ARHITECTURA CALCULATOARELOR 2003/2004
CURSUL 8

Capitolul 3: Organizarea sistemelor de calcul

Un calculator numeric este un sistem interconectat de **procesoare, memorii, şi dispozitive intrare/ieşire**. Aceste trei componente sunt conceptele cheie ale arhitecturii calculatorului.

3.1 Procesoare

3.1.1 Organizarea CPU

Organizarea unei structuri de calcul simple, bazată pe o magistrală, este prezentată în figura 3.1. **CPU - ul (Unitatea Centrala de Procesare**) este "creierul" calculatorului. Funcțiunea sa este de a executa programele stocate în memoria principală prin extragerea instrucțiunilor, decodificarea lor şi apoi executarea lor una după alta. Componentele sunt conectate printr-o magistrală (bus), care este, din punct de vedere fizic, o colecție de trasee electrice pentru transmiterea adreselor, datelor şi a semnalelor de control. Magistrala poate fi externă CPU-ului, conectând dispozitivele de I/ O şi memoria, sau poate fi internă CPU- ului.

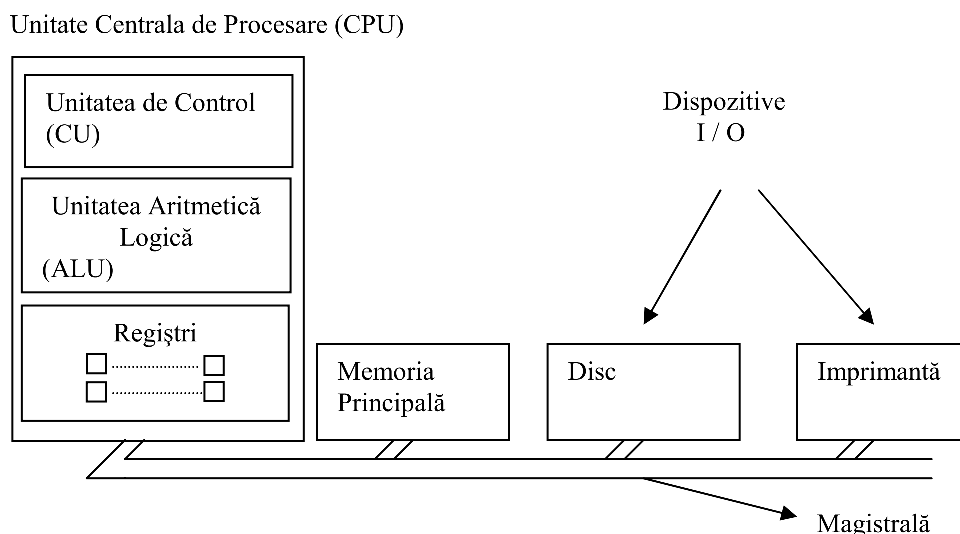

Figura 3.1 Organizarea unui calculator simplu cu CPU si doua dispozitive I/O

CPU este compus din mai multe părți distincte. **CU (Unitatea de control)** este responsabilă pentru extragerea instrucțiunilor din memoria principală şi determinarea tipului lor. **ALU (Unitatea Aritmetica – Logica)** execută operații aritmetice şi logice, cum ar fi adunarea sau AND logic, care sunt necesare pentru a realizarea instrucțiunilor.

CPU contine deasemenea o memorie mică de înaltă viteză folosită pentru a stoca rezultate temporare şi anumite informații de control. Memoria este alcatuită dintr-un număr de regiştri,

din care fiecare are o anumită mărime şi funcţiune. Deobicei, toti regiştrii au aceeaşi mărime (lungime). Fiecare registru poate conţine un număr, până la un maximum determinat de mărimea registrului. Regiştrii pot fi citiţi şi scrişi cu o mare viteză pentru că sunt interni CPU-ului.

