Capitolo 7

L'architettura dei calcolatori

7.1 L'architettura di von Neumann

È a tutti noto che i calcolatori sono *programmabili*, e dobbiamo a Charles Babbage l'idea di poter risolvere problemi diversi anche senza modificare l'*hardware* del sistema, e ciò proprio grazie a una riprogrammazione dello stesso.
Un programma $\mathcal{P}$ è genericamente costituito da un insieme di *istruzioni* $I(1), I(2), \ldots, I(n)$, che devono essere eseguite in sequenza, partendo cioè dall'esecuzione di $I(1)$, cui seguirà quella di $I(2)$ e così via, finché o sono state eseguite tutte le istruzioni o si è pervenuti a un blocco della computazione. Questo è perlomeno l'approccio seguito nell'ambito della *programmazione imperativa*, la quale è un paradigma di programmazione secondo cui un programma viene appunto inteso come un insieme di istruzioni (o comandi), ciascuna delle quali può essere pensata come un "ordine" che viene impartito alla macchina virtuale del linguaggio di programmazione utilizzato. Da un punto di vista sintattico, i costrutti di un linguaggio imperativo sono spesso identificati da verbi all'imperativo, quali per esempio *print, read, copy,...*.
Nelle prime realizzazioni di computer (Z1, Colossus, ENIAC), il programma $\mathcal{P}$ era esterno alla macchina o addirittura cablato, ma le limitazioni di tale impostazione portarono ben presto *John von Neumann* a proporre un modello architetturale di *computer a programma memorizzato*. Sulla base di questa nuova impostazione vennero poi realizzati i primi calcolatori con architettura di von Neumann, quali il SSEM (*Manchester Small-Scale Experimental Machine*), l'EDSAC (*Electronic Delay Storage Automatic Calculator*) e l'EDVAC (*Electronic Discrete Variable Automatic Computer*). Un tale approccio si poté attuare soprattutto grazie allo sviluppo delle memorie a ferrite (cfr. fig.2.24c), introdotte nei primi anni '50.
L'essenza di un computer a programma memorizzato è allora costituita da:

1. una *memoria indirizzabile*, che possa contenere il programma e i dati
2. un'*unità logico-aritmetica*, che possa lavorare sui dati della memoria
3. un *program counter*, cioè un registro che indica l'indirizzo di memoria dell'istruzione che deve essere eseguita

Se dunque il programma $\mathcal{P}$, costituito dalle istruzioni $I(1), I(2), \ldots, I(n)$, deve essere eseguito, si dovrà partire dall'esecuzione di $I(1)$, cui seguirà quella di $I(2)$ e così via. Se $\mathcal{P}$ è contenuto nella memoria del calcolatore, bisogna sapere *dove* reperire $I(1)$, cioè quale sia *l'indirizzo* di $I(1)$ all'interno della memoria. Questa informazione

viene memorizzata su un registro chiamato *program counter*. Quando l'istruzione $I(1)$ è stata individuata, bisogna acquisirla (*fetch*) e metterla a disposizione dell'unità logico-aritmetica, decodificarla (*decode*) e successivamente eseguirla (*execute*). Questa sequenza di operazioni prende il nome di *ciclo fetch-decode-execute* del processore. Eseguita l'istruzione $I(1)$ il *program counter* incrementerà di 1 il suo valore, punterà all'indirizzo dell'istruzione successiva $I(2)$ e il processore attiverà un nuovo ciclo *fetch-decode-execute* per l'esecuzione di $I(2)$. Dal punto di vista logico un computer programmabile svolge, per ogni programma $\mathcal{P}$, il seguente semplice ciclo:

```
PC ← 0;
repeat
    istruzione ← memoria[PC];
    decode(istruzione);
    fetch(operandi);
    execute
    PC ← PC+1
until istruzione = STOP
```

Vediamo ora nel dettaglio il significato di queste operazioni:

PC $\leftarrow 0$ significa che il registro PC, il *program counter*, viene caricata col valore 0, che per ipotesi corrisponde all'indirizzo della cella di memoria che contiene la prima istruzione che deve essere eseguita.

Il ciclo **repeat** I **until** C (ciclo *repeat*), dove C è una *condizione* e I una generica istruzione elementare (o un sottoprogramma costituito da istruzioni elementari, come nel nostro caso) significa:

ripeti (*repeat*) l'istruzione I, finché (*until*) si realizza la condizione C

Nel nostro caso la condizione C che ferma l'esecuzione è che l'istruzione corrente sia uno STOP.

istruzione ← *memoria*[PC] significa che la prima istruzione di $\mathcal{P}$, memorizzata nella postazione di memoria indicata dal PC, viene caricata su un registro opportuno di memoria e messa a disposizione dell'unità logico-aritmetica del processore.

decode(istruzione) significa che il processore provvede a decodificare il tipo di istruzione che deve essere eseguita.

fetch(operandi) significa che si raccolgono gli operandi sui quali interviene l'istruzione.

execute significa che il processore esegue l'istruzione sui propri operandi.

PC ← PC$+1$ significa il valore del *program counter* viene incrementato di 1.

A questo punto si controlla la condizione C del ciclo *repeat*, cioè se l'istruzione corrente appena eseguita è un'istruzione di STOP. In tal caso si termina la computazione, altrimenti si rientra nel ciclo *repeat* per eseguire $I(2)$.

L'architettura di von Neumann per un computer a programma memorizzato prevede in prima approssimazione i seguenti elementi (si veda figura 7.1):

RAM - *Random Access Memory* - una memoria indirizzabile che possa contenere il programma e i dati;

PC - *Program Counter* - un registro di memoria che indica l'indirizzo della cella di memoria che contiene l'istruzione che deve essere eseguita;

RI - *Registro Istruzioni* - registro sul quale viene caricata l'istruzione che deve essere eseguita;

ACC - *Accumulator* - registro sul quale viene caricato il risultato dell'elaborazione della ALU;

ALU - *Arithmetic Logic Unit* - unità logico-aritmetica che lavora sui dati della memoria;

CPU - *Central Processing Unit* - è il processore e costituisce il cuore del sistema; essa contiene la ALU e i registri necessari a memorizzare le istruzioni che devono essere eseguite e i risultati intermedi delle operazioni;

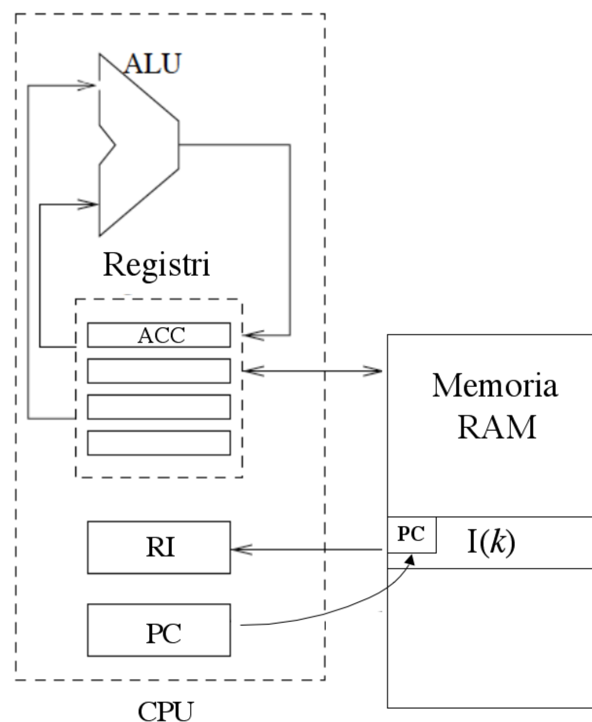


Figura 7.1: Architettura di von Neumann di un calcolatore a programma memorizzato

Nella struttura architetturale appena delineata la memoria RAM, detta anche *memoria principale*, contiene il programma e i dati intermedi; lavorando in stretta connessione col processore essa dovrà essere particolarmente veloce ed efficiente. D'altra parte quando il programma non serve esso deve risiedere da qualche parte, senza dover necessariamente occupare lo spazio della memoria RAM, che per le caratteristiche richieste sarà presumibilmente costosa, e di conseguenza non troppo capiente. È allora necessario aggiungere al sistema una memoria più esterna, chiamata *memoria secondaria* o *memoria di massa*, sulla quale potremo conservare tutti i programmi in uso all'utente dopo averli installati. Ecco allora che quando si attiva un'icona col doppio *clic* per aprire p.es. un file di testo, quello che accade è che il programma di elaborazione testi, inizialmente memorizzato sulla memoria di massa, viene caricato sulla memoria RAM e messo a disposizione del processore per la sua esecuzione, che in questo caso consisterà nella gestione e nella modifica del testo. L'interfaccia del computer con il mondo esterno passa attraverso le *periferiche*, che costituiscono le unità di ingresso e uscita dell'informazione; alcuni esempi tipici di periferiche sono: la tastiera, il *mouse*, un'unità di conversione A/D, per quanto riguarda i dati in ingresso; lo schermo e una memoria esterna per quanto riguarda dati in uscita. Un esempio della struttura completa di un calcolatore è fornito in figura 7.2.

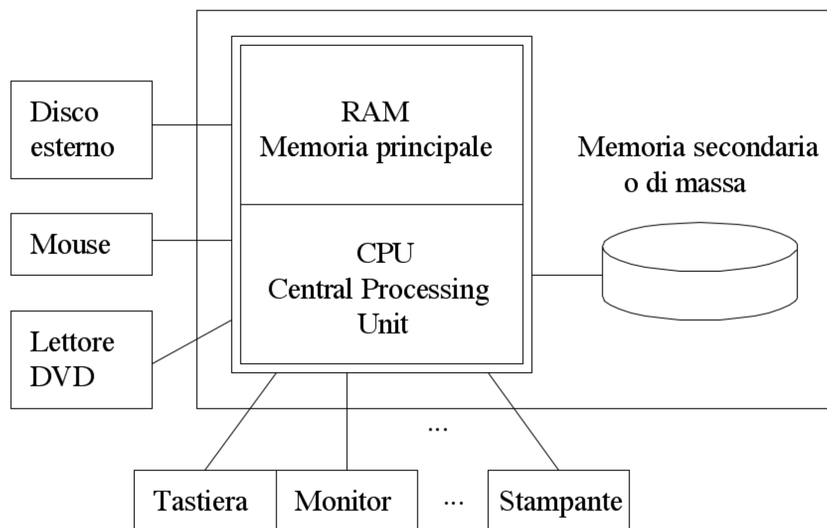


Figura 7.2: Struttura schematica di un calcolatore

Possiamo allora affermare che un computer è costituito sostanzialmente dai seguenti quattro componenti principali:

1. Processore (CPU)
2. Memoria primaria (RAM)
3. Memoria secondaria o di massa
4. Periferiche

Analizziamo ora queste componenti in dettaglio.

