SISTEM DE CALCUL IN TIMP REAL

SCTR

-SZOKE ENIKO -

Curs 3

Algoritmii de control utilizaţi depind de aplicaţie. Cei mai folosiţi sunt algoritmii care pleacă de la *algoritmul analogic de control cu 3 termeni* (proporţional+integral+derivativ - *PID*). Ecuaţia în domeniul timpului pentru controller-ul PID ideal este [Stu88]:

$$m(t) = K_c \left[e(t) + \frac{1}{T_i} \int e(t)dt + T_d \frac{de(t)}{dt} \right] \quad (1)$$

unde $e(t)=r(t)-y(t)$, cu $y(t)$ – variabila măsurată la ieşire, $r(t)$ - variabila de referinţă (set-point), $e(t)$ - eroarea. K_c este factorul de amplificare global al controller-ului, T_i este constanta de timp de integrare iar T_d constanta de timp a acţiunii derivative. Acest algoritm poate fi exprimat şi în alte forme. De exemplu, acţiunea derivativă este în mod frecvent neutilizată sau uneori *de/dt* este înlocuit prin *dy/dt* pentru a evita diferenţierea variabilei de referinţă etc.

Algoritmul poate fi implementat soft utilizând o ecuaţie echivalentă pentru (1). Astfel, dacă intervalul de eşantionare pentru calcule este T secunde, atunci pot fi utilizate aproximările

$$\frac{de}{dk}\Big|_k = \frac{e_k - e_{k-1}}{T} \quad si \quad \int e(t)dt = \sum_{k=0}^{n} e_k T$$

Ecuaţia de control devine în acest caz

$$m_n = K_c\left[T_d\,\frac{e_n - e_{n-1}}{T} + e_n + \frac{1}{T_i}\sum_{k=0}^{n} e_k T\right] \quad (2)$$

Dacă se fac înlocuirile

$$K_p = K_c$$

$$K_i = K_c\,\frac{T}{T_i}$$

$$K_d = K_c\,\frac{T_d}{T}$$

ecuaţia (2) poate fi exprimată ca un algoritm de forma

$$\begin{aligned} s_n &= s_{n-1} + e_n \\ m_n &= K_p e_n + K_i s_n + K_d(e_n - e_{n-1}) \end{aligned} \quad (3)$$

unde s_n este suma erorilor.

```
...
# define KPVAL          1.0
# define KIVAL          0.8
# define KDVAL          0.3
#define FALSE           0
#define TRUE            1
float s, kp, ki, kd, en, enold, mn;
unsigned char stop;
extern float can(void);/*funcţie care achizitioneaza de la convertorul
                        analog numeric, calculeaza si returneaza valoarea
                        erorii e; este dependenta de hardware utilizat.*/
extern void cna(float mn);/*  funcţie care primeste la intrare corectia
                              (valoarea de actionare) si o transmite la
                              convertorul numeric - analogic*/
void task_PID(void){
  …
  stop = FALSE;
  s = 0.0; kp = KPVAL; ki = KIVAL; kd = KDVAL;
  enold = can();
/* bucla de control*/
  while (!stop){
    en = can(); /* adc returnraza valoarea erorii actuale*/
    s = s+en;/* suma pentru integrala*/
    mn = kp*en + ki*s + kd * (en - enold);
    cna(mn);
    enold = en;
  }
  …
```

Cuprins

3. Componentele hard ale unui sistem de calcul in timp real

    3.1 Unitatea centrala de calcul

        3.1.1 Moduri de adresare

        3.1.2 Clase de arhitecturi ale unitatii de calcul

    3.2 Memorii

    3.3 Unitati de intrare/iesire

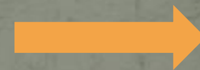
Componentele hard ale unui sistem de calcul in timp real

SCTR: - **componenta hardware**

- componenta software

Cunoasterea (de principiu)a componentelor fizice

utilizarea eficienta a resurselor (hard si soft).

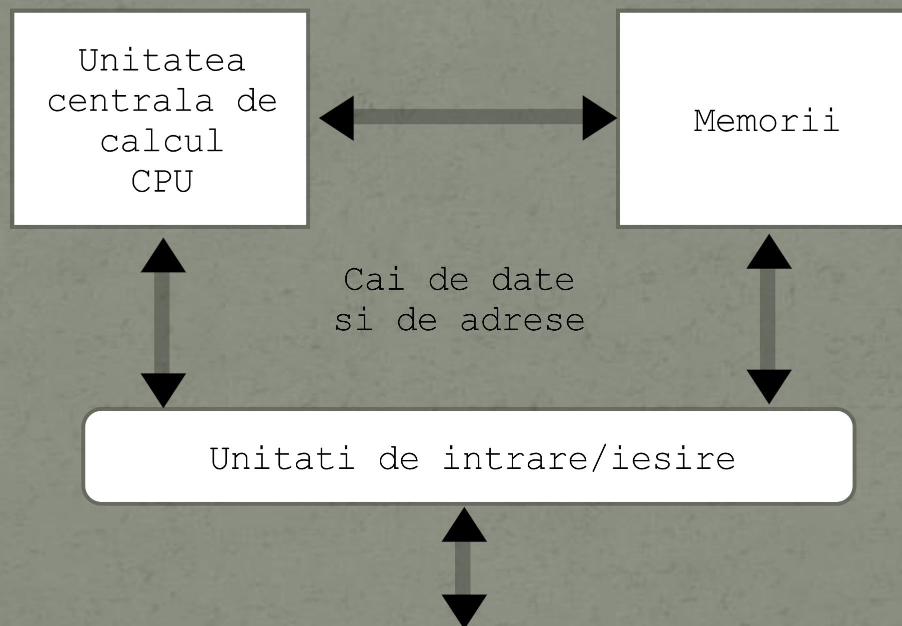


In general lb. de programare si SO moderne face ca programatorul sa fie izolat de interiorul calculatorului. Cele specifice lucrului in timp real presupun o altfel de abordare a relatiei calculator-utilizator.

