{16}$$

Volendo generalizzare questa procedura di passaggio diretto dalla rappresentazione in base b_i alla base b_j, ove $b_i = b_j^k$, si può dire:

> *Dato un valore rappresentato in base b_i per rappresentarlo in base b_j, ove $b_i = b_j^k$, si raggruppino le cifre del valore rappresentato in base b_i in sequenze di k cifre a partire da quelle meno significative. Se il gruppo contenente la cifra più significativa ha lunghezza inferiore a k si antepongano degli* 0 *per ottenere la lunghezza completa. Si converta quindi ogni gruppo di cifre in base b_i con la cifra corrispondente in base b_j. Ciò che si ottiene è la rappresentazione nella base b_j cercata.*

Più in dettaglio: Conversioni di base – i procedimenti

Il procedimento illustrato è valido per una conversione dalla base b_i alla base b_j, qualsiasi esse siano, sebbene non si abbia grande dimestichezza nel ragionare in basi diverse dalla base 10 comunemente adottata. L'obiettivo di questa nota è mostrare la validità generale dell'approccio. Per esempio, si converta il valore 35_{16} nelle basi 10, 8 e 2.

$$\begin{array}{rcl} A_{16} = 10_{10} & & \\ 35 & \Big| & 3(3_{16} = 3_{10}) \\ 5 & \Big| & 5(5_{16} = 5_{10}) \\ 0 & & \end{array}$$

La base $b_j = 10$ rappresentata in base 16 (base di partenza) è A_{16}. Al primo passo il valore 35_{16} viene diviso A_{16}: tale divisione (non proprio familiare) dà come risultato 5 e resto 3. A titolo di verifica si può effettuare l'operazione inversa (in base 16): $5 \times A + 3 = 35$. Procedendo fino al termine si ottiene $35_{16} = 53_{10}$. Sempre per verificare la correttezza del risultato è possibile effettuare il cambio di base opposto, da base 10 a base 16 del valore 53_{10} ottenendo appunto 35_{16}.

Per continuare l'esercizio si converta 35_{16} in base 8 con questo procedimento; in questo caso la base $b_j = 8$ rappresentata in base 16 (base di partenza) è ancora 8_{16}. il procedimento (riportato di seguito) produce il risultato $35_{16} = 65_8$.

$$\begin{array}{rcl} & 8 & \\ 35 & 5 & (5_{16} = 5_8) \\ 6 & 6 & (6_{16} = 6_8) \\ 0 & & \end{array}$$

Infine la conversione in base 2 produce il risultato $35_{16} = 110101_2$

$$\begin{array}{rcl} & 2 & \\ 35 & 1 & (1_{16} = 1_2) \\ 1\mathrm{A} & 0 & (0_{16} = 0_2) \\ \mathrm{D} & 1 & (1_{16} = 1_2) \\ 6 & 0 & (0_{16} = 0_2) \\ 3 & 1 & (1_{16} = 1_2) \\ 1 & 1 & (1_{16} = 1_2) \\ 0 & & \end{array}$$

Per generalizzare il secondo procedimento, si consideri la conversione della codifica 153_8 in base 2. In questo caso è necessario convertire sia i singoli coefficienti in base 2, sia la base stessa, ottenendo

$$\begin{aligned} v(N) &= 001_2 \cdot 1000_2^2 + 101_2 \cdot 1000_2^1 + 011_2 \cdot 1000_2^0 \\ &= 001 \cdot 1000000 + 101 \cdot 1000 + 011 \cdot 1 \\ &= 1000000_2 + 0101000_2 + 0000011_2 \\ &= 1101011_2 \end{aligned}$$

Come ulteriore esempio si converta il valore $N = 159_{10}$ in base 8 utilizzando il secondo procedimento invece del primo, al fine di verificare la generalità del metodo.

$$\begin{aligned} v(N) &= 1_8 \cdot 12_8^2 + 5_8 \cdot 12_8^1 + 11_8 \cdot 12_8^0 \\ &= 1 \cdot 144 + 5 \cdot 12 + 11 \cdot 1 \\ &= 144_8 + 62_8 + 11_8 \\ &= 237_8 \end{aligned}$$

Data la poca familiarità per le operazioni in basi diverse dalla base 10, e la facilità di errore, è abitudine comune convertire per prima cosa dalla base b_i alla base 10, quindi dalla base 10 alla base b_j. Per la prima conversione si utilizza il procedimento della somma dei prodotti per le potenze della base, per la seconda conversione si utilizza poi il procedimento delle divisioni ripetute.

2.2 Altri codici numerici

Oltre ai codici posizionali vi sono numerosi altri codice per la rappresentazione dei valori numerici; in questa trattazione viene solamente presentato il codice BCD, qui di seguito introdotto.

2.2.1 Binary-Coded Decimal – BCD

Il codice BCD si basa sulla codifica delle 10 cifre decimali mediante la codifica binaria naturale, e sulla loro giustapposizione per la rappresentazione dei valori. Più precisamente, poiché le cifre da rappresentare sono $\{0, 1, \ldots, 9\}$ è necessario utilizzare $\lceil ln_2(10) \rceil = 4$ bit, come mostrato in Figura 2.3.

Cifra decimale	Codice BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Figura 2.3: *Il codice BCD*

Volendo rappresentare il valore numerico 25_{10} nel codice BCD si giustappone la codifica della cifra $2_{10} = 0010_{BCD}$ a quella della $5_{10} = 0101_{BCD}$ ottenendo 00100101_{BCD}. Questo codice, per quanto possa sembrare poco efficiente poiché richiede la manipolazione delle singole cifre, trova applicazione nella rappresentazione di valori razionali con un numero fisso di cifre decimali, ove non si possa tollerare un'approssimazione. Un esempio di utilizzo è costituito dalla memorizzazione di importi monetari che fanno uso di centesimi (come per esempio l'euro o il dollaro), in cui il numero di cifre decimali è fisso (e pari a 2) e non si vogliono fare errori di arrotondamento. Ulteriori considerazioni in merito verranno fatte nella Sezione 5.

Riprendendo la considerazione fatta al termine della precedente sezione, alla codifica 10010011 corrispondono valori distinti in relazione alla base di rappresentazione, e all'insieme di valori che si sta rappresentando. Per cui, se 10010011 è la codifica di un valore naturale rappresentato in base 10 (che scriveremmo per maggior leggibilità come 10.010.011), il valore espresso è (in parole) dieci milioni diecimila e undici. Se invece si tratta della codifica di un valore naturale espresso in base 2, il valore espresso è centoquarantasette. Infine, se si trattasse di un naturale rappresentato mediante il codice BCD, il valore espresso sarebbe novantatre.

3 | Aritmetica binaria

Comune a qualsiasi sistema di rappresentazione dei valori numerici nel sistema binario, qua scelto come riferimento, sono le operazioni aritmetiche elementari di somma, sottrazione e prodotto (la divisione viene omessa dalla trattazione) tra numeri naturali. Tali operazioni vengono qui di seguito brevemente illustrate, con riferimento al caso più semplice di operandi a un bit, e poi estese ad operandi di dimensioni maggiori. Seguono infine alcune considerazioni sui possibili errori di rappresentazione.

3.1 Aritmetica in modulo 2^n

È importante ricordare che tutti i codici presi in considerazione sono a lunghezza fissa e, ove non indicato esplicitamente, si sottintende che le parole di codice hanno lunghezza n e pertanto tutte le operazioni aritmetiche avvengono in modulo 2^n.

Più in dettaglio: Operatore modulo

Un valore m modulo p è il resto della divisione intera m/p e cioè $m \pmod p = m - p \cdot \lfloor m/p \rfloor$. Alternativamente, si dice che $m \pmod p = q$ se esiste un valore $k \in \mathbb{Z}$ tale che $m = k \cdot p + q$ e che $0 \leq q < p$. Per esempio, si voglia calcolare $11 \pmod 4$. Secondo la prima definizione si ha:

$$11 \pmod 4 = 11 - 4 \cdot \lfloor 11/4 \rfloor = 11 - 4 \cdot \lfloor 2,75 \rfloor = 11 - 4 \cdot 2 = 3$$

La stessa operazione, svolta secondo la seconda definizione, fornisce $k = 2$ e $q = 3$ infatti risulta:

$$k \cdot p + q = 2 \cdot 4 + 3 = 11$$

e q soddisfa la disuguaglianza $0 \leq 3 < 4$. La seconda definizione risulta particolarmente utile quando è necessario calcolare il modulo di valori negativi. Come ulteriore esempio si consideri il calcolo di $-11 \pmod 4$. In questo caso si ottiene $k = -3$ e $q = 1$ poichè $-3 \cdot 4 + 1 = -11$ e q soddisfa la condizione $0 \leq 1 < 4$. Si noti, inoltre, che $\lfloor -11/4 \rfloor = -3$ e non -2 come si potrebbe erroneamente pensare. Il valore -2 non è consistente infatti con la seconda definizione poichè darebbe come risultato del modulo $q = -11 - 4 \cdot (-2) = -11 + 8 = -3$, risultato che viola la condizione per cui deve essere $0 \leq q < 4$.

3.2 Somma

Si consideri per prima cosa un'aritmetica in modulo 2^1 in cui tutte le parole, ossia i dati (operandi e risultati) sono rappresentati su 1 bit. L'operazione di somma determina due elementi: il *bit di somma* ed il *bit di riporto*, così definiti:

risultato

		A	
	$+$	0	1
B	0	0	1
	1	1	0

riporto

		A	
	$+$	0	1
B	0	0	0
	1	0	1

Operando in modulo $2^1 = 2$ il riporto non fa parte del risultato, mentre viene utilizzato quando si opera in un'aritmetica in modulo 2^n con $n > 1$, come illustrato nel seguito.
Si consideri ora l'operazione di somma tra due operandi di n bit, $A = [a_{n-1} \ldots a_1\, a_0]$ e $B = [b_{n-1} \ldots b_1\, b_0]$:

c_n	c_{n-1}	$\ldots$	c_3	c_2	c_1		
	a_{n-1}	$\ldots$	a_3	a_2	a_1	a_0	$+$
	b_{n-1}	$\ldots$	b_3	b_2	b_1	b_0	$=$
	s_{n-1}	$\ldots$	s_3	s_2	s_1	s_0	

Figura 3.1: *Schema di somma di numeri binari di n bit.*

I valori c_i sono i riporti determinati dalla somma $x_{i-1} + y_{i-1}$ dei bit di posizione precedente, e sono indicati nella prima riga; è immediato rilevare che il bit c_0 non esiste (in alcune trattazioni viene messo a 0).
Dallo schema di Figura 3.1 si deduce che potrebbero essere necessari $n + 1$ bit per poter rappresentare il risultato completo della somma; poiché però si sta utilizzando un'aritmetica in modulo 2^n anche il risultato è rappresentato su soli n bit*. Questo vincolo implica una limitazione: dati due (o più) operandi di n bit, non tutte le operazioni producono un risultato corretto; quando ciò accade si verifica una situazione di *overflow* (o *traboccamento*), come discusso più in dettaglio nella Sezione .
Il bit di riporto c_n viene quindi utilizzato per determinare la correttezza del risultato ottenuto. Infatti, nel caso in cui il bit valga 0, trascurare tale bit non comporta alcun errore (le codifiche nel sistema binario naturale 0011 e 011 e 11 hanno tutte lo stesso significato) nel valore risultante; non è così quando $c_n = 1$: in tal caso il risultato su n bit e quello su $n + 1$ bit differiscono di un 1.
Per esempio, si consideri il caso di un'aritmetica modulo 2^4, in cui tutti i valori coinvolti sono rappresentati utilizzando 4 bit, con $a = 1010$ e $b = 0011$ e se ne effettui la somma.
L'operazione viene svolta sommando bit a bit gli operandi e utilizzando i riporti: l'eventuale riporto derivante dalla somma dei bit più significativi non fa parte del risultato in quanto va oltre la dimensione dei dati (nel caso in esame tale riporto è comunque nullo), e il risultato finale è 1101.

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

Un ulteriore esempio di una somma in modulo 2^4 è la seguente, con $A = 1011$ e $B = 1100$.

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

Il risultato che si ottiene è $a + b = 0111$, con un bit di riporto in posizione più significativa pari a 1.

3.3 Sottrazione

L'operazione di sottrazione $A - B$ tra operandi a 1 bit è definita come segue:

risultato

		A	
	-	0	1
B	0	0	1
	1	1	0

prestito

		A	
	-	0	1
B	0	0	0
	1	1	0

Estendendo il caso a numeri rappresentati su un numero n generico di bit, in modo analogo a quanto fatto per la somma, si ha:

*Si tenga presente che se si utilizzasse un risultato a $n + 1$ bit e lo si sommasse ad un altro operando si potrebbe poi dover utilizzare $n + 2$ bit per l'ulteriore risultato e di questo passo si arriverebbe a dimensioni variabili, ipotesi scartata in partenza.

p_n	p_{n-1}	$\dots$	p_3	p_2	p_1		
	a_{n-1}	$\dots$	a_3	a_2	a_1	a_0	$-$
	b_{n-1}	$\dots$	b_3	b_2	b_1	b_0	$=$
	s_{n-1}	$\dots$	s_3	s_2	s_1	s_0	

Figura 3.2: *Schema di sottrazione di numeri binari di n bit.*

Gli elementi indicati nella prima riga sono i prestiti eventualmente necessari per poter effettuare la sottrazione bit a bit. Si ricordi, che per i numeri naturali la sottrazione $A-B$ è definita solo nel caso in cui $A > B$, in caso contrario il risultato che si ottiene è errato, situazione che è possibile rilevare mediante l'analisi del prestito p_n.
A titolo di esempio, si effettui la sottrazione $a-b$ in aritmetica modulo 2^4 con operandi $A = 1101$ e $B = 0110$; il risultato che si ottiene è $A - B = 0111$.

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

Per verificare la correttezza del risultato è possibile effettuare l'operazione inversa $0111 + B$ e confrontare quanto si ottiene con A.

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

Come ulteriore esempio si effettui l'operazione $A - B$ con $A = 1001$ e $B = 1010$.

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

Il risultato che si ottiene è 1111; in questo caso $B > A$, fatto indicato dalla situazione $p_5 = 1$. Anche in questo caso è possibile fare l'operazione inversa per controllare la correttezza del risultato ottenuto.

