• LA SOMMA DI 2 NUMERI NATURALI

- la somma deve fornire lo stesso risultato a prescindere dalla base in cui si opera

- per eseguire la somma di 2 numeri naturali in base 2, esistono opportuni circuiti. Tali circuiti prendono il nome di sommatori; ~~sta~~ sono delle reti combinatorie e sono espressi pertanto tramite tabelle di verità.

Vediamo il circuito più semplice, il sommatore di un bit. AVRÀ

- 2 ingressi (i bit da sommare)
- 1 uscita (il risultato)
- la seguente tabella di verità:

I_1	I_0	V
0	0	0
0	1	1
1	0	1
1	1	0

Tale circuito può essere utilizzato per costruire sommatori a più bit, e ognuno è in generale per fare la somma fra 2 numeri binari naturali. Non è però completo.

Se infatti devo sommare 1 + 1 = 2

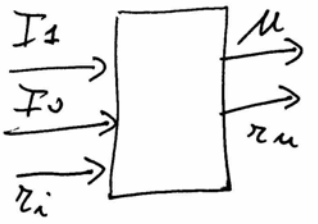
1 + 2 fa 1 con riporto di 1. Il riporto si aggiunge a somma a 1, per cui il risultato sarà come atteso 21. Lo stesso vale per il sommatore ad 1 bit. 1 + 1 in binario fa 0 con riporto di 1.

Il circuito sommatore ~~è pertanto del tipo~~ effettivo pertanto un riporto in uscita, ma avrà anche un riporto in ingresso.

Nel caso di sopra 1 + 2 = 21 perché

1 + 2 = 1 con riporto di 1 e 1 + riporto = 2

il circuito sommatore sarà perlato



I_1	I_0	r_i	u	r_u
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Con tale tabella ogni volta è possibile sommare Numeri ~~interi in base~~ naturali in base 2 su qualunque numero di bit

- A tal punto è possibile capire perché si usa la rappresentazione in ~~se~~ C2 e non quella in modulo e segno.

Proviamo ad eseguire l'algoritmo precedente ch sommo, per 2 numeri rappresentati in m.s e c2.

- Calcoliamo 2+ -1 lavorando in binario (il risultato deve venire 1, in qualunque modo si faccia l'operazione).

- lo vediamo in modulo i segno.

2 è rappresentato come 010

-1 " " come 101

$$\begin{array}{r} 2+ \\ -1 = \end{array} \begin{array}{r} 010+ \\ 101= \\ \hline 111 \end{array}$$

ma 111 è -3, sempre è sbagliato.

- lo vediamo in C2

2 è 010 allora $\begin{array}{r} 2+ \\ -1 = \end{array} \begin{array}{r} 010+ \\ 111 \\ \hline 001 \end{array}$ che è 1 proprio

-1 è 111

riporto 1

dunque, per realizzare la somma in ~~modulo~~ c2 posso utilizzare gli stessi circuiti dell'aritmetica intera. Questo è il motivo per cui i calcolatori utilizzano il c2 (non solo per moltiplicazione e divisione)

- Resta da capire quando la somma di 2 numeri ad N bit è rappresentabile su N bit.

Consideriamo 2 numeri naturali su 3 bit.
Il max numero naturale rappresentabile su 3 bit è 7.

Dunque se devo calcolare 5+3, la somma è ~~calcolabile su~~ rappresentabile su 3 bit (il risultato viene 6 che è rappresentabile)

- Ma se lavoro in binario, posso accorgermi?
  Sì: Se la somma dà NATURALMENTE un riporto, cioè dice che non è rappresentabile.

- proviamo 5+1 e 5+3

5 è 101 ; 1 è 001 , 3 è 011

$\bar{5}+\bar{1} \Rightarrow \bar{e}$

```
101+
001
---
110   riporto è 0
```

$\bar{5}+\bar{3} \Rightarrow \bar{e}$

```
101+
011
---
000   riporto 1
```

$\Rightarrow$ ho un riporto e allora non è valida l'operazione.

