Appunti Calcolatori Elettronici

Bozzo Francesco Conti Samuele Daniotti Filippo

Febbraio 2019

Indice

1 Introduzione ai calcolatori **1**
- 1.1 Un po' di storia dell'informatica 1
  - 1.1.1 Differenziazione dei calcolatori 1
- 1.2 Le prestazioni dei calcolatori 2
  - 1.2.1 La memoria e il processore 2
  - 1.2.2 Software di sistema 3
  - 1.2.3 Periferiche di IO 4
  - 1.2.4 Misurazione delle prestazioni 4
  - 1.2.5 La legge di Moore 5
  - 1.2.6 La barriera dell'energia 6

2 Aritmetica dei calcolatori **7**
- 2.1 Informazione nei computer 7
  - 2.1.1 I transistor 7
  - 2.1.2 Interpretazione delle informazioni 7
- 2.2 I numeri 8
  - 2.2.1 Regole di conversione tra basi 8
  - 2.2.2 I naturali 9
  - 2.2.3 Gli interi 10
  - 2.2.4 I Reali 13

Capitolo 1

Introduzione ai calcolatori

1.1 Un po' di storia dell'informatica

ENIAC, il primo computer della storia Commissionato nel 1946 dal Dipartimento di Difesa degli Stati Uniti, **ENIAC** (*Electonic Numerical Integrator and Computer*) diventa il primo calcolatore della storia. Dotato di 18000 valvole termoioniche, esso occupava una stanza di 9x30 metri, consumando un ammontare spropositato di energia.
Il suo impiego principale consisteva nel calcolare traiettorie dei proiettili di artiglieria. Infatti, è doveroso specificare come i primi calcolatori della storia siano stati concepiti per essere sfruttati in applicazioni belliche.

Apollo Guidance Computer Prodotto da IBM nel 1969, disponeva di 2800 circuiti integrati, un processore da 0.043 MHz e 152 KByte complessivi di memoria ROM/RAM. Presentava un'interfaccia display&keyboard: i comandi utilizzano una sintassi del tipo "verbo + nome".

Programma 101 Nel 1964 l'Olivetti rilascia il primo *personal computer* della storia, sfortunatamente non avrà successo.

1.1.1 Differenziazione dei calcolatori

Seppur i calcolatori di oggi condividano la stessa idea base, le soluzioni per ciascuna tipologia di applicazione sono piuttosto diverse. A seguire alcuni esempi di diverso tipo di calcolatori: Per ovvi motivi, ogni tipologia di calcolatore è meglio adatta per differenti scopi. Ottenere delle buone prestazioni senza eccedere in prezzo e potenza è ciò che decreta il successo commerciale di un prodotto.

Tipologia di calcolatore	Costo	Prestazioni	Applicazioni
Personale	Ridotto	Buone	Software prodotti da terze parti
Server	Elevato	Eccellenti	Poche ma complesse (calcolo scientifico) o tante ma semplici (web server)
Embedded	Minimo	Minime necessarie	Dedicate e molto specifiche; esiste un vasto spettro di scopi possibili

1.2 Le prestazioni dei calcolatori

Un buon programmatore, oltre a saper utilizzare degli efficienti paradgmi, deve comprendere la gerarchia di memoria e il concetto di parallelismo: conoscere a fondo il calcolatore è fondamentale.
Fino a qualche tempo fa le prestazioni di qualsiasi calcolatore erano in balia della disponibilità di memoria. Al giorno d'oggi invece risulta un problema risolto, tranne per qualche criticità per le applicazioni embedded.
Ecco una lista degli elementi che influenzano le prestazioni del calcolatore e il loro ruolo:

- *Algoritmi*: determinano il numero di istruzioni di alto livello e di operazioni di IO.
- *Linguaggi di programmazione, compilatori e architetture*: determinano il numero di istruzioni macchina per ogni istruzione di basso livello.
- *Processore e sistema di memoria*: determina quanto velocemente è possibile eseguire ciascuna istruzione.
- *Sistema di I/O (Hardware e sistema operativo)*: determina quanto velocemente possono essere eseguite le istruzioni.