Registrul cel mai important este **PC (Numărătorul de program)**, care indică adresa următoarei instrucţiuni care urmează să fie executată. Numele “Numărător de program” este într-un fel greşit deoarece acest registru nu are nimic de a face cu numărarea, dar termenul este foarte folosit. Deasemenea important este **IR (Registrul de instrucţiune)**, care memorează codul instrucţiunii curente care se execută. Majoritatea calculatoarelor au numeroase alte registre, unele de uz general ca şi altele pentru scopuri specifice.

Un exemplu de organizare tipică von Neumann pentru CPU este prezentată în Figura 3.2. Această structură este numită **cale de date (data path)** şi este alcatuită din regiştri (tipic de la 1 la 32), ALU şi mai multe magistrale conectând componentele. Regiştrii sunt conectaţi la doi regiştri de intrare ai ALU, etichetaţi A şi B in figura 3.1. Aceşti regiştri memorează intrarea ALU în timp ce ALU calculează. Calea de date este foarte importantă în toate sistemele de calcul şi vom reveni asupra acestui subiect.

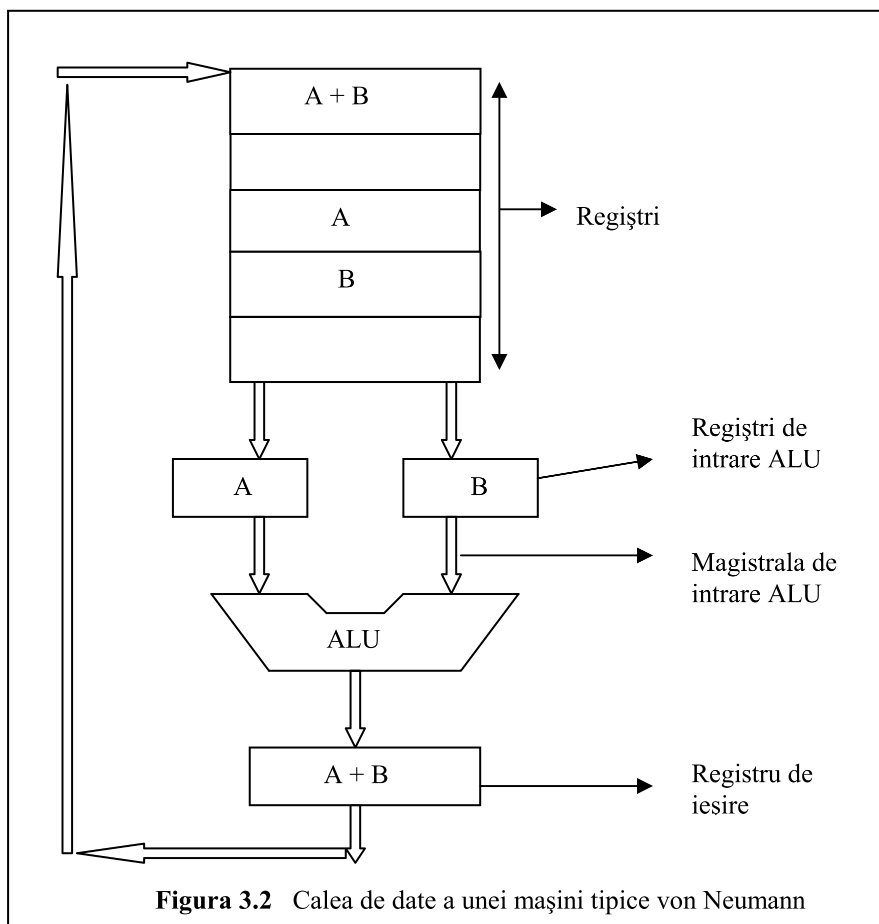

Figura 3.2 Calea de date a unei maşini tipice von Neumann

Chiar ALU realizează adunarea, scăderea, şi alte operaţii asupra intrarilor sale şi oferă rezultatul în registrul de ieşire. Acest registru de ieşire poate fi stocat înapoi într-un registru. Mai tarziu, registrul poate fi scris (adică stocat) în memorie, dacă se doreşte. Nu toate

tructurile de calcul au A, B, şi registru de ieşire. În exemplu este presupusă o operație de adunare.