Dunque, dati 2 numeri naturali, per capire se la somma è rappresentabile, posso procedere in 2 modi:

a) Lavoro in base 10 e calcolo la somma e vedo se è rappresentabile (ma così non può fare il calcolatore)

b) lavoro in base 2, faccio la somma e ~~genero~~ vedo se genero un riporto.
   1) se lo genero non è rappresentabile la somma
   2) Se non lo genero è rappresentabile.

⇒ Tale procedura è automatica e può essere implementata in un calcolatore.

- Le regole seguenti si applicano ~~a numeri~~ ~~in~~ ~~con~~ ~~cui~~ sono dei numeri in complemento a 2

REGOLE per l'operazione di SOMMA in
COMPLEMENTO A 2

- SOMMA DI 2 NUMERI CON SEGNO CONCORDE IN COMPLEMENTO A 2:

=> se cambia il segno vuol dire che è rappresentabile. (operazione oltre il limite)

- Esempio (a 4 bit).

6 = 0110
2 = 0010

6+2 = 0110
0010
1000

↑ il segno è cambiato: la somma non è rappresentabile.

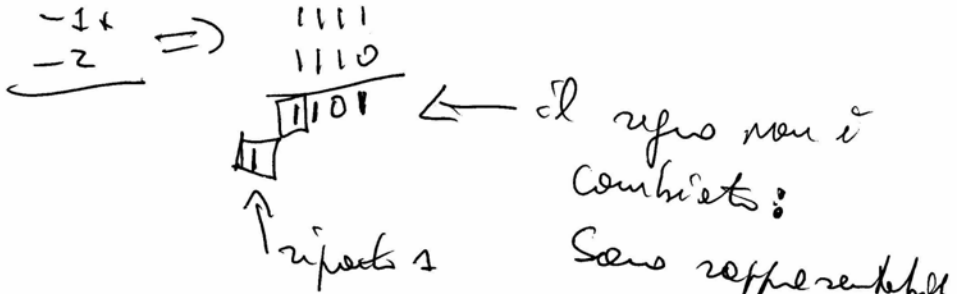
6 = 0110
1 = 0001

6+1 = 0110
0001
0111

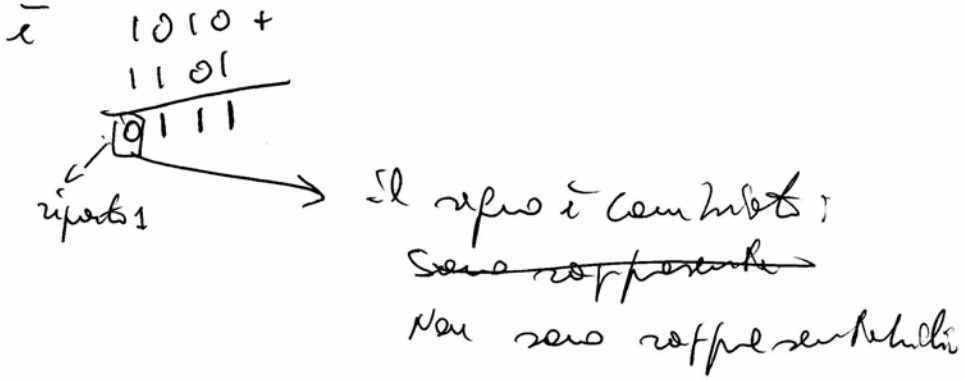
↑ Il segno non è cambiato: la somma è rappresentabile

• Esempio per numeri negativi

$-1 = 1111$
$-2 = 1110$



-6 1010
-3 1101



• SOMMA DI 2 NUMERI IN COMPLEMENTO A 2

DISCORSI: la somma è sempre rappresentabile

• esempio: $-8+$ è $1000+$
1 0001
1001 che è -7 O.K.

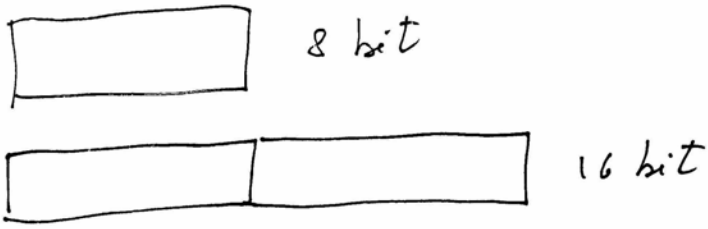
$-8+$ è $1000+$
7 0111
1111 è -1 O.K.