1.2.1 La memoria e il processore

La memoria È possibile classificare la memoria in:

- *volatile*: è costituita da vari banchi di (tipicamente) 8 chip di RAM dinamica. È dominata dalle *DRAM.*
- *permanente*: è costituita da memorie flash (es. SSD), dischi rigidi e CD/DVD.

La prima viene utilizzata per memorizzare dati e programmi mentre vengono eseguiti (per questo motivo viene chiamata anche *memoria principale*): allo

spegnimento i dati vengono persi. La seconda viene usata per memorizzare grandi quantità di dati e programmi fra esecuzioni diverse.
Il principio di funzionamento dell'hard disk è di magnetizzare delle particelle metalliche distribuite su un substrato:

- I dischi sono organizzati in strutture sovrapposte (cilindri).
- Le particelle vengono lette da un dispositivo meccanico (testina) che si sposta radialmente su un braccio (in grado di fare movimenti angolari).
- Questa componente rallenta i tempi di accesso ma aumenta al densità di memorizzazione (è possibile arrivare facilmente ai Terabyte).

Diversamente dai dischi rigidi elettromeccanici, la memorizzazione su una memoria flash avviene intrappolando una carica elettrica in maniera permanente.
I dischi ottici funzionano sul principio di riflessione della luce: viene emesso un raggio laser che viene riflesso dai rilievi, nel caso bit 1, e assorbito dalle buche, nel caso bit 0. Nei dischi riscrivibili è inoltre presente un particolare substrato che se riscaldato torna alla condizione di partenza, eliminando tutti i dati memorizzati.

Il processore La **CPU** è l'unità centrale di ogni calcolatore. Si compone di:

- *Datapath*: esegue operazioni aritmetiche sui dati.
- *Control Unit*: impartisce ordini al datapath, alla memoria e alle componenti IO, sulla base di quanto stabilito dal programma.

Il processore offre l'**ISA** (*Instruction Set Architecture*), un'interfaccia che permette di utilizzarlo senza conoscerne i dettagli. In aggiunta all'ulteriore interfaccia del sistema operativo, insieme formano l'interfaccia binaria delle applicazioni **ABI** (*Application Binary Interface*). Ciò permette allo sviluppatore di svincolarsi dal livello hardware sottostante, secondo il principio di *astrazione*.
Esistono due diversi tipi di architetture ISA:

- **RISC**: offre un numero ridotto di istruzioni, rivolta soprattutto a sistemi *embedded*;
- **CISC**: offre Permette di usufruire di un maggior numero di istruzioni, rivolta prevalentemente ai PC.

1.2.2 Software di sistema

Il *sistema operativo*:

- gestisce le operazioni di I/O.
- alloca la memoria.
- consente il multitasking.

Il *compilatore* traduce da linguaggio ad alto livello a linguaggio macchina. Ogni istruzione di quest'ultimo è costituita da una determinata sequenza di *bit*, ossia l'unità di base dell'informazione (1 o 0).
L'utilità del compilatore consiste nel facilitare il lavoro del programmatore: non è più necessario implementare il software in codice binario, bensì è possibile utilizzare un linguaggio di programmazione più vicino al linguaggio naturale. Il primo linguaggio sviluppato per questo scopo è stato l'*assembly*. Il software che traduce l'assembly in codice binario si chiama *assembler*.
È possibile riassumere la traduzione di un linguaggio di programmazione di alto livello in codice macchina in due semplici passi:

1. traduzione del linguaggio ad alto livello in linguaggio assembly, per via del compilatore
2. traduzione del linguaggio assembly in linguaggio macchina, attraverso l'assemblatore.

Spesso fanno eccezione alcuni compilatori che trasformano direttamente il linguaggio ad alto livello in codice macchina.

1.2.3 Periferiche di IO

Il **mouse** è stato inventato nel 1967 da *Doug Engelbart* nei laboratori della *Xerox*. Attualmente sfrutta la tecnologia ottica: attraverso le variazioni di luce provocate da alcuni led che illuminano il piano, il mouse è in grado di rilevare gli spostamenti.