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

La somma $1111 + b$ produce il valore A, come atteso, e così come nell'operazione inversa era presente un bit di prestito $p_5 = 1$ qui c'è un bit di riporto $c_5 = 1$.
Dualmente a quanto accade nell'operazione di somma, la presenza di un bit di prestito in posizione $n + 1$ denota l'occorrenza di una situazione anomala che prende il nome di *underflow*, discussa nella sezione successiva.

3.4 Overflow

Come visto, la rappresentazione di un valore numerico richiede un certo numero di bit; quando si effettuano operazioni aritmetiche tra operandi possono accadere due cose:

1. il risultato è rappresentabile sul numero di bit a disposizione, ossia non si ha un riporto o un prestito nella posizione successiva al bit in posizione più significativa;

2. il risultato *non* è rappresentabile sul numero di bit a disposizione: sarebbe necessario aggiungere un ulteriore bit.

Nel secondo caso si verifica quello che viene definito *overflow* (o *traboccamento*), ossia un errore nel risultato dovuto all'impossibilità di rappresentare il valore numerico risultante sul numero di bit a disposizione. Poiché nulla si può fare per ampliare la finestra di valori rappresentabile, tale impossibilità di rappresentare il valore porterà a ottenere un risultato errato nell'operazione, situazione cui non si può porre rimedio: la cosa importante in questi casi è *rilevare* tale situazione.
Nel caso della sottrazione $A - B$, si verifica uno dei due seguenti casi:

- $A > B$ e il risultato è senza dubbio corretto, in quanto il valore $A - B$ è certamente inferiore ad entrambi gli operandi e quindi rappresentabile sui bit a disposizione;
- $A < B$ e quindi l'operazione *non* è definita nell'ambito dei numeri naturali, e quindi il risultato che si ottiene non è corretto.

Quando si effettuano le operazioni di somma e sottrazione, poiché un risultato viene sempre prodotto, è necessario andare a verificare se tale valore è corretto oppure no, se si è verificata una condizione di overflow o underflow o se tutto è andato a buon fine.
È importante ricordare che i risultati che si ottengono sono di fatto corretti in relazione all'aritmetica in modulo 2^n: poiché però il sistema binario è utilizzato "solo" per effettuare i calcoli mentre si è abituati a lavorare in base 10 senza alcuna limitazione sul numero di cifre a disposizione, quel che si percepisce è un errore nel risultato.
A questo scopo si considerino i valori naturali $A = 13_{10}$ e $B = 7_{10}$; si convertano entrambi in base 2 sul numero minimo di bit necessari per rappresentare entrambi gli operandi e si effettui l'operazione $A + B$.
Utilizzando i procedimenti di conversione di base (vedi Sezione 2.1) si ottiene $A = 1101$ e $B = 111$. In primo luogo si effettua un'estensione di B per avere operandi di uguale dimensione ($n = 4$) ottenendo $B = 0111$; a questo punto si può procedere con la somma.

$$\begin{array}{ccccc} {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & \\ 1 & 1 & 0 & 1 & + \\ 0 & 1 & 1 & 1 & = \\ \hline 0 & 1 & 0 & 0 & \end{array}$$

Il risultato è 0100 che convertito in base 10 è $0100_2 = 4_{10}$. D'altra parte $13_{10} + 7_{10} = 20_{10}$ che è diverso dal 4_{10} ottenuto dall'operazione svolta nel sistema binario utilizzando $n = 4$ bit. Il valore 20_{10} richiede infatti $\lceil log_2(20 + 1) \rceil = 5$ bit e quindi l'operazione non può dare risultato corretto su $n = 4$ bit. Il bit di riporto $c_5 = 1$ segnala la situazione di errore. D'altra parte, considerando invece l'aritmetica in modulo 2^4, il risultato ottenuto è:

$$20 \pmod{16} = 20 - 16 \cdot \lfloor 20/16 \rfloor = 20 - 16 \cdot \lfloor 1,25 \rfloor = 20 - 16 \cdot 1 = 4$$

che è quanto risulta dall'operazione svolta.
Come detto quindi, le operazioni di somma e sottrazione nel sistema binario sono corrette rispetto alle definizioni date in relazione all'aritmetica in modulo 2^n, le condizioni di overflow e underflow sono tali rispetto ad un sistema di partenza diverso, che si attende risultati *non* in modulo 2^n.
La modalità di rilevazione dell'overflow e dell'underflow verranno di volta in volta studiate in relazione al sistema di rappresentazione utilizzato, soprattutto per quanto converne l'insieme dei numeri relativi.

3.5 Prodotto

L'ultima operazione analizzata in questo contesto è il prodotto, per prima cosa introdotto tra operandi a 1 bit, come segue:

		A	
	×	0	1
B	0	0	0
	1	0	1

Passando al caso generale di operandi a n bit, lo schema del prodotto è quello illustrato in Figura 3.3 tra gli operandi $A = [a_{n-1} \cdots a_1\ a_0]$ ed $B = [b_{n-1} \cdots b_1\ b_0]$.

				a_{n-1}	$\cdots$	a_1	a_0	$\times$
				b_{n-1}	$\cdots$	b_1	b_0	$=$
				$a_{n-1}b_0$	$\cdots$	a_1b_0	a_0b_0	
			$a_{n-1}b_1$	$\cdots$	a_1b_1	a_0b_1		
		$\cdots$	$\cdots$	$\cdots$	$\cdots$			
	$a_{n-1}b_{n-1}$	$\cdots$	a_1b_{n-1}	a_0b_{n-1}				
q_{2n-1}	q_{2n-2}	$\cdots$	q_4	q_3	q_2	q_1	q_0	

Figura 3.3: *Schema di moltiplicazione di numeri binari di n bit.*

Il bit q_{2n-1} deriva dagli eventuali riporti prodotti dalle somme dei prodotti parziali.
Si nota che il risultato del prodotto di due parole di n bit richiede $2n$ bit, infatti in generale, il valore massimo rappresentabile su n bit è $2^n - 1$ e risulta:

$$(2^n - 1)(2^n - 1) = 2^{2n} - 2^n - 2^n + 1 > 2^{2n-1}$$

cioè un valore rappresentabile appunto su $2n$ bit.

4 | Numeri interi relativi

Nell'insieme dei numeri naturali $\mathbb{N}$, l'equazione $m + x = n$ non ha sempre soluzione (per esempio $4 + x = 2$). Tale problema ha portato alla costruzione di un nuovo insieme $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, 4, \ldots\}$, che è un'estensione propria di $\mathbb{N}$, cioè $\mathbb{N} \subset \mathbb{Z}$. Dati $m, n \in \mathbb{Z}$ si ha che $m < n$ se $n - m \in \mathbb{N}$.
La rappresentazione posizionale (2.1) deve essere a questo punto "arricchita" per poter esprimere anche la connotazione positiva o negativa del valore che si desidera rappresentare. Tre sono i codici o notazioni qui introdotti per rappresentare i numeri interi relativi: a) la rappresentazione in modulo e segno, comunemente utilizzata con riferimento alla base 10, b) la notazione in complemento alla base diminuita, che costituisce un primo passo avanti in relazione alla facilità di manipolare aritmeticamente i numeri, e infine c) la notazione in complemento alla base, utilizzata con riferimento alla base 2 per la memorizzazione e manipolazione aritmetica dei numeri nei sistemi di calcolo.
All'interno di un sistema di calcolo, l'informazione viene rappresentata nel codice base 2 o sistema binario; per quanto riguarda i valori numerici questi vengono manipolati mediante le operazioni elementari di somma, sottrazione, prodotto e divisione (anche se in questo contesto non viene presentata), e la codifica adottata attualmente deve tale scelta alla particolare efficienza con cui vengono manipolati i numeri. In realtà la definizione del sistema di codifica è indipendente dalla definizione delle operazioni utilizzate per manipolare i valori rappresentati, ciononostante la definizione stessa del sistema nasce dall'esigenza di trovare una rappresentazione tale per cui le operazioni elementari più frequenti possano essere svolte in modo estremamente efficiente. Per questo motivo, la presentazione dei sistemi di codifica dei numeri qui di seguito presentati viene subito fatta seguire dalla definizione delle operazioni aritmetiche in tale sistema, così da evidenziare vantaggi e svantaggi di ogni rappresentazione.
Poiché questo documento si focalizza sulla rappresentazione dell'informazione nell'ambito dei sistemi di calcolo (e quindi al codice binario), la trattazione è introdotta in forma generale ma subito riportata al caso di interesse del codice base 2.

4.1 Modulo e segno

Questa notazione è quella utilizzata abitualmente, con riferimento alla base 10. In questa notazione è possibile individuare due elementi distinti:

- il segno, positivo o negativo
- il valore numerico

Poiché si dispone oltre ai caratteri numerici anche di altri caratteri, il segno viene abitualmente indicato mediante i simboli + e −, a cui si fa poi seguire il valore assoluto del numero che si vuole rappresentare. Sono esempi di questa notazione +7 e -4 a indicare rispettivamente il valore positivo sette ed il valore negativo quattro.
Nella generica base b, per utilizzare solo i simboli dell'alfabeto, si utilizza convenzionalmente il valore 0 per rappresentare il segno positivo (l'equivalente del simbolo +), e il simbolo corrispondente a $b - 1$ per rappresentare il segno negativo (l'equivalente del simbolo −). Il bit di segno occupa la posizione più significativa. Con riferimento al sistema binario, il simbolo per il segno negativo è $b - 1 = 2 - 1 = 1$.

Anche in questo caso è necessario specificare quale codice si sta utilizzando per rappresentare un valore, quindi oltre a indicare la base si indica anche la notazione specifica, utilizzando MS per la notazione modulo e segno: 0101101_{2MS} indica quindi un numero relativo espresso in base 2 con la notazione modulo e segno*. Le codifiche appartenenti a questo codice sono quindi del tipo

$$N = \alpha_{n-1}\alpha_{n-2}\alpha_{n-3}\ldots\alpha_0$$

ove α_{n-1} rappresenta il segno e la restante sequenza di simboli il valore assoluto del numero.
Questo codice è caratterizzato dai seguenti aspetti:

- il valore 0 ammette una duplice codifica: $0\alpha_{n-2}\alpha_{n-3}\ldots\alpha_0$ e $b-1\alpha_{n-2}\alpha_{n-3}\ldots\alpha_0$, quindi la codifica non è univoca;
- non vengono sfruttate tutte le b^n configurazioni, ma solo $2\times b^{n-1}$†;
- la finestra di valori rappresentabili è costituita dall'intervallo chiuso $\left[-b^{n-1}, b^{n-1}\right]$;
- dato il valore N la sua rappresentazione dà origine ad una codifica di $n = \lceil \log_b(|N|+1)\rceil + 1$ cifre.

Il procedimento di conversione da una base b_i ad una base b_j può essere ricondotto a quelli visti per i numeri naturali, per quanto riguarda il valore assoluto, cui si prepone il simbolo del segno.
Per esempio, si converta il valore $N = +35_{10}$ in base 2 notazione modulo e segno. Per prima cosa viene calcolata la lunghezza della codifica da calcolare: $n = \lceil \log_b(|N|+1)\rceil + 1 = \lceil \log_2(|35|+1)\rceil + 1 = 7$ Si procede quindi alla conversione del valore assoluto di N, quindi si prepone il segno. Da quanto visto nella Sezione 2.1, $|N| = 35_{10} = 100011_2$. Poiché il valore è positivo il bit di segno è 0, da cui si ottiene $N = +35_{10} = 0100011_{2MS}$.
La conversione di un numero negativo, per esempio $N = -14_{10}$ viene fatta in modo del tutto analogo: $|N| = 14_{10} = 1110_2$; il bit di segno è 1 e il risultato finale è: $N = -14_{10} = 11110_{2MS}$.
Come esempio di conversione dalla base 2 alla base 10, si utilizzi il valore $N = 110101_{2MS}$. Per prima cosa si isola il bit di segno 1 che indica che il valore è negativo, quindi si procede alla conversione del valore assoluto $|N| = 10101_2$ che in base a quanto visto nella Sezione corrisponde a $|N| = 21_{10}$. Il risultato completo è quindi $N = 110101_{2MS} = -21_{10}$.

4.1.1 Calcolo del valore opposto e estensione in segno

Il passaggio da un numero N al suo opposto $-N$ è immediato: è sufficiente cambiare il bit di segno lasciando invariato il modulo.
Per esempio, dato $N = 110101001_{2MS}$ il valore opposto è $N = 010101001_{2MS}$; dato $N = 01101_{2MS}$ il valore opposto è $N = 11101_{2MS}$.
L'operazione di *estensione in segno* consiste nel aumentare o ridurre (nel caso ve ne siano più di quante strettamente necessarie) il numero di cifre utilizzate per rappresentare il valore. Tale operazione è necessaria nel momento in cui si sta utilizzando un'aritmetica modulo 2^n e l'operando è rappresentato con un numero m di cifre diverso da n. Quel che si fa è quindi aggiungere o eliminare cifre, senza modificare il valore rappresentato. Possono verificarsi due casi:

- $m < n$: per rappresentare il valore N sono necessarie meno cifre di quante a disposizione. Nella codifica modulo e segno, questo significa rappresentare il modulo su un numero di cifre maggiore, ossia anteporre alla codifica del modulo $n-m$ cifre 0.

*Si noti che in questo ambito un eventuale 0 in posizione più significativa non deve e non può più essere rimosso in quanto cambierebbe il valore rappresentato, essendo esso l'indicazione del segno.

†Nel caso della base 2 vengono sfruttate tutte in quanto $2^n = 2\times 2^{n-1}$, quindi l'inefficienza è limitata.

- $m > n$: in questo caso è possibile ridurre il numero di cifre utilizzate mantenendo invariato il valore rappresentato solo se ne sono state utilizzate in eccesso, in caso contrario si commette un errore dovuto al limitato numero di cifre dell'aritmetica (il valore che si vuol rappresentare è troppo grande rispetto alle cifre a disposizione $|N| > 2^n$). Quando possibile, l'operazione consiste nell'eliminare $m - n$ cifre 0 a partire dalla cifra più significativa.

Si estenda in segno il valore $N = 011001_{2MS}$ cosicché venga rappresentato con 8 bit.

