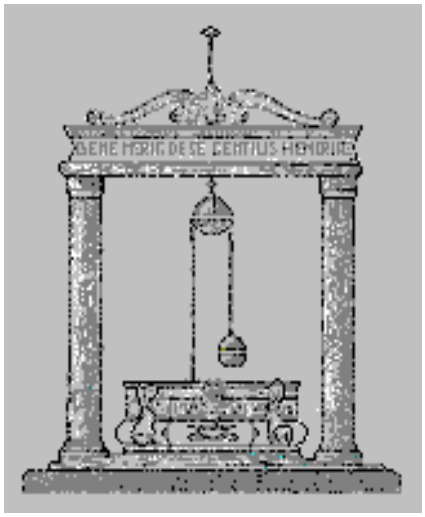
Corso di Laurea in Ingegneria Elettronica

Complementi al corso di
Fondamenti di Informatica

Architettura del calcolatore e rappresentazione dell'informazione

Daniela D'Aloisi



Anno Accademico 2000-2001

Indice

Parte I
Architettura di un calcolatore

1 Introduzione

Il calcolatore (computer, elaboratore o più semplicemente *macchina*) è nato negli anni quaranta come strumento dedicato alla risoluzione di complessi calcoli scientifici. Attualmente è diventato un oggetto di uso quotidiano, impiegato per moltissime e diverse attività come archiviare dati, navigare in rete, giocare, etc. L'attività a cui siamo interessati riguarda il modo in cui il calcolatore risolve problemi in modo automatico attraverso l'uso di un linguaggio di programmazione.

Studieremo quindi gli aspetti di un calcolatore che servono per imparare a scrivere un programma, ossia a costruire una soluzione automatica per un problema dato attraverso l'acquisizione di una metodologia di programmazione, di algoritmi standard e di un linguaggio programmazione. La capacità di programmare, ossia di risolvere problemi in modo automatico attraverso il progetto di un *algoritmo risolutivo*, dovrebbe essere acquisita indipendentemente dal linguaggio usato, il quale deve diventare lo strumento attraverso cui l'algoritmo diventa effettivo.

Durante tutto il corso si vedrà come costruire il procedimento risolutivo e si studierà il linguaggio C per rendere esecutivo tale procedimento.

In questa dispensa introdurremo alcuni concetti di base che sono utili per capire il funzionamento di un calcolatore e come un programma viene trattato al suo interno. Introdurremo alcune nozioni fondamentali cercando di capire come si arriva alla scrittura di un programma partendo da un problema. Studieremo in modo schematico l'architettura di un calcolatore e quale è il linguaggio che esso comprende. Infine vedremo come sono rappresentate le informazioni in un calcolatore.

2 Algoritmi e programmi

Scopo del corso non è imparare ad usare il calcolatore in modo "generico", ma imparare a *programmare*, ossia imparare a risolvere classi di problemi in modo automatico tramite una procedura che possa poi essere tradotta in un linguaggio interpretabile dal calcolatore.
Dato un problema (o più specificamente una classe di problemi) si deve:

- trovare un procedimento risolutivo, ossia un *algoritmo*;
- tradurre l'agoritmo in un linguaggio comprensibile dal calcolatore, ossia trasformarlo in un *programma* scritto in un determinato *linguaggio di programmazione*;
- fornire dei *dati in ingresso (input)* al programma;
- verificare che i *dati di uscita (output)* siano quello attesi.

Quindi un calcolatore è una macchina che esegue *ordini* dati sotto forma di *istruzioni* in un *linguaggio di programmazione*. Le istruzioni sono sintetizzate in un *programma*. Il programma permette l'elaborazione dei dati in ingresso per produrre dei dati in uscita.
Definiamo un po' meglio i concetti di algoritmo, linguaggio di programmazione e programma.

Algoritmo *L'algoritmo è un processo di trasformazione di un insieme di dati iniziali in un insieme di risultati finali mediante una sequenza di passi risolutivi (istruzioni).*

Linguaggio di programmazione *Un linguaggio di programmazione è un linguaggio o strumento per rappresentare le istruzioni di un algoritmo e la loro concatenazione.*

Programma *Un programma è un algoritmo scritto in un linguaggio di programmazione al fine di comunicare al calcolatore elettronico le azioni da intraprendere, quindi è la traduzione dell'algoritmo nel linguaggio di programmazione scelto.*

Dato un problema, l'operatore scriverà un algoritmo risolutivo: l'algoritmo esprime i passi elementari che devono essere compiuti per arrivare alla soluzione. Ogni passo elementare dell'algoritmo deve rappresentare operazioni che il calcolatore è in grado di compiere. L'algoritmo sarà poi tradotto in un corrispondente programma scritto in C (o in un qualsiasi altro linguaggio di programmazione). Il programma sarà composto da una sequenza di istruzioni, traduzione dei passi dell'algoritmo, che rispettano un ordine di esecuzione. Alla fine il programma in C sarà sottoposto ad una serie di trasformazioni che lo renderanno eseguibile dal calcolatore.

Un algoritmo deve necessariamente soddisfare tre condizioni per essere definito tale, ossia deve essere *non ambiguo, eseguibile, finito.*

Non ambiguo significa che le istruzioni devono essere univocamente interpretate dall'esecutore. Una istruzione deve essere interpretabile in un solo modo.

Eseguibile significa che il calcolatore deve essere in grado, con le risorse a sua disposizione, di eseguire le istruzioni.

Finito significa che l'esecuzione dell'algoritmo deve terminare in tempo finito per ogni insieme dei valori d'ingresso.

Inoltre, quando si progetta un algoritmo, si devono rispettare alcuni prerequisiti necessari per ottenere poi un buon programma:

- L'algoritmo deve essere indipendente dal linguaggio di programmazione scelto. Esistono dei linguaggi per rappresentare algoritmi (il più noto è quello dei grafi di flusso), ma possiamo usare anche frasi in linguaggio naturale oppure appositi linguaggi chiamati *pseudo-codici*.
- Il procedimento risolutivo deve essere riusabile, ossia non deve essere scritto per risolvere una sola istanza di un dato problema.
- L'algoritmo è la soluzione al problema.
- L'algoritmo deve essere traducibile in un linguaggio di programmazione.

La progettazione di un algoritmo può essere divisa in due fasi:

1. Analisi e definizione dei dati: si analizza il problema e si sceglie come rappresentare i dati che si hanno a disposizione e quelli che si vogliono ottenere.
2. Specifica della soluzione: corrisponde alla scelta dei passi che portano alla soluzione, quindi alla scrittura dell'algorimo.

Le due fasi sono strettamente legate: la fase di analisi dei dati è importante quanto la seconda fase. Una cattiva analisi può pregiudicare la costruzione dell'algoritmo. Inoltre la soluzione può essere condizionata dal tipo di rappresentazione dei dati scelta. Facciamo un esempio pratico. Supponiamo che il problema sia cercare un numero di telefono. Il procedimento che adotteremo sarà diverso se partiamo da un elenco telefonico oppure da una rubrica. L'elenco telefonico e la rubrica sono le rappresentazioni che scegliamo per i nostri dati. Nel caso dell'elenco telefonico dovremo effettuare una ricerca per trovare la lettera muovendoci avanti e indietro a seconda se siamo posizionati prima o dopo quella cercata. Nell'agenda potremo posizionarci direttamente sulla pagina relativa alle lettera. In entrambi i casi, una volta arrivati alla lettera giusta, effettueremo una ricerca sequenziale per individuare il nome cercato.

3 L'architettura di von Neumann

La struttura di un calcolatore, ossia la sua architettura, è ancora quella classica sviluppata nel 1947 da John von Neumann. L'avanzamento della tecnologia ha enormemente migliorato le componenti, ma in linea di massima le funzionalità e il coordinamento sono rimasti quelli classici. Sono state anche introdotte architetture diverse (per esempio macchine parallele), ma la stragrande maggioranza dei calcolatori mantiene la stessa struttura della macchina di von Neumann.

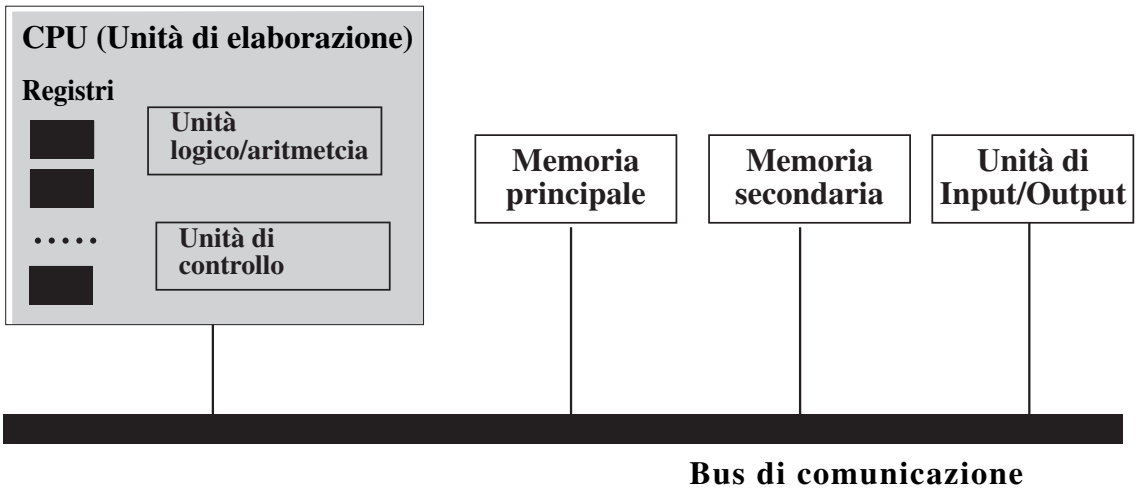


Figura 1: L'architettura di von Neumann

Introduciamo l'architettura di von Neumann tramite un parallelo tra le funzionalità richieste a un calcolatore e le componenti dell'architettura reale (Figura 1).

La principale caratteristica di un calcolatore è la capacità di memorizzare i dati e il programma che sta eseguendo: i componenti delegati a tale compito sono le *unità di memoria*.

I dati necessitano poi di essere elaborati tramite diversi tipi di operazioni. Si devono anche mantenere informazioni opportune sugli operandi e sulle istruzioni che si stanno eseguendo al momento. Tali compiti spettano rispettivamente all'*unità logico/aritmetica* e ai *registri*.

È necessario inoltre che il calcolatore sia capace di colloquiare con l'esterno, poiché i dati devono essere scambiati sia in uscita che in ingresso: queste funzionalità sono a carico delle *unità di ingresso/uscita (input/output)*.

Le componenti del sistema si scambiano dati e informazioni attraverso il *bus di comunicazione*.

L'*unità di controllo* svolge un'azione di coordinamento, governando il funzionamento complessivo della macchina attraverso il controllo dell'esecuzione delle singole istruzioni di un programma.