7.1.1 Il processore (CPU)

Il processore è costituito dall'insieme dei circuiti e dei registri di memoria necessari per decodificare ed eseguire le istruzioni del programma. Il suo componente più importante è l'*Unità di Controllo* UC (fig.7.3), che contiene i circuiti logici necessari per coordinare e attuare tutte le operazioni necessarie per realizzare il ciclo *fetch-decode-execute*; in particolare essa scandisce, col suo orologio interno, tutta l'attività del processore, attiva il PC, coordina la fase di lettura dalla memoria RAM dell'istruzione da eseguire, la decodifica e la manda in esecuzione avvalendosi anche della ALU. Quest'ultima è supportata anche da un *processore matematico* per velocizzare le operazioni in virgola mobile; nei primi modelli di calcolatore esso era venduto a parte su richiesta e installato successivamente su uno zoccolo della scheda madre; oggi fa parte integrante del *chip* del processore. All'interno del processore si trova anche la batteria dei *registri* che costituiscono il primo livello di memoria, quello immediatamente disponibile all'*Unità di Controllo*. Abbiamo già anticipato la funzione del *Program Counter* PC, che contiene in ogni istante l'indirizzo della cella di memoria RAM con la prossima istruzione da eseguire. Quest'ultima viene poi caricata sul *Registro delle Istruzioni* RI, per essere decodificata dall'*Unità Centrale* e successivamente eseguita. Il *Registro di Stato* RS contiene un insieme di variabili booleane (*flag*) che specificano lo stato di alcune operazioni matematiche a seguito di test effettuati su condizioni richieste dai programmi. Come esempio di *flag*

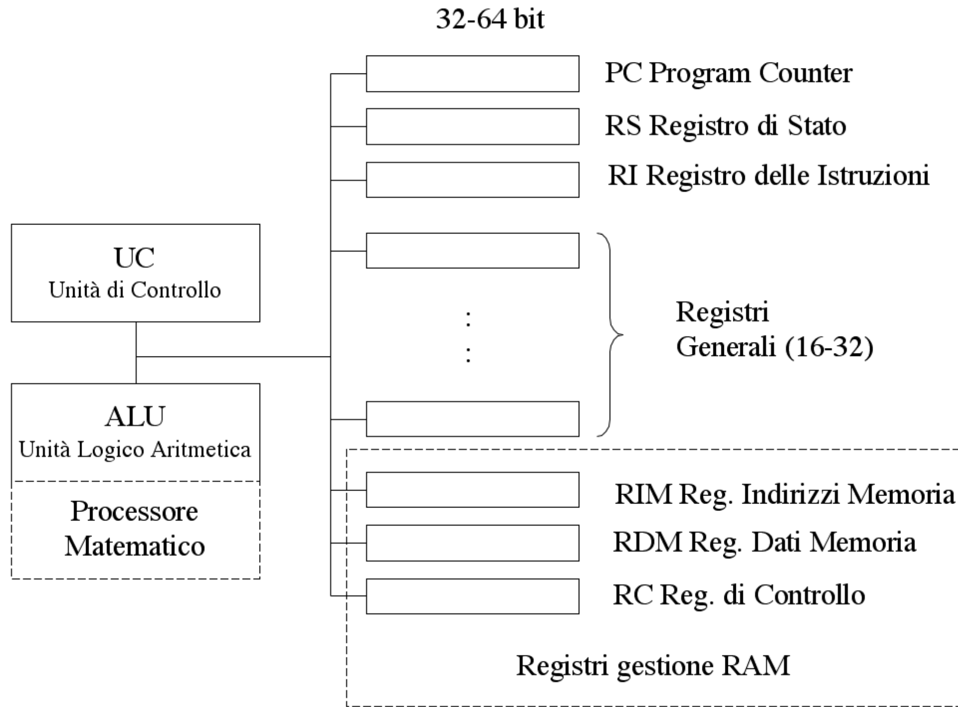


Figura 7.3: Struttura del processore

possiamo citare CF (*Carry flag*), che indica se il risultato di un'operazione produce un riporto non contenibile nei bit usati per il calcolo, oppure OF (*Overflow*), che indica se il risultato di un'operazione è in *overflow*, secondo la rappresentazione in complemento a due. I *Registri Generali* sono invece impiegati per memorizzare i risultati intermedi delle operazioni (si veda un esempio in figura 7.4).

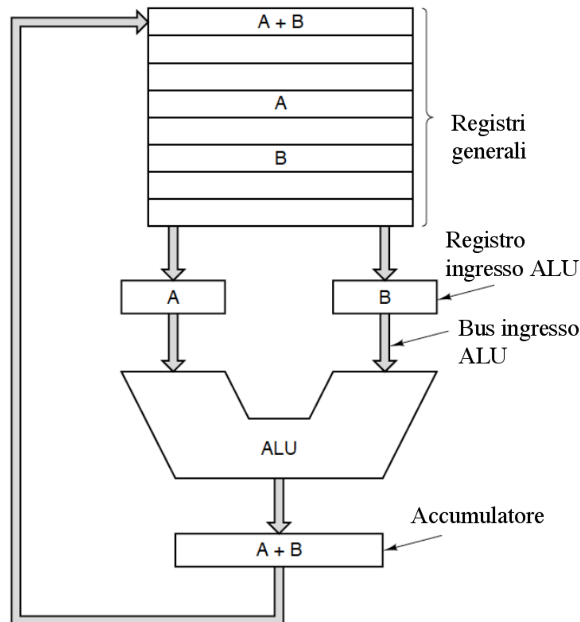
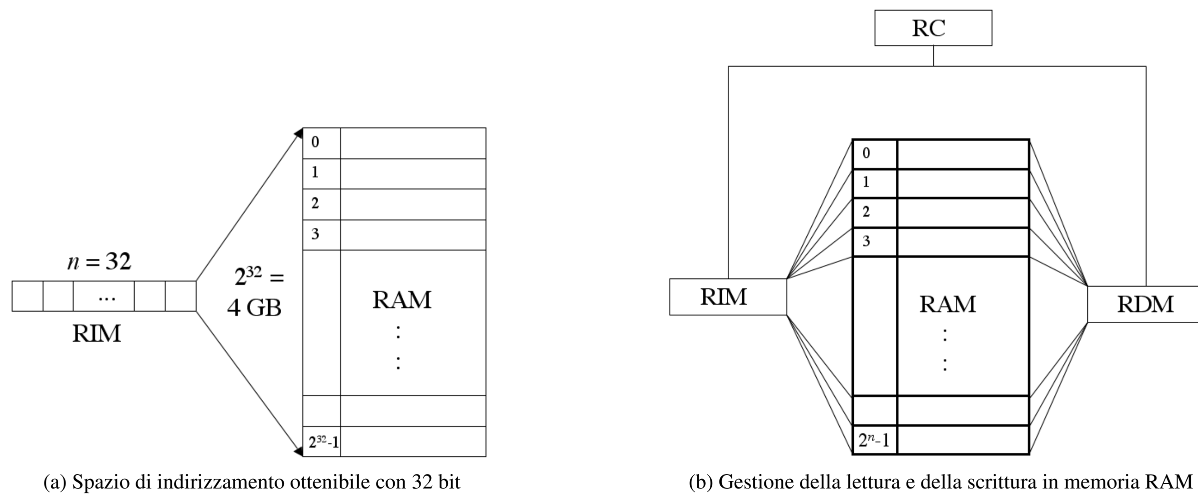


Figura 7.4: Esempio di operazione eseguita dalla ALU usando i Registri Generali

La schiera di registri viene completata dai registri necessari per gestire le operazioni lettura e la scrittura in memoria RAM. Essi sono il *Registro Indirizzi di Memoria* (RIM), sul quale viene caricato l'indirizzo della memoria interessato alla lettura o alla scrittura del dato; il *Registro Dati di Memoria* (RDM), che contiene il dato che deve essere scritto o letto e il *Registro di Controllo* (RC), che specifica se si tratta di lettura o scrittura.
Per poter accedere alla memoria RAM, il RIM deve indicare l'indirizzo della cella che si vuole usare in lettura o scrittura; se p.es. esso possiede 32 bit, allora saranno disponibili tutti gli indirizzi che vanno da 0 a $2^{32} - 1$; tale intervallo viene chiamato *spazio di indirizzamento* (si veda fig.7.5a).