Il primo bit è il bit di segno, positivo in questo caso, il modulo del valore è 11001_2: l'estensione in segno consiste nell'anteporre due bit a 0 davanti al modulo, ottenendo $|N| = 0011001_2$, da cui $N = 00011001_{2MS}$ (il bit di segno è sempre quello più a sinistra).

Si estenda in segno il valore $N = 111001_{2MS}$ cosicché venga rappresentato con 8 bit.

In questo caso il bit di segno è negativo, ma il procedimento non cambia, per cui il risultato a cui si perviene è $N = 10011001_{2MS}$.
Come ultimo esempio si consideri il seguente problema:

si rappresenti il valore $N = 1000110_{2MS}$ sul numero minimo di bit necessari

In questo caso è possibile rilevare la presenza di bit in eccesso per la rappresentazione del modulo, che infatti non ha come bit più significativo un 1. Il modulo del valore rappresentato è 000110_2, caratterizzato dall'avere tre cifre più di quante strettamente necessarie, e che quindi possono essere eliminate, ottenendo $|N| = 110_2$, da cui si ottiene infine $N = 1110_{2MS}$.

4.1.2 Aritmetica

Le operazioni aritmetiche elementari su dati rappresentati in modulo e segno possono essere ricondotte direttamente all'aritmetica binaria vista in precedenza, pur di lavorare esclusivamente sul modulo degli operandi e determinare il segno del risultato dell'operazione in base ai segni degli operandi.
Le operazioni aritmetiche elementari di somma e sottrazione richiedono un'analisi preliminare degli operandi prima di poter effettuare l'operazione richiesta: è necessario confrontare il segno e in alcuni casi anche il valore degli operandi prima di poter stabilire quale operazione debba essere effettivamente realizzata.

Somma algebrica

Si consideri l'operazione $A_{2MS} + B_{2MS}$ tra operandi rappresentati in modulo e segno; l'operazione può essere realizzata mediante un'operazione di somma nel caso in cui gli operandi siano di segno concorde, mediante un'operazione di sottrazione in caso contrario. Più precisamente, possono verificarsi le seguenti situazioni:

1. $A > 0$,$B > 0$: si effettua l'operazione $|A| + |B|$ come visto per l'aritmetica binaria per valori naturali, mettendo poi al risultato il segno positivo;
2. $A < 0$,$B < 0$: si effettua l'operazione $|A| + |B|$, mettendo poi al risultato il segno negativo;
3. $A > 0$, $B < 0$: si confrontano i moduli e si hanno i seguenti casi:
   - $|A| > |B|$: si effettua l'operazione $|A| - |B|$, mettendo al risultato il segno positivo;
   - $|A| < |B|$: si effettua l'operazione $|B| - |A|$, mettendo al risultato il segno negativo;
4. $A < 0$, $B > 0$: si confrontano i moduli e si hanno i seguenti casi:
   - $|A| > |B|$: si effettua l'operazione $|A| - |B|$, mettendo al risultato il segno negativo;
   - $|A| < |B|$: si effettua l'operazione $|B| - |A|$, mettendo al risultato il segno positivo;

Nei primi due casi il risultato dell'operazione $|A| + |B|$ che si ottiene applicando le regole per l'aritmetica per valori naturali è senz'altro corretto nell'aritmetica *modulo* 2^n; nel caso in cui però il valore risultante sia tale per cui $|A| + |B| > 2^n$ si avrà overflow. Negli altri due casi, con operandi discordi, l'operazione di sottrazione produce il risultato corretto, sia nell'aritmetica *modulo* 2^n, sia rispetto al valore finale, in quanto si sottrae sempre dal valore maggiore il valore minore.
L'operazione di sottrazione $A - B$ viene ricondotta all'operazione di somma algebrica $A + (-B)$, cui si applicano le regole viste prima.
È intuitivo rilevare come le operazioni aritmetiche per la codifica modulo e segno non siano immediate in quanto richiedono un'analisi preliminare del segno per poter stabilire che tipo di operazione debba essere svolta. Le operazioni sono di due tipi diversi, somma e sottrazione, richiedendo quindi dei componenti hardware di natura diversa (o un unico modulo in grado di effettuare entrambi i tipi di operazioni).

Prodotto

Si consideri ora il prodotto tra due operandi $A_{2MS} \times B_{2MS}$. Anche in questo caso ci si riferisce al prodotto tra i valori in modulo, determinando poi il segno del risultato; in particolare, i casi sono i seguenti:

1. A_{2MS} e B_{2MS} discordi: si effettua l'operazione $|A| \times |B|$, mettendo al risultato il segno negativo;
2. A_{2MS} e B_{2MS} concordi: si effettua l'operazione $|A| \times |B|$, mettendo al risultato il segno positivo.

Il risultato per essere rappresentato correttamente richiede $2^{n-1} \times 2 + 1$ bit.
Le operazione aritmetiche per operandi rappresentati in modulo e segno vengono ricondotte alle operazioni per numeri naturali considerando i soli valori assoluti (i moduli) e determinando il segno del risultato in modo del tutto indipendente. Caratteristica di questo codice è la netta separazione tra valore rappresentato e informazione sulla sua positività o negatività, che si riflette poi sulla manipolazione dei dati.
Al fine di trovare un'alternativa alla complessità dell'algoritmo e dell'hardware necessario per poter effettuare il calcolo, è stata studiata una codifica diversa per la rappresentazione dei numeri relativi, che include l'informazione della positività o negatività di un numero relativo direttamente all'interno della codifica, come parte integrante e non nettamente separata, in modo tale che sia possibile effettuare le operazioni senza dover tenere esplicitamente conto del segno degli operandi.
Due sono le notazioni caratterizzate da queste proprietà, che prendono il nome di *notazione in complemento*, di seguito presentate.

4.2 Complemento alla base diminuita $b - 1$

La notazione in complemento a base diminuita (o complemento a $b-1$) può essere vista come il primo passo verso una notazione che includa nella codifica del valore numerico anche l'informazione del segno. L'esigenza è quella di identificare un modo efficiente di rappresentare i numeri cosi che le operazioni aritmetiche elementari possano essere realizzate in modo immediato, prescindendo dal "segno" degli operandi.
Si consideri sempre un'aritmetica modulo b^n, con codifiche di n cifre: le configurazioni a disposizione sono b^n, da $00\ldots00$ a $b-1b-1\ldots b-1b-1$; poiché anche questo codice ha una duplica codifica per il valore nullo, l'intervallo dei valori rappresentabili è $\left[-\frac{b^n}{2}+1; +\frac{b^n}{2}-1\right]$.
La codifica definisce quindi il legame che sussiste tra i valori da rappresentare e le configurazioni disponibili.
Per quanto riguarda i valori positivi si mantiene la notazione posizionale vista in precedenza, con codifiche di n cifre.
Per i numeri positivi si adotta la codifica utilizzata nella codifica in modulo e segno, con n bit (in questo modo il bit più significativo non è sempre 0). La codifica dei numeri negativi fa riferimento alla definizione del legame tra un valore $N_{b,Cb-1}$ ed il suo valore opposto $-N_{b,Cb-1}$, che è:

$$(-N)_{b,Cb-1} = b^n - 1 - N_{b,Cb-1} \tag{4.1}$$

Si noti che $N + (-N) = b^n - 1$, ossia la base diminuita, da cui il nome della codifica, che risulta essere completamente definita.
Questo codice è caratterizzato dai seguenti aspetti:

c_n	c_{n-1}	$\dots$	c_3	c_2	c_1			
	a_{n-1}	$\dots$	a_3	a_2	a_1	a_0	$+$	N
	b_{n-1}	$\dots$	b_3	b_2	b_1	b_0	$=$	$-N$
	$b-1$	$\dots$	$b-1$	$b-1$	$b-1$	$b-1$		(b^n-1)

Figura 4.1: *Definizione del valore opposto in complemento alla base diminuita.*

- il valore 0 ammette una duplice codifica: $0\alpha_{n-2}\alpha_{n-3}\dots\alpha_0$ e $b-1b-1b-1\dots b-1$, quindi la codifica non è univoca;
- vengono sfruttate tutte le b^n configurazioni;
- la finestra di valori rappresentabili è costituita dall'intervallo chiuso $\left[-\frac{b^n-2}{2};+\frac{b^n-2}{2}\right]$ nel caso di una base pari, dall'intervallo chiuso $[-\frac{b^n-1}{2};+\frac{b^n-1}{2}]$ nel caso di base dispari, ovvero la configurazione addizionale viene convenzionalmente utilizzata per un valore negativo;
- dato il valore N la sua rappresentazione dà origine ad una codifica di $n = \lceil \log_b(|N|+1) \rceil + 1$ cifre.

Il procedimento di conversione da una base b_i alla base b_j in complemento a $b_j - 1$ può essere fatto in modo diretto per valori positivi, mentre per valori negativi è necessario calcolare prima il valore opposto (e positivo) quindi effettuarne il complemento.

4.2.1 Calcolo del valore opposto

Il procedimento operativo per calcolare dato un valore N il suo opposto $-N$ è il seguente:

> *La codifica di $-N$ si ottiene sostituendo ogni cifra con il suo complemento alla base diminuita, ossia quel valore che sommato alla cifra dà come risultato $b-1$. Il procedimento prescinde dal segno di N.*

Poiché in questa trattazione ciò che interessa è il sistema binario, nel seguito le considerazioni fatte vengono esposte in relazione al codice base 2, complemento a 1 (pari a $b-1$), senza che questo leda la generalità[‡].

4.2.2 Codice base 2 complemento a 1

In modo informale, la relazione (4.1) indica che sommando a N il suo valore opposto si ottiene la configurazione $b-1b-1\cdots b-1b-1$, ovvero $11\cdots11$.
In questa base, considerata un'aritmetica modulo 2^n l'intervallo dei valori rappresentabili è $\left[-2^{n-1}+1;+2^{n-1}-1\right]$, simmetrico rispetto allo 0 (con doppia rappresentazione). Poiché i simboli sono solamente 2, si verifica una situazione tale per cui tutti i numeri positivi hanno bit più significativo 0, tutti i numeri negativi hanno bit più significativo 1. È fondamentale ricordare che questo bit, per quanto consenta di determinare la positività o meno di un valore, *non è* un bit di segno ma parte integrante della codifica.
Come esempio di rappresentazione di un numero in base 2 notazione complemento a 1, si risolva il seguente problema:

si converta $N = +25$ in base 2 complemento a 1 sul numero minimo di bit necessari

Il numero di bit necessari a rappresentare il valore è $n = \lceil \log_2(|25|+1) \rceil + 1 = 6$. Poiché N è un numero positivo, la sua codifica è analoga a quella in modulo e segno, ovvero $N = +25_{B10} = 011001_{2C1}$.
Volendo calcolare il valore opposto $-N$ è possibile procedere come indicato: in questa base il complemento di 0 è 1 e viceversa, quindi dato $N = 011001_{2C1}$ si ottiene $-N = 100110_{2C1}$.

[‡]Per ricostruire la forma generale di quanto esposto è sufficiente sostituire a 2 la generica base b e a 1 il suo valore diminuito $b-1$.

La notazione in complemento alla base diminuita ha in questo contesto lo scopo di illustrare "l'evoluzione" della codifica alla ricerca di una soluzione più efficiente per quanto concerne la manipolazione dei dati, senza approfondire tutti gli aspetti aritmetici. Per questo motivo ci si limita all'analisi della somma algebrica, senza affrontare invece l'operazione prodotto.

4.2.3 Aritmetica

Le operazioni elementari possono essere ricondotte a quelle definite per i numeri naturali per operandi ad una o più cifre, introducendo però degli opportuni accorgimenti per ottenere il risultato corretto rispetto sia all'aritmetica modulo 2^n sia al significato del risultato rispetto al valore calcolato.

Somma algebrica

Si consideri l'operazione $A_{2,C1} + B_{2,C1}$, con operandi rappresentati in base 2 complemento a 1; a differenza della notazione in modulo e segno, in questa codifica si effettua esclusivamente un'operazione di addizione, qualsiasi sia il valore degli operandi. Si considerino infatti le situazioni possibili:

- $A > 0,\ B > 0$: si opera su valori la cui rappresentazione è uguale a quella in modulo e segno e il risultato è immediatamente corretto.
- $A > 0,\ B < 0$: in questo caso l'operazione diviene $A + (2^n - 1) - B$ ed il risultato corretto atteso è $(A - B) + (2^n - 1)$. I casi possibili sono due:
  - $|B| > |A|$: si ottiene $(2^n - 1) - (B - A)$ che rappresenta la codifica in complemento 1 del numero *negativo* $-(B - A)$; il risultato che si ottiene è corretto.
  - $|A| > |B|$: si ottiene

$$\begin{aligned}(2^n - 1) - (B - A) &= (2^n - 1) + (A - B) \\ &= 2^n + (A - B - 1)\end{aligned}$$

    in cui il risultato *non* è corretto e differisce di una unità rispetto a quanto atteso $(2^n + (A - B))$, inoltre l'addizione è caratterizzata da un bit di riporto in posizione $n + 1$ (come indicato dalla presenza del termine 2^n). Sommando tale valore del riporto al risultato della prima addizione (e non considerando il riporto) si ottiene il risultato complessivo corretto.
- $A < 0,\ B > 0$: analogo al caso precedente.
- $A < 0,\ B < 0$: il risultato della somma è:

$$\begin{aligned}A + B &= (2^n - 1) - A + (2^n - 1) - B \\ &= (2^n - 1) - (A + B) + (2^n - 1) \\ &= 2^n + (2^n - 1) - (A + B - 1)\end{aligned}$$

  Anche in questo caso il risultato ottenuto $2^n + (2^n - 1) - (A + B - 1)$ differisce dal risultato corretto $(2^n - 1) - (A + B)$ di una unità (-1) ed è presente bit di riporto (valore 2^n). Come nel caso precedente, per ottenere il risultato corretto è necessario sommare una unità e trascurare il riporto.