3.1 Le memorie

La memoria principale e la memoria secondaria svolgono compiti diversi e hanno anche diverse caratteristiche. La memoria principale gioca un ruolo cruciale in fase di esecuzione del programma, mentre la memoria secondaria è usata per mantenere in modo permanente dati e programmi.

3.1.1 La memoria principale

La *memoria principale (o centrale)* è una struttura mono-dimensionale suddivisa in celle della stessa dimensione (Figura 2). La dimensione di una cella (parola) è costante e dipende dal particolare tipo di elaboratore: la dimensione viene misurata in *bit* o in *byte*. La parola *bit* sta per **bi**nary digi**t**: un byte è formato da 8 bit. Un bit può assumere solo due valori: 0 e 1. La dimensione della singola cella è data da multipli di 8. Alcuni esempi: il PDP-11 ha 8 bit per cella, quindi parole di 8 bit o 1 byte; i processori della serie Pentium della Intel hanno 32 bit per cella (parola di 32 bit o 4 byte).

Una cella di memoria è formata da una coppia $\langle indirizzo, contenuto \rangle$. La funzione che associa un indirizzo all'informazione è una funzione non invertibile: una cella è identificata in modo *univoco* dal suo *indirizzo*, che è di tipo numerico (espresso in codice binario); celle di indirizzo diverso possono contenere la stessa informazione. Il *contenuto* di una cella viene rappresentato come una sequenza di k valori binari. Quindi sia la codifica dei dati che la codifica dell'indirizzo avviene in forma di sequenze di bit.

indirizzo	contenuto
0	00101100
1	10111001
2	00000000
3	11110000
4	00000000
	⋮

Figura 2: Struttura della memoria principale

Accedere ad un dato significa selezionare mediante l'indirizzo la cella in cui esso è memorizzato e prelevare il valore. *Memorizzare* un dato richiede di specificare la cella in cui si vuole inserirlo.

Il numero di bit necessari per indirizzare una memoria, ossia per assegnare un indirizzo univoco ad ogni cella, dipende dal numero di celle: per esempio, per indirizzare 30 celle, serviranno 5 bit, infatti $2^5 = 32$. Si noti che 4 bit sarebbero insufficienti.

L'indirizzo fisico di una cella è importante perché permette al calcolatore di individuare le celle. Nei linguaggi di programmazione ad alto livello non sarà necessario riferirsi ad essa con l'indirizzo fisico, ma si userà un indirizzamento *simbolico*, ossia ci si riferirà ad essa attraverso un nome.

Per misurare la dimensione di una memoria si fa riferimento al numero di celle e alla lunghezza in bit di una singola parola. La capacità è misurata in bit (o byte se dividiamo per il fattore 8): abbiamo varie volte visto che una memoria è misura *256K* o *128M* o *1G*. Il simbolo K è un'abbreviazione per *kilo*, che non sta esattamente per 1000 (ossia 10^3), ma per 1024 che è pari a 2^{10}. Anche i simboli M e G non indicano potenze di 10 ma di 2. Nella Tabella 1 sono riportate le corrispondenze esatte.

Per ottenere la dimensione di una memoria dobbiamo moltiplicare il numero di celle per la loro lunghezza. Una memoria di 128KB (Kilo byte) con parole di 16 bit contiene:

- 64×1024 parole, oppure
- $128 \times 1024 \times 8$ bit.

La memoria principale è anche indicata come *RAM* (Random Access Memory). In essa vengono caricati sia il programma da eseguire che i dati, ossia tutta l'informazione necessaria alla esecuzione del processo corrente.

Una RAM contiene informazione detta *a breve o medio termine*, poiché viene mantenuta per il tempo necessario all'esecuzione. Al termine dell'esecuzione del programma il contenuto della RAM non è più disponibile. Questo genere di memoria è detta *volatile* perché i dati in essa contenuti si perdono con lo spegnimento del calcolatore.

Le dimensioni della memoria principale sono ridotte se comparate a quelle delle memorie secondarie, che raggiungono ormai l'ordine delle decine di giga. Le dimensioni possono essere relativamente ridotte, perché la memoria contiene solo dati e istruzioni necessari all'esecuzione in corso. Non deve contenere *tutti i dati* e *tutto il programma* in esecuzione, ma solo le parti necessarie all'elaborazione corrente.

Il programma e i dati saranno generalmente mantenuti nelle memorie secondarie: sono *trasferiti in*

Simbolo	Termine	corrisponde a	approssimato come
K	Kilo	2^{10}	10^3
M	Mega	2^{20}	10^6
G	Giga	2^{30}	10^9
T	Tera	2^{40}	10^{12}
P	Peta	2^{50}	10^{15}

Tabella 1: Potenze di 2 significative in informatica

memoria centrale solo al momento dell'esecuzione. La memoria centrale è un passaggio obbligato per tutte le informazioni che, per essere elaborate, devono essere prima acquisite da essa.

La memoria principale e quelle secondarie sono costruite con diverse tecnologie: la memoria principale è molto più veloce di quelle secondarie. La velocità di una memoria misura il suo *tempo di accesso*, ossia il tempo necessario a prelevare un dato. Il tempo di accesso si misura come l'intervallo di tempo che intecorre dall'istante in cui è dato l'ordine di recuperare (*fetch*) un'informazione dalla memoria e l'istante in cui tale informazione è disponibile per l'unità centrale.

Da un punto di vista tecnologico, la memoria principale ha tre importanti caratteristiche:

- È una memoria ad *accesso random* (RAM), ossia il tempo di accesso alle celle è indipendente dalla loro posizione ed è quindi costante per tutte le celle.
- È una memoria ad *alta velocità*, ossia il suo tempo di accesso è molto basso. Per esempio, per un PentiumIII attuale è circa 10 nanosecondi, ossia un 100-milionesimo di secondo. Tuttavia i progressi tecnologici continuano ad abbassare tale soglia.
- È una memoria a *lettura* e *scrittura* (read/write), ossia l'informazione può essere sia letta che scritta. Infatti, le due operazioni che possono essere effettuate su una RAM sono: *prelievo non distruttivo* del contenuto di una cella, che corrisponde all'operazione di lettura del contenuto; *scrittura distruttiva* del contenuto di una cella, che corrisponde all'operazione di scrittura. Queste due operazioni vengono effettuate con velocità di accesso costante (ossia indipendente dalla cella alla quale si sta accedendo) e hanno un tempo di esecuzione comparabile.

3.1.2 La memoria secondaria

La memoria secondaria è costituita da uno o più dispositivi per mantenere informazioni permanenti o a lungo termine. Le memorie secondarie sono memorie *non volatili* poiché il loro contenuto si mantiene anche dopo lo spegnimento del calcolatore.

Esistono diversi tipi di memoria secondaria. La piú comune è il *disco rigido* che memorizza in modo permanente dati e programmi applicativi. La permanenza dei dati è limitata solo dalla cancellazione volontaria degli stessi per mezzo di apposite istruzioni. I dischi rigidi raggiungono ormai dimensioni dell'ordine delle decine di giga anche in piccoli calcolatori.

Altre memorie secondarie sono i floppy disk (dischetti), gli zip, i CDRom, i DVD e i nastri. Tutte queste memorie hanno diverse caratteristiche strutturali (tecnologia, velocità di accesso, dimensioni) ma hanno in comune la capacità di mantenere i dati in modo permanente.

Il tempo di accesso alle memorie secondarie è molto lento rispetto alla memoria principale. Il tipo di accesso a queste memorie è in genere di tipo *sequenziale*: per accedere ad una locazione è necessario accedere a tutte le locazioni che la precedono, aumentando così il tempo di accesso. Il tempo di accesso dipende inoltre dalle caratteristiche costruttive del dispositivo: per esempio, l'accesso ad un disco rigido è molto più veloce di quello ad un floppy.

3.2 I dispositivi di input/output e il bus

I *dispositivi di input/output (I/O)* permettono l'immissione e/o l'uscita dei dati. Tipicamente tramite un dispositivo di input saranno prelevati dati necessari all'elaborazione del programma e tramite i dispositivi di output saranno comunicati i risultati di una elaborazione. Tipici dispositivi di input sono la tastiera e il mouse; dispositivi di output sono il video e la stampante.

Il *bus* è il canale centrale di comunicazione di un calcolatore. Attraverso il bus viaggiano i dati, gli indirizzi e i comandi che sono scambiati tra i dispositivi ad esso collegati. Il bus è costituito da diverse linee, ed è suddiviso in *bus dati*, *bus inidrizzi* e *bus comandi*.

3.3 La CPU

La *CPU (Central Processing Unit)* o unità di elaborazione è preposta al coordinamento di tutte le componenti che formano l'architettura di un calcolatore. Il suo compito principale è quello di interpretare ed

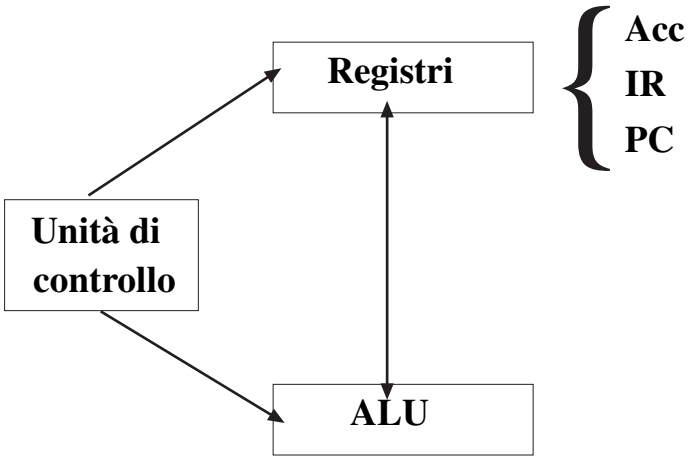


Figura 3: Struttura della CPU

eseguire le istruzioni elementari che compongono il programma. Essa consiste di tre componenti principali: l'*unità logico-aritmetica* (ALU), i *registri* e l'*unità di controllo* (Figura 3) che è la parte piú importante di questo blocco.

Infatti è l'unità di controllo che svolge il compito di coordinamento e mantiene il controllo sulla memoria centrale. Essa reperisce dalla memoria centrale le istruzioni da eseguire. Le istruzioni vengono trasferite in appositi registri, detti *registri istruzioni (IR)*, che si trovano all'interno della CPU e sono a disposizione dell'unità di controllo. Le istruzioni vengono decodificate (con il supporto di appositi organi di decodifica), ossia vengono interpretate e capite per potere essere eseguite.

L'esecuzione delle istruzioni è il compito successivo dell'unità di controllo: essa attiva le unità necessarie a seconda di quanto contenuto nelle istruzioni. Per esempio, attiverà la memoria per trasferire dei dati, l'unità logico-aritmetica per fare eseguire delle operazioni e le unità di I/O per trasferire i dati necessari o prodotti.