(a) Spazio di indirizzamento ottenibile con 32 bit

(b) Gestione della lettura e della scrittura in memoria RAM

Figura 7.5: Spazio di indirizzamento del RIM e gestione delle operazioni in memoria RAM

Per sfruttare al massimo tutta l'estensione dell'intervallo, sarà opportuno avere una memoria con un numero di celle esattamente pari allo spazio di indirizzamento (4 GB nell'esempio di cui sopra). Supponiamo ora che il dato debba essere letto dalla memoria; in tal caso il RC dà indicazione *read* e una volta individuata la cella con il RIM, il contenuto della stessa viene riversato nel RDM, che sarà poi a disposizione della UC. Se al contrario il dato deve essere memorizzato nella RAM, esso sarà già disponibile nel RDM, il RC darà indicazione *write* e il dato verrà caricato sulla cella con l'indirizzo specificato dal RIM (fig.7.5b).
La successiva figura 7.6 mette in evidenza la linea di connessione parallela (*Bus*) che mette in comunicazione le componenti di un processore e la memoria RAM. L'insieme delle istruzioni che il calcolatore è in grado di eseguire tramite la CPU costituisce il *linguaggio macchina*; esse devono essere molto semplici e direttamente gestibili dai circuiti dell'unità di controllo e della ALU. Per le sue caratteristiche un tale linguaggio, molto legato alla struttura del processore, è inadatto alla programmazione da parte di operatori umani, ai quali si riserva invece l'impiego di linguaggi ad *alto livello* (come p.es. C, C++, *Java, Fortran, Pascal, Cobol,...*), le cui istruzioni elementari consentono di effettuare operazioni complesse e astratte, molto vicine alla logica che guida il ragionamento umano nell'approccio algoritmico-procedurale. In una prima fase storica molte architetture per computer cercarono di colmare lo iato semantico che esisteva tra i comandi in linguaggio macchina e quelli dei linguaggi ad alto livello. Questi calcolatori offrivano istruzioni che consentivano di eseguire operazioni relativamente complesse, quali la gestione delle procedure, la gestione dei *loop* e dei salti, la gestione di strutture dati in memoria e altri compiti comuni. Un tale approccio, noto con l'acronimo CISC (*Complex Instruction Set Computer*), permetteva la realizzazione di programmi compatti e che quindi richiedevano poca memoria, una risorsa molto costosa negli anni '60. In seguito prese però forza anche la filosofia di progettazione chiamata RISC (*Reduced Instruction Set Computer*), basata su un numero ridotto di istruzioni in linguaggio macchina. Per esempio le architetture RISC permettono di accedere alla memoria unicamente tramite delle istruzioni specifiche (*load* e *store*) che provvedono a leggere e scrivere i dati nei registri del microprocessore, mentre tutte le altre istruzioni manipolano i dati contenuti all'interno del processore. Nei processori CISC vale l'esatto opposto, poiché praticamente tutte le istruzioni possono

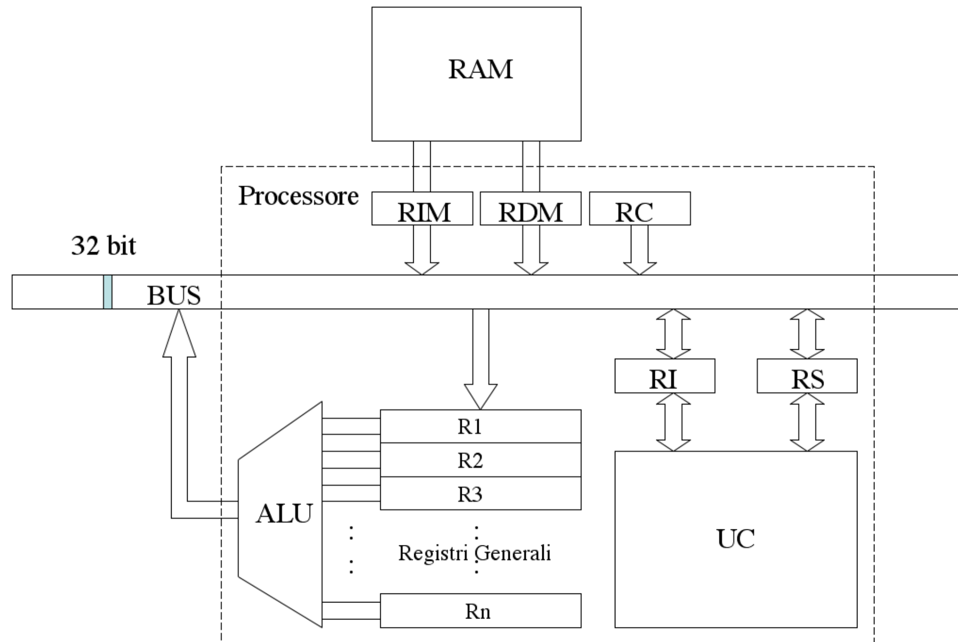


Figura 7.6: Il Bus dei dati mette in comunicazione le componenti di un processore e la memoria RAM

accedere ai registri o alla memoria in modo indifferente.
Tralasciando il problema troppo specifico della differenza tra CISC e RISC, che fra l'altro sta perdendo d'importanza con le nuove generazioni di processori paralleli, possiamo dire che le istruzioni in linguaggio macchina di un processore sono caratterizzate da un *codice operativo*, che specifica il tipo di istruzione, e da 2 o 3 operandi, che specificano i registri che contengono i dati sui quali si vuole eseguire l'istruzione. Esse appartengono alle seguenti categorie:

Istruzioni di I/O - servono per portare i dati all'interno o all'esterno della CPU; prevedono l'indicazione della periferica interessata e della locazione di memoria dove si trova il dato da portare in uscita o dove si vuole immettere il dato proveniente dall'esterno

READ INP 1322 Legge da tastiera e mette il dato nella cella di indirizzo 1322 della memoria RAM

WRITE OUT 1902 Scrive a video il contenuto della cella di indirizzo 1902 della memoria RAM

Istruzioni logico-aritmetiche - sono le istruzioni di manipolazione logico-aritmetica dei dati; devono essere indicati i dati su cui operare e la destinazione del risultato;

ADD R1 R2 Addiziona il contenuto del registro R1 con il contenuto del registro R2

MULT R1 1312 Moltiplica il contenuto della cella di indirizzo 1312 della memoria RAM con il contenuto del registro R2

COMP R1 R3 RC Compara il contenuto dei due registri R1 e R3 e scrive -1, 0 o 1 in RC a seconda che sia $R1 > R3$, $R1 = R3$, $R1 < R3$

Istruzioni di accesso alla memoria - servono a spostare dati da locazioni di memoria a registri della CPU o viceversa

LOAD 1672 R2 Carica sul registro R2 il contenuto della cella RAM d'indirizzo 1672

STORE R1 1559 Memorizza il contenuto del registro R1 nella cella RAM d'indirizzo 1559

Istruzioni di salto - servono a modificare la sequenza di esecuzione di un programma e devono contenere l'indirizzo di memoria dove si trova la prossima istruzione da eseguire

BRLT RC R1 1267 Se il contenuto del registro RC è minore del contenuto del registro R1 salta all'istruzione contenuta nella cella 1267, altrimenti incrementa PC di 1

JUMP 1721 Salta all'istruzione contenuta nella cella 1721

Siamo ora in grado di descrivere un ciclo completo *fetch-decode-execute* del processore. Supponiamo che il programma da eseguire sia contenuto nella memoria RAM a partire dalla cella di indirizzo 1000 e che le prime due istruzioni siano:

Load 1915 R1
Add R1 R2
⋮

Con riferimento alla figura 7.7 elenchiamo i passi del ciclo *fetch-decode-execute* necessari per eseguire la prima istruzione Load 1915 R1:

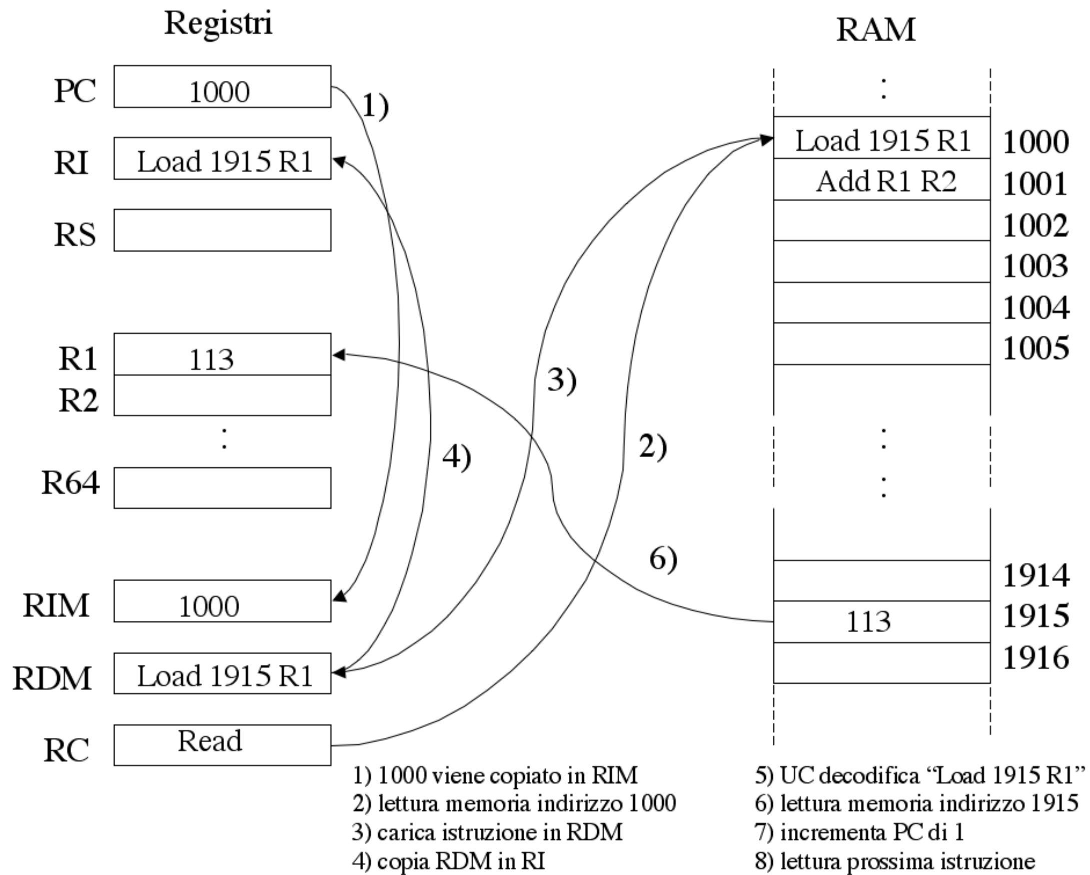


Figura 7.7: Ciclo *fetch-decode-execute* per l'esecuzione dell'istruzione Load 1915 R1

1. poiché l’istruzione si trova nella cella di indirizzo 1000 della RAM, è necessario che questo valore venga passato ai registri che si occupano della lettura/scrittura in memoria; il valore 1000 viene quindi copiato nel registro RIM
2. il registro di controllo RC dispone la lettura della memoria con il comando *Read*
3. il contenuto della memoria all’indirizzo 1000, costituito dall’istruzione da mandare in esecuzione, viene caricato nel registro RDM
4. il contenuto di RDM può ora essere posto a disposizione della UC, caricando l’istruzione che deve essere posta in esecuzione nel registro delle istruzioni RI
5. UC decodifica l’istruzione ”Load 1915 R1”, che richiede il caricamento nel registro generale R1 del contenuto della cella 1915
6. viene attivato un nuovo ciclo di lettura della memoria RAM, per caricare il contenuto della cella 1915, cioè il valore 113, nel registro R1
7. a questo punto l’istruzione è stata eseguita e si deve passare all’istruzione successiva; il contenuto del PC viene incrementato di 1
8. si riparte con un nuovo ciclo *fetch-decode-execute* per la lettura della prossima istruzione