Dallo studio delle casistiche è possibile dedurre che ogniqualvolta la prima addizione produce un riporto oltre le dimensioni degli operandi (in posizione n) è necessario effettuare una seconda addizione per aggiungere al primo risultato *parziale* proprio il valore del riporto. D'altra parte, se si effettua tale seconda addizione anche quando il riporto è nullo, sommando una quantità nulla il risultato finale è corretto.
Ne consegue che nella rappresentazione in complemento alla base diminuita (e in particolare in base 2 complemento a 1) data l'operazione $A_{b,Cb-1} + B_{b,Cb-1}$ si realizzano due addizioni, nella seconda delle quali si somma al risultato parziale il valore del riporto di tale operazione. Mediante due addizioni si realizza quindi

qualsiasi somma algebrica. Anche l'operazione di sottrazione $A_{b,Cb-1} - B_{b,Cb-1}$ si riconduce alla somma algebrica, effettuando l'operazione $A_{b,Cb-1} + (-B_{b,Cb-1})$, dopo aver calcolato il valore opposto $-B_{b,Cb-1}$. Con questo codice non è necessario disporre né di un sottrattore né di un comparatore di moduli.
Per applicare quanto visto, si risolva il seguente problema:

dati $A = 010111_{2,C1}$ *e* $B = 111000_{2,C1}$ *si effettuino le operazioni* $A + B$ *e* $A - B$

Gli operandi sono già rappresentati entrambi in base 2 complemento a 1 e sono entrambi di 6 bit ciascuno. È quindi possibile procedere con l'operazione.

1	1	0	0	0	0			
	0	1	0	1	1	1	+	(A)
	1	1	1	0	0	0	=	(B)
	0	0	1	1	1	1		*parziale*
0	0	1	1	1	1			
	0	0	1	1	1	1	+	*parziale*
	0	0	0	0	0	1	=	*riporto*
	0	1	0	0	0	0		$A+B$

Il risultato è $010000_{2,C1}$ ed è corretto. Ora si procede al calcolo di $-B$ per poter effettuare l'operazione $A+(-B)$; in questa rappresentazione dato $B = 111000_{2,C1}$ si ha $-B = 000111_{2,C1}$ ottenuto complementando bit a bit la codifica di B. L'operazione, qui di seguito riportata dà come risultato $011110_{2,C1}$.

0	0	0	1	1	1			
	0	1	0	1	1	1	+	(A)
	0	0	0	1	1	1	=	$(-B)$
	0	1	1	1	1	0		*parziale*
0	0	0	0	0	1			
	0	1	1	1	1	0	+	*parziale*
	0	0	0	0	0	0	=	*riporto*
	0	1	1	1	1	0		$A-B$

Poiché la prima addizione ha un riporto nullo, la seconda addizione non è necessaria ed il risultato ottenuto è corretto così[§].

4.3 Complemento alla base b

Questa notazione costituisce l'ulteriore passo avanti nella ricerca di una rappresentazione dei numeri interi relativi che semplifichi la loro manipolazione; in particolare si vuole ridurre le operazioni $A + B$ e $A - B$ ad una unica operazione di somma algebrica. Per raggiungere tale scopo il codice base b notazione complemento alla base b definisce il seguente legame tra valore N e valore opposto $-N$:

$$(-N)_{b,Cb-2} = b^n - N_{b,Cb-2} \tag{4.2}$$

Si noti che $N_{b,Cb-2} + (-N)_{b,Cb-2} = b^n$, ossia la base elevata alla potenza n, da cui il nome della codifica, che risulta essere completamente definita.
È importante sottolineare che $N_{b,Cb-2} + (-N)_{b,Cb-2}$ non è uguale a 0, bensì è pari a $100\cdots00$. Infatti, si considerino due operandi ad una cifra, in qualsiasi base: non esiste la possibilità che la somma di tali valori dia come risultato 0 senza un riporto che va a creare la seconda cifra. Più precisamente, se N ha una lunghezza pari a $n-1$ cifre, la somma $N_{b,Cb-2} + (-N)_{b,Cb-2} = b^n = b - 100\cdots00$ produce un risultato su n cifre, costituito da $n-1$ cifre 0 e da una ulteriore cifra in posizione $n+1$ (iniziando a numerare da 1).

[§]In alcune realizzazioni hardware è più conveniente effettuare comunque sempre la seconda operazione senza dover differenziare il comportamento in base al valore del riporto.

c_n	c_{n-1}	$\dots$	c_3	c_2	c_1			
	a_{n-1}	$\dots$	a_3	a_2	a_1	a_0	$+$	N
	b_{n-1}	$\dots$	b_3	b_2	b_1	b_0	$=$	$-N$
1	0	$\dots$	0	0	0	0		(b^n)

Figura 4.2: *Definizione del valore opposto in complemento alla base.*

Come si può vedere, l'eventuale riporto in posizione n esiste per costruzione, poiché però questo bit va oltre le dimensioni dei dati nell'aritmetica modulo 2^n esso verrà sistematicamente trascurato come se non ci fosse. Questo comportamento è giustificato dall'analisi dei casi che possono insorgere nella definizione dell'operazione $A + B$, discussa tra breve.
Questo codice è caratterizzato dai seguenti aspetti:

- il valore 0 ammette una unica codifica: $00 \cdots 00$, pertanto la codifica è univoca;
- vengono sfruttate tutte le b^n configurazioni;
- la finestra di valori rappresentabili è costituita dall'intervallo chiuso $[-\frac{b^n}{2}; +\frac{b^n}{2} - 1]$ nel caso di una base pari, dall'intervallo chiuso $[-\frac{b^n-1}{2}; +\frac{b^n-1}{2}]$ nel caso di base dispari, ossia la configurazione addizionale viene convenzionalmente utilizzata per un valore negativo;
- dato il valore N la sua rappresentazione dà origine ad una codifica di $n = \lceil \log_b (|N| + 1) \rceil + 1$ cifre.

Il procedimento di conversione da una base b_i alla base b_j in complemento alla base b_j può essere fatto in modo diretto per valori positivi (che anche in questo caso hanno una codifica identica a quella in modulo e segno e a quella in complemento alla base diminuita), mentre per valori negativi è necessario calcolare prima il valore opposto (e positivo) quindi effettuarne il complemento.

4.3.1 Calcolo del valore opposto

Il procedimento operativo per calcolare dato un valore N il suo opposto $-N$ è il seguente:

> *La codifica di $-N$ si ottiene sostituendo ogni cifra con il suo complemento alla base, ossia quel valore che sommato alla cifra dà come risultato* 0*. Poiché può crearsi una catena di riporti, quello che va oltre la dimensione degli operandi, se esiste, viene trascurato. Il procedimento prescinde dal segno di N.*

Poiché in questa trattazione ciò che interessa è il sistema binario, nel seguito le considerazioni fatte vengono esposte in relazione al codice base 2, complemento a 2 (pari a b), senza che questo leda la validità generale di quanto esposto.
Per effettuare il calcolo del valore opposto, si utilizzano operativamente i seguenti procedimenti:

1. Procedendo dal bit meno significativo verso quello pi$\check{}$ significativo, si lasciano invariati tutti i bit fino al primo 1 incluso, quindi si complementano tutti gli altri.
2. Si complementa bit a bit il valore di partenza (complemento a 1), quindi si somma $0 \dots 01$ al valore risultante.

4.3.2 Codice base 2 complemento a 2

In modo informale, la relazione (4.2) indica che sommando a N il suo valore opposto si ottiene la configurazione $100 \cdots 00$.
In questa base, considerata un'aritmetica modulo 2^n l'intervallo dei valori rappresentabili è $\left[-(2^{n-1}; +2^{n-1} - 1\right]$, asimmetrico rispetto allo 0, che ha una unica rappresentazione $(00 \cdots 00)$. Anche in questo caso, poiché i

simboli sono solamente 2, si verifica una situazione tale per cui tutti i numeri positivi hanno bit più significativo 0, tutti i numeri negativi hanno bit più significativo 1. Anche in questo caso è fondamentale ricordare che questo bit, per quanto consenta di determinare la positività o meno di un valore, *non è* un bit di segno ma parte integrante della codifica.
Come esempio di rappresentazione di un numero in base 2 notazione complemento a 2, si risolva il seguente problema:

si converta $N = +25_{10}$ *in base 2 complemento a 2 sul numero minimo di bit necessari*

Il numero di bit necessari a rappresentare il valore è $n = \lceil \log_2 (|+25|+1) \rceil + 1 = 6$. Come fatto per la notazione in complemento a 1, anche per questo codice la codifica di un numero positivo è analoga a quella in modulo e segno, ovvero $N = +25_{B10} = 011001_{2C1}$.
Volendo calcolare il valore opposto $-N$ è possibile procedere come indicato: per questo esempio si utilizzerà il procedimento di calcolare prima il complemento a 1 di N, quindi si somma il valore 1. Pertanto, dato $N = 011001_{2C2}$ si ottiene $-N = 100110 + 000001 = 100111_{2C2}$. Allo stesso risultato si perviene utilizzando uno degli altri metodi.
Si consideri ora un altro esempio di conversione di un valore negativo, ed in particolare si risolva il seguente problema:

si converta $N = -13_{10}$ *in base 2 complemento a 2 sul numero minimo di bit necessari*

In primo luogo si valuta il numero di bit necessari per la codifica: $n = \lceil \log_2 (|-13|+1) \rceil + 1 = 5$. Per determinare la codifica del valore, essendo esso negativo, si determina prima la codifica del suo valore opposto ottenendo così $-N = +13_{10} = 01101_{2MS} = 01101_{2C2}$. Il passo finale consiste nel ricavare l'opposto della codifica appena trovata, ossia 10011_{2C2} che è il valore cercato. Quindi, per riassumere, si ha $N = -13_{10} = 10011_{2C2}$.

4.3.3 Estensione in segno

Si consideri il seguente problema:

si converta $N = +6_{10}$ *in base 2 complemento a 2 sul 10 bit*

La codifica di questo valore in base 2 complemento a 2 richiede esattamente $n = \lceil \log_2 (|+6|+1) \rceil + 1 = 4$ bit, ed il risultato che si ottiene è $N = 0110_{2C2}$. Voler rappresentare N utilizzando più cifre implica mantenere invariato il suo valore: trattandosi di un numero positivo la sua codifica è uguale a quella in modulo e segno, per cui si inizia l'analisi da lì. Per rappresentare $N = 0110_{2MS}$ mediante 10 cifre, si "allunga" la codifica del solo modulo aggiungendo il numero di 0 necessari a raggiungere la dimensione desiderata; in questo caso si ottiene $N = 0000000110_{2MS}$.
La codifica in complemento a 2 è uguale a quella in modulo e segno, per valori positivi, quindi si ottiene $N = 0000000110_{2MS} = 0000000110_{2C2}$. Si calcoli ora l'opposto di N in complemento a 2, utilizzando uno dei procedimenti proposti, a partire dalla codifica rappresentata su 10 bit. Il risultato a cui si perviene è $N = 1111111010_{2C2}$, mentre lo stesso valore rappresentato sul numero minimo di cifre è $N = 1010_{2C2}$.
Si deduce che l'operazione di "estensione in segno" in complemento a 2 può essere realizzata mediante la procedura di seguito riportata:

> *L'estensione in segno di un numero in complemento a 2 si effettua ripetendo il bit più significativo tante volte quante necessarie a raggiungere la dimensione desiderata.*

L'operazione inversa, ossia il passaggio da una codifica con un numero di cifre ridondante a quella sul numero minimo di bit, consiste nell'eliminare tutti i bit più significativi uguali tra loro ad eccezione dell'ultimo. Per esempio, la codifica 11101011_{2C2} può essere ridotta senza modificare il valore rappresentato in quanto il bit più significativo ha valore uguale ai due bit precedenti: di questi tre bit uno solo è necessario; è quindi possibile ridurre la dimensione della codifica ottenendo 101011_{2C2} che rappresenta lo stesso valore. Si faccia attenzione a non eliminare *tutti* e tre gli 1, ottenendo in modo errato 01011: il valore ottenuto è un valore positivo, quello di partenza è un valore negativo, quindi l'operazione non è stata svolta correttamente.

4.3.4 Aritmetica

L'aritmetica in complemento a 2 si rifa all'aritmetica vista per i numeri naturali; si tratta quindi di verificare come tale aritmetica consenta di ottenere i risultati corretti rispetto all'aritmetica modulo 2^n che è alla base della codifica, e come determinare eventuali condizioni di overflow.

Somma algebrica

Si consideri l'operazione $A_{2C2} + B_{2C2}$; in base al valore degli operandi e con riferimento alla definizione del valore opposto (4.2) possono verificarsi le seguenti situazioni:

- $A > 0,\ B > 0$: non ci sono riporti e il risultato è immediatamente corretto.
- $A > 0,\ B < 0$: in questo caso l'operazione diviene $A + 2^n - B$ ossia $(A - B) + 2^n$; tale risultato è corretto solamente nel caso in cui $A - B < 0$, ossia il valore assoluto di B sia non inferiore a A. In tal caso si avrebbe infatti $2^n - (B - A)$ che rappresenta la codifica di un numero negativo $B - A$. Se invece il valore assoluto di A è maggiore di quello di B allora il risultato non è corretto ed è pari a $2^n + (A - B)$, ossia eccede di 2^n. La correzione viene quindi effettuata trascurando il bit in posizione $n + 1$ (ovvero il 2^n).
- $A < 0,\ B > 0$: analogo al caso precedente.
- $A < 0,\ B < 0$: il risultato della somma è $(2^n - A) + (2^n - B) = (2^n - (A+B) + 2^n) = 2^n + (2^n - (A+B))$. Per ottenere il risultato corretto $(2^n - (A + B))$ è necessario applicare la correzione applicata nei casi precedenti, ossia trascurare il riporto in posizione n.

Da questa analisi risulta che l'unica regola da applicare per poter ottenere il corretto risultato nelle operazioni tra valori rappresentati in base 2 notazione complemento a 2 su n bit consiste nell'ignorare il riporto in posizione $n+1$. Ne consegue che qualsiasi operazione elementare di somma o sottrazione può essere realizzata mediante una unica somma algebrica, indipendentemente dagli operandi.
Per applicare quanto visto, si risolva il problema proposto in precedenza, ipotizzando che gli operandi siano rappresentati in base 2 complemento a 2[¶]:

$$\textit{dati } A = 010111_{2,C2} \textit{ e } B = 111000_{2,C2} \textit{ si effettuino le operazioni } A + B \textit{ e } A - B$$

Gli operandi sono già rappresentati entrambi in base 2 complemento a 2 e sono entrambi di 6 bit ciascuno. È quindi possibile procedere con l'operazione.