L'unità logico-aritmetica (ALU) effettua i calcoli logici e aritmetici. È in grado di effettuare delle semplici operazioni "matematiche" partendo da dati contenuti in certi registri memoria che sono a sua disposizione. Le istruzioni che l'ALU è in grado di compiere dipendono dal tipo di calcolatore. L'ALU può effettuare generalmente solo operazioni che hanno al più due operandi. Un esempio di operazione è la seguente:

somma il contenuto del registro R1 a quello del registro R2.

L'ALU riceve dall'unità di controllo il comando che deve eseguire e l'operando (o gli operandi) su cui l'operazione va effettuata. Per eseguire l'operazione l'ALU ha a disposizione alcuni registri speciali il cui numero dipende dal tipo di calcolatore.

In generale, i *registri* della CPU sono celle di memoria ad accesso molto veloce che contengono istruzioni e dati ed hanno un ruolo speciale nell'ambito dell'esecuzione delle istruzioni. I registri più importanti sono elencati di seguito:

- Il *registro istruzione* (IR per *Instruction Register*) contiene l'istruzione in corso di esecuzione nella CPU.
- Il *contatore di programma* (PC per *Program Counter*) contiene l'indirizzo della successiva istruzione da eseguire. Il contenuto di questo registro riflette il flusso del programma, perché mantiene traccia della sequenza di istruzioni eseguite.
- Gli *accumulatori* (ACC) contengono gli operandi di una data istruzione e alla fine dell'operazione contengono il risultato dell'istruzione eseguita. Come detto in precedenza, le istruzioni possono avere al massimo due operandi. È importante notare che l'esecuzione di una qualsiasi operazione

richiede che uno degli operandi sia memorizzato nell'accumulatore. Le operazioni avvengono tutte tra l'accumulatore ed un registro, e il risultato viene poi memorizzato nell'accumulatore. Il numero di registri accumulatori dipende dalla macchina.

- Il *registro dei flag*, che mantiene alcune informazioni sul risultato dell'ultima operazione eseguita dall'ALU. Ogni bit di questo registro ha un significato particolare e indica il verificarsi o meno di una certa condizione. I più importanti bit del registro dei flag sono:
  - bit di *zero*, che viene posto a 1 se il risultato dell'ultima operazione è pari a 0;
  - bit di *segno*, che viene posto a 1 se il risultato dell'ultima operazione è negativo;
  - bit di *riporto* (carry), che viene posto a 1 se l'ultima operazione ha generato un riporto;
  - bit di *trabocco* (overflow), che viene posto a 1 se l'ultima operazione ha generato un trabocco.

È da notare che la gestione dei registri è completamente trasparente per l'utente, nel senso che il linguaggio nasconde queste operazioni di basso livello.

Come detto in precedenza, l'unità di controllo di fatto coordina le unità del calcolatore in modo da rendere possibile l'esecuzione di un programma. Il programma è memorizzato in celle di memoria consecutive, e su questo l'unità di controllo lavora seguendo un ciclo continuo, detto ciclo di *prelievo-decodifica-esecuzione*.

1. La prima fase è quella di *prelievo (fetch)*. In questa fase, l'unità di controllo acquisisce una istruzione in memoria e individua l'istruzione successiva da eseguire.
2. Nella fase di *decodifica* (decode), viene decodificato il tipo di istruzione.
3. Nella fase di *esecuzione (execute)*, vengono attivati i comandi per realizzare l'azione specificata dall'istruzione:
   - se è una istruzione di *ingresso* dati, comanda all'unità di ingresso interessata di trasferire i dati nella memoria centrale;
   - se è una istruzione di *uscita* dati, comanda all'unità di uscita di trasferire i dati dalla memoria centrale all'esterno;
   - se è una istruzione di *elaborazione* dati, comanda il trasferimento dei dati nell'ALU, comanda all'ALU di eseguire l'operazione voluta e comanda poi il trasferimento dei dati nella memoria centrale;
   - se è un'istruzione di *salto*, aggiorna il program counter (PC) all'indirizzo a cui saltare (specificato nell'operando dell'istruzione).

In pratica, l'unità di controllo esegue all'infinito sempre lo stesso ciclo:

```
ripeti all'infinito
    accedi all'istruzione indirizzata dal PC;
    decodifica l'istruzione;
    esegui l'istruzione;
    se l'istruzione non è un salto
        allora incrementa PC alla prossima istruzione
```

Si noti come le istruzioni dettino il flusso del programma: le istruzioni vengono eseguite in sequenza (ossia come sono scritte nel programma) a meno che l'istruzione non comporti un salto ad un punto diverso del programma stesso.

Tutte le operazioni dell'unità di controllo sono regolate da un orologio interno al sistema. L'orologio è un circuito che genera impulsi regolari, ad una certa frequenza espressa in Mhz. Questo corrisponde al *clock* del sistema: generalmente diciamo che un certo processore funziona a 500, 600, 800 Mhz. I segnali regolari dell'orologio permettono alle varie parti di operare in modo coordinato. L'esecuzione di una istruzione richiede al massimo una decina di cicli di clock. Quindi un calcolatore con clock a 500 Mhz permette di eseguire ca. 50 milioni di istruzioni al secondo (50 MIPS, Millions of Instructions Per Second).

4 Dal codice sorgente al codice oggetto

Parlare di algoritmi e programmi permette di astrarre dalla reale struttura del calcolatore. A basso livello, un calcolatore è semplicemente uno strumento digitale che funziona comprendendo segnali di tipo 0/1 (assenza/presenza di segnale), ossia sequenze di 0 e 1, che costituiscono il cosiddetto *linguaggio macchina*.

Un programma può essere visto a diversi livelli di astrazione a seconda del linguaggio con cui viene espresso. I linguaggi aumentano di potere espressivo man mano che ci si allontana dalla macchina. Possiamo introdurre la seguente classificazione:

Linguaggio macchina (detto anche linguaggio assoluto o codice binario): è al più basso livello di astrazione. Consiste di sequenze di **0** e **1** che codificano le istruzioni da eseguire. Queste istruzioni corrispondono a operazioni elementari che sono però sufficienti all'implementazioni di operazioni molto complesse. Un linguaggio macchina dipende dallo specifico tipo di calcolatore, ed ogni calcolatore dispone di un proprio insieme di istruzioni, ossia l'insieme di istruzioni che è in grado di eseguire.

Linguaggio assemblativo: è intermedio tra il linguaggio macchina e quelli ad alto livello. Anche il linguaggio assemblativo dipende dallo specifico calcolatore, ma invece di una codifica binaria usa delle parole e dei simboli (tipicamente abbreviazioni di parole inglesi) di più facile memorizzazione. A parte questo, i linguaggi assemblativi corrispondono in tutto e per tutto al linguaggio macchina, ossia esiste una corrispondenza 1:1 tra istruzioni del linguaggio assemblativo e istruzioni del linguaggio macchina, ovvero operazioni che il calcolatore può eseguire:

Linguaggi ad alto livello: presentano la maggiore potenza espressiva. Sono costituiti da costrutti non elementari, facilmente comprensibili da un operatore. Le istruzioni disponibili in un linguaggio ad alto livello sono più complesse di quelle effettivamente eseguibili da un calcolatore. Una istruzione ad alto livello corrisponde a molte istruzioni in linguaggio macchina o assemblativo. I linguaggi ad alto livello sono in larga misura indipendenti dalla specifica macchina. Tutti i linguaggi comunemente usati appartengono a tale classe: C, Pascal, Java, Fortran, C++, Lisp, etc.

Quindi un programma può passare attraverso diversi stadi, ciascuno corrispondente ad un diverso livello di astrazione.

La prima fase di sviluppo di un programma è data da una descrizione del problema mediante la scrittura di un algoritmo. La fase successiva consisterà nella trasformazione dell'algoritmo in un programma scritto in un linguaggio di programmazione ad alto livello per produrre il *codice sorgente*. Tale codice sarà scritto mediante l'uso di un editor. Il programma può essere costituito da uno o più file sorgente. Il codice sorgente sarà poi compilato per produrre il *codice oggetto*. Se il programma è costituito da più file o se fa uso di file di libreria, il *linker* si occuperà di collegare i file oggetto per produrre il *codice eseguibile* (o binario), ossia il programma vero e proprio. L'ultima fase è quella di caricamento e viene effettuata da un programma chiamato *loader*: il programma da eseguire viene trasferito in memoria centrale e lanciato.

Nel caso del C, in laboratorio verrà usato un programma chiamato TurboC che offre un ambiente integrato, mettendo a disposizione tutti i programmi citati nel paragrafo precedente, dall'editor al loader.

Per capire meglio la trasformazione di un programma, vediamo come questo evolve partendo dall'algoritmo per arrivare al codice binario. Per fare questo, ipotizzeremo una macchina con un suo linguaggio assemblativo e un suo linguaggio macchina. Il passaggio al linguaggio assemblativo viene inserito solo per meglio comprendere la trasformazione da codice sorgente a codice eseguibile.

Nel seguito risolveremo un problema semplice, il calcolo del prodotto di due interi, partendo dalla definizione di un algoritmo risolutivo fino ad arrivare al sua codifica in codice binario.

Problema: Dati due numeri interi X e Y, eseguire il loro prodotto usando solo le operazioni di somma e sottrazione.

4.1 Algoritmo e programma in C

La soluzione del problema è data dalla definizione stessa di moltiplicazione. Moltiplicare un numero X e un numero Y significa sommare X a se stesso per Y volte (partendo da 0)

$Sum = 0$
per Y volte ripeti
    $Sum = Sum + X$

La somma va eseguita per Y volte: il conteggio verrà tenuto da un contatore che incrementerà il suo valore ad ogni somma fino ad arrivare a Y. Il risultato finale e i passaggi parziali richiedono l'introduzione di una variabile intera Sum. L'algoritmo, scritto in linguaggio naturale, è il seguente:

leggi X ed Y
inizializzazione della variabile contatore: $I = 0$
inizializzazione della variabile che contiene il risultato: $Sum = 0$
fino a quando $I \neq Y$, esegui
    $Sum = Sum + X$
    $I = I + 1$
stampa Sum

Il passaggio successivo consiste nella codifica dell'algoritmo nel linguaggio di destinazione, ossia il C. Si ottiene il seguente programma.

```
#include <stdio.h>
int main (void)
{
  int x, y;
  int i = 0;
  int sum = 0;

  printf("Introduci due interi da moltiplicare\n");
  scanf("%d%d", &x, &y);
  while (i != y) {
    sum = sum + x;
    i = i + 1;
  }
  printf("La somma di %3d%3d e‘ pari a %6d\n", x, y, sum);
  return 0;
}
```

Sia le istruzioni che i dati sono poi codificati in forma binaria (codice macchina) e quindi caricati nella memoria centrale per permettere l'esecuzione del programma. Prima però studiamo il passaggio da codice C a codice assembler.

4.2 Il linguaggio assemblativo

Iniziamo con ipotizzare un linguaggio assemblativo per la macchina su cui dovremo eseguire il programma. L'ipotesi è che la macchina lavori su due registri accumulatori $R1$ e $R2$. Le istruzioni del linguaggio sono riportate nella Tabella 2.