Gli schermi **LCD** (*Liquid Crystal Display*) sono costituiti da alcuni cristalli che galleggiano in un fluido: ciascuno di essi corrisponde ad un pixel. Attraverso un campo elettrico è possibile ruotare di 90 gradi ogni singolo cristallo, che di conseguenza può impedire o meno il passaggio della luce secondo il fenomeno fisico della *luce polarizzata*.
L'immagine dunque viene rappresentata da una matrice di pixel: ciascuno di essi ha associata una componente rosso, verde, blu (sistema RGB). Questa immagine viene memorizzata in un *frame buffer*, una RAM che viene aggiornata fino a 100 volte al secondo.

1.2.4 Misurazione delle prestazioni

I moderni processori sono costruiti usando un segnale periodico che ne sincronizza le operazioni. Il *ciclo di clock* è l'intervallo di tempo che intercorre tra due colpi di clock. La frequenza è definita come $\frac{1}{\text{ciclo clock}}$. Il ciclo di clock è misurato in secondi, la frequenza in Hertz.

Il tempo necessario per l'esecuzione di un programma dipende da tre fattori:

- il numero di istruzioni dell'algoritmo.
- i cicli di clock per istruzione (*CPI*).
- la frequenza di clock.

$$\text{Tempo CPU} = \text{Num Istruzioni} \cdot \text{CPI} \cdot \text{Periodo Clock} = \frac{\text{Num Istruzioni} \cdot \text{CPI}}{\text{Frequenza Clock}}$$

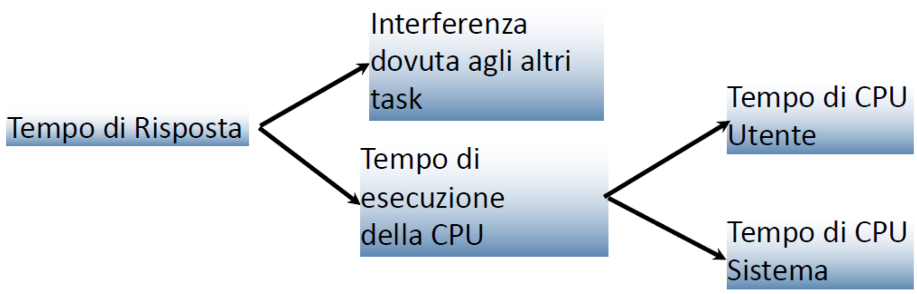


Figura 1.1: Diagramma del tempo di risposta

Un algoritmo è efficiente se viene strutturato in modo da risparmiare istruzioni e, per una particolare architettura, se utilizza le istruzioni più efficienti (quelle con un basso CPI, poichè il CPI dei diversi tipi di istruzione varia in base all'architettura che deve svolgere le istruzioni).
Il linguaggio di programmazione influenza il numero di istruzioni e il CPI: più di alto livello sono i costrutti, più lunghe sono le sequenze di istruzioni macchina ottenute traducendo il codice di partenza.
Il compilatore sicuramente influenza sia il numero di istruzioni che il CPI in base alla propria efficienza e ottimizzazione.
Anche l'*ISA* ha impatto sul numero di istruzioni, sul CPI, e sulla frequenza di clock attraverso la sua progettazione: essa può fornire istruzioni di basso o alto livello (più o meno istruzioni per eseguire un'operazione).

1.2.5 La legge di Moore

Negli scorsi anni le prestazioni dei calcolatori sono aumentati costantemente, secondo l'andamento previsto da *Moore* attraverso una legge informale.
Di recente si è assistito a una diminuzione dell'incremento tra una generazione

e l'altra, sintomo di una evidente saturazione, casuata da limiti fisici della materia.

1.2.6 La barriera dell'energia

Dagli anni Ottanta ad oggi, la frequenza media dei processori è stata aumentata di più di mille volte, con conseguente aumento dei consumi di 30 volte.

$$\text{Potenza} = \text{capacità} * \text{tensione}^2 * \text{frequenza di commutazione}$$

Dove la frequenza di commutazione è legata alla frequenza di clock.
Incrementare la frequenza senza eccedere in consumi è stato permesso grazie un abbassamento della tensione di alimentazione (termine quadratico, quindi il più influente) da 5 V a 1.2 V. Al di sotto di questo valore avvengono fenomeni di tipo elettrostatico che portano il sistema in una condizione anomala. C'è inoltre un limite alla capacità di estrarre la potenza prodotta dai processori (e il conseguente calore) tramite ventole o radiatori. Il processo di refrigerazione diventa molto costoso e difficilmente attuabile in dispositivi desktop e laptop.