1	1	0	0	0	0			
	0	1	0	1	1	1	+	(A)
	1	1	1	0	0	0	=	(B)
	0	0	1	1	1	1		$A+B$

Si trascura il bit di riporto e il risultato è $001111_{2,C2}$ ed è corretto. Ora si procede al calcolo di $-B$ per poter effettuare l'operazione $A + (-B)$; in questa rappresentazione dato $B = 111000_{2,C2}$ si ha $-B = 001000_{2,C2}$ ottenuto procedendo fa destra verso sinistra e complementando bit a bit la codifica di B *dopo* il primo 1. L'operazione, qui di seguito riportata dà come risultato $011111_{2,C2}$.

0	0	0	0	0	0			
	0	1	0	1	1	1	+	(A)
	0	0	1	0	0	0	=	$(-B)$
	0	1	1	1	1	1		$A-B$

[¶]Questo significa che i valori rappresentati potrebbero essere diversi rispetto al caso precedente; poiché la differenza esiste solo per i numeri negativi di fatto solo B cambia di valore tra l'esempio precedente e questo. In particolare $B1 = 111000_{2,C1} = -7_{10}$ e $B2 = 111000_{2,C2} = -8_{10}$

Overflow

Anche in complemento alla base può verificarsi un problema di rappresentazione del risultato, di overflow, che però non può essere rilevato utilizzando quanto visto per la notazione modulo e segno. Infatti, nella notazione in complemento alla base, il riporto in posizione n, se esiste, c'è per costruzione e quindi non può essere adottato come presenza di overflow.
In questo codice la rilevazione dell'overflow viene fatta confrontando gli operandi e il risultato dell'operazione. Dati due operandi discordi è possibile dimostrare che non si può mai verificare overflow, qualsiasi sia il valore assoluto degli operandi. D'altra parte, quando gli operandi sono entrambi positivi o entrambi negativi (operandi discordi) se il risultato è di segno opposto significa che il valore risultante è al di fuori dell'intervallo di valori rappresentabili. Si consideri a questo proposito il seguente esempio:

$$\textit{dati } A = 010101_{2,C2} \textit{ e } B = 011100_{2,C2} \textit{ si effettui le operazioni } A + B$$

$$\begin{array}{cccccccccc} 0 & 1 & 1 & 1 & 0 & 0 & & & & \\ & 0 & 1 & 0 & 1 & 0 & 1 & + & (A) \\ & 0 & 1 & 1 & 1 & 0 & 0 & = & (B) \\ \hline & 1 & 1 & 0 & 0 & 0 & 1 & & A+B \end{array}$$

I due operandi sono concordi (entrambi valori positivi) mentre il risultato è negativo: l'operazione è corretta se vista nell'ambito dell'aritmetica modulo 2^n ma non lo è rispetto ai valori rappresentati; infatti $A = +21_{10}$ e $B = +28_{10}$ ed il risultato $A + B = 110001_{2,C2} = -15_{10}$ [‖].
Osservando il valore assunto dai bit di riporto derivanti da ogni somma bit a bit nel caso in cui si verifica overflow è possibile rilevare che $c_n \neq c_{n-1}$. La rilevazione può quindi essere fatta analizzando i bit di riporto.

Prodotto

L'operazione prodotto nell'ambito del sistema binario in complemento a 2 si svolge come nel sistema binario naturale, effettuando però un'operazione di estensione in segno dei prodotti parziali che si ottengono man mano. Tale operazione viene di fatto svolta in modo implicito anche nel sistema binario naturale: in tale situazione quando si rappresenta un valore mediante un numero di bit superiore a quello strettamente necessario si prepongono degli 0. Tali 0, anche quando non scritti in modo esplicito non hanno alcun impatto sulla somma dei prodotti parziali.
In complemento alla 2 l'operazione di estensione in segno aggiunge alla codifica sul numero minimo di bit un numero di cifre uguali a quella più significativa; quando il numero è negativo quindi si devono replicare degli 1. Per lavorare correttamente è quindi necessario effettuare esplicitamente l'estensione in segno dei prodotti parziali ed effettuare quindi la somma di tali contributi. La Figura 4.3 propone lo schema di moltiplicazione tra operandi a n bit rappresentati in complemento a 2.

$$\begin{array}{cccccccc} & & & a_{n-1} & \cdots & a_1 & a_0 & \times \\ & & & b_{n-1} & \cdots & b_1 & b_0 & = \\ \hline & a_{n-1}b_0 & a_{n-1}b_0 & a_{n-1}b_0 & \cdots & a_1b_0 & a_0b_0 & \\ & a_{n-1}b_1 & a_{n-1}b_1 & \cdots & a_1b_1 & a_0b_1 & & \\ & & \cdots & \cdots & \cdots & \cdots & & \\ & a_{n-1}b_{n-1} & \cdots & a_1b_{n-1} & a_0b_{n-1} & & & \\ \hline q_{2n-1} & q_{2n-2} & \cdots & q_4 & q_3 & q_2 & q_1 & q_0 \end{array}$$

Figura 4.3: *Schema di moltiplicazione di numeri binari in complemento a 2 di n bit.*

Riprendiamo la considerazione relativa alla codifica 10010011. Se essa è la codifica di un valore relativo rappresentato in base due, mediante notazione modulo e segno, il valore espresso è un valore negativo il cui

[‖] Si ricordi che l'overflow va rilevato nella base in cui si sta lavorando analizzando operandi e risultato, la conversione in base 10 può essere un'ulteriore conferma di quanto dedotto.

modulo è 10010011, ossia il valore negativo diciannove. Se la stessa codifica è invece di un valore relativo sempre rappresentato in base 2, ma con notazione complemento a due, il valore espresso è centonove.

Più in dettaglio: Codice base 3 in complemento a 2 e base 3 complemento a 3

L'obiettivo di questa nota è derivare per prima cosa la codifica base 3 con notazione in complemento alla base diminuita, ossia complemento a 2, per un'aritmetica modulo 3^2, in cui le codifiche sono di lunghezza 2. In primo luogo è possibile elencare tutte le configurazioni appartenenti alla codifica (da 00 a 22); l'intervallo non è simmetrico, essendoci un numero dispari di configurazioni 3^2 e la doppia codifica dello 0; convenzionalmente la codifica *addizionale* viene associata al valore negativo, con un intervallo di rappresentazione pari a $[-4;+3]$**.

A questo punto si può procedere e determinare il legame tra le codifiche e i valori da rappresentare. Adottando per la codifica dei numeri positivi la notazione posizionale utilizzata per i numeri positivi in modulo e segno, la codifica del valore $N = +1_{10}$ in questa base si ottiene mediante divisioni ripetute che danno come risultato $N = +1_{10} = 01_{3C2}$ (si ricordi che le codifiche sono di 2 cifre, quindi si prepongono degli 0 nel caso la codifica ottenuta abbia un numero di cifre minore). In modo analogo si determinano: $N = +2_{10} = 02_{3C2}$ e $N = +3_{10} = 10_{3C2}$.

La codifica dei numeri negativi può essere ottenuta operativamente con uno qualsiasi dei procedimenti indicati e ricordando che in questa base il complemento di 1 è 1, il complemento di 0 è 2 e viceversa, ottenendo così il risultato di seguito riportato:

Codifica	Valore
00	0
01	+1
02	+2
10	+3
11	-4
12	-3
20	-2
21	-1
22	0

Per quanto riguarda la codifica in complemento alla base, ciò che cambia è la definizione della codifica per i numeri negativi. Per prima cosa si calcola l'intervallo di valori rappresentabili come $[-\frac{b^n-1}{2}; +\frac{b^n-1}{2}] = [-4;+4]$, simmetrico e che include un valore in più rispetto alla codifica in complemento a 2. Per questo motivo è necessario calcolare la codifica dei numeri positivi come fatto in precedenza, rilevando che la configurazione 11 è associata in questo caso al valore $N = +4_{10}$.

Per quanto riguarda la codifica dei valori negativi si procede utilizzando la definizione; Per esempio a partire dalla codifica di $N = +1_{10} = 01_{3C2} = 01_{3C3}$ è possibile determinare la codifica di $-N$ con uno dei procedimenti visti. Con riferimento alla definizione stessa:

$$\begin{array}{rr} c_2\,c_1 & \\ 0\;\;1 & +\;\;(+1) \\ x_1 x_0 = & \;\;(-1) \\ \hline 1\;0\;0 & 3^2 \end{array} \qquad \begin{array}{rr} c_2\;1 & \\ 0\;1 & +\;\;(+1) \\ x_1 2 = & \;\;(-1) \\ \hline 1\;0\;0 & 3^2 \end{array}$$

valore x_0 valore x_1

x_0 è tale per cui $1 + x_0 = 0$ e in base 3 tale valore è 2, con $c_1 = 1$. In modo analogo si determina poi il valore di x_1 considerando che $1 + 0 + x_1 = 0$ con il riporto di 1, da cui si deriva $x_1 = 2$. La codifica in base 3 in complemento a 3 su 2 bit per il valore $N = -1_{10}$ è 22_{3C3}.

Anche per gli altri valori è possibile procedere così, derivando per esempio $N = -2_{10} = 21_{3C3}$

**È lasciata al lettore la verifica che associando alla codifica addizionale un valore positivo, +4 in questo caso, la rappresentazione non perde di validità.

Codifica	Valore
00	0
01	+1
02	+2
10	+3
11	+4
12	-4
20	-3
21	-2
22	-1

Anche in questo caso il bit più significativo non consente di determinare immediatamente il segno del valore rappresentato.

Per eseguire operazioni aritmetiche su operandi rappresentati secondo questa codifica è necessario far riferimento alle regole viste per l'aritmetica in complemento alla base.

5 | Numeri razionali

L'ultimo insieme di numeri qui considerato è l'insieme dei numeri razionali*, $\mathbb{Q}$, ossia quei numeri che possono essere rappresentati come il rapporto tra due numeri interi; tale rapporto viene anche detto "frazione". Non ci si vuol soffermare sulla costruzione di tale insieme, si ricorda solo che i numeri razionali si rappresentano con i simboli:

$$\frac{m}{n},\ n \in \mathbb{N} \setminus \{0\},\ m \in \mathbb{Z} \tag{5.1}$$

in cui i simboli $\frac{0}{n}$ con $n \in \mathbb{N} \setminus \{0\}$ si identificano con 0 e indicano lo zero di $\mathbb{Z}$.
I seguenti sono esempi di numeri razionali:

- $\frac{7}{4} = 1.75$ ossia un numero decimale razionale che può essere finito (che contiene una sequenza finita di cifre dopo la virgola);
- $\frac{1}{3} = 0.\bar{3}$ ossia un numero illimitato e periodico (che contiene una sequenza di cifre che si ripete all'infinito dopo la virgola);
- $\frac{8}{4} = 2$ ossia un numero intero.

Un'interessante proprietà dei numeri razionali è che fra due numeri razionali qualsiasi, anche vicinissimi, ci sono sempre infiniti altri numeri razionali. Per questo motivo si dice che l'insieme $\mathbb{Q}$ è *uniformemente denso*. Proprio questa proprietà metterà in evidenza i limiti della capacità di rappresentazione di questo insieme di numeri, soprattutto da parte di un calcolatore la cui informazione è basata sul sistema binario. Più precisamente ci si trova davanti al problema di avere un insieme continuo di valori e di essere in grado, invece, di rappresentare un insieme discreto di valori, ne consegue che si potrà realizzare solo una rappresentazione "approssimata" dei numeri razionali, e il livello di approssimazione (o *errore di rappresentazione*) dipenderà dal numero di cifre a disposizione.

5.1 Rappresentazione

La rappresentazione dei valori frazionari costituisce un'estensione della notazione posizionale, il valore della parte frazionaria è ottenuta come somma di potenze negative della base. Più precisamente, la relazione (2.1) diventa:

$$v(X) = \alpha_{n-1}b^{n-1} + \ldots + \alpha_0 b^0 + \alpha_{-1}b^{-1} + \ldots + \alpha_{-k}b^{-k} = \sum_{i=-k}^{n-1} \alpha_i b^i \tag{5.2}$$

Con riferimento al sistema binario, una rappresentazione di questa relazione è proposta in Figura 5.1.
Esempi di questa rappresentazione sono riportati qui di seguito:

101.11_2
10.111_2
0.101111_2

*In questa sede si utilizza il . come separatore tra parte intera e parte frazionaria

2^i	2^{i-1}		4	2	1						
b_i	b_{i-1}	$\cdots$	b_2	b_1	b_0	$\cdot$	b_{-1}	b_{-2}	b_{-3}	$\cdots$	b_{-k}
							$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$		$\frac{1}{2^k}$

Figura 5.1: *Rappresentazione dei numeri razionali*

Il primo valore ($\texttt{101.11}_2$) equivale a $5 + \frac{3}{4}$, ossia a 5.75, il secondo ($\texttt{10.111}_2$) a $2 + \frac{7}{8}$, ossia a 2.875, mentre l'ultimo ($\texttt{0.101111}_2$) a $0 + \frac{46}{64}$, ovvero 0.71875 [†].
Con questa rappresentazione è possibile dunque rappresentare in modo esatto (pur di avere un numero di bit sufficiente) valori razionali della forma $\frac{x}{2^k}$, tutti gli altri verranno rappresentati solamente in modo approssimato, con un livello di precisione che dipende sia dalla notazione scelta, sia dal numero di cifre a disposizione.
Di seguito viene prima introdotta la rappresentazione in virgola fissa, quindi seguirà quella in virgola mobile con riferimento alla notazione standard.