Usando le istruzioni contenute nella Tabella 2, il programma in C viene trasformato nell'equivalente programma in linguaggio assemblativo, mostrato nella Tabella 3. Le istruzioni sono eseguite in sequenza e sono memorizzate in celle adiacenti nella memoria che è una struttura mono-dimensionale. Le variabili del programma (X, Y, Sum, I) sono memorizzate in locazioni di memoria apposite, mentre i registri $R1$ e $R2$ sono i registri accumulatori. Le variabili sono nomi simbolici che corrispondono ad indirizzi fisici nella memoria del calcolatore. All'inizio del programma le variabili e i registri hanno un valore indefinito; il loro valore sarà poi definito in fase di assegnazione.

Le istruzioni del programma in C devono essere tradotte in sequenze di istruzioni elementari nel linguaggio assembler. La corrispondenza tra le istruzioni in C e in assembler non è di 1:1. Ad esempio, l'istruzione `sum = sum + x` non corrisponde ad una semplice istruzione di somma, ma necessita di quattro diverse istruzioni:

Operazione	**Codice assembler**	**Significato**
Caricamento di un dato	`LAOD R1 X` `LAOD R2 X`	Carica nel registro $R1$ (o $R2$) il dato memorizzato nella cella di memoria identificata dal nome simbolico X
Somma e sottrazione	`SUM R1 R2` `SUB R1 R2`	Somma (sottrae) il contenuto di $R2$ al contenuto di $R1$ e memorizza il risultato in $R1$
Memorizzazione	`STORE R1 X` `STORE R2 X`	Memorizza il contenuto di $R1$ ($R2$) nella cella di nome simbolico X
Lettura	`READ X`	Legge un dato e lo memorizza nella cella di nome simb. X
Scrittura	`WRITE X`	Scrive il valore contenuto nella cella di nome simbolico X
Salto incondizionato	`JUMP A`	La prossima istruzione da eseguire è quella con etichetta A
Salto condizionato	`JUMPZ A`	Se il contenuto di $R1$ è uguale a 0, la prossima istruzione da eseguire è quella con etichetta A
Termine dell'esecuzione	`STOP`	Ferma l'esecuzione del programma

Tabella 2: Un esempio di linguaggio assemblativo

carica il valore di X in un registro;
carica il valore di Sum in un registro;
effettuata la somma tra i due registri;
memorizza il risultato nella locazione di memoria di Sum.

Discutiamo brevemente le istruzioni del programma assembler.

- Le prime due istruzioni di lettura assegnano i valori alle variabili X e Y.
- Le due successive coppie di istruzioni (2-3 e 4-5) corrispondono a due operazioni di assegnamento. Questa operazione corrisponde a due istruzioni in assembler (`LOAD` e `STORE`): nel primo passo il valore da assegnare viene caricato nell'accumulatore ($R1$) e successivamente trasferito da $R1$ alla locazione di memoria corrispondente alla variabile cui si assegna il valore. Sono necessarie due istruzioni poiché non è possibile assegnare un valore direttamente a registri che non siano accumulatori. Per esempio, l'istruzione di indirizzo 2 modifica il contenuto del registro $R1$ caricando il valore 0, mentre l'istruzione 3 trasferisce tale valore nel registro indirizzato da I.
- Ciascuna operazione di somma necessita di 4 passi elementari (10-13 e 14-17), provocando il cambiamento di due registri ($R1$ e $R2$) e delle variabili SUM e I rispettivamente.
- L'istruzione di ciclo presente nel programma, ossia l'esecuzione delle due somme fino a quando il confronto `(i != y)` rimane vero, viene tradotta in linguaggio assembler usando due istruzioni di salto: se il test è falso (ovvero $I = Y$), allora salta all'istruzione di stampa, etichettata con `FINE` (salto condizionato `JUMPZ`); altrimenti esegui le due somme e ritorna ad effettuare il test (salto incondizionato `JUMP`). L'esecuzione del test richiede tre istruzioni (6-8) e lascia nel registro $R1$ un valore che condiziona la successiva istruzione di salto `JUMPZ`.

Si noti come le operazioni `JUMPZ` e `JUMP` interrompano la sequenzialità dell'esecuzione. Ad esempio, dopo `JUMP`, invece dell'istruzione 19, viene eseguita l'istruzione etichettata con `INIZ`.

4.3 Il linguaggio macchina

Il programma nella Tabella 3 non può ancora essere eseguito da un calcolatore se non viene prima trasformato in linguaggio macchina, ossia in un linguaggio binario costituito da sequenze di 0 e 1. Per capire il linguaggio macchina dobbiamo vedere come il programma viene rappresentato in memoria:

- ogni istruzione occupa una o più celle di memoria; nel codice macchina mostrato come esempio ogni istruzione occupa una singola cella di memoria;

	Etic	Istr. assembler	Istruzione C	Significato
0		`READ X`	`scanf`	Contenuto cella identificata da X: 3
1		`READ Y`	`scanf`	Contenuto cella identificata da Y: 2
2		`LOAD R1 ZERO`	`i = 0`	Inizializzazione di I; metti 0 in $R1$
3		`STORE R1 I`		Metti il valore di $R1$ in I
4		`LOAD R1 ZERO`	`sum = 0`	Inizializzazione di SUM; metti 0 in $R1$
5		`STORE R1 SUM`		Metti il valore di $R1$ in SUM
6	`INIZ`	`LOAD R1 I`	`if (i == y)`	Esecuzione del test; metti in $R1$ il valore di I
7		`LOAD R2 Y`	`jump to FINE`	Metti in $R2$ il valore di Y
8		`SUB R1 R2`		Sottrai $R2$ (ossia Y) da $R1$
9		`JUMPZ FINE`		Se $R1 = 0$ (quindi $I = Y$) salta a $FINE$
10		`LOAD R1 SUM`	`sum = sum + x`	Somma parziale; metti in $R1$ il valore di SUM
11		`LOAD R2 X`		Metti in $R2$ il valore di X
12		`SUM R1 R2`		Metti in $R1$ la somma tra $R1$ ed $R2$
13		`STORE R1 SUM`		Metti il valore di $R1$ in SUM
14		`LOAD R1 I`	`i = i + 1`	Incremento contatore; metti in $R1$ il valore di I
15		`LOAD R2 UNO`		Metti 1 in $R2$
16		`SUM R1 R2`		Metti in $R1$ la somma tra $R1$ ed $R2$
17		`STORE R1 I`		Metti il valore di $R1$ in I
18		`JUMP INIZ`		Salta a $INIZ$
19	`FINE`	`WRITE SUM`	`printf`	Scrive il contenuto di SUM
20		`STOP`		Fine dell'esecuzione

Tabella 3: Programma assembler per il prodotto tra due numeri

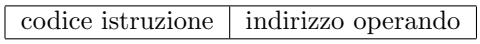


Figura 4: Rappresentazione di una istruzione

- ogni istruzione è codificata come una sequenza di bit (0 e 1);
- la rappresentazione di una istruzione consiste di due parti (Figura 4):
  - il *codice operativo* (che identifica l'operazione), e
  - gli *operandi*, che specificano come e dove reperire i dati su cui eseguire l'oeprazione; nel codice macchina mostrato come esempio ogni istruzione ha un singolo operando, che è l'indirizzo della cella di memoria contenente il dato sul quale eseguire l'operazione.

Nel codice mostrato come esempio, benché la maggior parte delle operazioni siano a due operandi, la rappresentazione dell'istruzione richiede solo il codice e al più un operando. Il secondo operando non è necessario in quanto è specificato implicitamente dall'istruzione (ad esempio un registro per un'istruzione di `LOAD` o `STORE`, oppure un accumulatore per un'operazione aritmetica). Si noti infatti che il codice operativo di un'istruzione indica non solo l'operazione ma anche *se* e *quale* registro è coinvolto.

Considerando che la Tabella 2 contiene 11 istruzioni, dobbiamo codificare 11 diverse sequenze binarie, ossia una per ogni istruzione in assembler. Per rappresentare almeno 11 sequenze distinte servono 4 bit ($2^4 = 16$ ma $2^3 = 8$) per la codifica dei codici operativi. Per rapprentare i codici operativi dobbiamo considerare le 21 linee di programma (ogni istruzione ha un indirizzo), le quattro variabili (X, Y, I, SUM) e le due costanti (0 e 1). Servono un totale di 27 indirizzi: sono necessari quindi 5 bit ($2^5 = 32$) per la codifica degli indirizzi degli operandi.

La trasformazione delle operazioni da un codice simbolico ad uno binario dipende dallo specifico calcolatore, che determina sia l'insieme delle operazioni che la sequenza di bit che le rappresenta. Non dipende dallo specifico programma, ossia un'operazione avrà un certo codice qualunque sia il programma.

Ipotizziamo che la codifica delle operazioni sia quella in Tabella 4: ricordiamo che c'è una corrispondeza 1:1 tra istruzioni in assembler e in linguaggio macchina. Si noti che le istruzioni seguite da *ind* avranno

Istruzione assembler	Codifica binaria
`LOAD R1` ind	0000
`LOAD R2` ind	0001
`STORE R1` ind	0010
`STORE R2` ind	0011
`SUM R1 R2`	0100
`SUB R1 R2`	0101
`JUMP` ind	0110
`JUMPZ` ind	0111
`READ` ind	1000
`WRITE` ind	1001
`STOP`	1011

Tabella 4: Corrispondenza tra istruzioni assembler e codici operativi

bisogno del codice dell'operando, mentre le altre sono univocamente definite perché sono implicitamente specificati i registri su cui operano.

La Tabella 5 mostra l'assegnazione degli indirizzi alle variabili, alle costanti e alle istruzioni. Sostituendo alle operazioni il corrispondente codice binario e agli operandi i corrispondenti indirizzi otteniamo il programma in linguaggio macchina (colonne 2 e 3 della Tabella 5).

5 Programmi di supporto

5.1 Compilatori e interpreti

Il processo di trasformazione di un programma da codice sorgente a codice macchina è completamente trasparente all'utente: esistono programmi generali che si occupano di processare il codice sorgente richiedendo poco intervento da parte dell'utente. Questi programmi si occupano di tradurre in linguaggio assoluto il programma scritto in un linguaggio ad alto livello. Permettono quindi al programmatore di scrivere programmi con una certa facilità senza dovere conoscere la struttura interna della macchina, dei suoi registri, della sua memoria.

I programmi che traducono codice sorgente in codice macchina possono essere di due tipi:

Compilatore è un programma che, eseguito su data macchina M, legge un programma P_L, scritto in un linguaggio ad alto livello L, e produce un corrispondente programma P_B, scritto in linguaggio binario, che viene eseguito dalla macchina M. Quindi i compilatori sono programmi che effettuano la traduzione dell'intero programma in linguaggio macchina. Il programma viene poi eseguito "tutto insieme" a basso livello.