Arhitectura de baza a unui SC:

- unitatea centrala de calcul (CPU) - procesorul (Central Processing Unit)
- memorii
- unitatile de intrare/iesire

Arhitectura de baza a unui SC



Arhitectura de baza a unui SC

Cai:

- busul de alimentare
- busul de date
- busul de adrese

Busul de alimentare - distribuie diferitele tensiuni necesare functionarii componentelor fizice

Busul de adrese - multime de cai prin care se acceseaza adresele de memorie

Busul de date - vehiculeaza datele intre diferitele componenete fizice

Adresa + Data = **informatie**

busul de date + busul de adrese = **busul de sistem**

Unitatea centrala de calcul

- Unitatea centrala de calcul (CPU)este acea componenta fizica care proceseaza numeric informatiile intr-un calculator.
- Cel mai cunoscut tip de CPU este **microprocesorul -** PC
- **- microcontroler**
- **- DSP - Procesoare numerice de control**
- **- Transputere**

Microcontrolerul

Microcontrolerul este o structura electronica destinata controlului unui proces sau, mai general, este un microcircuit care incorporeaza o unitate centrala (CPU) si o memorie împreuna cu resurse care-i permit interaciunea cu mediul exterior.

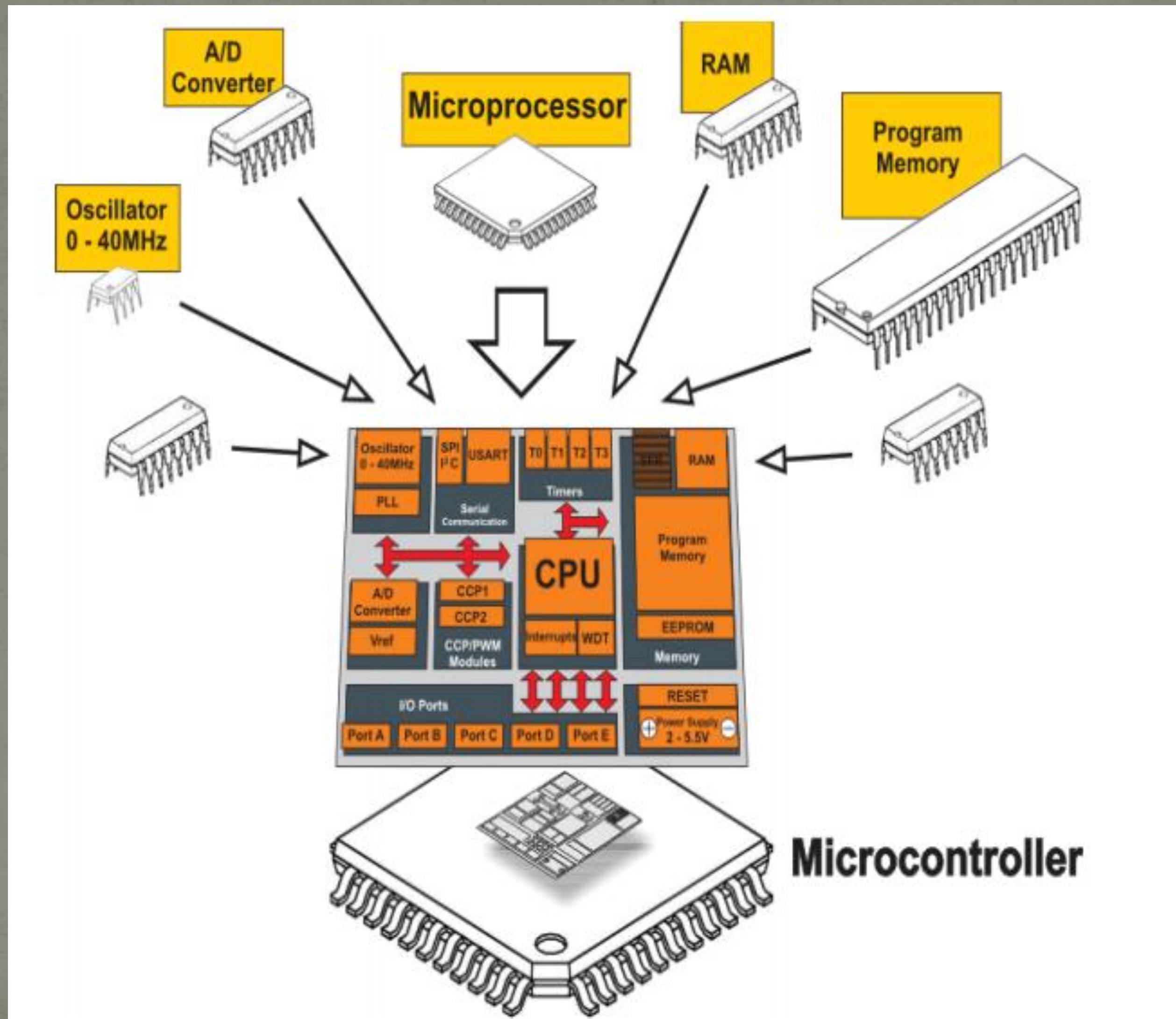
Este un sistem programabil prin microinstructiuni si care recunoaste un set restrans de instructiuni elementare.

Familia 8051 - Intel Corporation

PIC Peripheral Interface Controller - Micro-chip Technology

AVR - Virtual RISC Atmel.

Microcontroler contra microprocesor



Procesoarele numerice de semnal

Procesoarele numerice de semnal, numite în literatura de specialitate DSP-uri (DSP - Digital Signal Processors), sunt sisteme de calcul programabile de tip "single-chip", destinate prelucrării complexe a semnalelor digitale.