Volendo riassumere i caratteri essenziali dell’esecuzione di una qualunque istruzione possiamo dire che il procedimento richiede cinque *microistruzioni* fondamentali:

IF (Instruction Fetch) : Lettura dell’istruzione da memoria

ID (Instruction Decode) : Decodifica istruzione e lettura operandi dai registri

EX (Execution) : Esecuzione dell’istruzione

MEM (Memory) : Attivazione della memoria (solo per certe istruzioni)

WB (Write Back) : Scrittura del risultato nel registro opportuno

Ogni CPU in commercio è gestita da un orologio centrale (*clock*) che scandisce in tempo discreto il funzionamento del sistema, e ogni microistruzione richiede almeno un ciclo di *clock* per poter essere eseguita. Le prime CPU erano formate da un’unità polifunzionale che svolgeva in rigida sequenza tutti e cinque i passaggi legati all’elaborazione delle istruzioni. Una CPU classica richiedeva quindi almeno cinque cicli di *clock* per eseguire una singola istruzione. Nel seguito, con il progresso della miniaturizzazione su larga scala assicurata dalla legge di Moore (sez.1.5) è stato possibile integrare un numero maggiore di transistor in un microprocessore, consentendo di parallelizzare alcune operazioni, tra cui le cinque descritte poc’anzi. Nasce a questo punto il concetto di *pipeline* dei dati. Una CPU dotata di *pipeline* è composta da cinque linee parallele di elaborazione, ciascuna capace di eseguire sequenzialmente le microistruzioni sopra descritte; le linee lavorano sulla base di una traslazione temporale unitaria, in modo che quando la prima linea sta elaborando la seconda microistruzione ID della prima istruzione, la seconda linea elabora la prima microistruzione della seconda istruzione, e così via (fig.7.8). Quando la *pipeline* è a regime, dall’ultimo stadio esce ciclicamente a ogni istante di *clock* un’istruzione completata. Si guadagna quindi una maggior velocità di esecuzione a prezzo di una maggior complessità circuitale del microprocessore, che deve contenere cinque linee di elaborazione che lavorano in parallelo tra loro.

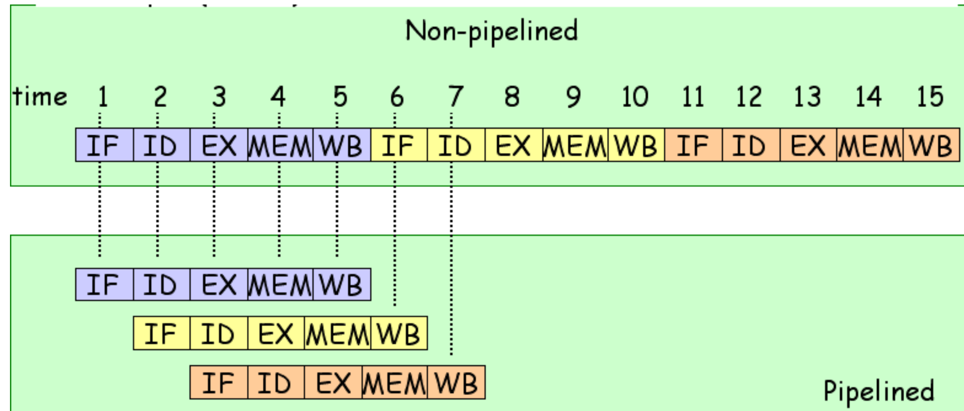


Figura 7.8: *Pipeline* dei dati per un microprocessore

7.1.2 La gerarchia delle macchine virtuali

L'esecuzione ripetuta del ciclo *fetch-decode-execute* consente l'esecuzione delle istruzioni che costituiscono i programmi che via via immettiamo nel calcolatore. Essa si realizza a un livello che è essenzialmente quello dei circuiti e delle memorie a registri del microprocessore. Si è più volte osservato che tali istruzioni sono estremamente semplici, quali caricare un dato in memoria, sommare il contenuto di due registri, leggere dalla memoria, fare un confronto tra due valori ecc. È evidente che progettare una procedura complessa per la soluzione di un problema della vita reale lavorando a questo livello di dettaglio per le istruzioni del programma sarebbe un'impresa (quasi) irrealizzabile. Per di più le istruzioni che abbiamo indicato come Load 1915 R1, COMP R1 R3 RC, JUMP 1721 ecc., descritte in un linguaggio chiamato *assembly*, non sarebbero comprensibili a livello circuitale, poichè come sappiamo il calcolatore opera col solo alfabeto binario; le vere istruzioni in *linguaggio macchina* si presentano in effetti come stringhe binarie; la figura 7.9 mostra un'ipotetica istruzione da 16 bit che compara il contenuto dei due registri R1 e R3 e scrive un valore su un terzo registro RC. Nella figura viene evidenziata anche la stringa in

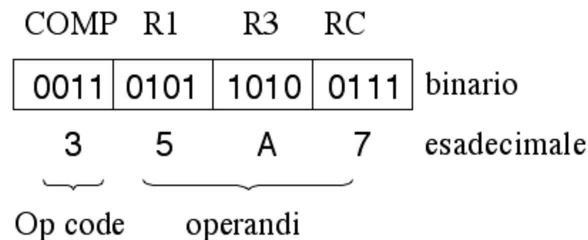


Figura 7.9: Istruzione in linguaggio *assembly* e corrispondente codifica binaria ed esadecimale

esadecimale, più compatta da trattare nel caso si voglia riprodurre in modo efficiente il listato del programma.
La programmazione basata sul linguaggio *assembly* segue le regole imposte dal corrispondente *Instruction Set Architecture* o *ISA*, che specifica l'insieme di tutte le istruzioni che possono essere attuate a questo livello; la traduzione delle istruzioni dal linguaggio *assembly* al linguaggio macchina viene attuata da un programma chiamato *assembler*. Tuttavia programmare in *assembly* non è efficiente, e i programmi che si realizzano sono vincolati alla tipologia architetturale della macchina in uso, e come tali non sono esportabili su altre macchine. È quindi necessario fare in modo che il programmatore possa operare a un livello logico superiore, usando linguaggi come C, C++, *Java, Fortran, Pascal*, ecc; per questi linguaggi le istruzioni elementari consentono di effettuare operazioni complesse e astratte, molto vicine alla logica che guida il ragionamento umano nell'approccio algoritmico-procedurale; inoltre i programmi in tali linguaggi sono indipendenti dall'architettura della macchina. La traduzione tra un linguaggio ad alto livello di questo tipo e il linguaggio *assembly* viene garantita da un programma che si chiama

compilatore, che andrà adattato alle diverse piattaforme architetturali in uso.
Da quanto detto finora si intuisce che il *computer* è una struttura molto complessa, in grado di lavorare solo a un livello circuitale e binario. Per poter creare una connessione operativa con l'utente, che è un programmatore esterno, è allora necessario costruire una *gerarchia* di ambienti organizzata in *livelli di astrazione* o *macchine virtuali* i quali, partendo dai transistor in condizioni di saturazione o interdizione del livello circuitale più basso, possano giungere sino all'interfaccia grafica, basata sulle icone e sulla metafora della scrivania, che si presenta all'utente in procinto di programmare usando un linguaggio tra quelli prima menzionati. Ogni livello i, associato alla macchina virtuale M_i, è caratterizzato da un proprio linguaggio L_i che utilizza i comandi e i servizi messi a disposizione dal livello inferiore. A ogni livello superiore aumenta l'astrazione e la facilità d'uso, ma diminuisce la velocità di esecuzione. Per scrivere i programmi a un certo livello non è necessario conoscere come viene effettuata la traduzione e quindi l'esecuzione al livello inferiore. Nel passaggio tra un livello e l'altro si possono usare, in generale, due diversi approcci operativi distinti, quello basato sui *compilatori* e quello basato sugli *interpreti*.
Un *compilatore* traduce il programma $P(L_i)$, scritto in linguaggio L_i, in un programma $P(L_{i-1})$, scritto in linguaggio L_{i-1}. Il nuovo programma $P(L_{i-1})$ viene quindi eseguito in blocco.
Un *interprete* esamina il programma $P(L_i)$, scritto in linguaggio L_i e, istruzione per istruzione, lo traduce nel linguaggio L_{i-1} eseguendolo.
Se dunque chiamiamo M_0 la macchina virtuale cha abbiamo a disposizione al livello più basso e L_0 il corrispondente linguaggio, possiamo pensare di costruire una macchina virtuale M_1 con il relativo linguaggio L_1, che si pone a un livello astratto superiore e più vicino all'utente. Se seguiamo l'approccio del compilatore, possiamo eseguire un programma $P(L_1)$ nella macchina M_0 traducendo ogni istruzione di L_1 in una sequenza di istruzioni di L_0 ad essa equivalente. L'insieme delle istruzioni costituisce il programma $P(L_0)$ che può essere eseguito direttamente dalla macchina.
Un altro modo per eseguire un programma scritto nel linguaggio L_1 tramite le funzionalità di M_0 consiste nello scrivere un programma in L_0, detto appunto *interprete*, che sia in grado di eseguire *tutti i programmi in* L_1. Si osservi che la possibilità di costruire un interprete è un importante acquisizione teorica legata all'esistenza della *Macchina Universale*. Essa completa il percorso di astrazione delle funzioni delle macchine, che portò in un primo momento al passaggio dalle *macchine cablate* per la soluzione di un certo problema alle *macchine programmabili*, consentendo di risolvere problemi di natura diversa cambiando solo il programma della macchina e non la sua struttura (si veda la sez. 1.2). Successivamente il percorso si completò con la dimostrazione della possibilità di costruire un *programma universale*, che consente di risolvere qualunque problema (risolubile) usando la stessa struttura per la macchina e lo stesso *software*.
Tornando alla gerarchia dei livelli, osserviamo che per rendere traduzione e interpretazione utilizzabili in pratica, è opportuno che i linguaggi L_0 e L_1 non siano "troppo" diversi tra loro. La stessa idea si può applicare a L_2 e M_2, L_3 e M_3 ecc., dando luogo alla gerarchia che ora descriveremo brevemente a partire, per completezza, da un livello (-1) inferiore a quello d'interesse per l'architettura dei *computer*, cioè dal livello dei transistor.