Interprete è un programma che, eseguito su una macchina M, legge le istruzioni del programma P_L, scritto nel linguaggio ad alto livello L, e le esegue man mano che queste vengono lette. Quindi gli interpreti traducono il programma istruzione per istruzione: ogni nuova istruzione viene interpretata (tradotta) ed eseguita direttamente a livello basso.

Il programma TurboC è un compilatore per il linguaggio C che, oltre a trasformare il codice sorgente in codice macchina, offre un ambiente di sviluppo per editare il codice sorgente, per seguire lo svolgimento del programma (trace) e degli ausili per la correzione degli errori (debugger).

5.2 Il sistema operativo

Il *sistema operativo* è un insieme di programmi (generalmente complessi) che permettono di utilizzare e gestire al meglio tutte le risorse messe a disposizione da un calcolatore (processore, memoria centrale, memoria di massa, periferiche, ...). In particolare il sistema operativo permette di mandare in esecuzione i programmi utente. Il sistema operativo è in grado gestire (anche contemporaneamente) molti diversi tipi

	Indirizzo	Codice operativo	Indirizzo operando	Istr. assembler
0	00000	1000	10101	READ X
1	00001	1000	10110	READ Y
2	00010	0000	10111	LOAD R1 ZERO
3	00011	0010	11001	STORE R1 I
4	00100	0000	10111	LOAD R1 ZERO
5	00101	0010	11000	STORE R1 SUM
6	00110	0000	11001	LOAD R1 I
7	00111	0001	10110	LOAD R2 Y
8	01000	0101	-----	SUB R1 R2
9	01001	0111	10011	JUMPZ FINE
10	01010	0000	11000	LOAD R1 SUM
11	01011	0001	10101	LOAD R2 X
12	01100	0100	-----	SUM R1 R2
13	01101	0010	11000	STORE R1 SUM
14	01110	0000	11001	LOAD R1 I
15	01111	0001	11010	LOAD R2 UNO
16	10000	0100	-----	SUM R1 R2
17	10001	0010	11001	STORE R1 I
18	10010	0110	00110	JUMP INIZ
19	10011	1001	11000	WRITE SUM
20	10100	1011	-----	STOP
21	10101			X
22	10110			Y
23	10111			ZERO (0)
24	11000			SUM
25	11001			I
26	11010			UNO (1)

Tabella 5: Programma in codice binario

di *processi* (ovvero programmi in esecuzione) su una stessa macchina, decidendo le attività da compiere e le attività delle periferiche (stampanti, dischi esterni, scanner, etc.).

Per le macchine di piccole dimensioni, uno dei sistemi più usati è lo MS-DOS, sistema operativo che gestisce un solo processo alla volta. Le più recenti versioni, note come WindowsXX, offrono un interfaccia più sofisticata ma le funzionalità sono quelle dello MS-DOS.

Un altro sistema per piccole macchine è il MAC OS (ora arrivato alla versione 10) usato esclusivamente sulle macchine Apple. Tale sistema operativo è sempre stato all'avanguardia dal punto di vista dell'interfaccia utente.

Un sistema operativo molto usato per le workstation è Unix: è un sistema molto sofisticato in grado di gestire più processori contemporaneamente. Piattaforma tipica dei sistemi Unix sono le macchine della famiglia Sun. Unix è stato realizzato in C.

Un sistema operativo che ha molte affinità con Unix è Linux che si sta sempre più affermando come sistema aperto e multi-piattaforma. Il suo codice sorgente è disponibile e quindi può essere modificato da chiunque ne sia in grado. La sua caratteristica più importante è però la sua portabilità da una piattaforma all'altra: esistono infatti versioni per macchine di tipo diverso, ossia per macchine nate con sistemi operativi come Unix, MAC-OS, Windows.

Parte II
La rappresentazione dell'informazione

6 Introduzione

La rappresentazione a basso livello di tutto quello che viene manipolato da un calcolatore (programmi, dati, indirizzi, etc.) consiste di sequenze di 0 e 1 che hanno un legame stretto con la struttura fisica del calcolatore.

In questo capitolo vedremo come l'informazione è rappresentata a basso livello, con riferimento alle *rappresentazioni numeriche*.

La rappresentazione di altro tipo di informazione non è diversa, ma la trattazione di numeri è particolarmente delicata perché, per problemi di memoria e lunghezza delle parole all'interno della memoria, possono sorgere problemi nella rappresentazione di numeri grandi.

Il sistema di rappresentazione cui dedicheremo maggiore attenzione sarà ovviamente quello binario.

7 Sistemi di numerazione

Un *numero* è un oggetto matematico che può essere rappresentato mediante una stringa di caratteri, detti cifre, di un fissato alfabeto. Si deve operare una distinzione tra numero e sua *rappresentazione*. Se si scrive "567", la maggior parte di noi direbbe: "Questo è un numero". Non è esatto: "567" è la sua rappresentazione in cifre decimali. Quella che noi scriviamo è quindi la rappresentazione in un sistema di numerazione. Se avessimo adottato la rappresentazione in cifre romane avremmo scritto DLXVII. Si distinguerà in seguito tra numero e *numerale*, intendendo con quest'ultimo termine la rappresentazione del numero in un sistema di rappresentazione.

Un numerale è costituito da una sequenza di *cifre*, ovvero caratteri di un certo alfabeto (ad es. $\{0, 1, \ldots, 9\}$ per le cifre decimali), ad ognuna delle quali è associato un valore. Le rappresentazioni di un numero possono essere di due tipi: *additiva*, come quella romana, o *posizionale*, come quella araba. I sistemi di rappresentazione che tratteremo (decimale, binaria, ottale ed esadecimale) sono tutti posizionali.

7.1 Rappresentazione dei numeri naturali

In una rappresentazione posizionale una cifra contribuisce al numero con un valore diverso a seconda della posizione che occupa.

Per esempio, il numerale 101 nella notazione decimale viene interpretato come

$$1 \cdot 10^2 + 0 \cdot 10^1 + 1 \cdot 10^0 = (101)_{10}$$

mentre nella rappresentazione binaria è interpretato come

$$1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = (5)_2.$$

In entrambi i casi abbiamo una *base* (10 e 2 rispettivamente) che viene elevata ad un esponente che dipende dalla posizione. Tale quantità viene poi moltiplicata per la cifra che si trova nella posizione corrispondente a quell'esponente.

La cifra fa parte dell'insieme delle possibili cifre per quel sistema di numerazione. Il numero di cifre diverse è pari alla base: se la base è b l'insieme dei valori associati alle cifre sarà dato dall'insieme $\{0, 1, 2, \ldots, b-1\}$. Nella *rappresentazione binaria* la base è $b = 2$ e le cifre sono $\{0, 1\}$; nella *rappresentazione decimale* la base è $b = 10$ e le cifre sono $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Definizione 1 (rappresentazione dei numeri naturali) *Data una qualsiasi base b e l'insieme delle cifre $\{0, 1, 2, \ldots, b-1\}$ di rappresentazione, un numero naturale può essere rappresentato come una sequenza finita di cifre (appartenenti all'insieme)*

$$c_{n-1} c_{n-2} \cdots c_1 c_0$$

che vanno interpretate come i coefficienti dell'espressione polinomiale

$$c_{n-1}b^{n-1} + c_{n-2}b^{n-2} + \cdots + c_1 b^1 + c_0 b^0 \tag{1}$$

il cui valore è il numero rappresentato.

Per evidenziare la dipendenza dalla base b scelta per la rappresentazione, si usa la seguente notazione:

$(c_{n-1}c_{n-2}\ldots c_1 c_0)_b$

Vediamo qualche esempio di rappresentazione diversa da quella decimale. Oltre la base 2, le basi più usate sono le basi 8 e 16.

La rappresentazione in base $b = 8$, detta rappresentazione *ottale*, usa le cifre da 0 a 7. Il valore di un numerale rappresentato in ottale si calcola applicando la formula di conversione (1). Ad esempio:

$$(354)_8 = 3 \cdot 8^2 + 5 \cdot 8^1 + 4 \cdot 8^0 = 3 \cdot 64 + 5 \cdot 8 + 4 = 192 + 40 + 4 = (236)_{10}$$

La rappresentazione in base $b = 16$, detta rappresentazione *esadecimale*, necessita di 16 cifre, e quindi usa i numeri da 0 a 9 più le prime sei lettere dell'alfabeto A, B, C, D, E, F (che corrispondono ai valori 10, 11, 12, 13, 14, 15). Ad esempio, il valore del numerale $(B7F)_{16}$ è dato dalla seguente formula:

$$(B7F)_{16} = 11 \cdot 16^2 + 7 \cdot 16^1 + 15 \cdot 16^0 = 2816 + 112 + 15 = (2943)_{10}$$

7.2 Rappresentazione dei numeri reali in virgola fissa

Un modo per rappresentare *numeri reali* è di rappresentare, oltre alla parte intera, anche la parte *frazionaria* usando cifre in base b. Per separare la parte frazionaria dalla parte intera si inserisce una ",". La sequenza (che può essere illimitata verso destra)

$c_{n-1}c_{n-2}\cdots c_1 c_0, c_{-1}c_{-2}\cdots$

rappresenta il valore reale assoluto

$$c_{n-1} \cdot b^{n-1} + c_{n-2} \cdot b^{n-2} + \cdots + c_1 \cdot b^1 + c_0 \cdot b^0 + c_{-1} \cdot b^{-1} + c_{-2} \cdot b^{-2} + \cdots$$

Per esempio, la sequenza di cifre $(42, 3)_{10}$ in base dieci rappresenta il numero

$$4 \cdot 10^1 + 2 \cdot 10^0 + 3 \cdot 10^{-1}$$

Tale rappresentazione viene detta in *virgola fissa* in quanto la posizione della virgola che separa parte intera da parte frazionaria è determinata dal numero di cifre scelte per le due parti.

8 Conversione di base

Dato che un numero può avere rappresentazioni diverse in basi diverse, è opportuno introdurre delle regole per operare delle *trasformazioni di base*, ossia delle regole che permettano di passare da una base ad un'altra.

8.1 Una base è potenza dell'altra

Il caso più semplice di conversione si presenta quando la base del sistema di partenza è una potenza k-esima della base del sistema di arrivo. Ad esempio, la base di partenza è 16 e la base di arrivo è 2 ($16 = 2^4$). Limitiamo le trasformazioni ai numeri naturali assoluti.