$+7$ è $0111+$
-1 1111
0110 è $+6$ O.K.
riporto 1 (il riporto si trascura)

• Se sono rappresentabili i 2 numeri, e sono distanti la somma avrà valore assoluto inferiore ai 2 e dunque è rappresentabile

Esercizi:

1) • Si dispone delle seguenti locazioni:



• Scegliere in quale locazione, e con quale formato, è possibile rappresentare (e rappresentarli)

- ± 5
- $\pm 254 \mid -255$
- ± 125.000

2)

- Si dispone di 8 bit
- Si vogliono svolgere le seguenti somme:

  $2+5$; $12-27$; $-125-100$

- Discutere l'implementabilità dell'operazione su 8 bit e calcolare il risultato dell'operazione
  - Nei decimali
  - IN complemento a 2
- Si vogliono sommare 125 e 100 su 8 bit in complemento a 2.
- Discutere la rappresentabilità.
  - Rispondendo nei decimali
  - Rispondendo in complemento a 2.

(3)

- Discutere l'implementabilità su 3 bit delle seguenti somme:

1) $111 + 010$
2) $010 + 001$
3) $010 + 010$
4) $111 + 110$
5) $101 + 110$

fettuare. La scelta delle configurazioni da assegnare alle diverse istruzioni viene compiuta cercando di rendere efficiente la fase di decodifica, un obiettivo talmente rilevante che spesso i progettisti preferiscono utilizzare un numero superiore di bit (quindi accettando di non utilizzare alcune configurazioni) pur di rendere la più semplice e quindi la più veloce possibile la fase di decodifica delle istruzioni.

Tabella 3.2 Esempio di codifica a 8 bit delle istruzioni per un linguaggio macchina.

Istruzioni aritmetico-logiche		Istruzioni per il trasferimento dati		Istruzioni di controllo	
Codice	Istruzione	Codice	Istruzione	Codice	Istruzione
01100000	ADD	00010101	LOAD	10011001	IF_EQ
01100100	SUB	00110110	STORE	10110110	GOSUB
01111110	AND	... ...	... ...	10101100	RETURN
... ...	... ...	... ...	... ...	... ...	... ...

Oltre al codice operativo, un'istruzione deve anche contenere gli eventuali riferimenti ai dati necessari per completare l'esecuzione. Nel caso di una tipica operazione aritmetica come l'addizione, per esempio, è necessario che sia specificato dove leggere i due operandi da sommare e dove scrivere il risultato. Naturalmente il numero dei dati da specificare è variabile, in funzione delle istruzioni (si noti inoltre che alcuni dati potrebbero essere indicati implicitamente, per esempio imponendo che il risultato sia memorizzato nella stessa locazione di memoria da cui è prelevato il primo operando). La Figura 3.4 riporta alcuni esempi di formato delle istruzioni in linguaggio macchina.

Figura 3.4 Esempi di alcuni formati utilizzati per istruzioni in linguaggio macchina.

3.2.4 La codifica binaria di dati numerici

Per comprendere le tecniche di codifica dei numeri in un calcolatore è utile far riferimento alla notazione decimale che si adotta abitualmente. Questa rappresentazione è basata sulla dipendenza del valore di ogni cifra dalla sua posizione nella successione di simboli che rappresenta il numero. Per esempio, la successione "4243" è interpretata $4\times10^3 + 2\times10^2 + 4\times10^1 + 3\times10^0$. Una stessa cifra in posizioni diffe-

renti corrisponde a valori diversi: i due simboli "4" nell'esempio sono i fattori moltiplicativi per $10^3 = 1000$ e $10^1 = 10$ rispettivamente. Per un generico numero intero composto di n cifre si ha che:

$$c_{n-1}c_{n-2}\ldots c_1c_0 = c_{n-1}\times 10^{n-1} + c_{n-2}\times 10^{n-2} + \ldots + c_1\times 10^1 + c_0\times 10^0$$

dove ogni c_i è una cifra tra 0 e 9.
Questa stessa rappresentazione può essere adottata anche per numeri espressi in una base di numerazione B diversa da 10, la quale mette a disposizione B cifre, comprese tra 0 e B–1. Quindi:

$$c_{n-1}c_{n-2}\ldots c_1c_0 = c_{n-1}\times B^{n-1} + c_{n-2}\times B^{n-2} + \ldots + c_1\times B^1 + c_0\times B^0$$