La soluzione principale che è stata adottata consiste nelle *architetture multicore* sfruttando il concetto di *parallelismo*. Ciò comporta una maggiore difficoltà e responsabilità nella scrittura di codice da parte del programmatore (debugging complicato, bilanciare il carico di lavoro fra le CPU).

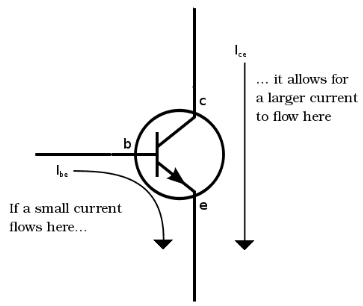
Capitolo 2

Aritmetica dei calcolatori

2.1 Informazione nei computer

2.1.1 I transistor

Tutti i computer moderni sono composti da *transistor* (detti anche triodi) che sono strutture bistabili, ovvero che possono assumere due stati come un interruttore: 0=spento e 1=acceso. Molti *transistor* collegati tra loro in una sorta di matrice possono rappresentare delle serie di porte logiche, ed è questa la struttura che sta alla base dell'architettura dei moderni calcolatori.



Un computer memorizza (e manipola) solo sequenze di 0 e 1 (sequenze di bit); anche l'ENIAC funzionava allo stesso modo, solo che al posto dei *transistor* era composto da *valvole termoioniche*, che in ultima analisi non erano differenti nel funzionamento (assumono sempre i due stati descritti in precedenza, solo più lentamente).

2.1.2 Interpretazione delle informazioni

In seguito le sequenze di bit che i computer elaborano/memorizzano possono essere interpretate in tantissimi modi diversi, tra cui numeri, caratteri, suoni, immagini, istruzioni e molti altri.
Alla base di tutte le interpretazioni che si possono dare alle stringhe di simboli (che nel caso del computer sono proprio i due stati del bit) sta il concetto di **codifica**. La **codifica** è appunto uno schema, una legge, un *mapping* che permette di tradurre prima ed interpretare poi stringhe di simboli. Nell'am-

bito dell'IT le codifiche sono fortemente caratterizzate dalla lunghezza (dal numero di bit) delle "parole" elementari della codifica. Ad esempio per rappresentare tutti i caratteri (tabella ASCII) utilizziamo 8 bit [256 caratteri]. Si osservi che essendo i *transistor* sistemi bistabili la base 2 (con cifre 0 e 1) è perfetta per rappresentare la codifica dei dati memorizzati/elaborati da essi.

2.2 I numeri

Limitiamoci al caso dei numeri per spiegare il concetto di **codifica**:
Ricordiamo che i metodi di rappresentazione numerica che studiamo sono posizionali, ovvero il peso di ogni cifra varia in base alla posizione che essa occupa.
In generale se una base è composta da B elementi essa ha B cifre (da 0 a B-1) utilizzate per scrivere ogni numero. Questa formula ci restituisce il valore di ogni numero scritto in una qualsiasi base B:

$$c_i c_{i-1} c_{i-2} ... c_0 = c_i \cdot B^i + ... + c_0 \cdot B^0$$

dove c_i è la cifra in posizione i.

2.2.1 Regole di conversione tra basi

(un elenco delle principali basi è troppo mainstream?)