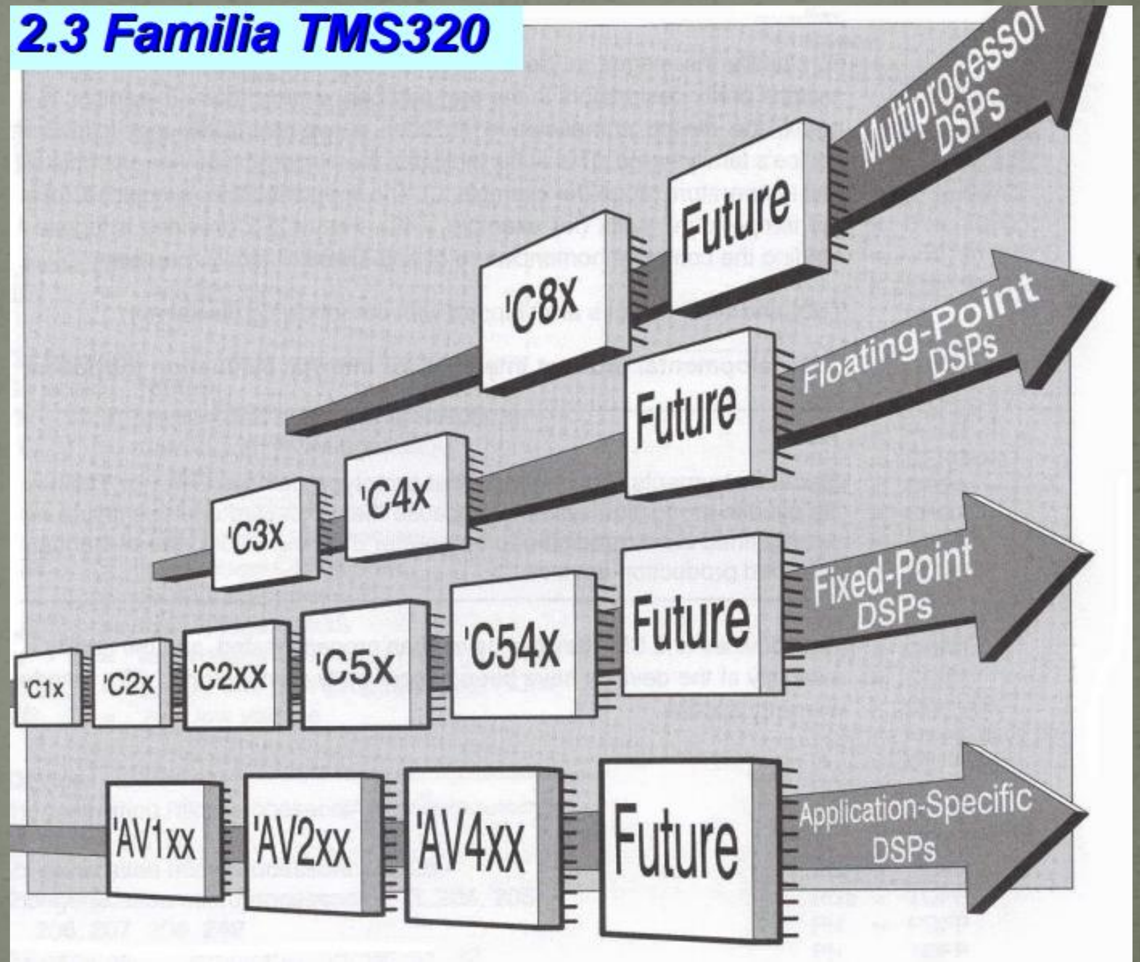
exp: Unitati de conversie CAN/CAN sau componente de prelucrare a seriilor Fourier.

Deşi se numesc procesoare, ele înglobează într-un singur circuit integrat principalele subsisteme componente ale unui sistem de calcul (unitate centrală, subsistem de memorie, subsistem de intrare / ieşire, etc.), realizând funcții complexe de transfer şi de prelucrare a datelor.

În 1982, Texas Instruments a introdus primul procesor DSP, TMS32010, din familia TMS320.

Texas Instruments (57%)

Astăzi, această familie conține atât procesoare în virgulă fixă, cât şi în virgulă mobilă. Procesoarele pe 16 biți în virgulă fixă sunt cuprinse în generațiile: TMS320C2000 (cu C24x şi C28x), C5000 (C54x şi C55x) şi C6000 (C62x şi C64x). Procesoarele pe 32 biți în virgulă mobilă sunt cuprinse în generațiile: C3x, C4x şi C7x.



Transputerele

Este un microcontroler de 16/32 de biti, cu arhitectură de tip RISC (Reduced Instruction Set Computer) proiectat pentru a fi utilizat în unitătile centrale ale sistemelor de calcul cu prelucrare paralelă, orientate spre comunicare eficientă de date.

- -IMS T222 procesor pe 16 biți;
- -IMS T414 şi IMS T425 procesoare pe 32 de biți;
- -IMS M212 controlor de periferice inteligent (procesor pe 16 biți, memorie pe cip şi legături de comunicație, cu logică pentru interfațarea de discuri sau periferice generale);
- -IMS C011 şi IMS C012 adaptoare de legături permițând conectarea legăturilor seriale INMOS la porturi paralele sau magistrale
- -IMS C004 comutator de legături programabil (rețea grilă de comutatoare cu 32 de intrări şi 32 de ieşiri).

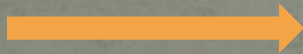
Coprosesoarele

Coprocesoarele sunt unitati de calcul specializate pentru anumite tipuri de operatii si lucreaza intotdeauna cu un alt procesor, fiind subordonat acestuia ca nivel de decizie.

Exp: Un PC poate avea unul sau mai multe procesoare . Plăcile de bază normale permit prezența unui singur procesor, însă sunt producători ce oferă opțiunea de dual processor. Astfel în sistemele produse de Digital, HP se pot întâlni între 2-8 procesoare. Problema este ca numai anumite sisteme de operare ştiu sa folosească multiprocesarea (Linux, SunOs, Unix, WindowsNT). Astfel în Windows 9x prezența unui processor suplimentar nu va influența cu nimic performanța sistemului. Sistemele multiprocesor sunt folosite în servere sau în stații de lucru cu flux mare de date (CAD, GIS, etc). Un alt motiv de a folosi un sistem multiprocesor este securitatea oferită. Astfel în cazul unei defecțiuni produse la unul din procesoare conducerea va fi luată de celălalt.