Majoritatea instrucțiunilor pot fi împărțite în două categorii: registru - memorie sau registru - registru. Unele instrucțiuni registru - memorie permit aducerea cuvintelor de memorie în regiştri, unde ele pot fi folosite ca intrări ALU pentru instrucțiunile corespunzătoare. Cuvintele (word) sunt unități de date care se mută între memorie şi regiştrii. Un cuvânt poate fi şi un intreg. Alte instrucțiuni registru - memorie permit ca registrele să fie stocate înapoi în memorie.

Celalalt tip de instrucțiune este registru - registru. O instrucțiune registru - registru tipică aduce cei doi operanzi din registre, îi depune în regiştrii de intrare ALU, realizează unele operații cu ele (de exemplu, adunarea sau AND logic) şi stochează rezultatul într-unul dintre regiştri. Procesul trecerii a doi operanzii prin ALU şi stocarea rezultatului este numit **ciclu căii de date ş**i este partea cea mai importantă a majorității CPU - urilor. Prin extensie, el defineşte ceea ce poate face maşina. Cu cât ciclul căii de date este mai rapid cu atât maşina respectivă este mai rapidă.

3.1.2 Executarea instrucțiunilor

CPU execută fiecare instrucțiune într- o secvență de paşi. În linii mari paşii sunt următorii:

1. Extrage instrucțiunea următoare din memorie în registrul de intrare,
2. Schimbă numărătorul de programe pentru a indica instrucțiunea următoare,
3. Determină tipul instrucțiunii care tocmai a fost încărcată,
4. Dacă instrucțiunea foloseşte un cuvânt din memorie determină unde este el,
5. Dacă este necesar, aduce cuvântul într- un registru CPU,
6. Execută instrucțiunea,
7. Reia de la pasul 1 pentru a începe executarea următoarei instrucțiuni.

Această succesiune de paşi este frecvent referită drept ciclul **extragere (fetch) - decodificare - execuție.** Ea este activitatea centrală pentru CPU în toate calculatoarele.

Această descriere a modului cum lucrează CPU - urile seamănă foarte mult cu un program scris în limbaj natural. În figura 3.3 se prezintă un astfel de program rescris într-o manieră procedurală (Java) şi numit interpretor.

Maşina care este interpretată are doi regiştri vizibili din punctul de vedere al utilizatorului:

- numarător de programe (PC), pentru memorarea suitei de adrese ale următoarei instrucțiuni care trebuie extrase şi
- acumulatorul (AC), pentru memorarea rezultateor aritmetice.

Ea are deasemenea regiştri interni pentru memorarea:

- instrucțiunii curente (instr),
- tipului instrucțiunii curente (instr_type),
- adresei operandului instrucțiunii (data_loc),
- a operandului curent însuşi (data).