Vediamo ora come operare la conversione tra le principali basi:

- **da base 2 a base 16**: partendo da destra, si dividono le cifre binarie in gruppi da 4 cifre ciascuno, e ognuno di questi corrisponderà a una cifra esadecimale; qualora il numero di cifre binarie non sia divisibile per 4, si completi con degli 0 in modo da ottenere gruppi da 4 (*esempio:* $(11011)_2$ *diventerà* $(0001\ 1011)_2$, *ovvero* $(1B)_{16}$);
- **da base 2 a base 8**: analogamente, si dividano le cifre binarie in gruppi da 3 cifre ciascuno, e ognuno di questi corrisponderà a una cifra ottale;
- **da una qualsiasi base B a base 10**: come visto sopra, si moltiplica ogni cifra c_i per B^i, dipendentemente dalla sua posizione;
- **da base 10 a una qualsiasi base B**:
  1. si consideri un numero x_{10} e una base B;
  2. si divida x per B;
  3. il resto della divisione è la cifra da inserire a sinistra nel numero convertito;

4. si assegni a x il quoziente della divisione;
5. si torni al punto 2 e si ripeta finché $x \neq 0$.

$1000_{10} = ?_2$

X	$X/2$	$X\%2$
1000	500	0
500	250	0
250	125	0
125	62	1
62	31	0
31	15	1
15	7	1
7	3	1
3	1	1
1	0	1

$1000_{10} = 1111101000_2$

$1000_{10} = ?_{16}$

X	$X/16$	$X\%16$
1000	62	8
62	3	14 (E)
3	0	3

$1000_{10} = 3E8_{16}$

Figura 2.1: Alcuni esempi di conversione

2.2.2 I naturali

Nella codifica binaria un numero naturale è rappresentato su k cifre binarie, dove con k cifre si possono rappresentare i numeri tra 0 e $2^k - 1$.
La codifica più comune per gli interi spesso usa i byte, sequenze di 8 bit, che tuttavia richiedono molte cifre per rappresentare un numero e quindi spesso, per semplificare la lettura, si usa scrivere i numeri in esadecimale (quindi scrivendo un quarto delle cifre rispetto alla binaria).

Somma e sottrazione Le operazioni somma e sottrazione funzionano allo stesso modo del sistema utilizzato nella numerazione decimale con il riporto. Si guardino gli esempi qui sotto riportati, già di per sè esplicativi del processo:

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

Figura 2.2: Esempio di somma

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

Figura 2.3: Esempio di differenza

Moltiplicazione La moltiplicazione in base binaria si può semplificare con il metodo dello shifting, che è più facile illustrare con un esempio:

				1	1	0	1	x
			1	0	0	0	1	=
				1	1	0	1	+
						-		+
					-			+
				-				+
1	1	0	1					
1	1	0	1	1	1	0	1	

Figura 2.4: Esempio di prodotto

Si noti che quando è presente un 1 si riscrive il numero (nella posizione corrispondente) mentre quando è presente uno 0 semplicemente non si scrive nulla (e si va solo avanti con le posizioni).

2.2.3 Gli interi

Finora abbiamo visto come si codificano solo i numeri naturali, ma esistono metodi di codifica che permettono di rappresentare anche i numeri negativi (quindi l'insieme degli interi). Le tecniche più comuni sono: modulo e segno, complemento a 1 (*CA1*) e complemento a 2 (*CA2*).

Codifica con Modulo e Segno: Idea semplice: si usano $k-1$ bit per rappresentare il valore assoluto (modulo) del numero ed un bit per codificare il segno (0=positivo, 1=negativo). In questo modo con k bit si possono codificare valori tra $-2^{k-1}+1$ e $+2^{k-1}-1$.
N.B.: esistono due codifiche per lo 0, -0 e $+0$, e questo è un spreco.

Codifica in Complemento a 1: L'idea alla base è semplice, il primo bit indica sempre il segno e per ottenere l'opposto di un numero positivo si invertono tutti gli 0 in 1 e gli 1 in 0 (e lo stesso si fa per ottenere l'opposto di un numero negativo). Si hanno ancora due rappresentazioni per +0 e -0 (rispettivamente, utilizzando 4 bit, 0000 e 1111). È più facile da sommare rispetto al modulo e segno ma solo se il bit significativo non da riporto. Quando si sommano due numeri in *CA1*:

Esempio: $6 + -3$ (codifica su 5 bit)

$6 = 00110; -3 = \overline{00011} = 11100$

1	1	1				(Riga dei riporti)
	0	0	1	1	0	
	1	1	1	0	0	
	0	0	0	1	0	

$00010 + 1 = 00011 \; (= 3)$

Figura 2.5: Esempio di somma in CA1

- Si deve verificare che i riporti delle prime due cifre significative siano uguali, altrimenti il numero che si ottiene non è rappresentabile in CA1 su k bit.
- Alla fine si deve sommare il riporto della prima cifra al risultato ottenuto (ultima riga).