Data una base di partenza b^k, il valore di un numero in questa base può essere scritto come:

$$c_{n-1} \cdot (b^k)^{n-1} + c_{n-2} \cdot (b^k)^{n-2} + \cdots + c_0 \cdot (b^k)^0 \tag{2}$$

La generica cifra c_i è compresa tra 0 e $b^k - 1$ e può quindi essere rappresentata nella base b di arrivo usando al più k cifre. Ogni c_i sarà rappresentato nella base b da una sequenza di cifre

$c_{i,k-1}c_{i,k-2}\cdots c_{i,1}c_{i,0}$

Il numero rappresentato da questa sequenza può essere espresso moltiplicando ciascuna cifra $c_{i,j}$ per la potenza b^j della base b, ottenendo

$$c_i \;=\; c_{i,k-1} \cdot b^{k-1} + c_{i,k-2} \cdot b^{k-2} + \cdots + c_{i,1} \cdot b^1 + c_{i,0} \cdot b^0$$

Effettuando le sostituzioni nella formula (2) si ottiene

$$\begin{array}{ll} (c_{n-1,k-1} \cdot b^{k-1} + c_{n-1,k-2} \cdot b^{k-2} + \cdots + c_{n-1,0} \cdot b^0) \cdot (b^k)^{n-1} & + \\ (c_{n-2,k-1} \cdot b^{k-1} + c_{n-2,k-2} \cdot b^{k-2} + \cdots + c_{n-2,0} \cdot b^0) \cdot (b^k)^{n-2} & + \\ \cdots & + \\ (c_{0,k-1} \cdot b^{k-1} + c_{0,k-2} \cdot b^{k-2} + \cdots + c_{0,0} \cdot b^0) \cdot (b^k)^0 & \end{array} \tag{3}$$

e quindi i coefficienti nella nuova base sono:

$$c_{n-1,k-1} c_{n-1,k-2} \cdots c_{0,0}$$

Per capire meglio come funziona la formula di conversione (3), operiamo una conversione del numero $(653)_8$ espresso in ottale nel suo corrispondente in base 2, quindi con $b = 2$ e $k = 3$.

$$(653)_8 = (6 \cdot 8^2 + 5 \cdot 8^1 + 3 \cdot 8^0) = (6 \cdot (2^3)^2 + 5 \cdot (2^3)^1 + 3 \cdot (2^3)^0)$$

Il secondo passo consiste nel rappresentare anche i coefficienti del polimonio, ossia le cifre 6, 5 e 3, in base 2.

$$\begin{array}{l} (1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) \cdot (2^3)^2 + (1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0) \cdot (2^3)^1 + (0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) \cdot (2^3)^0 = \\ 1 \cdot 2^8 + 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = \\ (110\,101\,011)_2 \end{array}$$

Applicando una regola duale alla (3) si può tradurre un numero da una base b in basi che siano potenze di b, per esempio b^k. Il procedimento consiste nel raggruppare le cifre in gruppi di k a partire da destra e poi scrivere la traduzione dei gruppi di cifre nella base b^k. Ad esempio, il numerale $(1\,010\,011\,001)_2$ si traduce nella sua rappresentazione ottale ($k = 3$) raggruppando le cifre a tre a tre a partire da destra, completando con zeri a sinistra per avere un ultimo gruppo completo di tre cifre, e sostituendo ad ogni gruppo il valore corrispondente nel sistema ottale:

$$(001\ 010\ 011\ 001)_2 = (1\,2\,3\,1)_8$$

Nel caso di conversione da sistema binario a sistema esadecimale, la cui base 16 corrisponde a 2^4, le cifre vengono raggruppate a quattro a quattro partendo da destra e trasformate nel corrispondente valore esadecimale. Per esempio, il numerale $(1011\,0111\,1111)_2$ corrisponde al numerale $(\mathrm{B7F})_{16}$ in esadecimale.

La Tabella 6 illustra le rappresentazioni dei primi sedici interi nelle basi 16, 10, 8 e 2.

La formula (3) non può essere applicata per le trasformazioni tra base 10 e base 2 non essendo l'una potenza dell'altra. È necessaria una formula più generale che permetta di passare da una base b_1 ad una base b_2 qualsivoglia. Esistono due diversi metodi di conversione: il primo opera nella base di arrivo b_2, l'altro nella base di partenza b_1.

8.2 Operazione di conversione nella base di arrivo b_2

Data la rappresentazione di un numerale nella base b_1 si trasformano sia la base che i coefficienti nella base b_2. Si eseguono le operazioni necessarie, ossia le moltiplicazioni e i calcoli delle potenze. Le varie potenze della nuova base b_2 costituiscono le cifre della nuova rappresentazione.

Supponiamo di avere un numero nella base b_1:

$$c_{n-1} \cdot b_1^{n-1} + c_{n-2} \cdot b_1^{n-2} + \cdots + c_1 \cdot b_1^1 + c_0 \cdot b_1^0 + c_{-1} \cdot b_1^{-1} + \cdots$$

Si sostituiscono sia ai coefficienti c_i che alla base b_1 la loro rappresentazione nella base b_2.

Esadecimale	Decimale	Ottale	Binario
0	0	0	0000
1	1	1	0001
2	2	2	0010
3	3	3	0011
4	4	4	0100
5	5	5	0101
6	6	6	0110
7	7	7	0111
8	8	10	1000
9	9	11	1001
A	10	12	1010
B	11	13	1011
C	12	14	1100
D	13	15	1101
E	14	16	1110
F	15	17	1111

Tabella 6: Conversione tra basi

Supponiamo di volere trasformare $(37)_{10}$, rappresentato in base 10, nel corrispondente numerale in base 7.

$$\begin{aligned}(37)_{10} &= (3\cdot 10^1 + 7\cdot 10^0) = (3\cdot 7^0)\cdot(1\cdot 7^1 + 3\cdot 7^0)^1 + (1\cdot 7^1)\cdot(1\cdot 7^1 + 3\cdot 7^0)^0 \\ &= 3\cdot 7^1 + 9\cdot 7^0 + 1\cdot 7^1 = 4\cdot 7^1 + (1\cdot 7^1 + 2\cdot 7^0)\cdot 7^0 = 5\cdot 7^1 + 2\cdot 7^0 \\ &= (52)_7\end{aligned}$$

Si noti che in un passaggio intermedio è stato necessario portare in base 7 il coefficiente 9 (che non fa parte dell'insieme di cifre usato per rappresentare la base 7).

8.3 Operazione di conversione nella base di partenza b_1

Dato un qualsiasi numero in base b_1, esisterà la sua rappresentazione nella base b_2 di cui non conosciamo i coefficienti. Data la seguente rappresentazione

$$c_{n-1}\cdot b_2^{n-1} + c_{n-2}\cdot b_2^{n-2} + \cdots + c_1\cdot b_2^1 + c_0\cdot b_2^0 + c_{-1}\cdot b_2^{-1} + c_{-2}\cdot b_2^{-2} + \cdots \tag{4}$$

le cifre c_i sono i valori incogniti che dobbiamo determinare. La rappresentazione (4) può essere riscritta al seguente modo, mettendo ripetutamente in evidenza b_2 e b_2^{-1}

$$\begin{aligned}&c_0 + b_2\cdot(c_1 + b_2\cdot(c_2 + b_2\cdots(c_{n-2} + b_2\cdot c_{n-1})\cdots)) + \\ &b_2^{-1}\cdot(c_{-1} + b_2^{-1}\cdot(c_{-2} + b_2^{-1}\cdot(\cdots)))\end{aligned} \tag{5}$$

Il calcolo dei coefficienti avviene adottando due diversi criteri per la parte intera e frazionaria. La parte intera della rappresentazione in b_2 viene ricavata dai resti della divisione della parte intera di b_1 per la base b_2. Il resto della prima divisione coincide con la cifra meno significativa della rappresentazione in b_2. Il quoziente viene di nuovo diviso per b_2 ottenendo la seconda cifra meno significativa: si prosegue fino ad arrivare ad un quoziente nullo. La spiegazione è semplice: se consideriamo la parte intera della rappresentazione (4) del numero nella base b_2, e la sua forma equivalente (5), vediamo che il resto della divisione per b_2 ci fornisce proprio il coefficiente c_0, ovvero la cifra meno significativa della rappresentazione. Al quoziente della divisione possiamo poi applicare lo stesso procedimento, per ottenere via via i coefficienti $c_1, c_2, \ldots$; il processo termina a quoziente nullo.

Il calcolo della parte decimale richiede il procedimento inverso, ossia la parte decimale della rappresentazione (4) va moltiplicata per la base b_2. Come risulta evidente dalla forma equivalente (5), la prima operazione ci fornisce il coefficiente c_{-1}. Continuando a moltipicare per b_2 la parte decimale rimasta

divisione	quoz.	resto	cifra
35:2	17	1	c_0
17:2	8	1	c_1
8:2	4	0	c_2
4:2	2	0	c_3
2:2	1	0	c_4
1:2	0	1	c_5

Tabella 7: Conversione da base 10 a base 2

otteniamo via via i coefficienti $c_{-2}, c_{-3}, \ldots$. Il procedimento si ferma quando si è ottenuto il numero di cifre desiderato (ricordiamo che la sequenza può essere infinita a destra).

La scelta di uno dei due metodi dipende dalla facilità di applicazione. Se una delle due basi è 10, si preferirà fare operazioni in quella base indipendentemente dal fatto che sia quella di partenza o di arrivo.

Supponiamo di dovere trasformare il numerale $(100011)_2$ rappresentato in base $b_1 = 2$ in base decimale $b_2 = 10$. In questo caso applicando il primo metodo otteniamo:

$$(100011)_2 = (1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 32 + 2 + 1 = (35)_2$$

Se invece la base di partenza è 10, è più facile applicare il secondo metodo. Supponiamo di dovere covertire il numerale $(35)_{10}$ nella base $b_2 = 2$ applicando il secondo metodo. Il numerale $(35)_{10}$ sarà ripetutamente diviso per 2 fino ad arrivare ad un quoziente nullo: i resti delle divisioni successive formeranno la rappresentazione binaria del numero a partire dal coefficiente meno significativo. La Tabella 7 mostra il procedimento. Questo è il metodo che viene comunemente applicato passando dalla base 10 alla rappresentazione binaria.

Consideriamo ora il caso di un numero frazionario, ossia un numero reale compreso tra 0 e 1. Nella rappresentazione in base 10, le cifre a destra della virgola sono i coefficienti di potenze negative di 10. Per esempio: $(0,58710)_2 = 5 \cdot 10^{-1} + 8 \cdot 10^{-2} + 7 \cdot 10^{-3} + 1 \cdot 10^{-4}$. Questo può essere esteso a qualsiasi base:

$$c_{-1} \cdot b_1^{-1} + c_{-2} \cdot b_1^{-2} + c_{-3} \cdot b_1^{-3} + \cdots$$

Un numero frazionario può essere esteso infinitamente a destra ponendo i coefficienti uguali allo zero, così come un numero intero può essere esteso a sinistra. Data questa caratteristica, la conversione dei numeri frazionari presenta dei problemi di approssimazione, come risulterà chiaro dall'esempio che segue. Supponiamo di dovere convertire il numerale $(0.14)_7$ dalla base 7 alla base 10. Se applichiamo il primo metodo, avremo:

$$(0.14)_7 = 1 \cdot 7^{-1} + 4 \cdot 7^{-2} = 11/49 = 0.22448959 \cdots$$

È evidente che potremmo approssimare anche con meno cifre, ma anche con un numero maggiore: c'è quindi un problema di approssimazione.