Unitatea centrala de calcul

- O unitate centrală are un limbaj propriu, care diferă de la o unitate centrală la alta, instrucțiunile unității centrale fiind reprezentate de şiruri de numere binare. Producătorul unității centrale stabileşte tipurile de instrucțiuni, codificarea, structura şi modul de utilizare a acestora. Un program scris în binar cu ajutorul acestor instrucțiuni se numeşte program maşină iar codul în care este scris se numeşte cod obiect (sau cod binar) direct executabil. Programele executabile se afla in memoria interna a calculatorului si sunt incarcate in CPU si executate continuu.



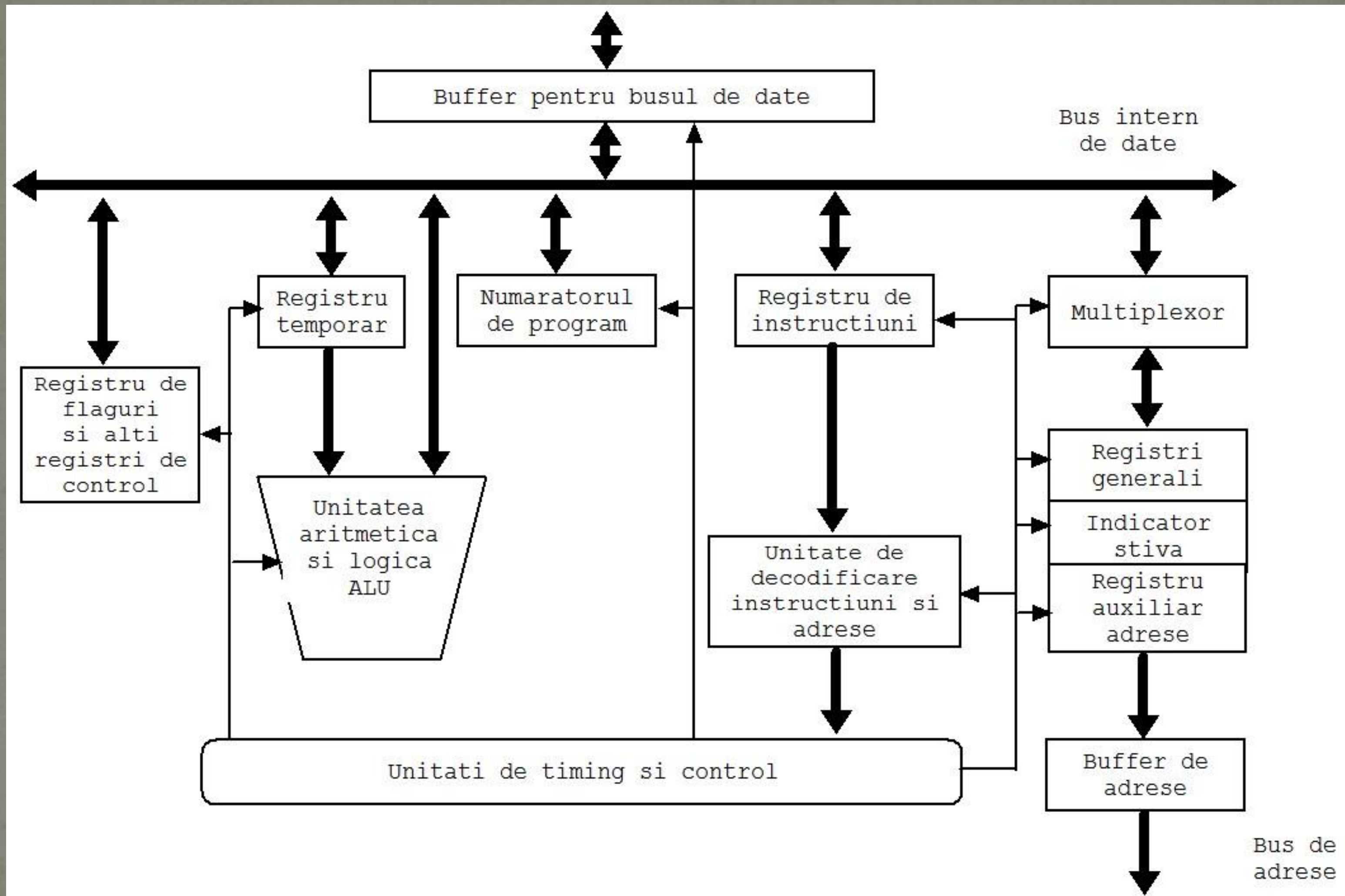
sistem de calcul bazate pe arhitectura von Neumann

Unitatea centrala de calcul

- CPU alcatuit din mai multe componente conectate printr-un bus intern.
- unitatea aritmetica si logica
- unitatea de comanda si control
- registrii

- Registrii - Reprezinta locatii de memorie temporare aflate ın interiorul CPU. Registrele sunt fie dedicate (program counter PC), fie generale. Un procesor are multi registrii. Fiecare registru este un grup de celule de memorie folosite pentru a întreține stocarea temporară de cuvinte (octeti) înăuntrul procesorului.

Unitatea centrala de calcul



Unitatea centrala de calcul

- Rolul CPU este de a prelucra programele primite. Primele programe au fost scrise în cod masina dar evident, scrierea unor astfel de programe este dificilă iar riscul de eroare este ridicat. Pentru simplificare, producătorii unităților centrale asociază codului binar corespunzător unei instrucțiuni, un nume care să fie semnificativ şi care să sugereze acțiunea realizată de instrucțiune. Acest nume poartă denumirea de **mnemonică.** Programarea cu mnemonici este mai uşor de realizat dar este necesar un program de traducere din mnemonici în cod binar. Un astfel de program prevăzut cu o serie de facilități care să uşureze munca programatorului se numeşte **asamblor** iar programele scrise cu ajutorul mnemonicelor, pentru asamblor, se numesc programe în **limbaj de asamblare.**

Unitatea centrala de calcul

Programele scrise în limbaj de asamblare nu pot fi rulate decât pe unitatea centrală pentru care au fost scrise şi din acest motiv se spune că programele scrise în limbaj de asamblare nu sunt portabile. Avantajul utilizării programelor în limbaj de asamblare este reprezentat de faptul că ele permit accesul programatorului la structurile de nivel jos ale SC (ceea ce nu se întâmplă la limbajele de nivel înalt) şi permit scrierea unor programe de dimensiuni mici ce se execută în timp scurt iar uneori astfel de cerințe sunt impuse. Din acest motiv şi limbajele de programare de nivel înalt permit mecanisme de inserare a unor secvențe de program scrise în limbaj de asamblare.