Se presupune că instrucțiunile conțin o singură adresă de memorie. Locația de memorie adresată conține operandul, de exemplu valoarea care să fie adăugată la acumulator.

```
public class Interp {
        static int PC;                    // registru pentru adresa instrucțiunii următoare
        static int AC;                    // acumulatorul, un registru pentru operații aritmetice
        static int instr;                 // registru pentru instrucțiunea curentă
        static int instr_type;            // tipul de instrucțiune (opcode)
        static int data_loc;              // adresa operandului, sau –1 dacă nu este niciunul
        static int data;                  // registru pentru operandul curent
        static boolean run_bit = true;    // un comutator care poate fi trecut pe off pentru a
                                          // opri calculatorul

public static void interpret(int memory[ ] , int starting_address )  {
// Acest program interpretează programe pentru un calculator simplu cu instrucțiuni care au un
//singur operand din memorie. Această maşină are un registru AC (acumulator), folosit pentru
//operații aritmetice. Spre exemplu, instrucțiunea ADD adaugă un întreg din memoria la AC.
//Interpretorul continuă rularea până ce o instrucțiune HALT aduce run_bit la valoarea 0.
//Starea rulării proceselor pe această maşină constă din memorie, numărătorul de program,
//run_bit şi AC. Parametrii de intrare sunt imaginea memoriei şi adresa de început.
        PC = starting_address;
        While (run_bit)         {
                instr = memory[PC];                    // aduce instrucțiunea următoare în instr
                PC = PC + 1;                           // incrementează numărătorul de program
                instr_type = get_instr_type(instr);    // determină tipul de instrucțiune
                data_loc=find_data(instr, instr_type); // localizează data, -1 dacă nu e niciuna
                if ( data_loc >= 0)                    // dacă –1 atunci nu este nici un operand
                data = memory [data_loc];              // încarcă operand
                execute(instr_type, data);             // execută instrucțiunea
                        }
                                                                }
private static int get_instr_type(int addr)          { … }
private static int find_data(int instr, int type)    { … }
private static void execute(int type, int data)      { … }
                }
```

Figura 3.3 Un interpretor pentru un calculator simplu (scris în Java).

Faptul că este posibil să se scrie un program care să simuleze funcționarea CPU arată că nu este necesar ca un program să fie executat de către un CPU hardware constând dintr-o cutie plină de electronice. În schimb, un program poate fi executat de alt program care îi extrage, decodifică şi execută instrucțiunile. Un program (ca în figura 3.3) care extrage, decodifică şi execută instrucțiunile altui program este numit **interpretor**.

Această echivalență între procesoarele hardware şi interpretoarele software are implicații importante pentru organizarea şi proiectarea sistemelor de calcul. După ce a specificat limbajul maşinii, L, pentru un calculator nou, echipa de proiectare decide dacă vor să construiască un procesor hardware pentru a executa direct programe în L sau, alternativ, dacă vor să scrie un interpretor pentru a interpreta programe pentru L. Dacă ei aleg să scrie un interpretator, ei trebuie deasemenea să ofere o altă maşină hardware care să ruleze interpretatorul. Anumite construcții hibrid sunt deasemenea posibile cu unele execuții hardware, dar şi cu interpretare software.

Un interpretor sparge instrucţiunile unei maşini ţintă în paşi mici. Ca o consecinţă, maşina pe care rulează interpretorul poate fi mult mai simplă şi mai puţin scumpă decât un procesor hardware pentru maşina ţintă. Această economie este semnificativă dacă maşina ţintă are un număr mare de instrucţiuni şi instrucţiunile sunt complicate, cu multe opţiuni. Economia provine în esenţă din faptul că un sistem hardware este înlocuit de un sistem software (interpretator).

Primele computere aveau seturi simple de instrucţiuni. Dar goana pentru computere mai puternice conduce, printre alte lucruri, la instrucţiuni individuale mai puternice. Foarte repede s-a descoperit că instrucţiunile mai complexe conduc la execuţii mai rapide ale programelor, chiar dacă instrucţiunile individuale durează mai mult. O instrucţiune în virgulă mobilă este un exemplu de instrucţiune complexă. Instrucţiunile pentru accesarea blocurilor de elemente este alt exemplu. Tot aşa de simplă este observaţia că pentru două instrucţiuni care apar frevcent consecutiv este preferabilă o singură instrucţiune care poate să realizeze mai rapid efectul celor două.

Instrucţiunile mai complexe sunt preferabile deoarece executarea operaţiilor individuale poate fi executată în paralel folosind resurse hardware diferite. Pentru computerele scumpe de înaltă performanţă costul acesta suplimentar ar putea fi justificat. De aceea computerele scumpe de înaltă performanţă tind să aibă mult mai multe instrucţiuni decât calculatoarele cu un preţ mai scăzut. Totuşi creşterea costului dezvoltării de software şi cerinţele de compatibilitate a instrucţiunilor au creat necesitatea de a implementa instrucţiuni complexe chiar şi pe calculatoarele de preţ scăzut, unde preţul este mai important decât viteza.