Livello -1 - Elettronico Siamo al livello dei transistor che formano i circuiti elettronici che costituiscono le *porte logiche* di cui è composto un calcolatore (si vedano le figure 1.22). A questo livello il singolo transistor è in conduzione o in interdizione (si vedano le figure 2.6a e 2.6b), e si raggiunge un livello di dettaglio che viene in genere trascurato nella progettazione dei calcolatori.

Livello 0 - Logico Questo è il primo livello utile nella progettazione di un computer. La macchina è formata da porte logiche o *gate* studiate nei capitoli precedenti. Ogni porta riceve in ingresso dei segnali binari e calcola una semplice funzione Booleana (AND, OR, ...). Collegando opportunamente le porte di base si ottengono relazioni logiche complesse per i circuiti costituenti, p.es. si può realizzare una memoria di un bit (bistabile). Combinando n memorie di un bit si può formare un registro capace di memorizzare un numero binario compreso tra 0 e $2^n - 1$. Mediante le porte si realizzano i circuiti logici il cui funzionamento è regolato dalle leggi dell'algebra Booleana e dell'elettronica digitale. La macchina logica del livello 0 viene progettata dal costruttore dei vari componenti ed è puramente hardware.

Livello 1 - Microarchitettura A questo livello troviamo tutti gli elementi nell'architettura di base del computer, e cioè i registri generali usati come memoria locale, la ALU che esegue semplici operazioni logico-aritmetiche,

gli elementi di connessione tra registri e ALU, i registri dedicati al controllo (PC, RI, ...) e la circuiteria dell'Unità di Controllo. Il percorso dei dati può essere gestito da un programma controllabile dall'esterno, chiamato *microprogramma* (come nel caso dell'architettura CISC), oppure da una specifica circuiteria (come nel caso dell'architettura RISC).

Livello 2 - ISA È costituito dall'*Instruction Set Architecture* ed è quindi governato dal linguaggio macchina. Offre visione ed accesso diretto a tutte le risorse fisiche del sistema, tramite una specifica interfaccia di livello. Fornisce un'interfaccia tipicamente *software* (e quindi programmabile) ai livelli superiori. Si può agire al livello 1 tramite interpretazione (microprogramma) o disporre di un'esecuzione diretta a livello 0.

Livello 3 - Sistema Operativo È un'estensione del livello ISA ottenuta aggiungendo servizi che vengono eseguiti (interpretati) da un programma del livello ISA chiamato appunto *Sistema Operativo*. Esso garantisce l'operatività di base del calcolatore, coordinando e gestendo le risorse *hardware* di processamento e di memorizzazione, le periferiche, le risorse e le attività *software* legate ai vari processi in uso, funge da interfaccia con l'utente, consentendo l'impiego di altri *software* d'utente, come le varie applicazioni o le librerie.

Livello 4 - Assembler A questo livello si fornisce una rappresentazione simbolica di uno dei livelli sottostanti, impiegando sequenze alfanumerice pensate per essere mnemoniche e comprensibili. Infatti i linguaggi binari dei livelli più bassi sono difficili (o praticamente impossibili) da usare per un programmatore. Il programma che traduce da programmi in linguaggio assemblativo a programmi a livello ISA è detto *assembler* o *assemblatore*. A ogni istruzione del linguaggio assemblatore corrisponde un'istruzione del linguaggio macchina che viene eseguita direttamente. I programmi a questo livello non sono usati dal programmatore medio, che deve realizzare programmi applicativi, ma solo dai programmatori di sistema.

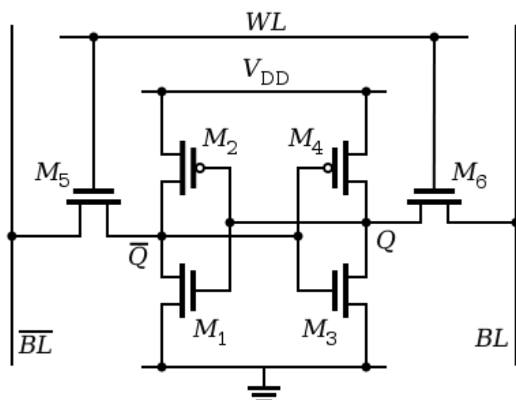
Livello 5 - Linguaggi applicativi È caratterizzato dall'impiego di inguaggi come C, C++, *Java, BASIC, LISP, Prolog*, ..., chiamati per l'appunto *linguaggi di alto livello*. Sono impiegati per la realizzazione di programmi applicativi. Il più delle volte la traduzione è affidata a un compilatore, mentre in alcuni casi si usa un interprete.

7.1.3 La gerarchia delle memorie

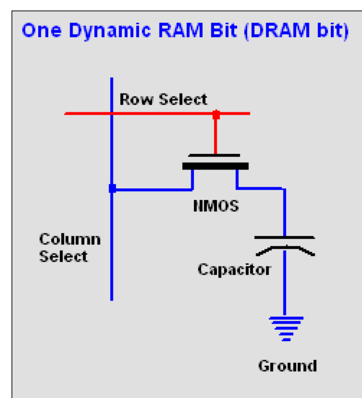
L'architettura di von Neumann si caratterizza per il caricamento del programma in esecuzione nella memoria principale del *computer*, la memoria RAM. L'esecuzione delle istruzioni che si susseguono viene attuata secondo il ciclo *fetch-decode-execute* illustrato nella figura 7.7. Dalla stessa figura appare che ci sono alcuni dati, quelli memorizzati nei registri, che vengono elaborati direttamente a livello di UC, e poi ce ne sono altri che derivano dalla lettura della memoria RAM. Inoltre il programma caricato sulla RAM deve essere reperito dalla *memoria di massa* visibile in figura 7.2, che costituisce l'ambiente all'interno del quale vengono *installati* tutti i programmi in uso all'utente. Come vedremo tra poco l'esistenza di alcuni vincoli di carattere tecnologico e funzionale impongono a queste tre tipologie di memoria (registri, RAM e memoria di massa) delle caratteristiche che sono in qualche misura complementari. In prossimità dell'UC, lavorando a livelli di registri, è necessario disporre di memorie ad accesso veloce, che saranno per forza di cose anche costose. Per contro non è richiesta una elevata capacità di memoria. Man mano che ci si allontana dal cuore del sistema, cioè dalla CPU, è sempre meno importante avere velocità elevate; bisogna però disporre di memorie molto capienti e, di conseguenza, anche molto economiche. Tutto ciò porta a una struttura gerarchica delle memorie, che ora ci prestiamo ad analizzare.

I registri - Abbiamo sottolineato il fatto che i registri sono il primo livello di memoria direttamente impiegato dall'*Unità Centrale* della CPU, al punto che si trovano circuitalmente inseriti all'interno del *case* del microprocessore; nei computer più recenti hanno una dimensione di 32 o 64 *bit*, ed è a questo valore che ci si riferisce per specificare la corrispondente architettura. Poiché costituiscono la memoria immediatamente disponibile per le istruzioni a livello di linguaggio macchina, essi devono operare alle velocità imposte dal ciclo macchina *fetch-decode-execute* e quindi devono essere memorie estremamente veloci. I registri sono realizzati con una tecnologia

elettronica denominata *RAM statica* (SRAM), basata sull’impiego dei MOS-FET; questi sono collegati come appare nel circuito di figura 7.10a, che è in grado di memorizzare un singolo bit. Il principio di funzionamento è quello già visto per il circuito *Flip-Flop* incontrato in figura 2.26. Ricordiamo che RAM significa *Random Access*



(a) Memoria RAM statica (SRAM) in grado di memorizzare un bit



(b) Memoria RAM dinamica (DRAM)

Figura 7.10: Circuiti per la realizzazione delle memorie RAM statiche e dinamiche

Memory, ed evidenzia un aspetto tecnologicamente rilevante della memoria, vale a dire la possibilità di accedere a una cella qualunque della stessa (da cui la specificazione *Random*) in un tempo costante. Questa caratteristica non vale, per esempio, per la tecnologia delle memorie a disco, che costituiscono le memorie di massa del tipo *Hard Disk* (HD) di molti *computer*.
Le unità di memoria SRAM sono di tipo *volatile* (si perde la memoria quando il circuito non è più alimentato) e necessita molti componenti (solitamente 6 MOS-FET per bit); è di conseguenza una memoria estremamente costosa e caratterizzata da una bassa densità di memorizzazione (in genere basterebbe un singolo transistor per bit, contro i 6 usati in questa caso). In cambio la tecnologia dei MOS-FET garantisce un bassissimo consumo di potenza e un tempo di accesso che è il migliore realizzabile con le odierne tecnologie; il valore si attesta di $1-2$ ns, ma può arrivare anche a $0,5$ ns; come conseguenza si ottiene un’altissima velocità di trasferimento dati (tipicamente > 12000 MB/s).

La RAM - La struttura schematica del calcolatore riportata in figura 7.2 evidenzia la stretta connessione tra CPU e *memoria principale* di tipo RAM. Su questa memoria, attualmente di tipo volatile, viene infatti caricato il programma in esecuzione secondo il paradigma dell’architettura di von Neumann. Anche la memoria principale deve possedere delle buone caratteristiche di velocità di accesso, ma il costo delle memorie SRAM rende proibitivo il loro impiego come memoria principale. Si ricorre pertanto a una tipologia più economica di memoria RAM, denominata DRAM (*Dynamic RAM*), che è almeno $5-10$ volte più lenta ($5-10$ ns come tempo di accesso), ma ben 100 volte più economica della SRAM. Poiché sulla memoria principale devono trovare posto, oltre ai dati intermedi della computazione, anche tutti i programmi che sono contemporaneamente in esecuzione, essa deve possedere una capacità rilevante; attualmente, per un *computer* di tipo portatile, la memoria principale è dell’ordine di qualche unità di GB ($2-16$ GB).