9 L'aritmetica intera nella rappresentazione binaria

Limitiamo la nostra attenzione ai *numeri interi* positivi e negativi nella rappresentazione binaria che è quella usata dai calcolatori. I numeri interi, come qualsiasi altra informazione, sono rappresentati in una o più celle di memoria, utilizzando il sistema di numerazione binario.

Se la cella ha una dimensione di n bit, allora è possibile rappresentare 2^n numeri. Questo numero di bit potrebbe non essere sufficiente per rappresentare il risultato di alcune operazioni. Se il risultato richiedesse un numero di cifre binarie maggiore della dimensione della cella, si avrebbe situazione di *trabocco (overflow)*: le cifre più significative non potranno essere rappresentate e saranno perdute, alterando così il risultato dell'operazione.

Per la rappresentazione di numeri interi positivi e negativi abbiamo bisogno di rappresentare anche il *segno*. Nell'aritmetica decimale il segno viene posto a sinistra del numero. Se adottassimo lo stesso

sistema per l'aritmetica binaria, ad esempio usando il bit più sigificativo per rappresentare il segno, si presenterebbero dei problemi di calcolo. Infatti la rappresentazione del segno richiederebbe due diversi sistemi di calcolo, uno per l'addizione e l'altro per la sottrazione, e conseguentemente per la moltiplicazione e la divisione che non sono altro che addizioni e sottrazioni ripetute. Si dovrebbe prima analizzare il segno, poi decidere se addizionare o sottrarre. Se per esempio dovessimo effettuare l'operazione $x + y$, la somma dovrebbere essere fatta in modulo e segno. Se x e y avessero lo stesso segno, si sommerebbero i moduli e il segno sarebbe lo stesso dei due operandi. Se invece avessero segno diverso, si dovrebbe determinare l'operando con il modulo più grande e sottrarre a questo l'operando piú piccolo. Il segno sarebbe quello del numero con il modulo più grande.

Un altro problema in questo tipo di soluzione è quello della doppia rappresentazione per lo zero: ci sarebbero uno zero positivo e uno negativo.

Per ovviare a questi inconvenienti, si usa un particolare sistema detto *rappresentazione in complemento* rispetto alla base del sistema di numerazione scelto. Nell'aritmetica binaria si usano due sistemi di complementazione, il *complemento a uno* e il *complemento a due*. Vedremo il significato e i vantaggi della rappresentazione in complemento a due.

9.1 La rappresentazione in complemento

Il maggiore vantaggio della rappresentazione in complemento è la possibilità di avere un solo sistema di calcolo per l'addizione e per la sottrazione. Quando si sommano due numeri non è necessario preoccuparci dei segni, perché l'algoritmo è lo stesso in tutti i casi. L'altro vantaggio (limitatamente alla rappresentazione in complemento a due) è che lo zero avrà una sola rappresentazione.

Definizione 2 (residuo modulo b^n) *Dati un numero X e due interi b ed n (dove b rappresenta la base e n il numero di cifre usate per la rappresentazione), il* residuo modulo b^n *di X, indicato come $|X|_{b^n}$, è dato dall'espressione:*

$$X - \lfloor X/b^n \rfloor \cdot b^n$$

dove il simbolo $\lfloor \cdot \rfloor$ indica la parte intera inferiore.

Consideriamo il caso di $b = 10$ e $n = 2$ per $X = 25$ e $X = -25$.

$$|25|_{10^2} = 25 - \lfloor 25/10^2 \rfloor \cdot 10^2 = 25 - 0 = 25$$
$$|-25|_{10^2} = -25 - \lfloor -25/10^2 \rfloor \cdot 10^2 = -25 + 1 \cdot 100 = 75$$

Nell'esempio il residuo modulo 10^2 di un numero positivo è uguale al numero stesso, mentre per il numero negativo è proprio il suo complemento rispetto a 100. Per ogni intero relativo X esiste solo un $|X|_{b^n}$.

Si noti che, mentre per ogni X esiste un solo $|X|_{b^n}$, ad un residuo corrispondono in generale due numeri, uno positivo e uno negativo. Infatti al residuo indicato con 75 corrisponderebbe sia il numerale 75 che il numerale -25. Questo avviene se consideriamo numeri il cui valore assoluto supera $b^n/2$. Se invece consideriamo solo numeri nell'intervallo $[-b^n/2, b^n/2)$, allora vale anche la proprietà inversa, ovvero ad un residuo corrisponde un unico numero in tale intervallo.

Questa restrizione semplifica la definizione di residuo modulo b^n nel seguente modo:

$$|X|_{b^n} = \begin{cases} X, & \text{se } 0 \leq X < b^n/2 \\ b^n - |X|, & \text{se } -b^n/2 \leq X < 0 \end{cases}$$

Dati una base b e un numero n di cifre usate per la rappresentazione, è possibile rappresentare numeri interi relativi X nell'intervallo $[-b^n/2, b^n/2)$: tale rappresentazione tramite il residuo in modulo b^n è detta *rappresentazione in complemento alla base*.

Se $X \geq 0$, la sua rappresentazione in complemento alla base b con n cifre è compresa nell'intervallo $[0, b^n/2)$ e coincide con la rappresentazione del suo valore assoluto nella base b. Se $X < 0$ la sua rappresentazione in complemento alla base b è compresa nell'intervallo $[b^n/2, b^n)$. Si noti che lo zero è uno solo ed è positivo.

Questa proprietà permette di distinguere i numeri positivi dai negativi in base alla cifra più significativa del numerale. Nel caso di $b = 2$, i negativi, che sono rappresentati nel segmento superiore dell'intervallo,

avranno come prima cifra (cifra più significativa) il valore 1; i numeri positivi avranno come cifra più significativa uno 0, dato che sono rappresentati nella metà inferiore dell'intervallo.

Vediamo un esempio per $b = 2$ e $n = 6$. Vediamo la rappresentazione in complemento alla base (quindi la rappresentazione *in complemento a due*) del numerale binario $X = -1001$. Essa si ottiene come:

$$b^n - |X| = 2^6 - 1001 = 1000000 - 1001 = 110111$$

C'è un metodo più rapido per calcolare il complemento a due di un numero negativo X:

1. si parte dalla rappresentazione in binario del valore assoluto $|X|$, usando un numero di bit pari ad n (aggiungendo zeri a sinistra se necessario);
2. si complementano tutte le cifre (ossia si trasforma ogni 0 in 1 e viceversa);
3. a tale numero si somma 1.

Tale calcolo deriva dalla seguente considerazione:

$$2^n - |X| = 2^n - |X| + 1 - 1 = (2^n - 1) - |X| + 1$$

notando che $2^n - 1$ è rappresentato da n bit tutti pari ad 1.
Esiste un metodo ancora più rapido:

1. si rappresenta il valore assoluto $|X|$ in binario, completando a sinistra con degli zeri fino a raggiungere n bit;
2. a partire da destra, si lasciano inalterate tutte le cifre fino al primo 1 compreso;
3. le rimanenti cifre sono invertite, ossia ogni 0 è trasformato in 1 e viceversa.

Si noti che il complemento a due di un numero binario positivo coincide con se stesso in quanto si rappresenta nel segmento inferiore dell'intervallo.

10 Operazioni su interi relativi

Come detto all'inizio il vantaggio della notazione in complemento è che quando si sommano due numeri non è necessario preoccuparci dei segni, perché l'algoritmo di calcolo è lo stesso in tutti i casi. Ogni operazione diventa una somma di numeri nella loro rappresentazione in complemento a due, e il risultato va a sua volta complementato.

Un problema può sorgere solo nel caso di trabocco, ossia quando le cifre a dispozione non sono sufficienti a rappresentare il numero.

Esistono 4 casi possibili di somma algebrica tra X e Y: entrambi positivi, entrambi negativi, il primo positivo e il secondo negativo, il primo negativo e il secondo negativo. Vediamo come tutte si possono ridurre a somme in complemento a due. Consideriamo una rappresentazione a 5 bit (compreso il segno). Con x e y indichiamo le rappresentazioni in complemento rispettivamente di X e Y.

1. $+X + Y$
   In questo caso la somma coincide con la somma in complemento:

   $$+X + Y = x + y$$

   L'unico problema può presentarsi in caso di trabocco, perché il numero insufficiente di bit elimina la cifra più significativa dando luogo a un risultato errato. Se per esempio dovessimo sommare $X = 01001 = (9)_{10}$ con $Y = 01001 = (9)_{10}$ il risultato sarebbe 10001, ossia un numero negativo.

2. $+X - Y$
   In questo caso la somma in complemento corrisponde alla somma di un numero positivo e della rappresentazione in complemento a due del numero negativo:

$$x + y = X + (2^n - Y) \tag{6}$$

   Si hanno due sottocasi a seconda del segno di $X - Y$:

   (a) $X - Y > 0$, ossia $X > Y$
   Per ottenere $X - Y$ da $x + y$ basta sottrarre 2^n (si veda (6)), ossia trascurare il bit più significativo. Consideriamo l'esempio di $(9 - 4)$:

01001	9	
11100	-4	
100101		
00101	5	(eliminando il bit più significativo)

   (b) $X - Y < 0$, ossia $X < Y$
   Il risultato che si ottiene calcolando $x + y$ coincide con la rappresentazione in complemento di $X - Y$, ossia a $2^n - (X - Y)$. Consideriamo il caso di $(9 - 13)$:

01001	9	
10011	-13	
11100	-4	(in complemento a due)
00100	4	(rappresentazione non in complemento, il segno era dato dal primo 1)

   Si noti che -13 in complemento a due è ottenuto rappresentando 13 con 5 bit (01101) e complementandolo a due. Il segno negativo del risultato si educe dalla cifra più significativa. Il risultato è proprio la rappresentazione in complemento a due di -4.

3. $-X + Y$
   È identico al secondo caso, scambiando gli addendi tra di loro.

4. $-X - Y$
   La somma tra le notazioni in complemento dà luogo alla seguente formula:

$$x + y = (2^n - X) + (2^n - Y) = 2^n + 2^n - (X + Y)$$

   Il risultato differisce di 2^n dal valore corretto espresso in complemento. Quindi, se non si è avuto trabocco, basta trascurare il bit più significativo per avere la risposta corretta (rappresentata in complemento). Vediamo i due casi.

   (a) $|X + Y| < b^n/2$
   Supponiamo che $X = -1$ e $Y = -2$.

11111	-1	
11110	-2	
111101	-3	(in complemento a due con 6 bit)
11101	-3	(risultato corretto)

   (b) $|X + Y| > b^n/2$
   Supponiamo che $X = -14$ e $Y = -15$.