Codifica in Complemento a 2: Per tradurre un numero da intero con segno a complemento a 2 (*CA2*) basta invertire tutti i bit e poi sommare 1. Viceversa, per convertire da *CA2* a binario, si sottrae 1 e poi si invertono tutti i bit.
Ancora una volta il bit più significativo indica il segno, ma in questo caso la codifica dello 0 è unica, quindi con k bit si possono rappresentare i numeri da -2^{k-1} a $2^{k-1} - 1$. Anche la somma è più semplice ma questo lo vedremo in futuro.

Somma algebrica in Complemento a 2

L'utilizzo della codifica *CA2* permette di eseguire somme algebriche senza alcuna differenza operativa, ma rende necessario prestare attenzione agli *overflows*.
Con il termine *overflow* si intende la situazione in cui, date in input delle informazioni codificate in un numero k di bit arbitrariamente scelto, lo stesso

numero k di bit non è sufficiente a codificare le informazioni in output.
Diamo subito un esempio: supponiamo di voler eseguire $-64 - 8$ utilizzando il *CA2* e $k = 7$ bit. Ricordiamo che:

$-64 = 1000000_2 = \overline{1000000} + 1 = 0111111 + 1 = 1000000$ in *CA2;*
$-8 = 0001000_2 = \overline{0001000} + 1 = 1110111 + 1 = 1111000$ in *CA2*.

A questo punto eseguiamo la somma algebrica

$$\begin{array}{cccccccc}
(1) & & & & & & & \\
 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
\hline
 & 0 & 1 & 1 & 1 & 0 & 0 & 0
\end{array}$$

Figura 2.6: (Il numero fra parentesi indica l'ultimo riporto, che chiaramente non possiamo inserire poiché sforerebbe i 7 bit)

Osserviamo così che, per $k = 7$ bit, $-64 - 8 = 0111000_2 = 56_{10}$, che ovviamente non ci piace; concludiamo quindi che la somma algebrica $-64-8 = -72$ *non* è rappresentabile con soli $k = 7$ bit.

Possiamo dire che, in generale, dati due numeri a, b aventi lo stesso segno, avremo probelmi di *overflow* se

- $a + b > 2^{k-1} - 1$
- $a + b < -2^{k-1}$

ricordando che $-2^{k-1} \leq a, b \leq 2^{k-1} - 1$.

Aritmetica modulare

Incosapevolmente noi utilizziamo l'*aritmetica modulare* quasi continuamente, nella nostra quotidianità; l'esempio più comune è quando sommiamo le ore, che sono organizzate in modulo 24 (da cui la denominazione ufficiosa di "Matematica dell'orologio"), ma anche minuti e secondi (modulo 60) e gli angoli (modulo 360).
Diamone una descrizione intuitiva: fissato un numero n finito di valori, arrivato al numero più grande $(n - 1)$, al successivo avrò che $(n - 1) + 1 = 0$.
Per completezza, forniamo ora la relazione di equivalenza di due numeri in modulo n:

$$a \equiv b \longleftrightarrow a\%n = b\%n$$

Quando noi eseguiamo delle operazioni sui numeri in base 2 su k bit, stiamo lavorando in un ambiente in modulo 2^k, e calcolando $a+b$ stiamo in realtà trovando $(a+b)\%2^k$, e i due valori saranno uguali se e solo se $a+b < 2^k$, perché altrimenti ogni valore successivo ricomincerebbe da 0: questo è l'*overflow*.
In particolare quando lavoriamo su numeri interi in *CA2* dobbiamo ricordare che, se $-2^{k-1} \leq a+b \leq 2^{k-1}-1$ non avremo *overflow*, altrimenti