Nel caso della numerazione binaria ($B = 2$), l'alfabeto comprende dunque solo due cifre, 0 e 1, e il valore associato per esempio al numero 101100_{due} (con il pedice indichiamo la base della numerazione adottata, e quindi il codice per interpretare correttamente il numero) può essere calcolato nel seguente modo:

$$\begin{aligned} 101100_{due} &= 1_{dieci}\times 2^5_{dieci} + 0_{dieci}\times 2^4_{dieci} + 1_{dieci}\times 2^3_{dieci} + 1_{dieci}\times 2^2_{dieci} + 0_{dieci}\times 2^1_{dieci} + 0_{dieci}\times 2^0_{dieci} \\ &= 32_{dieci} + 8_{dieci} + 4_{dieci} = 44_{dieci} \end{aligned}$$

Questo esempio mostra come convertire un numero binario in un numero decimale; con una procedura analoga si trasformano i numeri decimali in numeri binari. Per esempio, 565_{dieci} viene convertito in rappresentazione binaria come segue:

$$\begin{aligned} 565_{dieci} &= 5_{dieci}\times 10^2_{dieci} + 6_{dieci}\times 10^1_{dieci} + 5_{dieci}\times 10^0_{dieci} \\ &= 101_{due}\times 1010^2_{due} + 110_{due}\times 1010^1_{due} + 101_{dieci}\times 1010^0_{due} \\ &= 101_{due}\times 1100100_{due} + 110_{due}\times 1010_{due} + 101_{due}\times 1_{due} \\ &= 111110100_{due} + 111100_{due} + 101_{due} = 1000110101_{due} \end{aligned}$$

La complessità di questa operazione risiede nella scarsa abitudine a operare in basi diverse da quella decimale. Per questo motivo si può adottare un metodo più comodo per convertire i numeri decimali in numeri binari (e comunque in numeri espressi in una base diversa da dieci). Si consideri di nuovo l'espressione:

$$\begin{aligned} c_{n-1}c_{n-2}\ldots c_1c_0 &= c_{n-1}\times B^{n-1} + c_{n-2}\times B^{n-2} + \ldots + c_1\times B^1 + c_0\times B^0 \\ &= c_{n-1}\times B^{n-1} + c_{n-2}\times B^{n-2} + \ldots + c_1\times B + c_0 \qquad \text{(infatti } B^1 = B \text{ e } B^0 = 1\text{)} \end{aligned}$$

che rappresenta un generico numero scritto in una base B diversa da 10. Dividendo numero per il valore della base, il risultato che si ottiene è:

$$\frac{c_{n-1}\times B^{n-1} + c_{n-2}\times B^{n-2} + \ldots + c_1\times B + c_0}{B}$$

che può essere scomposto in modo da evidenziare quoziente e resto:

$$\frac{c_{n-1}\times B^{n-1} + c_{n-2}\times B^{n-2} + \ldots + c_1\times B}{B} + \frac{c_0}{B} = c_{n-1}\times B^{n-2} + c_{n-2}\times B^{n-3} + \ldots + c_1 + \frac{c_0}{B}$$

$$\textit{quoziente} = c_{n-1}\times B^{n-2} + c_{n-2}\times B^{n-3} + \ldots + c_1 \qquad \textit{resto} = c_0$$

Il resto della division
zione caratteristica d
possibile ottenere tu
vertire il numero 57

$573_{dieci} : 2_{dieci} \Rightarrow$ (
$286_{dieci} : 2_{dieci} \Rightarrow$ (
$143_{dieci} : 2_{dieci} \Rightarrow$
$71_{dieci} : 2_{dieci} \Rightarrow$
$35_{dieci} : 2_{dieci} \Rightarrow$
$17_{dieci} : 2_{dieci} \Rightarrow$
$8_{dieci} : 2_{dieci} \Rightarrow$
$4_{dieci} : 2_{dieci} \Rightarrow$
$2_{dieci} : 2_{dieci} \Rightarrow$
$1_{dieci} : 2_{dieci} \Rightarrow$

Alla fine, leggend
1000111101_{due}.