O implementare hardware directă, adica neinterpretată, a fost folosită numai în implementarea modelelor scumpe. Calculatoarele simple cu instrucţiuni interpretate au avut deasemenea alte beneficii. Printre cele mai importante importante au fost:

- Abilitatea de a repara instrucţiunile implementate incorect sau a unor deficienţe de proiectare în hardware – ul de baza,
- Oportunitatea de a adăuga unele noi instrucţiuni, cu cost minimal, chiar după livrarea maşinii,
- Proiectarea structurată care a permis dezvoltarea eficientă, testarea şi documentarea instrucţiunilor complexe.

Prin 1950 IBM a sesizat că dezvoltarea unei singure familii de maşini, toate executând aceleaşi instrucţiuni, are multe avantaje atâat pentru IBM cât şi pentru clienţii ei. IBM a introdus termenul de **arhitectură** pentru a descrie acest nivel de compatibilitate. O nouă familie de calculatoare ar avea o aceeaşi arhitectură, dar multe implementări diferite care să poată toate să execute acelaşi program, diferind numai preţul şi viteza. Dar cum să construim un computer de preţ scăzut care ar putea executa toate instrucţiunile complicate ale unei maşini scumpe de înaltă performanţă? Raspunsul sta în **interpretare**. Această tehnică, care a fost prima dată sugerată de Wilkes (1951), a permis proiectarea calculatoarelor simple, de preţ mai scăzut, care totuşi au putut să execute un mare număr de instrucţiuni. Rezultatul a fost arhitectura sistemului IBM/ 360, o familie de calculatoare compatibile, traversând aproape doua ordine de magnitudine, atât în preţ cât şi în capacitate.

Pentru că piaţa de calculatoare a explodat în anii 1970 şi capacităţile de calcul au crescut rapid, cereare de calculatoare cu preţ scăzut a favorizat proiectarea calculatoarele folosind interpretoare. Abilitatea de a imbina hardware şi interpretoare pentru un set de instrucţiuni

particulare s-a dovedit un factor important pentru proiectarea efecientă. Pentru că tehnologia de semiconductori a avansat rapid avantajele costului au depăşit ponderea acordată performanţelor mai înalte şi **astfel arhitecturile bazate pe interpretor au devenit calea convenţională de a proiecta calculatoarele**. Aproape toate noile calculatoare proiectate începând din 1970, de la microcalculatoare la calculatoare mainframes, s-au bazat pe o interpretoare.

Prin anii '70 folosirea procesoarelor simple care folosesc interpretoare a devenit foarte larg răspândită exceptând cele mai scumpe modele de înaltă performanţă, ca şi Cray - 1 şi seriile Control Data Cyber. Folosirea unui interpretor a eliminat inevitabilele limitări de cost ale instrucţiunilor complexe şi arhitecturile au început să includă mult mai mult instrucţiunile complexe, în particular cu căi multiple de a specifica operanzii. Curentul a atins zenitul cu calculatorul VAX de la DEC care au avut mai multe sute de instrucţiuni şi mai multe de 200 de căi diferite de specificare a operanzilor care să fie folosiţi în fiecare instrucţiune. Din nefericire arhitectura VAX a fost concepută de la început să fie implementată cu un interpretor, făra a lua în considerare posibilitatea de a implementa un model de performanţă. Ca urmare s-au inclus un număr foarte mare de instrucţiuni de valoare marginală care erau dificil de executat direct. Această omisiune s-a dovedit a fi fatală lui VAX. DEC a fost cumparat de Compaq în 1998.