5.2 Virgola fissa

Il termine "virgola fissa" di questo tipo di notazione deriva dal fatto di stabilire a priori la posizione del separatore tra la parte intera e la parte frazionaria, definendo così il numero di cifre destinate a ciascuna delle due parti (da cui il nome *fixed point*). Peraltro, poiché le dimensioni di ciascuna parte sono note e fissate, ma non necessariamente uguali tra loro, non serve rappresentare il separatore. Si consideri un codice che utilizza n cifre per la rappresentazione della parte intera e k cifre per la parte frazionaria: una parola di codice è costituita dalla sequenza di simboli seguente:

$$X = \alpha_{n-1}\alpha_{n-2}\ldots\alpha_1\alpha_0\alpha_{-1}\alpha_{-k} \tag{5.3}$$

in cui la virgola viene omessa ed è da intendersi come posizionata alla destra di α_0.
Si consideri ora la conversione da una base b_i ad una base b_j di un numero razionale rappresentato in virgola fissa. Il procedimento utilizzato è un'estensione di quello visto per i numeri naturali, basato sulle divisioni ripetute. Le due parti del numero possono essere convertite in modo indipendente, poiché la codifica finale è la giustapposizione dei due contributi, e quindi si porrà l'attenzione solo sulla conversione della parte frazionaria. Il metodo delle divisioni ripetute deriva dalla relazione (2.1), in cui si cercano i coefficienti delle potenze della base da sommare: la divisione è l'operazione inversa della moltiplicazione e quindi anche dell'elevamento a potenza. In modo analogo si vogliono cercare i coefficienti delle potenze, questa volta negative, per quanto riguarda la parte frazionaria. Il procedimento si basa su moltiplicazioni ripetute della parte frazionaria (la moltiplicazione è in questo caso l'operazione inversa dell'elevamento a potenza, quando la potenza è negativa), di cui si considera ad ogni iterazione la sola parte intera, che va a costituire uno dei coefficienti della codifica che si sta cercando. In modo informale il procedimento è il seguente:

> *Il valore decimale N viene moltiplicato per la base b_j in cui si vuole rappresentare il valore: la parte intera I del valore risultante costituisce un elemento della codifica, mentre la restante parte frazionaria F viene presa come nuovo valore di partenza. Il procedimento viene ripetuto per un numero di passi pari al numero di cifre a disposizione per la parte frazionaria. Tutte le parti intere I_l ottenute, prese nell'ordine in cui sono state ottenute e rappresentate nella base b_j costituiscono la codifica cercata.*

Per esempio, si converta il valore 21.45_{10} in base 2, utilizzando la notazione in virgola fissa costituita da $n = 6$ bit per la parte intera e $n = 7$ per la parte frazionaria. Per quanto riguarda la conversione della parte intera, questa viene fatta per divisioni ripetute, con risultato $21_{10} = 010101_2$ utilizzando i 6 bit a disposizione della parte intera. Per quanto riguarda la parte frazionaria, si ha:

[†]Come per i numeri interi, la divisione per la base (2 in questi esempi) corrisponde ad uno scorrimento a destra, mentre il prodotto per la base ad uno scorrimento a sinistra con l'aggiunta di uno in posizione meno significativa.

0.45	2	$\leftarrow$ base
0.90	0	$(0_{10} = 0_2)$
1.80	1	$(1_{10} = 1_2)$
1.60	1	
1.20	1	
0.40	0	
0.80	0	
1.60	1	

Figura 5.2: *Metodo delle moltiplicazioni ripetute per la conversione di base della parte frazionaria*

La codifica risultante è $0.45_{10} = 0,0111001_2$. Si noti che in Figura 5.2 le cifre nella colonna di destra sono le parti intere ottenute ad ogni iterazione, mentre nella colonna di sinistra c'è il risultato della moltiplicazione della *sola* parte frazionaria. Il risultato finale è dunque $21.45_{10} = 0101010111001_2$, in cui è stato omesso il separatore.
Da questo esempio appare chiaro come il procedimento avrebbe potuto proseguire dando ulteriori contributi non nulli alla codifica se il numero di bit a disposizione per la parte frazionaria fosse stato maggiore. Per meglio comprendere gli effetti di quanto detto, si riconverta il valore 0101010111001_2 appena ottenuto in base 10. Come visto in precedenza con i numeri naturali, per la conversione dalla base 2 alla base 10 è più immediato utilizzare il procedimento della somma di potenze. Nel caso in esame si ha:

$$\begin{aligned} v(X) &= 2^4 + 2^2 + 2^0 + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-7} \\ &= 21 + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{128} = 21 + \frac{57}{128} = 21.4453125 \end{aligned}$$

Come è possibile vedere il numero codificato in base 2 è un'approssimazione del valore in base 10 e in particolare un valore approssimato per difetto. Con il procedimento adottato infatti si commettono errori di arrotondamento per difetto o troncamenti; maggiore è il numero di bit a disposizione per la parte frazionaria e minore è l'errore che si può avere. Naturalmente non tutti i valori rappresentati sono soggetti all'errore: si consideri a questo scopo la conversione di 17.25_{10} in base 2. Utilizzando il procedimento prima illustrato si ottiene per la parte intera $17_{10} = 010001_2$ e per la parte frazionaria $0.25_{10} = 0.0100000_2$ (Figura 5.3), ossia $17.25_{10} = 0100010100000$.

0.25	2	$\leftarrow$ base
0.50	0	
1.00	1	
0.00	0	
0.00	0	
0.00	0	
0.00	0	
0.00	0	

Figura 5.3: *Conversione di base della parte frazionaria* 0.25

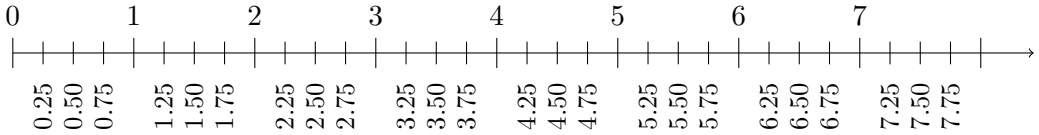
In questo caso l'errore commesso è nullo, e il valore rappresentato è identico in entrambe le basi.
La rappresentazione in virgola fissa ha come caratteristica quella di avere una distribuzione uniforme dei valori rappresentabili, dovuta al fatto che il numero di bit a disposizione per la parte intera e parte frazionaria non cambiano.
Per meglio comprendere questa caratteristica, si prenda in considerazione una rappresentazione di $n = 5$ bit, di cui tre per la parte intera, due per la parte frazionaria. Questo significa che qualsiasi numero

rappresentabile è compreso tra $[0, 2^3)$ e oltre ai numeri interi (in cui la parte frazionaria è nulla), è possibile rappresentare tre valori compresi tra un intero e il successivo, cosí come illustrato in Figura 5.4.

___.00	x
___.01	x + $\frac{1}{4}$
___.10	x + $\frac{1}{2}$
___.11	x + $\frac{3}{4}$

Figura 5.4: *Numeri razionali rappresentabili usando 3 bit per la parte intera e 2 per la parte frazionaria.*

Qualsiasi valore razionale verrà quindi convertito in uno dei valori elencati, e la distribuzione dei valori rappresentabili è riportata in Figura 5.5.



0.25 0.50 0.75 1.25 1.50 1.75 2.25 2.50 2.75 3.25 3.50 3.75 4.25 4.50 4.75 5.25 5.50 5.75 6.25 6.50 6.75 7.25 7.50 7.75

Figura 5.5: *Distribuzione dei valori rappresentabili usando 3 bit per la parte intera e 2 per la parte frazionaria.*

5.2.1 Errore relativo ed assoluto

Quando si deve rappresentare un valore che non è della forma $\frac{x}{2^k}$ si commette un errore: ci sono due diversi modi di guardare a tale differenza tra il valore rappresentato e quello reale. Più precisamente, si parla di *errore assoluto*, ϵ_A quando si indica la differenza tra il valore vero V_{vero} e quello rappresentato V_{rapp}; si parla di *errore relativo*, ϵ_R quando si indica la differenza tra il valore vero e quello rappresentato, messa in rapporto con il valore vero stesso, ossia il rapporto tra errore assoluto e valore vero: $\epsilon_R = \frac{\epsilon_A}{V_{vero}}$. Questo secondo indice ci aiuta a dare una misura al peso che l'errore ha sul valore che stiamo rappresentando.
Nel caso della rappresentazione in virgola fissa, l'errore assoluto rimane costante, l'errore relativo invece decresce al crescere del valore che si vuole rappresentare. Utilizzando come riferimento una rappresentazione in virgola fissa, su $n = 10$ bit con 4 bit per la parte intera, e 6 per quella frazionaria, i valori $X_{10} = 0.30$ e $Y_{10} = 15.30$ verrebbero arrotondati in $X_2 = 0000010011$ e $Y_2 = 1111010011$, che riconvertiti in base 10 corrispondono a $X'_{10} = 0.296875$ e $Y'_{10} = 15.296875$. L'errore assoluto commesso è uguale in entrambi i casi, $\epsilon_{XA} = \epsilon_{YA} = 0.003125$; l'errore relativo invece è diverso nei due casi: $\epsilon_{XR} = 1.0416 \times 10^{-2}$ rispetto e $\epsilon_{XR} = 2.042 \times 10^{-4}$.

5.2.2 Codice BCD

Una variante della notazione in virgola fissa, che non si basa sulla notazione posizionale, utilizza il già citato codice BCD (Sezione 2.2.1), che risulta essere particolarmente adatto per memorizzare numeri razionali per i quali è prefissato il numero di cifre decimali. Si consideri a titolo d'esempio la rappresentazione di importi monetari in euro: le cifre decimali significative sono 2, quindi il codice BCD consente di rappresentare *esattamente* qualsiasi importo utilizzando 8 bit per la parte decimale, senza alcun problema di approssimazione. È pratica comune, in quei sistemi dedicati alla manipolazione di questo tipo di dati, "impacchettare" due cifre BCD in un byte. La notazione in complemento a 2 non è però utilizzabile con questo codice, quindi è necessario utilizzare quella in modulo e segno, che richiede un bit per il segno, al quale però verranno poi dedicati o 4 o 8 bit, per mantenere una certa regolarità.
Per esempio, si rappresenti il valore -4325120.25_{10} nel codice BCD. Convertendo ogni singola cifra, secondo la codifica binaria naturale (riportata in Figura 2.3), si ottiene:

00010100 00110010 01010001 00100000 00100101

o in esadecimale:

$$14_{16} \qquad 32_{16} \qquad 51_{16} \qquad 20_{16} \qquad 25_{16}$$

I primi quattro bit sono 0001 ad indicare il numero negativo; si fosse trattato di un numero positivo la configurazione utilizzata sarebbe stata 0000.
Questo codice, dunque, se il numero di cifre decimali k da rappresentare è noto e predefinito, garantisce l'assenza d'errore pur di utilizzare $4k$ bit.
La rappresentazione in virgola fissa ha come vantaggio la sua semplicità, caratteristica che la rende la rappresentazione convenzionale, ma ha come svantaggio la sua "uniformità". Infatti, poiché su calcolatore si ha a disposizione uno spazio limitato di memoria per la memorizzazione del numero, di tutti i numeri reali è possibile rappresentarne solo una parte discreta. Supposto di aver fissato un numero di cifre massimo per la parte intera e un numero di cifre massimo per la mantissa, i numeri rappresentabili risulteranno equispaziati. Pertanto, avendo fissato tali due numeri di cifre massime in modo adatto per numeri molto grandi (per i quali ha molta importanza la parte intera), risulterà impossibile apprezzare le differenze fra numeri molto piccoli (per i quali ha molta importanza la parte decimale) e viceversa.

5.3 Virgola mobile

La determinazione della posizione della virgola per una rappresentazione di numeri frazionari in virgola fissa può non essere agevole nella pratica. Infatti, riservando molti bit alla parte frazionaria è possibile rappresentare numeri molto piccoli diversi da zero o anche numeri non riducibili a somme di potenze (negative) di 2 con un elevato grado di approssimazione. D'altra parte, lasciare pochi bit per la rappresentazione della parte intera introduce delle forti limitazioni sul valore assoluto massimo rappresentabile. È quindi conveniente poter "spostare" la virgola in modo da adattare dinamicamente il numero di bit a disposizione per le due componenti intera e frazionaria alle esigenze della rappresentazione di valori diversi. Nella realtà questo spostamento non può essere fatto, si può però adottare un codice che sia tale da consentire di "includere" nella notazione stessa anche la virgola: tale metodo di rappresentazione viene chiamato "virgola mobile" o *floating point*, in quanto permette di adattare la posizione della virgola alle esigenze del particolare valore che si deve codificare.
Questa rappresentazione si basa su 3 elementi distinti: il segno s, la mantissa m e l'esponente e mentre la base b è implicita e per le rappresentazioni binarie è 2. Il valore assoluto numerico corrispondente è $m \cdot 2^e$, il segno s completa l'informazione. La rappresentazione è una riformulazione della notazione posizionale, secondo la seguente relazione:

$$v(X) = (-1)^s \cdot \sum_{i=-k}^{n-1} \alpha_i b^i = (-1)^s \cdot b^e \sum_{j=0}^{h} \alpha_j b^{-j} \tag{5.4}$$

dove $\sum_{j=0}^{h} \alpha_j b^{-j}$ è la mantissa $m = 1 + M$ ed ha un valore compreso tra $1 \leq m < 2$.
Tali elementi sono organizzati nella rappresentazione in questo modo:

s	`exp`	M

In questo contesto ci si riferisce alla codifica in virgola mobile secondo lo standard più diffuso, che è quello stabilito dall'IEEE (Institute for Electrical and Electronics Engineering) e identificato dalla sigla 754 (IEEE 754).
Si noti che la mantissa m è un valore compreso nell'intervallo $[1, 2)$, l'esponente e è un valore intero, positivo o negativo. Lo standard IEEE 754 definisce due notazioni che differiscono per il numero complessivo di bit utilizzati per la rappresentazione, e conseguentemente la precisione dei valori che possono rappresentare, così come illustrato dalla Tabella 5.1.
Il codice adotta un approccio simile alla codifica dei numeri interi in modulo e segno per quanto riguarda la positività o negatività del valore, ossia $s = 0$ determina un valore positivo mentre $s = 1$ un valore negativo.

Standard	Segno	Esponente n_e	Mantissa n_m	Eccesso
Singola precisione	1 bit	8 bit	23 bit	$2^{8-1}-1=127$
Doppia precisione	1 bit	11 bit	52 bit	$2^{11-1}-1=1023$

Tabella 5.1: *Codice floating point secondo lo standard IEEE 754 per la rappresentazione dei numeri frazionari in singola e doppia precisione.*

La mantissa m è rappresentata in modo *normalizzato* su n_m bit: si rappresenta la sola parte frazionaria, sottintendendo l'1 (se $m = 1 + M$ si rappresenta solo M). Ad esempio, se $m = 1.5932$ si codifica nel campo mantissa il valore $0,5932$ sugli n_m, utilizzando lo stesso procedimento visto per la notazione in virgola fissa. L'esponente e viene codificato su n_e bit; i valori possibili quindi per l'esponente sono quelli compresi nell'intervallo $[-(2^{n_e-1}-2); +2^{n_e-1}-1] = [-126; +127]$. La codifica Lo standard IEEE 754 prevede però una notazione *in eccesso* (in inglese *biased*) per l'esponente: nel campo esponente invece di codificare il valore e viene codificato il valore $e + (2^{n_e-1}-1)$. Con questo accorgimento nel campo esponente viene sempre codificato un valore positivo: tale scelta costituisce un vantaggio per il tipo di operazioni che frequentemente vengono svolte sugli esponenti del numeri frazionari.
Se il valore X da rappresentare in virgola mobile è già nella forma $\pm(1 + M) \times 2^e$, non resta che derivare la codifica da mettere nel campo esponente come $e + (2^{n_e-1}-1)$, e convertire la parte frazionaria M da mettere nel campo della mantissa.
Nel caso in cui il valore di partenza sia in una forma diversa da questa, è possibile procedere alla conversione in due modi, di seguito illustrati.