10010	-14	
10001	-15	
100011	-29	(in complemento a due con 6 bit)
00011	3	(risultato errato a causa del trabocco)

   Come si vede il risultato risulta errato (pari a 3) a causa del trabocco: in questo caso la sesta cifra è significativa per la rappresentazione del numero, mentre non lo era nel caso precedente.

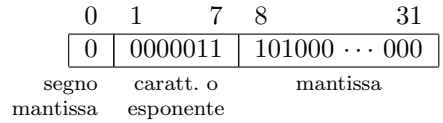


Figura 5: Rappresentazione di $X = 5$ in forma normalizzata in base $b = 2$ con 32 bit

Abbiamo quindi dimostrato che, tranne i casi di trabocco dovuti ai limiti fisici della memoria, è possibile eseguire le operazioni di somma e sottrazione usando il medesimo sistema di calcolo, ossia la somma nella notazione in complemento a due.

11 L'aritmetica reale

Date le loro caratteristiche, i numeri reali necessitano di una diversa rappresentazione rispetto ai numeri interi. Un modo di rappresentazione è dato dalla *forma normalizzata*.

Definizione 3 (Forma normalizzata in base b) *Data una base b, un numero reale X può essere rappresentato in una forma detta* forma normalizzata in base b *di X che è espressa nel modo seguente:*

$$X = m \cdot b^e$$

in cui:

- *e, detta* caratteristica in base b di X*, è un intero relativo;*
- *m, detta* mantissa in base b di X*, è un numero reale tale che $1/b \leq |m| < 1$. Se la mantissa è espressa dalla sequenza di cifre $c_1 c_2 c_3 \cdots$, allora rappresenta il valore*

  $$c_1 \cdot b^{-1} + c_2 \cdot b^{-2} + c_3 \cdot b^{-3} + \cdots$$

  Se questa espansione è finita o periodica*, X è un* numero razionale*, altrimenti è un* numero irrazionale*.*

L'insieme dei reali è un insieme infinito, e quindi non sarà possibile rappresentare tutti i suoi elementi. Inoltre molti degli elementi saranno rappresentati in modo approssimato o saranno troncati a causa della limitatezza dei bit a disposizione.

Vediamo come funziona la rappresentazione dei reali, secondo la forma normalizzata, nella memoria di un calcolatore. Supponiamo che la dimensione delle celle di memoria sia di 32 bit. In caso di lunghezze diverse il tipo di rappresentazione non cambia, sarà diversa la suddivisione dei bit. La rappresentazione è di tipo binario e quindi con base $b = 2$. I 32 bit della cella sono così suddivisi:

- 24 bit per rappresentare la mantissa m;
- 7 bit per rappresentare la caratteristica e nella notazione in complemento;
- 1 bit per il segno della mantissa m, ove 0 indica un numero positivo e 1 un numero negativo.

Applichiamo la forma normalizzata alla rappresentazione binaria del numero $X = 5$, ossia a $(101)_2$:

$$\begin{aligned} m &= |m| = (0.101 \ldots 0000)_2 \\ e &= (11)_2 \end{aligned}$$

Il numero reale 5 sarà rappresentato in una cella di memoria a 32 bit come in Figura 5.

L'operazione inversa consiste nell'interpretare una sequenza di 32 bit come numero reale. Per esempio, la rappresentazione di Figura 6 indica che:

- il segno della mantissa è positivo: il primo bit è pari a 0;

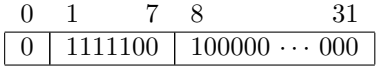


Figura 6: Rappresentazione di $X = 0.03125$ in forma normalizzata in base $b = 2$ con 32 bit

- il valore della mantissa è $m = |m| = (0.1000 \cdots 000)_2 = 1 \cdot 2^{-1} = 1/2$;
- l'esponente, rappresentato nella forma in complemento a due, è pari a -4.

Si ha allora che:

$$X = m \cdot 2^e = 1/2 \cdot 2^{-4} = 1/32 = 0.03125$$

In un elaboratore con celle da n bit di cui k sono per la mantissa, h per la caratteristica e uno per il segno, per un totale di $n = k + h + 1$, si possono rappresentare solo quei numeri per cui valgono le due seguenti condizioni:

$$1 \cdot 2^{-1} \leq |m| \leq 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} + \cdots + 1 \cdot 2^{-k}$$
$$|e| \leq 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + \cdots + 1 \cdot 2^{h-2}$$

Questo limita l'insieme dei numeri trattabili: non si possono trattare gli irrazionali e solo un sottoinsieme limitato dei razionali. Quindi l'insieme dei numeri rappresentabili in un calcolatore è un sottoinsieme finito dei numeri razionali. Questo insieme, indicato con F, è noto come l'insieme dei numeri a virgola mobile (*floating point*): il termine "virgola mobile" deriva dal fatto che la virgola può essere resa mobile modificando la caratteristica. Dato che l'insieme F è finito avrà un valore minimo e un valore massimo rappresentabili. Tale insieme è simmetrico rispetto allo zero, quindi possiamo caratterizzarlo preoccupandoci solo della parte positiva.

Il calcolo del limite massimo X_{max} si ottiene per il massimo valore di m e di e ottenibili, ossia:

$$m = |m| = 1 - 2^{24}$$
$$e = 2^6 - 1 = 63$$
$$X_{max} = (1 - 2^{24}) \cdot 2^{63} = 2^{63} - 2^{39} \approx 2^{63}$$

Il calcolo del limite minimo X_{min} si ottiene per il minimo valore di m e di e ottenibili, ossia:

$$m = |m| = 1 \cdot 2^{-1}$$
$$e = -2^6 = -63$$
$$X_{min} = 1 \cdot 2^{-1} \cdot 2^{-63} = 2^{-64}$$

Si noti come i valori rappresentabili non sono distribuiti uniformemente sull'asse dei reali. Due numeri nell'intorno di X_{min} sono estremamente vicini perché 2^{-64} è una quantità estremamente piccola e quindi possiamo rappresentare il suo intorno in maniera molto densa. Nell'intorno di X_{max} due valori adiacenti distano circa 2^{39}. Quindi i valori non sono distribuiti uniformemente sull'asse reale.

L'intervallo F gode di alcune proprietà di seguito elencate:

- è un sottoinsieme *finito* dei numeri razionali;
- è simmetrico rispetto allo zero, di cui però abbiamo due diverse rappresentazioni;
- i suoi elementi *non sono uniformemente distribuiti* sull'asse reale;
- molti numeri razionali non appartengono a F. Per esempio $1/3$, $1/5$ e $1/10$ non sono elementi di F, perché la proprietà di appartenenza a F è limitata solo ad alcuni numeri razionali con denominatore dato da una potenza di 2;
- non gode delle proprietà commutativa, associativa e distributiva. Si noti che componendo con addizioni e moltiplicazioni numeri di F si possono ottenere numeri che non appartengono ad F, un esempio per tutti $X_{min}/2$.

Data l'impossibilità di rappresentare ogni elemento dei reali, per rappresentare un numero reale X compreso in modulo tra 0 e 2^{63} si sceglie l'elemento x di F che sia più vicino a X. La funzione che associa ad ogni reale X un elemento x di F che lo rappresenta è detta *funzione di arrotondamento* ed è indicata con fl, per cui si ha:

$$X = fl(X)$$

L'aritmetica in virgola mobile è costruita in modo da garantire la *chiusura di* F, ossia che il risultato di un'operazione sia garantito essere in F, facendo uso della funzione di arrotondamento quando il risultato esatto non fa parte di F.

12 Limitazioni aritmetiche

I calcolatori operano su parole di memoria di dimensione fissa che fissa un limite ai numeri rappresentabili. Quando il calcolatore non dispone di un numero sufficiente di bit per rappresentare un certo numero, ci si può trovare di fronte a due diverse alternative: i calcoli proseguono usando valori approssimati; si abbandona l'elaborazione dopo avere segnalato l'errore.

Nel caso della perdita di precisione, il risultato approssimerà quello effettivo. Abbiamo appena visto come un numero reale sia rappresentato come mantissa ed esponente. Il numero di cifre della mantissa dà la precisione del numero, ad es. 3.56789E4 rispetto a 3.6E4. Supponiamo di operare con un calcolatore di piccole dimensioni che usi per esempio 4 cifre per la mantissa e 1 cifra per l'esponente. Il problema di approssimazione sorge in diversi casi, per esempio nel caso di una divisione (abbiamo detto numeri razionali con denominatore che non sono potenza di due non fanno parte dell'insieme F). In questo caso delle operazioni banali potrebbero dare dei risultati sorprendenti. Per esempio $(10/3) \cdot 3$ è diverso da 10. La causa è che 10/3 dà come risultato 3.333 che moltiplicato per 3 dà 9.999 che è ovviamente diverso da 10.

La precisione è andata persa. Nella rappresentazione in virgola mobile, l'inesattezza si vede sempre. Nella rappresentazione in virgola fissa, se si usa un numero di cifre minore di quelle necessarie il risultato potrebbe essere esatto per effetto dell'arrotondamento: 9.999 stampato con solo 2 cifre darebbe 10.

Un altro problema legato alla dimensione delle parole della memoria è legato alla rappresentazione di grandi numeri quale risultato, anche intermedio, di operazioni. C'è da notare un elemento importante che differenzia l'approssimazione dall'errore. Si parla di *approssimazione* quando la mantissa non è in grado di rappresentare tutte le cifre del numero. Siamo in presenza di *errore* quando tutta la caratteristica non è sufficiente a rappresentare il numero.

Ogni sistema ha un massimo numero rappresentabile che, a seconda del linguaggio di programmazione usato, viene memorizzato in una costante che indichiamo con *maxint*. Dato che non è possibile creare o memorizzare un numero più grande di questo, il sistema dovrebbe produrre un messaggio di errore nel caso si tentasse di farlo. L'errore risulta meno evidente nel caso in cui un'operazione destinata ad un risultato più piccolo di *maxint* possa incorrere nello stesso genere di problema: è il caso di risultato parziale più grande di *maxint*, come per esempio nell'operazione $maxint \cdot 2/3$, che porta ad un errore perché non è possibile rappresentare, nemmeno temporaneamente, un numero che sia il doppio di *maxint*.

Quando un numero è troppo grande per essere rappresentato siamo in presenza di *trabocco (overflow)*.

Può presentarsi anche il problema inverso, detto di *underflow*. Se consideriamo il caso di rappresentazione con 4 cifre di mantissa e una cifra di esponente, il più piccolo numero rappresentabile sarebbe 0.1000E-9, ossia 0.0000000001. Quindi qualsiasi valore più piccolo della metà di questo estremo verrà memorizzato come 0.

Il formato che in genere si usa per i numeri reali è quello proposto dallo IEEE (Institute of Electrical and Electronics Engineers). Sono stati proposti tre formati:

- *singola precisione* con parole di 32 bit,
- *doppia precisione* con 2 parole da 32 bit ciascuna (64 bit),
- *quadrupla precisione* con 4 parole (128 bit).

Vedremo poi come queste grandezze sono implementate in C.