Deşi la început procesoarele cu 8 biţi erau maşini foarte simple cu seturi de instrucţiuni foarte simple, prin anii 1970, chiar microprocesoarele au fost transformate în proiecte bazate pe interpretoare. În timpul acestei perioade una din provocările majore a fost creşterea complexităţii circuitelor integrate. Un avantaj major al abordării bazate pe interpretor a fost abilitatea de a proiecta un procesor simplu, care transfera complexitatea la memoria care găzduieşte interpretorul. De aceea o proiectare complexă hardware a putut fi transformată într-o proiectare complexă software. Succesul Motorola 68000 care a avut un mare set de instrucţiuni interpretate şi eşecul concurentului Zilog Z8000, care a avut un set de instrucţiuni egal de mare, dar fără un interpretator, a demonstrat avantajele unui interpretor prin aducerea rapidă pe piaţă a noului microprocesor. Succesul a fost tot mai mare surprinzând conducerea lui Zilog, mai ales că predecesorul lui Z8000, Z80, a fost mai popular decât predecesorul lui 68000, 6800). Bineînţeles alţi factori au fost deasemenea implicaţi aici, nu în ultimul rând a fost lunga istoriei a firmei Motorola ca fabricant de cipuri, în timp ce Exxon (detinatorul lui Zilog) este o companie de petrol nu un fabricant cipuri.

Alt actor lucrând în favoarea interpretării a fost existenţa memoriilor ROM rapide (control stores) capabile să stocheze interpretorul. Să presupunem că instrucţiunea tipică 68000 interpretată corespunde la 10 instrucţuni interpretor, numite **microinstrucţuni**, a câte 100 nanosecunde fiecare, şi două accesuri la memoria principală, a câte 500 nanosecunde fiecare. Timpul total de execuţie a fost atunci 2000 nanosecunde, de numai două ori mai rău decât ar fi putut să ofere cea mai bună execuţie directă. Dacă memoria rapidă nu ar fi fost disponibilă, atunci instrucţiunea ar fi luat 6000 de nanosecunde.

3.2 Memoria cache

De-a lungul timpului, **microprocesoarele au fost întotdeauna mai rapide decât memoriile**. Odată cu îmbunătăţirea memoriilor, se perfecţionau şi procesoarele, păstrând raportul. De fapt, cum devine posibil să se pună mai multe circuite pe un chip, proiectanţii de procesoare folosesc aceste noi facilităţi pentru operaţii de tip pipeline şi superscalare, făcând procesoarele

şi mai rapide. Proiectanții memoriilor au folosit de obicei noile tehnologii pentru creşterea capacității chipurilor, nu a vitezei, aşa că problema a avut o tendință de înrăutățire în timp. În practică, acest dezacord înseamna ca după ce procesorul emite o cerere către memorie, nu va primi cuvântul necesar timp de multe impulsuri de tact. **Cu cât memoria este mai lentă, cu atât mai multe tacturi va trebui să aştepte procesorul.**

Sunt două moduri de a trata problema. Cel mai simplu mod este de a începe citirile READ de memorie când sunt întâlnite, dar continuând execuția şi **oprind procesorul dacă o instrucțiune încearcă să folosească cuvântul de memorie înainte ca acesta să fie disponibil**. Cu cât este mai lentă memoria, cu atât mai des va apărea această problemă, şi cu atât mai grave vor fi sancțiunile atunci când se întamplă. De exemplu, dacă memoria are o întârziere de 10 perioade de tact, este foarte probabil ca una dintre următoarele 10 instrucțiuni să încerce folosirea cuvântului citit.

Cealaltă soluție este să se instruiască maşinile să nu se oprească, dar în schimb să ceară **compilatoarelor să nu genereze cod care să folosească cuvintele înainte de a sosi din memorie**. Problema este că această abordare este mult mai greu de implementat. După o instrucțiune de încărcare, nu mai este nimic de făcut, şi compilatorul este forțat să adauge instrucțiunea NOP (no operation), care nu face decât să ocupe o pauză şi să piardă timpul. Efectul este că această abordare conduce la o întârziere software, nu hardeware, dar pierderea de performanță este aceeaşi.