Tabella 3.3 I prin
decim

Base due	di
0000	
0001	
0010	
0011	

Con una successi
2^n–1 (nell'esempi
tati i primi $2^4 = 1$
quindi il massimo
Nei linguaggi di p
zano abitualment
numeri compresi
la lunghezza dell
nenzialmente; pe
tabile diventa 2^6
grandezza (si di
quando il primo
A causa del min
le, un numero c
impiegate in no
numeri elaborati

Il resto della divisione corrisponde all'ultima cifra del numero, espresso nella notazione caratteristica della base *B*. Applicando ripetutamente questo procedimento è possibile ottenere tutte le cifre del numero. Per esempio, i passi necessari per convertire il numero 573_{dieci} in un numero binario sono:

$573_{dieci} : 2_{dieci}$	$\Rightarrow$ quoziente	286_{dieci}	resto 1_{dieci}	(cifra binaria meno significativa)
$286_{dieci} : 2_{dieci}$	$\Rightarrow$ quoziente	143_{dieci}	resto 0_{dieci}	
$143_{dieci} : 2_{dieci}$	$\Rightarrow$ quoziente	71_{dieci}	resto 1_{dieci}	
$71_{dieci} : 2_{dieci}$	$\Rightarrow$ quoziente	35_{dieci}	resto 1_{dieci}	
$35_{dieci} : 2_{dieci}$	$\Rightarrow$ quoziente	17_{dieci}	resto 1_{dieci}	
$17_{dieci} : 2_{dieci}$	$\Rightarrow$ quoziente	8_{dieci}	resto 1_{dieci}	
$8_{dieci} : 2_{dieci}$	$\Rightarrow$ quoziente	4_{dieci}	resto 0_{dieci}	
$4_{dieci} : 2_{dieci}$	$\Rightarrow$ quoziente	2_{dieci}	resto 0_{dieci}	
$2_{dieci} : 2_{dieci}$	$\Rightarrow$ quoziente	1_{dieci}	resto 0_{dieci}	
$1_{dieci} : 2_{dieci}$	$\Rightarrow$ quoziente	0_{dieci}	resto 1_{dieci}	(cifra binaria più significativa)

Alla fine, leggendo i resti dal basso verso l'alto, si ottiene il numero binario 1000111101_{due}.

Tabella 3.3 I primi 16 numeri naturali rappresentati nel sistema binario e in quello decimale.

Base		Base		Base		Base	
due	dieci	due	dieci	due	dieci	due	dieci
0000	0	0100	4	1000	8	1100	12
0001	1	0101	5	1001	9	1101	13
0010	2	0110	6	1010	10	1110	14
0011	3	0111	7	1011	11	1111	15

Con una successione di n bit si possono rappresentare i 2^n numeri naturali da 0 a 2^n-1 (nell'esempio riportato in Tabella 3.3, n è uguale a 4 e quindi sono rappresentati i primi $2^4 = 16$ numeri). La lunghezza delle successioni di bit adottate stabilisce quindi il massimo numero che può essere rappresentato.

Nei linguaggi di programmazione, per la codifica dei numeri interi positivi si utilizzano abitualmente successioni di 32 bit, che consentono quindi di rappresentare i numeri compresi tra 0 e $2^{32}-1 = 4\,294\,967\,295 \cong 4\times10^9$. Si noti che raddoppiando la lunghezza delle successioni il massimo numero rappresentabile aumenta esponenzialmente; per esempio passando da 32 a 64 bit il massimo numero rappresentabile diventa $2^{64}-1 \cong 16\times10^{18} = 1.6\times10^{19}$ e cresce quindi di circa 10 ordini di grandezza (si dice che un valore è superiore a un altro di un ordine di grandezza quando il primo valore è circa dieci volte maggiore del secondo).