La *cache* - Una quantità di memoria così elevata non è integrabile all’interno del *chip* del processore, ed è quindi necessario tenere la memoria principale fisicamente all’esterno dello stesso; ciò comporta un ulteriore rallentamento nelle operazioni di lettura e scrittura in memoria principale, che diventa di fatto accessibile in tempi che si attestano nell’intervallo $30-90$ ns. A questo punto abbiamo una comunicazione molto veloce ed efficiente tra UC e registri, che consente una rapida esecuzione delle istruzioni, ma una comunicazione $30-90$ volte più lenta tra UC e memoria RAM esterna, il che ritarda la lettura delle istruzioni che devono essere eseguite. Questo fatto suggerisce l’introduzione di un ulteriore *livello di memoria* all’interno del *chip* del processore, denominato

cache, che serve per memorizzare in anticipo le informazioni che l'UC è in procinto di chiedere alla RAM per la successiva esecuzione. Per chiarire il concetto facciamo un esempio.
Supponiamo che l'UC sia in grado di eseguire un'istruzione in 10 ns e di leggere dalla memoria RAM in 60 ns. Immaginiamo che il programma in esecuzione abbia 100 istruzioni, $I(1), I(2), I(3), \ldots, I(100)$. Per leggere la prima istruzione, $I(1)$, l'UC ci mette 60 ns; poi la manda in esecuzione, impiegando 10 ns, e contemporaneamente inizia la lettura di $I(2)$, che dura sempre 60 ns. Ciò significa che il processore, dopo aver eseguito $I(1)$, deve attendere 50 ns prima di poter iniziare l'esecuzione di $I(2)$. In altre parole la cadenza con la quale vengono eseguite le istruzioni non è quella veloce alla portata dell'UC, di 1 istruzione ogni 10 ns, ma quella più lenta di 1 istruzione ogni 60 ns, a causa del rallentamento imposto dalla lettura della RAM esterna. Se introduciamo una memoria *cache* di prestazioni (e capacità) intermedie tra registri e memoria principale esterna, p.es. basata su una tecnologia SRAM che risponde con una velocità di 20 ns, possiamo memorizzare un primo blocco di istruzioni nella *cache* (per esempio $I(1), I(2), \ldots, I(10)$) ed eseguirle subito dopo al tasso di 1 istruzione ogni 20 ns, incrementando la velocià di un fattore pari a 3. La memoria *cache* è dunque una memoria tampone, che serve come contenitore intermedio e provvisorio della RAM nella sua interazione con l'UC.
Per questo motivo a partire dagli anni $'70$ iniziò a diffondersi l'uso della memoria *cache* per velocizzare la lettura dalla memoria principale; in un primo periodo essa era unica, ma nel seguito vennero costituiti più *livelli*, tipicamente due per processori singoli e tre per processori *multicore*, cioè con più di un'UC all'intreno di un unico *chip*. Il primo di questi livelli, L1, sta immediatamente a ridosso dell'UC e ha una capacità relativamente modesta, ma è caratterizzato da una memoria molto veloce e costosa. La *cache* di secondo livello, la L2, è più capiente, più lenta e meno costosa, mentre il terzo livello L3 si usa solitamente nelle realizzazioni *multicore*, ed è una *cache* condivisa tra i vari *core*; la figura 7.11 mostra una fotografia della locazione fisica delle memorie *cache* all'interno di un processore *quad-core*.

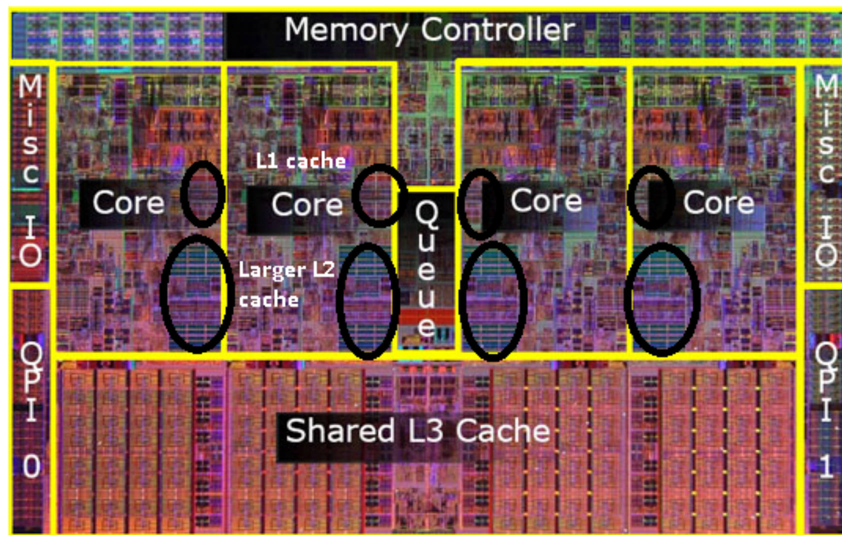


Figura 7.11: Locazione fisica delle memorie *cache* L1, L2 e L3 all'interno di un processore *quad-core*

La struttura di memoria che si sta delineando per il processore è allora quella di una *gerarchia delle memorie* la quale, partendo dai registri che sono i più veloci e costosi, passa per i tre livelli della *cache*, la RAM e infine la memoria di massa, cui corrisponde la capacità più elevata, ma una velocità di accesso più bassa; quest'ultima è disponibile tanto nella versione elettromeccanica (*Hard Disk*) che in quella a stato solido (*Solid State Disk*). La figura 7.12 offre una visione schematica di questa gerarchia, con un'indicazione di massima delle capacità in gioco. La memoria RAM (fig.7.13a) è solitamente alloggiata nelle immediate vicinanze del processore, su circuiti stampati connessi a pettine sulla scheda madre (fig.7.13b).

La memoria secondaria o di massa - L'ultimo livello della gerarchia di memoria che risiede all'interno della scheda madre è la *memoria di massa*. Su di essa vanno caricati tutti i programmi che vengono installati sul

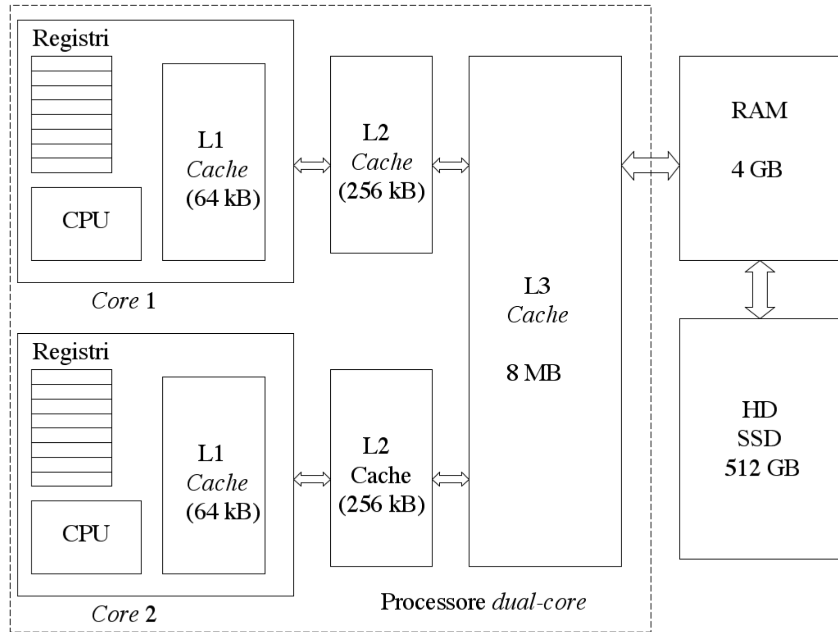
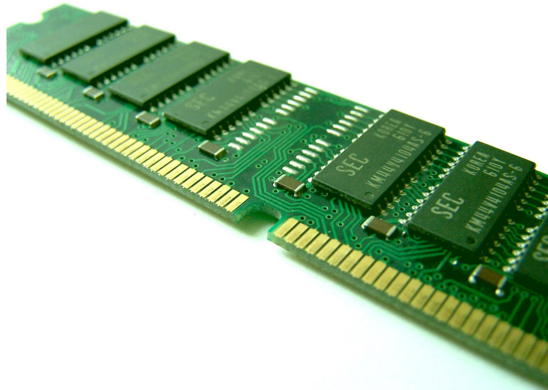
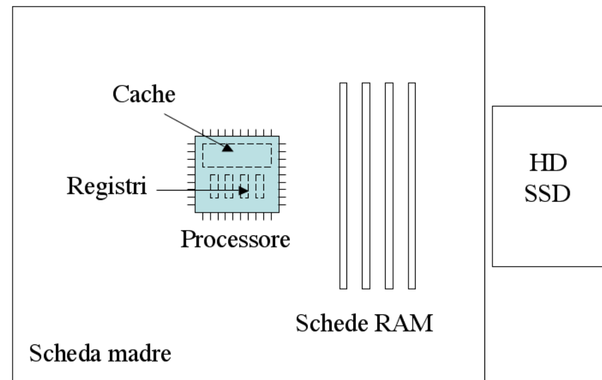


Figura 7.12: Struttura delle memorie *cache* di primo, secondo e terzo livello in un processore *dual-core*; in evidenza anche la memoria principale (RAM) e la memoria di massa (HD-SSD)



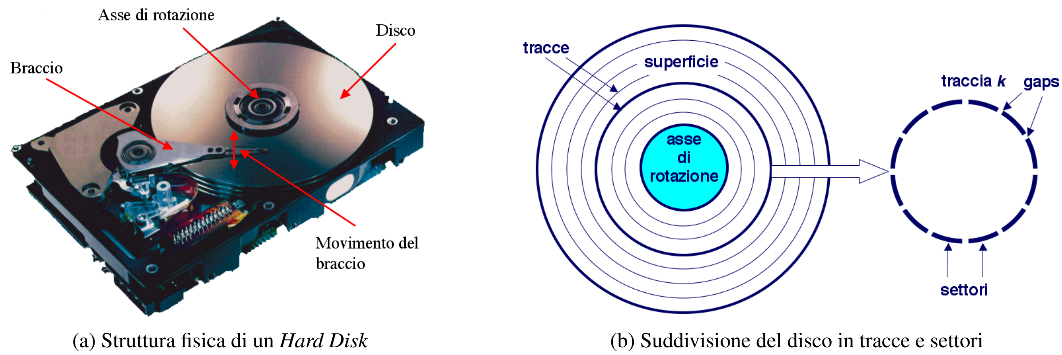
(a) Scheda di memoria RAM; ogni circuito integrato contiene 512 kB