De fapt, problema este mai mult de economie (bani) decât de tehnică. Inginerii ştiu cum să construiască memorii la fel de rapide ca procesorul, dar pentru a funcționa la viteza maximă, acestea trebuie să fie aşezate pe chipul procesorului (deoarece transferul prin magistrală este foarte lent). Adăugarea unei memorii mari pe chipul procesorului îl face mai mare, deci şi mai scump, şi chiar dacă nu costul ar fi problema, există limitări tehnologice ale dimensiunii chipului procesorului. **Astfel, alegerea devine să avem o cantitate mare de memorie lentă, sau o cantitate mică de memorie rapidă**. Bineințeles că noi am prefera o cantitate mare de memorie rapidă la un preț mic.

Se cunosc tehnici de combinare a unei cantități mari de mamorie lentă cu o cantitate mică de memorie rapidă, pentru a obține viteza memoriei rapide (aproape) şi capacitatea unei memorii mari, la un preț moderat. **Memoria mică şi rapidă se numeşte cache** (de la francezul "cacher", care înseamnă a ascunde). În continuare vom descrie pe scurt cum funcționează şi cum se folosesc memoriile cache.

Ideea de bază a cache-ului este simplă: **cuvintele de memorie cele mai utilizate se pastrează în cache**. Când procesorul are nevoie de un cuvânt, caută întâi în cache. Numai în cazul când cuvântul nu este acolo, îl va cauta în memoria principală. Dacă o parte substanțială a cuvintelor este în cache, timpul mediu de acces poate fi redus foarte mult. Succesul sau eşecul depinde de ce fracțiune din cuvinte utile sunt în cache. De muți ani, oamenii ştiu că programele nu accesează memoria complet aleator. Dacă se face o referință la locația de memorie de adresă A, **este probabil ca următoarea referință de memorie să fie în vecinătatea** lui A. Un exemplu simplu este chiar programul. În afară de salturi şi apeluri de procedură, **instrucțiunile se aduc din locații de memorie consecutive**. Mai mult, majoritatea timpului de execuție al programului este petrecut în **bucle**, unde un număr limitat de instrucțiuni se execută de mai multe ori. Asemănător, un program de manipulare a unei matrice este probabil să acceseze de multe ori aceeaşi matrice înainte de a trece la altceva.

Observația că referințele de memorie într-un interval de timp scurt tind să folosească numai o zonă mică din memoria totală se numeşte **"principiul localizării"** şi este baza tuturor sistemelor de cache. Ideea generală este că atunci când un cuvânt este accesat, acesta şi câțiva dintre vecinii săi sunt aduşi din memoria mare şi lentă în cache, astfel că data viitoare când se foloseşte, va fi accesat foarte rapid. O aşezare obişnuită a procesorului, memoriei cache şi memoriei principale este ilustrată în figura 3.4. Memoria cache se situează logic între CPU şi memoria principală. Fizic, sunt mai multe localizări posibile.

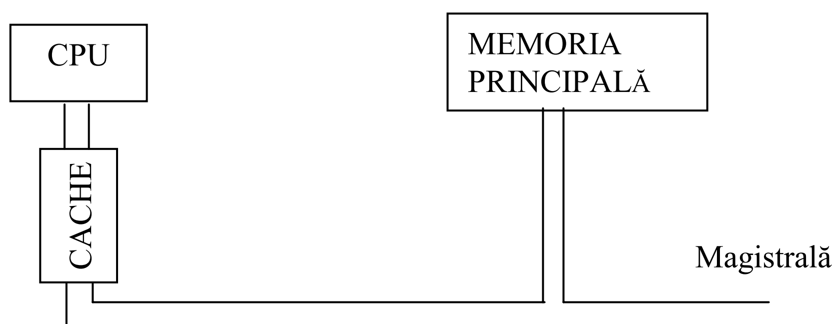

Figura 3.4 Poziția relativă a memoriei cache față de restul componentelor sistemului

Dacă un cuvânt este citit sau scris de k ori într-un interval scurt, calculatorul va folosi 1 referire către memoria lentă şi k-1 referiri către memoria rapidă. Cu cât mai mare este k, cu atât mai bună este performanța globală.