A causa del minor numero di simboli dell'alfabeto binario rispetto a quello decimale, un numero codificato in notazione binaria richiede più cifre rispetto a quelle impiegate in notazione decimale. Per rendere più sintetica la rappresentazione dei numeri elaborati dai calcolatori, si sceglie sovente di impiegare la base 16 (una no-

tazione detta *esadecimale*), dunque impiegando un alfabeto di sedici cifre: {0, …, 9, A, B, C, D, E, F}. Poiché $16 = 2^4$, questa notazione gode della proprietà che ciascuna cifra esadecimale corrisponde a un gruppo di quattro cifre binarie. Questa proprietà rende molto semplice la conversione dei numeri binari in numeri esadecimali. Considerando il numero ottenuto sopra, 1000111101_{due}, le cifre possono essere raggruppate a quattro a quattro, partendo da destra e spostandosi verso sinistra, ottenendo così $0010\ 0011\ 1101_{due}$. Se a ciascun gruppo si associa la cifra esadecimale corrispondente, si arriva al numero esadecimale $23D_{sedici}$.

3.2.5 La codifica binaria di numeri interi con segno

Il modo più semplice per codificare i numeri interi con segno, dunque positivi e negativi, consiste nell'indicarne il segno seguito dal valore assoluto, come succede normalmente nella codifica decimale. Questa rappresentazione, chiamata *codifica con modulo e segno*, utilizza il primo bit della successione per indicare il segno (0 positivo e 1 negativo) e i restanti per la rappresentazione del valore assoluto del numero.

Tabella 3.4 Numeri interi rappresentati nel sistema binario utilizzando 4 bit e una codifica con modulo e segno.

Base due	Base dieci	Base due	Base dieci	Base due	Base dieci	Base due	Base dieci
0000	0	0100	4	1000	– 0	1100	– 4
0001	1	0101	5	1001	– 1	1101	– 5
0010	2	0110	6	1010	– 2	1110	– 6
0011	3	0111	7	1011	– 3	1111	– 7

Come si può notare dall'esempio riportato nella Tabella 3.4, con n bit questa codifica consente di rappresentare i numeri interi da $-2^{n-1}-1$ a $2^{n-1}-1$: con 4 bit a disposizione, si rappresentano i numeri interi che vanno da -7 ($= -2^{4-1}-1$) a 7 ($= 2^{4-1}-1$).

La codifica dei numeri interi in complemento a due

La codifica in modulo e segno ha il difetto di duplicare la rappresentazione del numero 0, come se –0 fosse diverso da +0, cosa che può complicare l'esecuzione e il controllo delle operazioni aritmetiche. Per questo motivo si preferisce spesso adottare una codifica diversa, detta **rappresentazione in complemento a due**, in cui, date successioni di **n** bit, per rappresentare il numero **x** si utilizza il valore binario corrispondente a $\mathbf{2^n+x}$. Con successioni di 4 bit (2^n è quindi 16), per esempio, la rappresentazione del numero 5_{dieci} è $16_{dieci}+5_{dieci} = 21_{dieci} = 10101_{due}$, ma, siccome sono disponibili solo 4 bit, viene rimosso il bit più significativo (il primo della successione), conservando solo 0101_{due}, che peraltro corrisponde alla codifica binaria del numero naturale 5_{dieci}.

Se invece si vuole rappresentare un numero negativo, per esempio -5_{dieci}, si ottiene $16_{dieci}-5_{dieci} = 11_{dieci} = 01011_{due}$, da cui, eliminando ancora il bit più significativo, si ottiene 1011_{due}.

La Tabella 3.5 riporta i numeri che possono essere rappresentati in complemento a due utilizzando 4 bit. Si può notare come il primo bit di tutti i numeri positivi sia 0, mentre i numeri negativi hanno il primo bit uguale a 1: è come se il primo bit rappresentasse il segno del numero. Il numero 0 ha ora una rappresentazione unica, anche se ciò impedisce di bilancia-

re i numeri positivi e quelli negativi: per esempio, con quattro bit, oltre allo zero si possono rappresentare 8 numeri negativi (fino a –8) ma solo 7 numeri positivi (fino a 7). Più in generale, con una rappresentazione in complemento a due, ***n*** bit sono sufficienti per codificare i numeri che vanno da -2^{n-1} fino a $2^{n-1}-1$.