Unitatea centrala de calcul

Programele reprezinta secvente de macroinstructiuni. Secventele sunt stocate in memoria principala a calculatorului si asteapta sa fie executate. Cand incepe executia sa, o macroinstructiune este **incarcata** (fetch) de la adresa data de **registru numarator program** si asezata in **registrul de instructiuni. Unitatea de control** decodifica macroinstructiunea si se executa cu ajutorul **unitatii logice si aritmetice de control** (ALU)si a diversilor de **registri interni.**

Se repeta pt. urmatoarea macroinstructiune.

Operatie ciclica:

incarcare - decodificare - executie

fetch - decode - execute

Instructiuni in lb. de asamblare

- Prin limbajului de asamblare:
  - se face programe mai scurte şi care să lucreze mai repede;
  - se înțelege mai bine cum lucrează calculatoarele;
  - se scrie un cod eficient.

Dar
  - Limbajul de asamblare este puțin răspândit printre nespecialişti
  - este greu de învățat, de citit şi de înțeles, de depanat;
  - Calculatoarele actuale sunt atât de rapide încât nu mai este necesară programarea în limbaj de asamblare;
  - Limbajul de asamblare nu este portabil.

Instructiuni in lb. de asamblare

- Fara operand: NOP
- Cu un operand: PUSH AX
- Cu doi operanzi: MOV AX,BX
- Linie cu eticheta START: MOV AX,BX
- Linie de comentar ;

ADD AX,BX,CX

Instructiuni in lb. de asamblare

- Timpul de executie al unei macroinstructiuni, calculat in numar de cicli de masina depinde:
  - timpul de decodificare a instructiunii
  - lungimea microcodului
  - numarul de incarcari de adrese si date

- Modul de adresare a instructiunilor:
  - Adresare implicita
  - Adresare directa
  - Adresare indirecta
  - Adresare cu registri
  - Adresare indexata cu registru de index

Adresare implicita

- Codul operatiei specifica in mod implicit registri utilizati pentru efectuarea instructiunii
- Se foloseste registrul acumulator, care este registru de destinatie in operatiile aritmetice si logice
- Instrtructiunile cu adresare implicita necesita un singur ciclu masina ca CPU sa efectueze operatia de incarcare (fetch) - cele mai rapide.

ABS ;

calculeaza valoarea absoluta a marimii aflata in registru acumulator

Adresare imediata

- Operandul este o constanta
- Operandul este continut in codul instructiunii
- Operandul este citit odata cu instructiunea

```
LOAD acc,5
```

Se incarca in registrul accumulator valoarea numerica 5. Instructiunea necesita doua cicluri masina pt. ca CPU sa efectueze operatiunea de fetch, un ciclu pt. incarcarea instructiunii si unul pentru incarcarea operandului.

Adresarea directa

- Operandul este explicat printr-o adresa de memorie (adresa unde se afla data respectiva)
- Adresa operandului este continut in codul instructiunii
- Instructiunea poate lucra cu o singura locatie de memorie (octet,cuvant sau dubl-cuvant)
- Se foloseste pt. variabile simple date nestructurate
- Necesita ciclu suplimentar de transfer

  Timp de raspuns mai mare!!

ADD acc,A

Se adauga la continutul acumulatorului valoarea aflata in memorie la adresa A.Instructiunea necesita trei cicli masina pt. operatiunea fetch. Unul pt. incarcarea instructiunii, unul pt. cautarea operandului si unul pt. incarcarea sa.

Adresare indirecta

- Campul operand al instructiunii nu contine nici data necesara nici adresa unde se afla data, ci o adresa unde gasim adresa la care se afla data cautata (operandul).

LOAD acc,A,Ind

Ind - specifica adresarea indirecta

La adresa A procesorul stie ca va gasi o informatie (o adresa) de unde va putea sa incarce in acumulator operandul.

Necesita patru cicli masina pt. operatiunea de fetch.

Mod de adresare cel mai lung.

Adresare cu registri

- Poate o sa fie o adresare imediata, directa sau indirecta
- In campul cod operatie sunt specificate nu adrese ci registri interni ai CPU cu care se va lucra ca locatii de memorie

LOAD R1,5

nr ciclu

Se incarca valoarea 5 in registrul R1.

ADD R3,R1

Aduna la continutul registrului R3 continutul registrului R1.

LOAD acc,R3,Ind

Incarca in acumulator data care se afla la adresa data de R3.

Adresare indexata (cu registru de index)

- Foloseste unul sau mai multi registri speciali, numiti registri de index.
- Continutul acestui registru este o marime ce trebuie indexata la valoarea operandului sau a adresei la care se gaseste operandul.

LOAD *acc,5,RI1

Se incarca in acumulator valoarea 5 indexata cu continutul registrului de index RI1. (adresare imediata)

ADD *acc,A,RI2

Adresa A se indexeaza cu continutul registrului de index RI2 si se obtine astfel adresa de la care se aduce operandul si se aduna cu cel aflat deja in acumulator. (adresare directa)

* simbolizeaza adresarea indexata

Clase de arhitecturi ale unitatii centrale de calcul

In functie de complexitatea modului de adresare utilizat de un anumit tip de procesor:

- arhitecturi de 0-adrese
- arhitecturi de 1-adresa
- arhitecturi de 2-adrese
- arhitecturi de 3-adrese

Arhitecturi de 0-adrese sau arhitecturi de tip stiva

- CPU lucreaza numai cu un registru acumulator si o memorie de tip stiva ambele neadresabile.
- Instructiunile nu au in ele camp de adresa si SC nu are memorii adresabile.
- Modurile de adresare este cel implicit sau cel imediat.
- Se mai poate utiliza instructiuni de tip PUSH sau POP (stiva)

si operatii aritmetice sau logice (stiva)

Se folosesc la procesoare de tip microcontroler, cu aplicatii in -timp real - conducerea proceselor industriale de mica complexitate (masini unelte, brat robotic, instalatii de ridicat(ascensoare)) si in domeniul aparaturii electrocasnice (masini de spalat automate, miniroboti), aparatura audio-video.