Putem formaliza aceste operații prin introducerea următoarelor valori:
- c timpul de acces al cache,
- m timpul de acces al memoriei principale,
- h raportul de reuşită, care este raportul din toate referirile care pot fi satisfăcute de cache şi numărul total de referiri.

În exemplul din paragraful precedent, $h = \frac{k-1}{k}$.

Unii autori definesc deasemenea raportul de ratare, care este 1-h = $\frac{1}{k}$.

Cu aceste definiții, putem calcula timpul mediu de acces, t_{med}, astfel:

$$T_{med} = c + (1-h)m$$

Daca $h \rightarrow 1$, toate referirile pot fi satisfăcute din cache, şi timpul de acces se apropie de c. Din contră, dacă $h \rightarrow 0$, o referire la memoria principală este necesară de fiecare dată, aşa că timpul de acces se apropie de (c + m), c pentru a verifica memoria cache (fără succes), şi m pentru a efectua referirea la memorie.

La unele sisteme, referirea (accesul) la memorie poate fi pornită în paralel cu căutarea în cache, aşa că dacă nu se găseşte în cache, ciclul de memorie este deja început. Oricum, această strategie necesită ca memoria să poată fi oprită pe parcurs la o găsire a cuvântului în cache, ceea ce face implementarea mai complicată.

Funcționând pe principiul localizării, memoriile principale şi cache sunt divizate în blocuri de dimensiune fixă. Când se vorbeşte de aceste blocuri în cache, se foloseşte de obicei denumirea "linii de cache". Când se produce o ratare, toată linia de cache se încarcă din memorie, nu numai cuvântul necesar. De exemplu, pentru o linie de 64 biți, o referință către adresa de memorie 256 va încărca în cache linia formată din octeții de la 256 la 319. Cu puțin noroc, unii dinte octeții din linia de cache respectivă vor fi apelați în scurt timp. **Operarea în acest fel este mult mai rapidă decât extragerea câte unui cuvânt, deoarece este mai rapid să se extragă k cuvinte o dată, decât câte un cuvânt de k ori.**

Proiectarea memoriei cache este un subiect de importanță tot mai mare pentru procesoarele de înaltă performanță.

Observații:
1. O problemă este dimensiunea memoriei cache. Cu cât este mai mare, cu atât mai bune sunt rezultatele.
2. O altă problemă este dimensiunea liniei de cache. 16 k cache pot fi împărțiți în 1k linii de 16 octeți, 2k linii de 8 octeți şi alte combinații.
3. A treia problemă este organizarea, sau cum ştie memoria cache ce cuvinte păstrează la un moment dat, deci cum se selectează informațiile de păstrat în cache. Ideal ar fi să se poată face o predicție asupra informațiilor care urmează să fie folosite.
4. O a patra problema este dacă datele şi instrucțiunile sunt păstrate în acelaşi cache sau nu. Un cache unificat (instrucțiunile şi datele folosesc acelaşi cache), este mai simplu de proiectat şi favorizează automat extragerile de instrucțiuni. Oricum, astăzi tendința este către un cache separat, cu instrucțiunile într-o parte şi datele în alta. Aceasta se mai numeşte arhitectura Harvard. Forța care îi ghidează pe proiectanți în această direcție este răspândirea procesoarelor "pipeline". Zona de instrucțiuni trebuie să acceseze instrucțiunile în acelaşi timp în care în zona de operanzi se accesează datele. Un cache separat permite accesul simultan, pe când cel unificat nu. De asemenea, cum instrucțiunile nu se modifică în timpul execuției, conținutul memoriei cache de cod nu va fi niciodată rescris în memorie.
5. În final, o a cincea problemă este numărul memoriilor cache. Este obişnuit astăzi să avem chipuri cu cache primar pe chip, cache secundar în afara chipului, dar în aceeaşi capsulă, şi un al treilea nivel de cache mai departe.