(b) Collocazione delle schede di memoria RAM e della memoria di massa (HD-SSD) sulla scheda madre di un calcolatore

Figura 7.13:

computer e che sono quindi a disposizione dell'utente. Ciò implica la necessità di disporre di una memoria di tipo *permanente*, che non perda le informazioni memorizzate quando il circuito non è alimentato.
Alla fine degli anni ′50 le memorie di massa erano costruite impiegando i nuclei di ferrite visti in figura 2.24c, che sfruttano la permanenza di una polarizzazione magnetica del nucleo a seguito di una una corrente che scorre in un conduttore che attraversa il nucleo stesso. Successivamente vennero realizzati i primi *dischi rigidi*, o *Hard Disk*, che sono dei dischi di plastica rigida sui quali viene depositato uno stato sottilissimo di una sostanza ferromagnetica; essi vengono polarizzati secondo la tecnica illustrata nella figura 2.27a e vengono posti in rotazione a una velocità molto elevata, che varia tipicamente tra i 7000 e i 10000 giri/min. Gli *Hard Disk* sono dispositivi *elettromeccanici*, in quanto la lettura e la scrittura delle informazioni avviene mediante una testina magnetica montata su un braccio oscillante e comandata da servomeccanismi elettronici. La struttura di un *Hard Disk* è illustrata in figura 7.14a; in essa è visibile, oltre al disco e il suo asse di rotazione, anche il braccio che sorregge

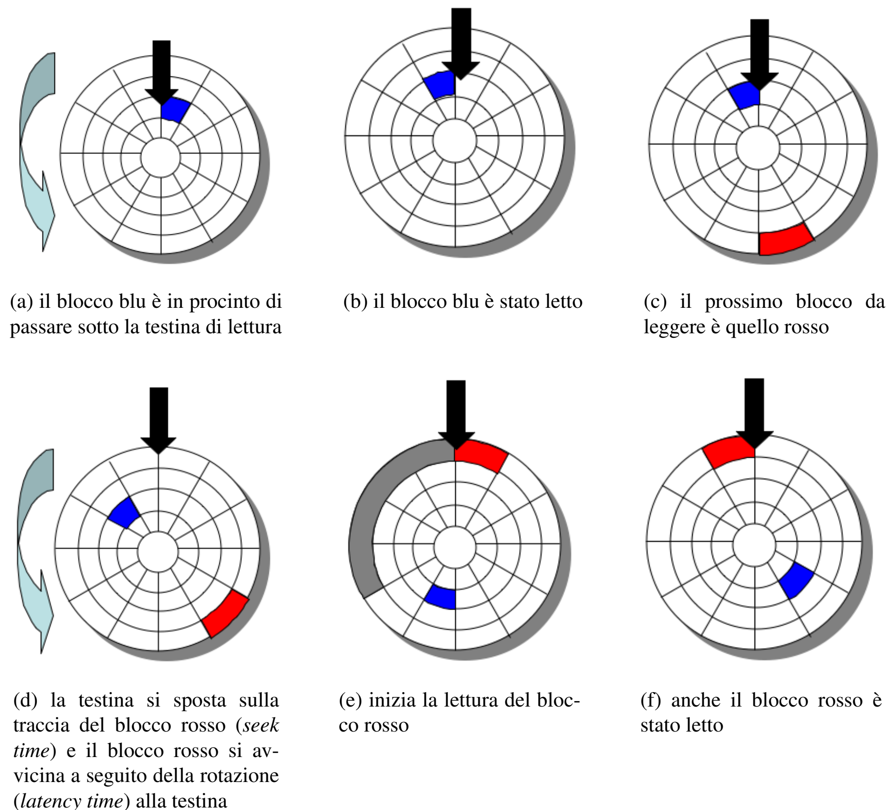


(a) Struttura fisica di un *Hard Disk*

(b) Suddivisione del disco in tracce e settori

Figura 7.14: Struttura interna di un *Hard Disk*

la testina magnetica di lettura/scrittura. Al disco viene poi attribuito un *formato* magnetico, basato su un certo numero di *tracce* concentriche e *settori*. Una caratteristica importante degli *Hard Disk* è che essi non hanno un indirizzamento sul singolo *byte*, bensì su un blocco la cui grandezza è compresa tra i 512 Byte, che sono un valore classico e standard, e i 4 kB delle ultime realizzazioni. L'accesso al singolo blocco non è effettuato in tempo costante, come nel caso delle memorie RAM, ma in un tempo che dipende dalla posizione della traccia cui il blocco appartiene e del blocco all'interno della traccia. Il tempo complessivo di accesso al disco dipende dal contributo



(a) il blocco blu è in procinto di passare sotto la testina di lettura

(b) il blocco blu è stato letto

(c) il prossimo blocco da leggere è quello rosso

(d) la testina si sposta sulla traccia del blocco rosso (*seek time*) e il blocco rosso si avvicina a seguito della rotazione (*latency time*) alla testina

(e) inizia la lettura del blocco rosso

(f) anche il blocco rosso è stato letto

Figura 7.15: *seek time* e *latency time* nel funzionamento di un disco rigido

di tre componenti, il *seek time* ($3 - 5$ ms), necessario alla testina per raggiungere la traccia sulla quale si trova il blocco che deve essere letto/scritto, il *latency time* (< 3 ms), necessario per attendere che il blocco passi per

effetto della rotazione del disco sotto la testina magnetica e il *transfer time* ($0,006$ ms), necessario per effettuare la lettura/scrittura. Il valore complessivo della lettura varia nell'intervallo $3-6$ ms, che è un tempo enorme rispetto ai tempi relativi alle memorie elettroniche; il rapporto è dell'ordine di 10^6 con i registri e di 10^5 con la memoria RAM. Come conseguenza anche per gli *Hard Disk* è necessario prevedere dei livelli di *cache* sulla RAM (*page cache*) o direttamente sull'elettronica del disco rigido (*disk buffer*) che agevolino la lettura dei dati.
Il meccanismo di lettura è illustrato in figura 7.15, tenendo conto del fatto che il disco sta girando in modo antiorario:

Fig. 7.15a il blocco blu è in procinto di passare sotto la testina di lettura

Fig. 7.15b il blocco blu è stato letto

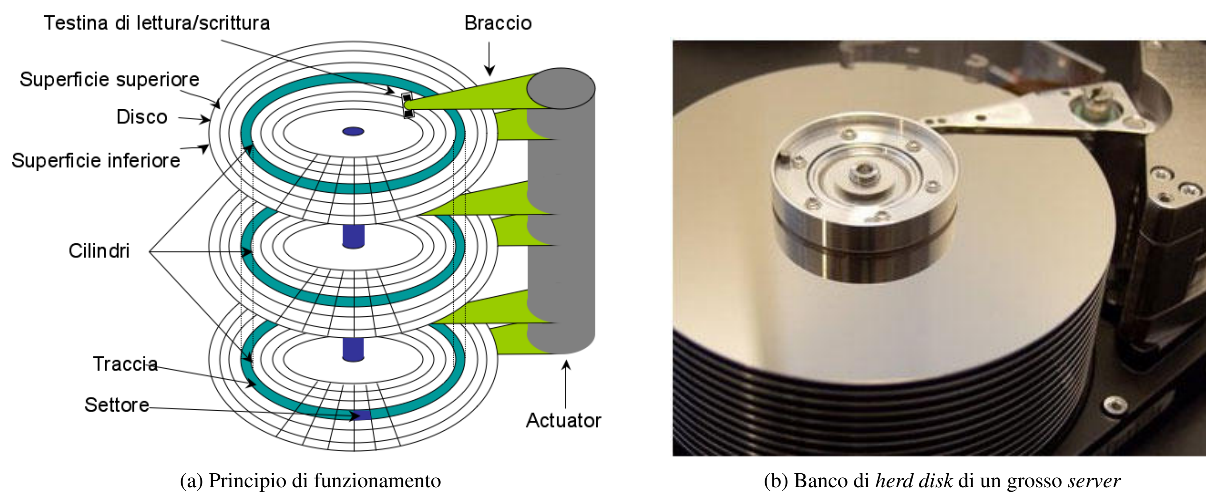
Fig. 7.15c il prossimo blocco da leggere è quello rosso

Fig. 7.15d la testina si sposta sulla traccia del blocco rosso (*seek time*) e il blocco rosso si avvicina a seguito della rotazione (*latency time*) alla testina

Fig. 7.15e inizia la lettura

Fig. 7.15f anche il blocco rosso è stato letto

Per quanto riguarda le altre caratteristiche dei dischi osserviamo che gli *Hard Disk* devono poter contenere tutti i programma in uso all'utente, oltre al *sistema operativo*, che è il programma che gestisce il funzionamento complessivo del *computer*; di conseguenza la capacità di memorizzazione deve essere notevole; nei modelli attuali arrivano a 512 GB per *computer* portatili, e a molti Tera Byte per le macchine più potenti. Nel caso di grossi *server* si usano più dischi rigidi organizzati in banchi, che operano in parallelo (spesso come dischi di *back-up*) ruotando sullo stesso asse (vedi fig7.16). Gli *hard-disk* sono caratterizzati da costi per unità di memoria molto contenuti e sempre in diminuzione.