> *Si effettui la conversione del valore X così come fatto per la rappresentazione in virgola fissa. Si faccia poi scorrere il valore ottenuto nella base b_j a sinistra (o a destra se più piccolo di 1) di k posizioni, ossia quante necessarie a ottenere un valore nella forma $1, M$. La codifica del campo esponente è pari $k + (2^{n_e-1}-1)$, mentre M costituisce la codifica della mantissa normalizzata.*

A titolo di esempio, prendiamo come riferimento il valore 17.25_{10} precedentemente convertito in base 2, notazione virgola fissa; il risultato ottenuto era (mettendo in evidenza la virgola): 10001.0100000_{2FX}. Il procedimento prima presentato richiede di effettuare uno scorrrimento della virgola di 4 posti a sinistra per trovarsi nella forma 1.00010100000, operazione che di fatto equivale a dividere il numero per 2^4. In particolare, è vero che $10001,0100000_2 = 1.00010100000_2 \times 2^4$. Il valore che deve essere quindi codificato per l'esponente è ($k = 4$) $4 + 127 = 131 = 10000011_2$.
La codifica completa del valore $17,25_{10}$ è quindi:

$\pm$	e+eccesso	M
0	10000011	00010100000000000000000

Un metodo alternativo consiste nell'effettuare la trasformazione del valore da rappresentarsi nella forma $\pm(1 + M) \times 2^e$ già in base 10, effettuando poi la conversione di e e di M. Così facendo si otterrebbe la seguente situazione: $17.25_{10} = 1.078125 \times 2^4$. La conversione dell'esponente cui si somma l'eccesso è $131 = 10000011_2$, mentre la conversione della mantissa sui 23 bit si ricava dalla sequenza di parti intere calcolate mediante il prodotto iterato della parte frazionaria riportato di seguito.

1,078125	2 ← base
0,15625	0
0,3125	0
0,625	0
1,25	1
0,50	0
1,00	1
0,00	0
0,00	0
···	0

Si ottiene dunque il risultato trovato in precedenza.
Come altro esempio, si consideri il seguente problema:

si rappresenti secondo lo standard IEEE 754 il valore $N = -38.53_{10}$ *in singola precisione*

La conversione del valore 38.53_{10} come virgola fissa produce: $38_{10} = 100110_2$ e $0.53_{10} = 10000111101011100001011_2$, come si vede dalla sequenza di prodotti.

0,53	2 ← base		
1,06	1	1,76	1
0,12	0	1,52	1
0,24	0	1,04	1
0,48	0	0,08	0
0,96	0	0,16	0
1,92	1	0,32	0
1,84	1	0,64	0
1,68	1	1,38	1
1,36	1	0,76	0
0,72	0	1,52	1
1,44	1	1,04	1
0,88	0		

Vale la pena notare che poichè si effettuerà uno scorrimento a destra della virgola di 5 posizioni, verranno persi i 5 bit meno significativi della parte frazionaria appena convertita, e dunque si poteva omettere il loro calcolo sin da subito. Si ha quindi come mantissa = 1.00110100001111010111000
Effettuando la conversione dell'esponente (con l'eccesso) $5 + 127 = 132$ si ottiene:

±	e+eccesso	M
1	10000100	00110100001111010111000

Utilizzando l'altro metodo si procede nel ricondurre il valore N alla forma $(-1)^s \times (1+M) \times 2^e$ di riferimento per la notazione in virgola mobile. Il valore da rappresentare è positivo, quindi il primo bit della codifica (s) è 0; inoltre, $N = -38.53_{10} = -1.2040625 \cdot 2^5$ (infatti 2^5 è la potenza di 2 più vicina a N). Ne consegue che la parte di mantissa da convertire $M = 0.2040625$ e il valore dell'esponente è $e = 5$.
La conversione della mantissa su 23 bit si ricava dalla sequenza di parti intere calcolate mediante il prodotto iterato della parte frazionaria riportato di seguito.

1,2040625	2 ← base		
0,408125	0	1,68	1
0,81625	0	1,36	1
1,6325	1	0,72	0
1,265	1	1,44	1
0,53	0	0,88	0
1,06	1	1,76	1
0,12	0	1,52	1
0,24	0	1,04	1
0,48	0	0,08	0
0,96	0	0,16	0
1,92	1	0,32	0
1,84	1		

Il risultato del procedimento è $M = 00110100001111010111000$. Rimane ora il calcolo della codifica del campo esponente: la notazione prevede un eccesso 127, quindi il valore da rappresentare è $127 + 5 = 132_{10}$ utilizzando $n_e = 8$ bit. La codifica del campo esponente è: 10000100. Si ottiene nuovamente $N = -38.53_{10} = 11000010000101000011110101110000$.

Valori limite

Nella codifica IEEE 754 sono definite configurazioni particolari del campo esponente e del campo mantissa per identificare dei casi limite.

Valori denormalizzati

Quando nel campo esponente (e + eccesso) si ha il valore 00000000 (o nel caso di doppia precisione 00000000000) ci si trova in una situazione particolare, utilizzata per gestire casi particolari in cui il valore che si vuol rappresentare sta raggiungengo il valore più piccolo rappresentabile. Invece di rappresentare il valore nella forma $(-1)^s \times 1.M \times 2^e$, si *denormalizza* la rappresentazione, adottando la forma $(-1)^s \times 0.M \times 2^{eccesso-1}$, per poter rappresentare numeri vicini allo 0.0.
Con e + eccesso = 00000000, i casi che possono verificarsi sono i seguenti:

- $M = 0$: rappresenta il valore 0
- $M \neq 0$: rappresenta numeri molto vicini a 0.0

Si consideri, per esempio, il valore $N = 10000000000101000011110000000000$ rappresentato in base due, notazione virgola mobile in singola precisione. Il valore è negativo, e ha il campo esponente nullo. Si tratta quindi di un valore espresso nella forma $(-1) \times (0 + M) \times 2^{-126}$.
Volendo mettere in relazione la forma normalizzata e denormalizzata con l'esponente più piccolo rappresentabile si ha:

$(-1)^s \times (0 + 0) \times 2^{eccesso-1}$	$\pm 00000000(0 + 0)$	zero
$(-1)^s \times (0 + M) \times 2^{eccesso-1}$	$\pm 00000000(0 + M)$	denormalizzata
$(-1)^s \times (1 + M) \times 2^{eccesso-1}$	$\pm 00000000(1 + M)$	normalizzata

Infinito e `NaN`

Anche il valore più elevato del campo esponente viene utilizzato in modo particolare, per rappresentare dei casi limite. Più precisamente, quando il campo esponente assume valore 11111111 (o 11111111111 nel caso di doppia precisione), ossia $2^{n_e} - 1$, ci si trova in una delle due situazioni seguenti, ciascuna con il proprio significato:

1. $M = 0$: infinito (positivo o negativo, in base al valore del segno)

2. $M \neq 0$: `NaN` (*Not a Number*), ossia un valore *non definito*

La prima situazione si verifica nel momento in cui l'operazione produce come risultato un numero in valore assoluto troppo grande per essere rappresentato con un qualche grado di precisione: può trattarsi di $+\infty$ nel caso in cui $s = 0$, oppure di $-\infty$ nel caso in cui $s = 1$.
La seconda situazione si verifica quando, ad esempio, si divide zero per zero, oppure si computa la radice quadrata di -1.
Si noti come qualsiasi operazione effettuata con un operando che è un valore limite, produca come risultato un valore limite. Le tre rappresentazioni di questi valori limite sono quindi:

$(-1)^0 \times (1+0) \times 2^{eccesso+1}$	$+11111111(1.)000\cdots000$	$+\infty$
$(-1)^1 \times (1+0) \times 2^{eccesso+1}$	$-11111111(1.)000\cdots000$	$-\infty$
$(-1)^s \times (1+M) \times 2^{eccesso+1}$	$\pm11111111(1.)M$ $(M \neq 0)$	`NaN`

Per avere una visione d'insieme di questi valori limite in relazione al valore del campo esponente, viene ora proposta una rappresentazione in virgola mobile su 8 bit, così utilizzati:

- 1 bit di segno
- 4 bit per la rappresentazione dell'esponente in *eccesso*-7
- 3 bit per la mantissa

$\texttt{exp}_{10}$	$\texttt{exp}_2$	e	2^e	
0	0000	-6	1/64	denormalizzati
1	0001	-6	1/64	
2	0010	-5	1/32	
3	0011	-4	1/16	
4	0100	-3	1/8	
5	0101	-2	1/4	
6	0110	-1	1/2	
7	0111	0	1	
8	1000	+1	2	
9	1001	+2	4	
10	1010	+3	8	
11	1011	+4	16	
12	1100	+5	32	
13	1101	+6	64	
14	1110	+7	128	
15	1111	n/a	∞, `NaN`	

I valori che si riescono a rappresentare sono dunque i seguenti:

	s	exp	M	e			Valore	
	0	0000	000	-7	0	=	0	
numeri	0	0000	001	-7	$1/8 \times 1/64$	=	1/512	più vicino a zero
denormalizzati	0	0000	010	-7	$2/8 \times 1/64$	=	2/512	
	...							
	0	0000	111	-7	$7/8 \times 1/64$	=	7/512	più grande denormalizzato
	0	0001	000	-6	$8/8 \times 1/64$	=	8/512	più piccolo normalizzato
	0	0001	001	-6	$9/8 \times 1/64$	=	9/512	
	...							
	0	0110	110	-1	$14/8 \times 1/2$	=	14/16	
	0	0110	111	-1	$15/8 \times 1/2$	=	15/16	più vicino a 1 (inferiore)
numeri	0	0111	000	0	$8/8 \times 1$	=	1	
normalizzati	0	0111	001	0	$9/8 \times 1$	=	9/8	più vicino a 1 (superiore)
	0	0111	010	0	$10/8 \times 1$	=	10/8	
	...							
	0	1110	110	7	$14/8 \times 128$	=	224	
	0	1110	111	7	$15/8 \times 128$	=	240	più grande normalizzato
	0	1111	000	n/a	∞			
	0	1111	001	n/a	`NaN`			
	...							
	0	1111	111	n/a	`NaN`			

La successiva Tabella 5.2 riporta la rappresentazione in virgola mobile, con tutti i casi limite visti, rappresentata graficamente in Figura 5.6.

Tipo	Esponente e (n_e bit)	Campo esponente $e + 2^{n_e-1} - 1$	Parte frazionaria M (n_m bit)
Zero	$-(2^{n_e-1} - 1)$	$00\cdots00$	$00\cdots00$
Denormalizzati	0	$00\cdots00$	diverso da $00\cdots00$
Normalizzati	$[1, 2^{n_e} - 2]$	$[00\cdots01, 11\cdots10]$	qualsiasi
Infinito	$2^{n_e} - 1$	$11\cdots11$	$00\cdots00$
`NaN`	$2^{n_e} - 1$	$11\cdots11$	diverso da $00\cdots00$

Tabella 5.2: *Rappresentazione di valori in virgola mobile, standard IEEE 754.*

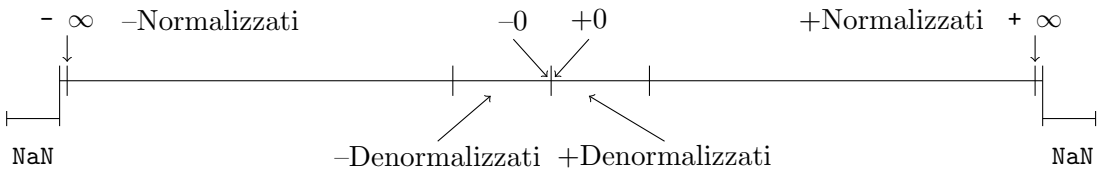


Figura 5.6: *Rappresentazione di valori in virgola mobile, standard IEEE 754.*

Rappresentazione esadecimale del numero

Dato un valore frazionario rappresentato in singola o doppia precisione, è conveniente rappresentarlo in base 16 per motivi di compattezza. La conversione è immediata, prendendo gruppi di 4 bit e sostituendoli con il corrispondente simbolo del sistema esadecimale.
Ad esempio, il valore $N = 10000000000101000011110000000000$ rappresentato in base 16 è $N = 80143C00$, ottenuto come di seguito illustrato.

1000	0000	0001	0100	0011	1100	0000	0000
8	0	1	4	3	C	0	0

Ha senso effettuare la conversione da e verso il sistema esadecimale sulla rappresentazione completa del valore in virgola mobile, pertanto viene fatta esclusivamente sull'insieme dei 32/64 bit.