Tabella 3.5 Numeri interi codificati nel sistema binario utilizzando 4 bit e una rappresentazione in complemento a due.

Base due	Base dieci	Base due	Base dieci	Base due	Base dieci	Base due	Base dieci
0000	0	0100	4	1000	–8	1100	–4
0001	1	0101	5	1001	–7	1101	–3
0010	2	0110	6	1010	–6	1110	–2
0011	3	0111	7	1011	–5	1111	–1

La codifica binaria di numeri razionali

Nelle applicazioni occorre spesso trattare con numeri razionali o reali. Il problema che si pone in questo caso è che per una rappresentazione esatta di molti di questi sarebbe richiesto un numero di cifre illimitato (si pensi al valore di $1/3 = 0.33333\ldots_{dieci}$ oppure di $\pi = 3.14159265359\ldots_{dieci}$). Poiché nella rappresentazione a uso di un esecutore automatico sono evidentemente utilizzabili solo successioni di bit di lunghezza finita, è necessario approssimare numeri di questo genere con numeri razionali il cui numero di cifre sia ridotto, ma comunque sufficiente a garantire il grado di precisione richiesto. Per esempio, π potrebbe venire approssimato come 3.14_{dieci} oppure 3.1416_{dieci}.

È utile allora definire il concetto di **cifre più significative**: nelle rappresentazioni basate sulla notazione posizionale, quale che sia la base utilizzata, le cifre più significative sono quelle associate ai pesi maggiori. Per esempio, nel numero $723\,456_{dieci}$ la cifra più significativa è 7, associata al peso 10^5, seguita da 2, con peso 10^4, e così via. Quando si considerano i numeri maggiori di 1, le cifre più significative sono quelle poste più a sinistra; nel caso di numeri minori di 1, che iniziano con 0 seguito dal separatore decimale, le cifre più significative sono le prime diverse da 0 che si incontrano andando da sinistra verso destra. Per esempio, nel numero 0.0072345_{dieci} la cifra più significativa è 7, con peso 10^{-3}, seguita da 2, con peso 10^{-4}, e così via.

Un ulteriore problema riguarda la rappresentazione di numeri con valore assoluto molto grande o molto piccolo. In questi casi, si considerano solo le cifre più significative, adottando una rappresentazione che viene generalmente indicata come **notazione scientifica**, per cui un numero viene rappresentato in base 10 come $\pm\mathbf{m}\times\mathbf{10}^{\mathbf{e}}$ (e più in generale come $\pm\mathbf{m}\times\mathbf{B}^{\mathbf{e}}$ in base **B**), cioè indicandone il segno (+ oppure –), il coefficiente **m**, detto **mantissa** (adottando la convenzione di inserire il separatore decimale dopo la prima cifra), e l'esponente **e** a cui elevare la base della numerazione. Il numero $-123\,450\,000\,000_{dieci}$ viene quindi rappresentato come $-1.2345\times10^{11}_{dieci}$, dove il segno è negativo, la mantissa è 12345_{dieci}, mentre l'esponente è 11_{dieci}; il numero 0.0000012345_{dieci}, che corrisponde a $1.2345\times10^{-6}_{dieci}$, ha invece segno positivo, mantissa 12345_{dieci} ed esponente -6_{dieci}.

Lo stesso principio viene adottato per la rappresentazione binaria dei numeri razionali: il numero 101010000_{due}, corrisponde per esempio a $1.0101\times10^{1000}_{due}$, dove il segno è positivo, la mantissa è *10101* e l'esponente è 1000. Per operare con numeri di questo tipo, è sufficiente conoscere, e quindi memorizzare, segno, mantissa ed esponente. Un problema sorge quando si deve stabilire il numero di bit da utilizzare: cambiando il numero di cifre dedicato alla rappresentazione di mantissa ed esponente si può infatti modificare la precisione dei risultati che si ottengono. L'interesse a uniformare la precisione di calcolo nelle diverse situazioni (calcolatori di produttori diversi e con strutture differenti) ha condotto alla definizione di uno standard internazionale, proposto dall'**Institute of Electrical and Electronic Engineers** (IEEE) che stabilisce la lunghezza di mantissa ed esponente.