Arhitecturi de 0-adrese sau arhitecturi de tip stiva

Avantaje

- Viteza de calcul mare deoarece sunt utilizate doar instructiuni rapide (cu unul sau doi cicli masina pentru operatiunea de fetch)
- Simplitate in utilizarea SC de catre nespecialisti datorita setului mic de instructiuni pe care le are
- Costuri mici ale SC

Dezavantaje

- Imposibilitatea de a efectua operatii complexe, implementarea algotimi de comanda sau conducere sofisticati
- Dificultatea sporita in a programa aceste SC pt operatii matematice simple dar cu mai multi operatori.

Arhitecturi de 0-adrese sau arhitecturi de tip stiva

Programarea unui procesor de 0-adrese. Presupune cunoasterea notiunii de **memorie de tip stiva** si a celei de **arbore binar**.

- Denumirea de **stivă** se datorează modului de operare, similar cu modul de accesare a obiectelor dispuse sub formă de **stivă** într-un depozit.
- Pentru a putea retrage dintr-o **stivă** un obiect, trebuie să luăm mai întâi toate obiectele care se află deasupra acestuia. Similar, într-o **stivă** poate fi accesat doar primul element – **ultimul intrat**. Pentru a accesa un element aflat în **stivă**, trebuiesc mai întâi retrase toate elementele aflate deasupra acestuia.
- **Stiva** este utilizată în mod curent pentru managementul rulării aplicațiilor **software** de către procesor.

  Exp: Dacă la un moment dat într-un program apare o instrucțiune de salt la o funcție ce sortează crescător un şir de numere, atunci, adresa curentă de execuție este memorată în **stivă.Procesorul** face saltul şi execută funcția respectivă după care se întoarce la execuția liniilor de cod conform primei valori memorate în stivă.

Asupra unei **stive** pot fi executate următoarele operații:

- **PUSH** – introducerea unui element în **stivă;**
- **POP** – extragerea unui element din stivă;
- **TOP** – Accesarea elementului din vârf, fără a modifica valoarea acestuia.
- Singurul element care descrie starea unei memorii stiva este **indicatorul de stiva** care arata cate elemente exista la un moment dat in stiva.

- Operatiile matematice si logice pe stiva se definesc in felul urmator:

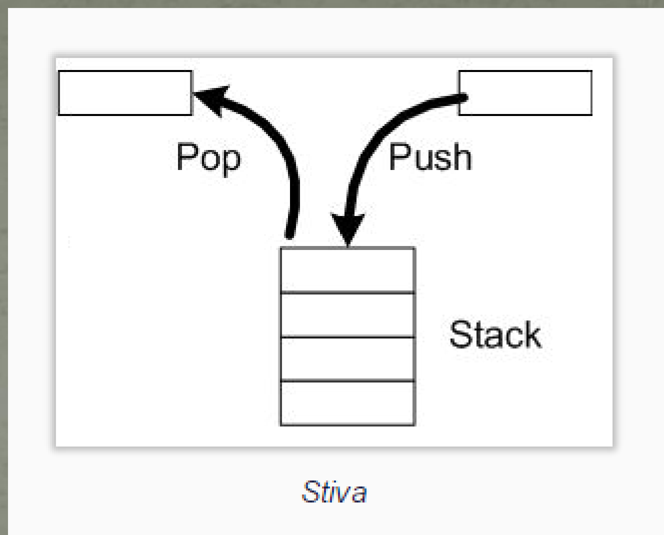
```
op(niv_stiva-1) ← op(niv_stiva-1) # op(niv_stiva)
niv_stiva ← niv_stiva-1
```

op() – reprezinta operandul

- reprezinta operatorul algebric sau logic

niv_stiva – reprezinta valoarea curenta a indicatorului de stiva

In urma unei operatii intre doi operanzi din stiva (cei mai de sus doi) valoarea celor doi operanzi se pierde iar rezultatul ramane in varful stivei (care coboara o unitate).



Stiva

Principiul de funcționare al stivei este "ultimul intrat, primul ieşit" sau **LIFO** (Last In, First Out).

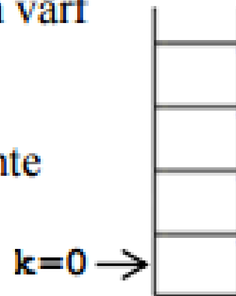
Stiva poate fi implementată static (folosind vectori) atunci când se lucrează cu fluxuri de date precise. Când nu se cunoaşte exact cantitatea şi ritmul informațiilor ce urmează a fi procesate, **stiva** se implementează dinamic (folosind pointeri şi alocări dinamice de memorie) .

Exemplu: Dorim ca in stiva sa fie cel mult 100 de elemente. C++

```
const int DMax=100; //numărul maxim de elemente
int S[DMax+1]; //stiva (vectorul) S
int k,x; //vârful stivei (egal cu numărul elementelor din listă)
         // x elementul din vârf
```

a) Crearea unei stive vide:

```
k=0; // stiva nu are elemente
```

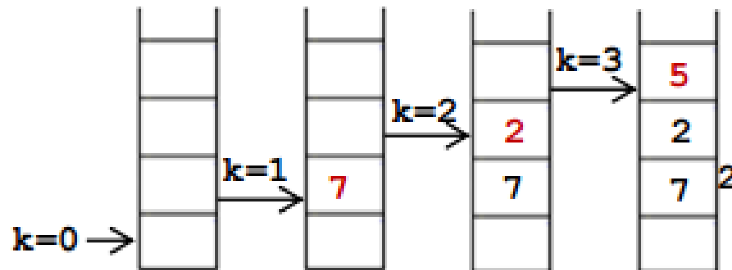


Exemplu: Dorim ca in stiva sa fie cel mult 100 de elemente. C++

b) Inserarea unui element x în vârful stivei, operația **PUSH(x)**:

```
if (k==DMax) //stiva este plină
     cout<<"Eroare: stiva plină!";
else //inserăm x în vârful stivei S
     S[++k] = x;
```