- se $a+b < 2^{k-1}-1$, otterremo $a+b+2^k \longrightarrow$ *OVERFLO(W)!*
- se $a+b > -2^{k-1}$, otterremo $a+b-2^k \longrightarrow$ *OVERFLO(W)!*

2.2.4 I Reali

Non è sempre semplice codificare un numero reale $\alpha \in \mathbb{R}$, poiché di questo insieme fanno parte i numeri cosiddetti *irrazionali*, che presentano infinite cifre non periodiche dopo la virgola (si pensi ad esempio a $\pi = 3,14159...$, o $e = 2.71828...$, o ancora $\sqrt{2} = 1.41421...$).
Alcuni numeri non possono quindi essere rappresentati con un numero finito di bit, per cui ne rappresenteremo solo un'approssimazione, e per farlo abbiamo sviluppato due metodi: la rappresentazione a **virgola fissa** (o *fixed point*) e quella a **virgola mobile** (o *floating point*).

Codifica in virgola fissa

Questo sistema di cofidica prevede che, posto un numero k di bit per la rappresentazione del numero, si impieghino $k-f$ bit per rappresentare la parte intera, e i restanti f bit per rappresentare la parte razioneale:

$$\overbrace{\underbrace{c_{k-1}c_{k-2}\cdots c_f}_{k-f}\cdot\underbrace{c_{f-1}\cdots c_0}_{f}}^{k}$$

Possiamo scrivere una formula generale più compatta. Supponiamo di voler convertire un numero x_{10}:

$$x_{10} = \sum_{i=0}^{k-1} c_i B^{i-f} = \sum_{i=0}^{k-1} c_i B^i \cdot B^{-f} = (\sum_{i=0}^{k-1} c_i B^i) \cdot B^{-f}$$

Detta in altri termini, convertiamo il numero da base 10 come se non avesse la virgola e poi moltiplichiamo per B^{-f}.
Presentiamo vantaggi e svantaggi di questo tipo di codifica:

- le operazioni si possono svolgere senza alcuna differenza, se non l'accortezza di scalare il risultato;

- il peso per la CPU è ridotto;
- non ci sono errori di approssimazione, *se il valore è rappresentabile*;
- iniziamo ad avere problemi quando lavoriamo con un ampio spettro di ordini di grandezza.

(esempio di conversione?)

Codifica in virgola mobile

Un qualsiasi numero reale può essere scritto anche nella forma $x = M \cdot B^E$ dove M si dice *mantissa*, E *esponente* e B è la base della rappresentazione (che nel nostro caso è la leggendaria base 2).
Questo metodo di rappresentazione, del tutto analogo notazione scientifica standard, è implementato anche nell'informatica in più modi diversi ma alla base di tutti si riconosce lo stesso schema:
Un numero x può essere rappresentato su k bit utilizzando m bit per il campo mantissa M ed $e = k - m$ bit per il campo esponente E; quest'ultimo permette in un certo senso di "spostare" la virgola; con esponenti negativi si avranno x "piccoli" e viceversa con esponenti positivi si avranno x "grandi".
Lo standard più comunemente utilizzato è l'IEEE 754 che, nei numeri *float* a precisione singola (che impiega 32 bit), ha questa struttura:

1 bit	8 bit	23 bit
Bit di segno	Esponente	Mantissa

Il bit di segno ha la funzione che già conosciamo, mentre il campo E può contenere valori compresi tra 0 e 255; questi ultimi due valori vengono utilizzati per funzioni speciali, mentre i valori da 1 a 254 compresi codificano l'esponente effettivo dei calcoli, che diremo *exp*, in questo modo:

$$exp = E - 2^{e-1} - 1$$

Nel nostro caso, $exp = E - 127$. L'ultimo campo, la *mantissa*, codifica un numero intero.

Come calcolare il valore di un *float* Il primo bit è utilizzato per indicare il segno, i successivi 8 per E e i restanti 23 per la mantissa. La codifica assume due forme in base al valore di E:

- se $E = 0$, allora il numero è $x_{10} = 0.M \cdot 2^0$;
- se $E > 0$, allora il numero è $x_{10} = 1.M \cdot 2^{exp}$, dove exp è definito come prima.