(a) Principio di funzionamento

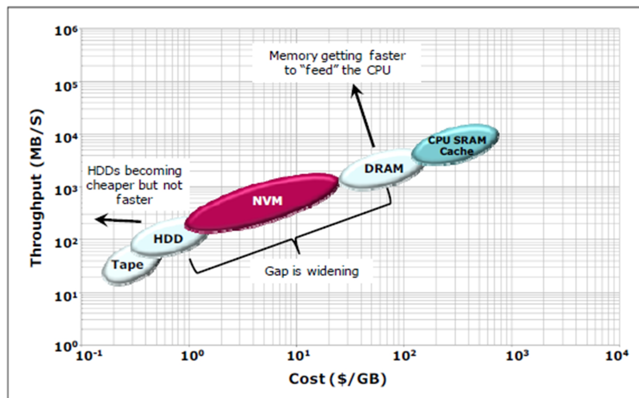
(b) Banco di *herd disk* di un grosso *server*

Figura 7.16: Banco di dischi rigidi che ruotano sullo stesso asse

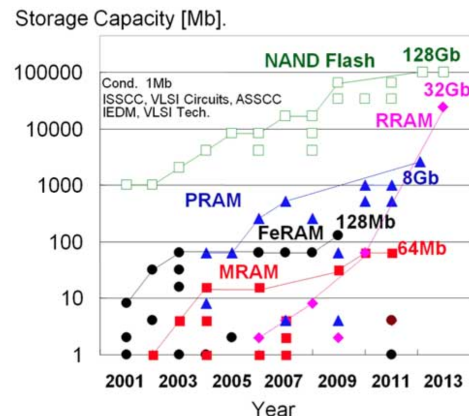
Se mettiamo in relazione il legame che esiste tra tipologia di memoria, la velocità di gestione dei dati e il costo per unità di GB (espresso in dollari USA), otteniamo il diagramma di figura 7.17a. Al vertice delle prestazioni (e dei costi) ci sono le memorie SRAM, impiegate per i registri e per la *cache* di primo livello. Segue la DRAM della memoria primaria e molto distanziati i dischi rigidi (HDD) e le memorie a *nastro magnetico*, di

cui non abbiamo parlato, ma che costituiscono l'ultimo gradino della gerarchia di memoria; essi sono disponibili solo come memoria di massa di *back-up* all'esterno del *computer* e sono caratterizzati da un costo bassissimo.
Come si vede bene dal diagramma, lo spazio tra DRAM e HDD è stato riempito di recente da una nuova tecnologia di memoria, denominata *Non Volatile Memory* (NVM). Si tratta di memorie elettroniche di tipo non volatile, nelle quali si riesce a mantenere l'informazione anche quando il circuito non viene alimentato. Ci sono diverse tecnologie a supporto delle memorie NVM, ma la più matura usata a tutt'oggi è quella delle memorie *flash*, usata in passato solo per apparecchiature sofisticate e ad alto costo (fotocamere digitali professionali), ma largamente diffusa ora per la produzione delle memorie a pennetta USB. Il principio fisico di funzionamento di una memoria *flash* è piuttosto complesso, e possiamo limitarci a dire che sfrutta la possibilità di costruire dei MOS-FET di tipo speciale, nei quali alcune cariche elettriche vengono imprigionate da un secondo *gate* isolato e a potenziale flottante. Recentemente sono state sviluppate anche delle NVM basate sulla sostituzione del dielettrico isolante dei MOS-FET con del materiale ferroelettrico (*Ferroelectric RAM* - FeRAM, F-RAM o FRAM) o addirittura sul fenomeno del *Magnetic Tunnel Junction* (MTJ), fenomeno descrivibile solo a livello di fisica quantistica, che porta alle promettenti memorie *Magnetoresistive* RAM (MRAM). Altri tipi di memoria NVM basati invece sulla variazione controllata di resistenza sono la *Resistive RAM* (RRAM), che sfruttano la proprietà di alcuni ossidi di cambiare il valore della resistenza in modo controllato elettricamente e la *Phase-change* RAM (PRAM), la cui variazione di resistenza deriva da una variazione dello stato cristallino dell'elemento costituente.
La disponibilità delle memorie NVM di tipo *flash*, a costi in veloce diminuzione, ha comportato la progressiva sostituzione dei dischi rigidi elettromeccanici con banchi di memoria elettronici denominati SSD (*Solid State Disk*), almeno per le tipologie di *computer* portatili. I tempi di accesso sono circa 200 volte migliori dei tempi per gli *Hard Disk*, poiché si attestano sui 25 μs; la figura 7.18 ci mostra un confronto tra un HD tradizionale e un suo equivalente nella versione SSD a stato solido.
Anche se la tecnologia elettromeccanica dei dischi rigidi è oramai sorpassata, questo tipo di disco continua a riscuotere un lusinghiero successo a causa del costo in continua diminuzione e alle doti di affidabilità e di maturità raggiunta da questa tecnologia; è comunque forza maggiore la sua progressiva alienazione man mano che saranno disponibili le memorie permanenti a stato solido a prezzi più convenienti.

Le memorie esterne - Facciamo solo qualche brevissimo cenno sulle memorie che si possono connettere esternamente al calcolatore tramite le porte d'ingresso o i dispositivi di lettura CD/DVD, poiché ne abbiamo già parlato precedentemente (vedi figura 2.27b). Per quanto riguarda i DVD possiamo dire che la tecnologia laser è nella fase finale del suo sviluppo e verrà presto soppiantata completamente da supporti basati su memorie NVM. Il motivo del declino è che anche in questo caso la presenza di una tecnologia elettromeccanica per la lettura dei dati, che non ha più altra giustificazione se non che quella economica, visto il bassissimo costo del supporto. Anche per



(a) Velocità di lettura dei dati delle varie memorie della gerarchia rapportata al costo



(b) Capacità raggiunte da varie tipologie di memorie NVM negli ultimi anni

Figura 7.17:

Figura 7.18: Confroto tra un disco rigido elettromeccanico e la corrispondente versione SSD a stato solido, basata sulle memorie NVM

quanto attiene i dischi rigidi esterni, che si usano come unità di *back-up* la loro sostituzione con memorie NVM è solo una questione di tempo. Dal diagramma 7.17a e dalla tendenza delle recenti tecnologie NVM si può intuire che si sta andando verso un'architettura del *computer* in cui la differenza tra memoria principale e memoria di massa del tipo SSD si farà sempre più sfumata, man mano che aumenteranno le prestazioni della tecnologia a stato solido del tipo NVM. È verosimile che tra qualche tempo i dischi rigidi non saranno più usati neanche sui grossi

Tipo di memoria	Tempo di accesso	Velocità trasferimento dati MB/s	Costo Euro/GB	Capacità
Registri	$0,5$ ns	$3000 \div 17000$	$110 \div 540$	2 kB
Cache L1	2 ns			64 kB
Cache L2	5 ns			256 kB
Cache L3	$20 \div 50$ ns			8 MB
Memoria principale (RAM)	$30 \div 90$ ns	$1000 \div 6400$	10	4 GB
Non Volatile Memory (MRAM)	35 ns	400	5000	4 MB
Solid State Disk (SSD)	$25 \div 100$ μs	250	$0,7$	512 GB
Hard Disk (HD)	$5 \div 20$ ms	80	$0,1$	512 GB
Dischi laser (DVD)	$0,1 \div 5$ s	$4,5 \div 20$	$0,15$	25 GB
Nastri magnetici	> 10 s	$80 \div 240$	$0,15 \div 0,40$	1 TB

Figura 7.19: Principali caratteristiche delle memorie di un *computer*. Nella parte alta troviamo le memorie più vicine all'UC, più veloci e più costose, e nella parte bassa le memorie più distanti, lente ed economiche

server, realizzando anche per questi dispositivi quella sorta di fusione tra i vari strati della gerarchia di memoria, che si distribuirà in modo molto più uniforme tra interno ed esterno della CPU, senza soluzione di continuità e basata tutta su tecnologie elettroniche.
Nella tabella di figura 7.19 riportiamo un quadro riassuntivo delle principali caratteristiche delle memorie di un *computer*, organizzate in una gerarchia che vede nella parte alta le memorie più vicine all'UC, più veloci e più costose, e nella parte bassa le memorie più distanti, lente ed economiche. Si noti il prezzo delle memorie MRAM,

che riflette una tecnologia che rimane ancora d'avanguardia.
Concludiamo con la figura 7.20, che offre uno sguardo d'insieme della topologia del calcolatore, che mette in evidenza i dispositivi principali e la loro connessione realizzata tramite *bus*.

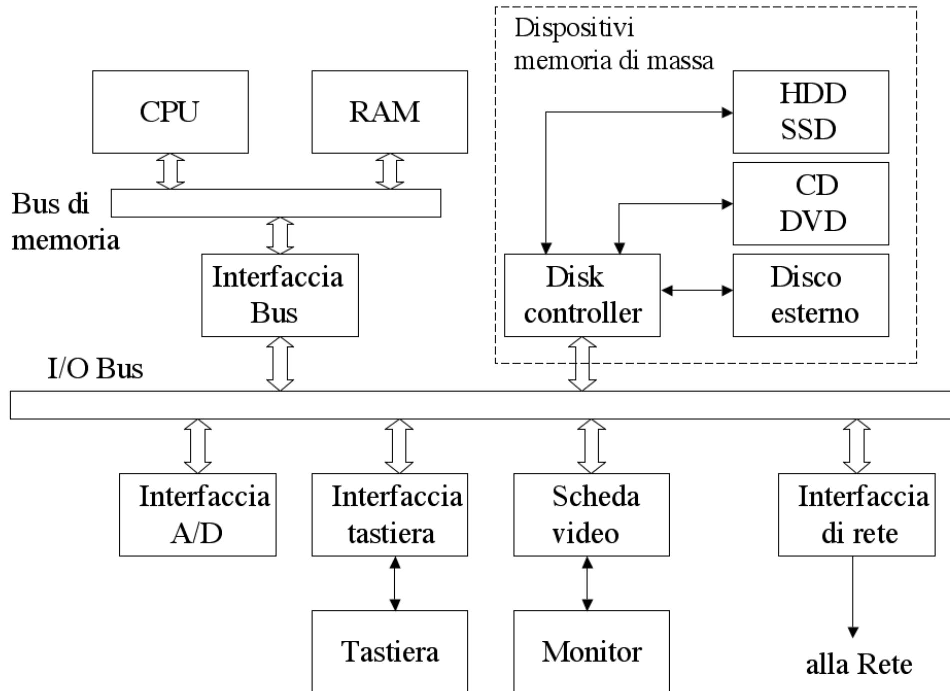


Figura 7.20: Dispositivi principali di un calcolatore