Modificările stivei prin operațiile **PUSH(7), PUSH(2), PUSH(5)** sunt reprezentate în figura următoare:

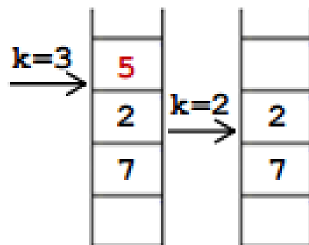


Exemplu: Dorim ca in stiva sa fie cel mult 100 de elemente. C++

c) Extragerea unui element x din vârful stivei, operaţia **POP(x)**:

```
if (k==0) //stiva este vidă
    cout<<"Eroare: stiva vidă!";
else //inserăm x în stiva S
    x = S[k--];
```

Prin extragerea elementului din vârful stivei, de pe nivelul k=3, aceasta revine la nivelul anterior, k=2. Se consideră că elementul de pe nivelul 3 (fostul vârf al stivei) nu mai aparţine stivei.



d) Operaţia **TOP** este o operaţie de informare, care permite consultarea valorii elementului din vârful stivei, fără a modifica valoarea acestuia.

```
x = S[k];
```

Aplicaţie: Verificarea parantezelor

1. O aplicaţie utilă a structurei de tip stivă este algoritmul prin care se verifică dacă o expresie ce conţine paranteze este bine formată. O expresie cu paranteze este un şir de paranteze deschise **'('** şi închise **')'**. Expresia este bine formată dacă:

-începe cu o paranteză deschisă **(**

-fiecare paranteză deschisă **(** are o paranteză închisă corespunzătoare **)**

De exemplu, expresia **(( ) ( ))** este bine formată, iar expresia **(( ) ( )** este incorectă.

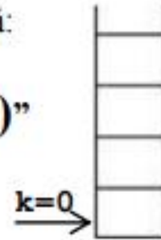
Algoritmul pentru verificarea corectitudinii unei expresii cu paranteze foloseşte o stivă de caractere, în care se memorează doar parantezele deschise (. Paşii algoritmului sunt:

- la citirea unei paranteze deschise, aceasta se adaugă în stivă cu ***PUSH (***
- la citirea unei paranteze închise, verificăm dacă stiva conţine o paranteză deschisă) şi o scoatem din stivă cu ***POP***; dacă apare eroare la operaţia POP atunci expresia este incorectă (avem o paranteză închisă fără corespondent)
- la sfârşitul şirului, stiva trebuie să fie vidă, k=0, dacă expresia este bine formată

În continuare o să urmărim aplicarea algoritmului pentru şirul bine format **(( ) ( )):**

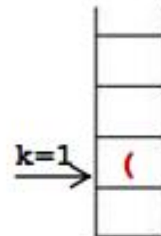
- La început, stiva este vidă:

  Şirul citit este "(( ) ( ))"



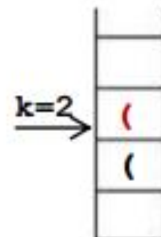
- Primul character din şir (este adăugat în stivă

(( ) ( ))

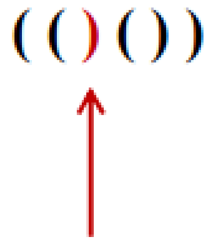


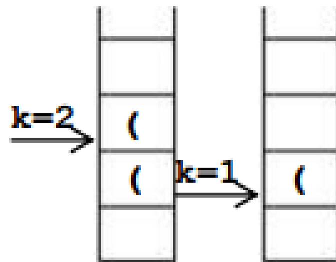
- Al doilea caracter este tot o paranteză deschisă (

(( ) ( ))

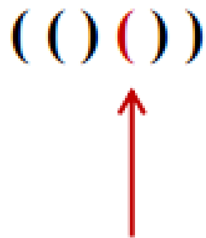


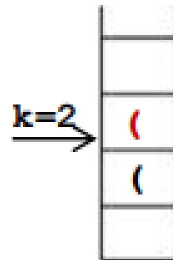
- Al treilea caracter citit este paranteza închisă pereche a ultimei paranteze adăugate, care va fi extrasă din stivă:





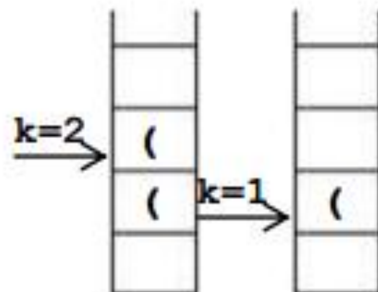
- Urmează al patrulea caracter o nouă paranteză deschisă:





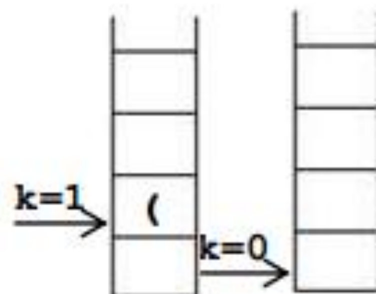
- Al cincilea caracter este paranteza închisă pereche a ultimei paranteze deschise:

(( ) ( ))



- Ultimul caracter este paranteza închisă corespunzătoare primului caracter citit:

(( ) ( ))



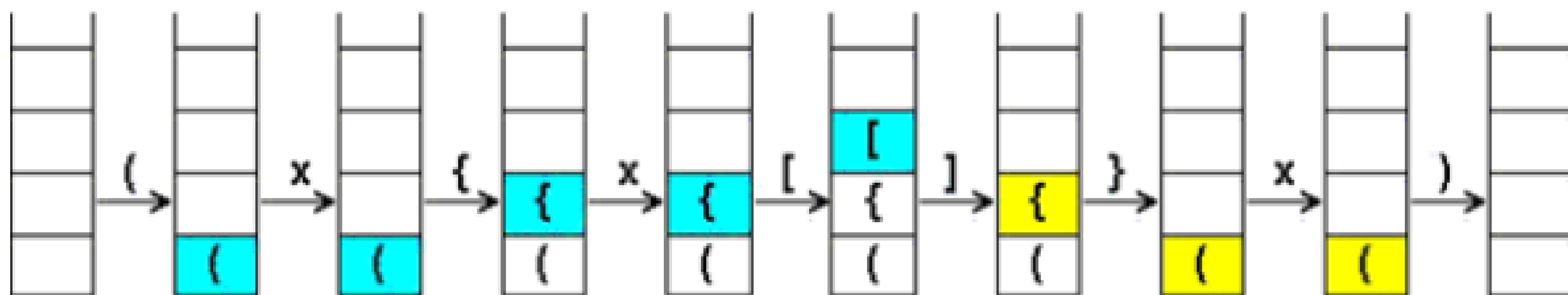
În final, stiva este vidă, ceea ce înseamnă că expresia este corect construită!