Per riepilogare, si riporta qua un esempio:

dato $+1.0545_{10MS}$ *rappresentato in base 10, lo si rappresenti in base 2, notazione IEEE 754, singola precisione, riportando tutti i bit della codifica, convertendo inoltre il valore risultante espresso in base 16*

Gli elementi che compongono la rappresentazione nella notazione IEEE 754, sono i seguenti:

- segno positivo: 0
- conversione della parte intera $1_{10} = 1_2$
- conversione della parte decimale: $0.0545_{10} = 00001101111100111011011_2$

La rappresentazione in virgola fissa è 1.00001101111100111011011.
Il valore è già nella forma $(1 + M) \times 2^e$ ossia $1.00001101111100111011011 \times 2^0$
La notazione è in eccesso e dunque nel campo esponente va rappresentato il valore $0 + 127 = 127$, che rappresentato in base 2 è $127_{10} = 01111111_2$.
Quindi, la sequenza di 32 bit che rappresenta la codifica in base 2, notazione IEEE 754, singola precisione è la seguente.
$+1.0545_{10MS} = 0\ 01111111\ 00001101111100111011011$
Per rappresentare questo valore in base 16, è sufficiente convertire in base 16 sequenze di 4 bit:

	00111111100001101111100111011011						
0011	1111	1000	0110	1111	1001	1101	1011
3	F	8	6	F	9	D	B
			$3F86F9DB$				

6 | Codici non numerici

I codici visti fino ad ora vengono utilizzati per rappresentare un'informazione numerica, più o meno complessa. Nella realtà oltre a tale tipo di informazione ci sono anche altri "valori" da rappresentare, tra cui i caratteri alfanumerici, ossia l'insieme di caratteri alfabetici quali *a*, *b*, $\cdots$, *A*, *B*, $\cdots$, i caratteri numerici *0*, *1*, $\cdots$, e caratteri di interpunzione *.*, *;*, $\cdots$. Certamente tale insieme di informazioni dipende molto dal linguaggio cui ci si riferisce (per esempio non fa parte dell'insieme dei caratteri italiani il carattere `\UTF{00C1}`).

6.1 ASCII

Per rappresentare l'informazione non numerica uno dei codici più comunemente utilizzati è il codice ASCII (American Standard Code for Information Interchange), basato sul sistema binario e che consta di parole di codice di 7 bit. Il codice ASCII è non ridondante, perchè i simboli codificati sono in numero pari alle configurazioni ottenibili con 7 cifre binarie. I primi 32 caratteri sono spesso chiamati "caratteri non stampabili" sono riservati per simboli di controllo (per esempio il ritorno a capo) e funzioni varie (campanello).
Poiché i caratteri sono però ben più numerosi di 127 anche nel nostro alfabeto, in realtà si utilizza il codice ASCII esteso, che utilizza 8 bit, avendo quindi a disposizione 256 diverse configurazioni. Questa parte della codifica però presenta varie versioni a carattere nazionale, quindi ci sono diverse codifiche per l'intervallo 128-255.
La Tabella 6.1 riporta il codice ASCII.
Questa coesistenza fra diverse versioni del codice ASCII esteso produce spesso discordanze nella visualizzazione dei file di testo, in base alla codifica cui ci si sta riferendo.

6.2 Unicode

Per cercare di ovviare al limite del codice ASCII è stato definito un nuovo standard internazionale detto *Unicode*, definito dalla Unicode Consortium e dalla International Organization for Standardization (ISO 10646), che rappresenta i caratteri usando 16 bit (2 byte) nella proposta iniziale ed ora codici nell'intervallo esadecimale 0..10FFFF. Si rappresentano non solo tutte le varianti dell'alfabeto latino, ma anche tutti gli altri alfabeti (greco, cirillico, arabo, ebraico, hindi, thai, $\cdots$) oltre all'insieme degli ideogrammi cinesi e giapponesi (che sono in tutto circa 30.000, anche se poi ne vengono effettivamente utilizzati solo poche migliaia). Lo scopo è rappresentare qualsiasi simbolo possa essere presente in un documento, in qualsiasi lingua. Più precisamente, è stata effettuata una classificazione in base alle tipologie linguistiche (ad esempio, Alfabeti europei, Scritture africane, ecc.) e a ogni simbolo è associato un codice ed una descrizione testuale estesa. Un esempio è il carattere A, corrispondente al codice U0041, appartenente al gruppo di Alfabeti europei, ed in particolare l'alfabeto Latino base, e la descrizione del carattere è il "il carattere latino maiuscolo A".
Lo svantaggio dell'Unicode, rispetto all'ASCII, è che le dimensioni dei file di testo risultano raddoppiate (vengono usati 2 byte per carattere, invece di 1 solo).

	Codifica				Codifica		
base 2	base 10	base 16	carattere	base 2	base 10	base 16	carattere
00000000	0	0	NUL	01000000	64	40	@
00000001	1	1	SOH	01000001	65	41	A
00000010	2	2	STX	01000010	66	42	B
00000011	3	3	ETX	01000011	67	43	C
00000100	4	4	EOT	01000100	68	44	D
00000101	5	5	ENQ	01000101	69	45	E
00000110	6	6	ACK	01000110	70	46	F
00000111	7	7	BEL	01000111	71	47	G
00001000	8	8	BS	01001000	72	48	H
00001001	9	9	HT	01001001	73	49	I
00001010	10	A	LF	01001010	74	4A	J
00001011	11	B	VT	01001011	75	4B	K
00001100	12	C	FF	01001100	76	4C	L
00001101	13	D	CR	01001101	77	4D	M
00001110	14	E	SO	01001110	78	4E	N
00001111	15	F	SI	01001111	79	4F	O
00010000	16	10	DLE	01010000	80	50	P
00010001	17	11	DC1	01010001	81	51	Q
00010010	18	12	DC2	01010010	82	52	R
00010011	19	13	DC3	01010011	83	53	S
00010100	20	14	DC4	01010100	84	54	T
00010101	21	15	NAK	01010101	85	55	U
00010110	22	16	SYN	01010110	86	56	V
00010111	23	17	ETB	01010111	87	57	W
00011000	24	18	CAN	01011000	88	58	X
00011001	25	19	EM	01011001	89	59	Y
00011010	26	1A	SUB	01011010	90	5A	Z
00011011	27	1B	ESC	01011011	91	5B	[
00011100	28	1C	FS	01011100	92	5C	\
00011101	29	1D	GS	01011101	93	5D	]
00011110	30	1E	RS	01011110	94	5E	^
00011111	31	1F	US	01011111	95	5F	_
00100000	32	20	SPAZIO	01100000	96	60	‘
00100001	33	21	!	01100001	97	61	a
00100010	34	22	"	01100010	98	62	b
00100011	35	23	#	01100011	99	63	c
00100100	36	24	$	01100100	100	64	d
00100101	37	25	%	01100101	101	65	e
00100110	38	26	&	01100110	102	66	f
00100111	39	27	’	01100111	103	67	g
00101000	40	28	(	01101000	104	68	h
00101001	41	29	)	01101001	105	69	i
00101010	42	2A	*	01101010	106	6A	j
00101011	43	2B	+	01101011	107	6B	k
00101100	44	2C	,	01101100	108	6C	l
00101101	45	2D	-	01101101	109	6D	m
00101110	46	2E	.	01101110	110	6E	n
00101111	47	2F	/	01101111	111	6F	o
00110000	48	30	0	01110000	112	70	p
00110001	49	31	1	01110001	113	71	q
00110010	50	32	2	01110010	114	72	r
00110011	51	33	3	01110011	115	73	s
00110100	52	34	4	01110100	116	74	t
00110101	53	35	5	01110101	117	75	u
00110110	54	36	6	01110110	118	76	v
00110111	55	37	7	01110111	119	77	w
00111000	56	38	8	01111000	120	78	x
00111001	57	39	9	01111001	121	79	y
00111010	58	3A	:	01111010	122	7A	z
00111011	59	3B	;	01111011	123	7B	{
00111100	60	3C	<	01111100	124	7C	—
00111101	61	3D	=	01111101	125	7D	}
00111110	62	3E	>	01111110	126	7E	~
00111111	63	3F	?	01111111	127	7F	DEL

Tabella 6.1: *Tabella del codice ASCII*

Codifica				Codifica			
base 2	base 10	base 16	carattere	base 2	base 10	base 16	carattere
10000000	128	80	\UTF{00C4}	11000000	192	C0	\UTF{00BF}
10000001	129	81	Å	11000001	193	C1	\UTF{00A1}
10000010	130	82	\UTF{00C7}	11000010	194	C2	¬
10000011	131	83	\UTF{00C9}	11000011	195	C3	
10000100	132	84	\UTF{00D1}	11000100	196	C4	\UTF{0192}
10000101	133	85	\UTF{00D6}	11000101	197	C5	\UTF{2248}
10000110	134	86	\UTF{00DC}	11000110	198	C6	\UTF{2206}
10000111	135	87	\UTF{00E1}	11000111	199	C7	\UTF{00AB}
10001000	136	88	\UTF{00E0}	11001000	200	C8	\UTF{00BB}
10001001	137	89	\UTF{00E2}	11001001	201	C9	...
10001010	138	8A	\UTF{00E4}	11001010	202	CA	\UTF{00A0}
10001011	139	8B	\UTF{00E3}	11001011	203	CB	\UTF{00C0}
10001100	140	8C	\UTF{00E5}	11001100	204	CC	\UTF{00C3}
10001101	141	8D	\UTF{00E7}	11001101	205	CD	\UTF{00D5}
10001110	142	8E	\UTF{00E9}	11001110	206	CE	\UTF{0152}
10001111	143	8F	\UTF{00E8}	11001111	207	CF	\UTF{0153}
10010000	144	90	\UTF{00EA}	11010000	208	D0	\UTF{2013}
10010001	145	91	\UTF{00EB}	11010001	209	D1		
10010010	146	92	\UTF{00ED}	11010010	210	D2	\
10010011	147	93	\UTF{00EC}	11010011	211	D3	"
10010100	148	94	\UTF{00EE}	11010100	212	D4	‘
10010101	149	95	\UTF{00EF}	11010101	213	D5	’
10010110	150	96	\UTF{00F1}	11010110	214	D6	÷
10010111	151	97	\UTF{00F3}	11010111	215	D7	\UTF{25CA}
10011000	152	98	\UTF{00F2}	11011000	216	D8	\UTF{00FF}
10011001	153	99	\UTF{00F4}	11011001	217	D9	\UTF{0178}
10011010	154	9A	\UTF{00F6}	11011010	218	DA	\UTF{2044}
10011011	155	9B	\UTF{00F5}	11011011	219	DB	\UTF{20AC}
10011100	156	9C	\UTF{00FA}	11011100	220	DC	\UTF{2039}
10011101	157	9D	\UTF{00F9}	11011101	221	DD	\UTF{203A}
10011110	158	9E	\UTF{00FB}	11011110	222	DE	\UTF{FB01}
10011111	159	9F	\UTF{00FC}	11011111	223	DF	\UTF{FB02}
10100000	160	A0	NBSP	11100000	224	E0	‡
10100001	161	A1	°	11100001	225	E1	\UTF{00B7}
10100010	162	A2	¢	11100010	226	E2	\UTF{201A}
10100011	163	A3	£	11100011	227	E3	\UTF{201E}
10100100	164	A4	§	11100100	228	E4	‰
10100101	165	A5	\UTF{2022}	11100101	229	E5	\UTF{00C2}
10100110	166	A6	¶	11100110	230	E6	\UTF{00CA}
10100111	167	A7	\UTF{00DF}	11100111	231	E7	\UTF{00C1}
10101000	168	A8	\UTF{00AE}	11101000	232	E8	\UTF{00CB}
10101001	169	A9	\UTF{00A9}	11101001	233	E9	\UTF{00C8}
10101010	170	AA	\UTF{2122}	11101010	234	EA	\UTF{00CD}
10101011	171	AB	´	11101011	235	EB	\UTF{00CE}
10101100	172	AC	¨	11101100	236	EC	\UTF{00CF}
10101101	173	AD		11101101	237	ED	\UTF{00CC}
10101110	174	AE	\UTF{00C6}	11101110	238	EE	\UTF{00D3}
10101111	175	AF	\UTF{00D8}	11101111	239	EF	\UTF{00D4}
10110000	176	B0		11110000	240	F0	\UTF{F8FF}
10110001	177	B1	±	11110001	241	F1	\UTF{00D2}
10110010	178	B2	\UTF{2264}	11110010	242	F2	\UTF{00DA}
10110011	179	B3	\UTF{2265}	11110011	243	F3	\UTF{00DB}
10110100	180	B4	¥	11110100	244	F4	\UTF{00D9}
10110101	181	B5	\UTF{00B5}	11110101	245	F5	\UTF{0131}
10110110	182	B6		11110110	246	F6	\UTF{02C6}
10110111	183	B7	\UTF{2211}	11110111	247	F7	\UTF{02DC}
10111000	184	B8	\UTF{220F}	11111000	248	F8	\UTF{00AF}
10111001	185	B9		11111001	249	F9	\UTF{02D8}
10111010	186	BA		11111010	250	FA	\UTF{02D9}
10111011	187	BB	\UTF{00AA}	11111011	251	FB	\UTF{02DA}
10111100	188	BC	\UTF{00BA}	11111100	252	FC	\UTF{00B8}
10111101	189	BD		11111101	253	FD	\UTF{02DD}
10111110	190	BE	\UTF{00E6}	11111110	254	FE	\UTF{02DB}
10111111	191	BF	\UTF{00F8}	11111111	255	FF	\UTF{02C7}

Tabella 6.2: *Tabella del codice ASCII (esteso)*

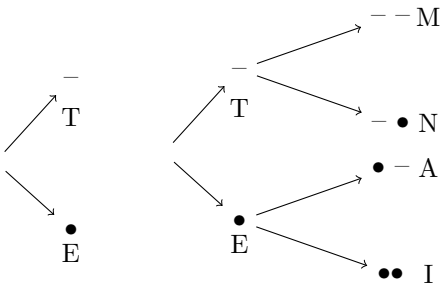


Figura 6.1: *Albero di costruzione del codice Morse*

6.3 Morse

Sebbene il codice Morse e i suoi impieghi abbiano poca attinenza con i sistemi di calcolo in termini di rappresentazione dell'informazione, questo paragrafo presenta le caratteristiche di tale codice, come esempio di codice binario a lunghezza variabile. La Figura 6.1 illustra il procedimento di costruzione del codice per le sequenze di lunghezza 1 e 2, poi riportato in forma completa in Tabella 6.3.

a	·−	n	−·	0	−−−−−
b	−···	o	−−−	1	·−−−−
c	−·−·	p	·−−·	2	··−−−
d	−··	q	−−·−	3	···−−
e	·	r	·−·	4	····−
f	··−·	s	···	5	·····
g	−−·	t	−	6	−····
h	····	u	··−	7	−−···
i	··	v	···−	8	−−−··
j	·−−−	w	·−−	9	−−−−·
k	−·−	x	−··−	Punto	·−·−·−
l	·−··	y	−·−−	Virgola	−−··−−
m	−−	z	−−··	Domanda	··−−··

Tabella 6.3: *Tabella del codice Morse*