2. Algoritmul de mai sus poate fi adaptat pentru expresii formate cu toate tipurile de paranteze folosite la matematică paranteze mici, paranteze drepte şi acolade. Condiţiile de funcţionare a algoritmului sunt aceleaşi, în plus mai trebuie să testăm ca o paranteză închisă să aibă ca pereche în stivă o paranteză de acelaşi tip. Parantezele nu au voie să se intersecteze, de exemplu şirul `( [ ) ]` este incorect format. Spre deosebire de regulile de la matematică într-o expresie cu mai multe tipuri de paranteze este permis ca parantezele drepte sau acoladele să fie incluse între paranteze mici, adică `([]{})` este o expresie corectă.

În acest caz, pașii algoritmului sunt:

- la citirea oricărui tip de paranteză deschisă, aceasta se adaugă în stivă cu ***PUSH***
- orice alt caractere care nu este o paranteză deschisă sau închisă se ignoră
- la citirea unei paranteze închise, verificăm dacă stiva conține o paranteză deschisă de același tip și o scoatem din stivă cu ***POP***; dacă apare eroare la operația POP atunci expresia este incorectă (avem o paranteză închisă fără corespondent)
- la sfârșitul șirului, stiva trebuie să fie vidă, k=0, dacă expresia este bine formată

De exemplu, expresia **`(x{x[]}x)`** este evaluată așa cum apare în imaginea următoare:



I.2. Sortare cu ajutorul a două stive (STL)

Se citește din fișier un șir de numere întregi pozitive, terminat cu -1. Să se sorteze acest șir prin **inserție directă**, cu ajutorul a două stive. În prima stivă se rețin numerele ordonate (descrescător, astfel încât la baza stivei se află cel mai mare). La citirea unui nou număr din fișier (fie acesta *x*) se extrag din această stivă toate numerele mai mici decât cel citit, salvându-le în a doua stivă. Inserăm numărul *x* în stiva ordonată și apoi reintroducem numerele salvate (temporar) din a doua stivă.

I.3. Folosirea stivei în verificarea unui șir dacă este *palindrom* (STL)

Un palindrom este un șir de caractere având aceeași semnificație dacă este citit de la oricare din capete. Scrieți un program care afișează un mesaj corespunzător "**ESTE / NU ESTE Palindrom!**" pentru un șir citit de la tastatură și care se bazează pe o stivă de caractere. Se introduc în stivă caracterele șirului analizat și apoi se extrag unul câte unul (parcurgerea în ordine inversă a șirului). Pentru ca șirul să fie palindrom, caracterele scoase trebuie să coincidă cu cele obținute prin parcurgerea normală a șirului. Extindeți problema și verificați care dintre liniile unui fișier conțin șiruri palindrom.

I.4. Conversia între baze de numerație (STL)

Pentru afișarea unui număr întreg *n* în altă bază *B* se poate folosi o stivă în care se pun resturile parțiale ale împărțirii repetate a numărului cu B. Scrieți un program care citește un număr întreg pozitiv și o bază $2 \le B \le 9$ și afișează reprezentarea numărului în baza B. Definiți o funcție pentru afișarea rezultatului astfel încât B să poată fi ales între 2 și 16. Scrieți o funcție recursivă pentru afișarea unui număr întreg zecimal *n* în baza *B*.

```cpp
#include    <iostream>

using namespace std;

bool numarPalindrom(int numar) // Functia returneaza doar true sau false - pentru ca nu avem nevoie de alte val
{
    int numarInitial, numarInvers = 0; // Creeam doua variabile pentru a salva numarul invers si numarul initial
    numarInitial = numar; // Salvam numarul initial
    while(numar)
    { // Descompunem numarul nostru
        int c = numar % 10; // Obtinem ultima cifre din numar
        numarInvers = numarInvers * 10 + c; // Construim numarul invers
        numar = numar / 10; // Taiem ultima cifra
    }
    if(numarInitial == numarInvers) // Daca numarul initial este egal cu cel invers
        return true;
    else
        return false;
}

int main()
{
    int nr;
    cin >> nr;
    if(numarPalindrom(nr) == true)
        cout << "Numarul este palindrom";
    else
        cout << "Numarul NU este palindrom";
    return 0;
}
```

1. Să se calculeze rezistența echivalentă, obținută prin legarea în serie și în paralel a mai multor rezistențe.
Configurația este descrisă postfixat sub forma: R_1R_2 **operație**.

Soluție:

```
#include "stiva.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){
   Stiva S;
   S = S_New(20);
   double *pR1, *pR2, *pR;
   char descr[20];
   int i;
   gets(descr);
   for(i=0; i<strlen(descr); i++)
       if(descr[i]=='R'){
          pR = (double*)malloc(sizeof(double));
           scanf("%lf", pR);
           Push(S, pR);
       }
```

```
    else
    {  pR1 = Pop(S);
       pR2 = Pop(S);
       pR = (double*)malloc(sizeof(double));
       if(descr[i]=='S')
          *pR = *pR1 + *pR2;
       else
         *pR=*pR1**pR2/(*pR1+*pR2);
       Push(S, pR);
       free(pR1);
       free(pR2);
    }
pR = Top(S);


   printf("rezistenta echivalenta = %6.2lf\n", *pR);
   pR = Pop(S);
   free(pR);
   if(!S_Empty(S))
      printf("descriere incorecta\n");
   return 0;
}
```

- Tatal Tantei are 5 fete: 1. Chacha 2. Cheche 3. Chichi 4. Chocho Intrebare: Care este numele celei de-a cincea? Gandeste repede...