



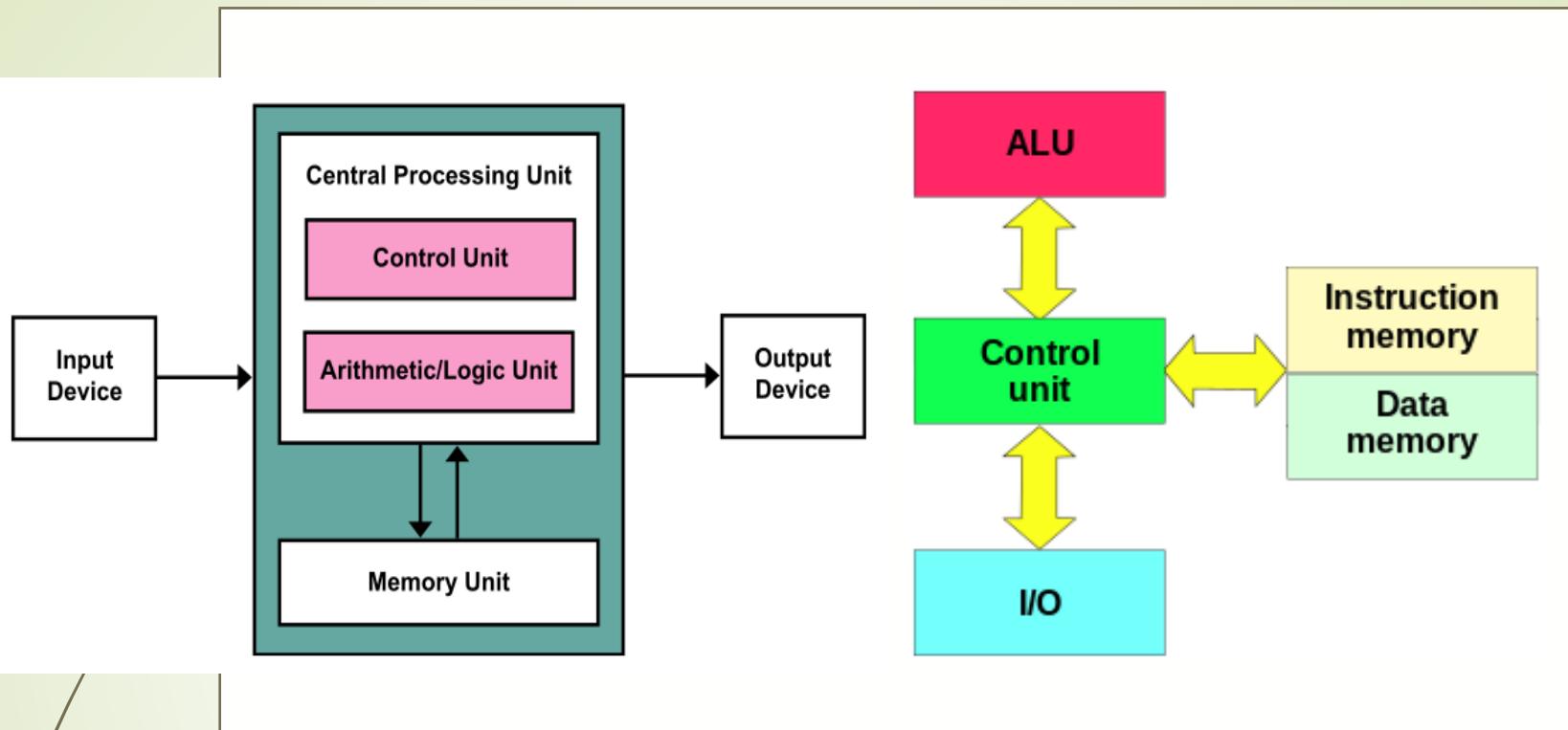
# **Arhitecturi si Prelucrari Paralele**

## **APP**

**Nicolae Tapus**

# Arhitectura Von Neumann

- ▶ Arhitectura a fost proiectată de matematicianul și fizicianul renumit John Von Neumann în 1945.
- ▶ Arhitectura Von Neumann este un model teoretic al calculatorului bazat pe conceptul de program stocat unde programele și datele sunt stocate în aceeași memorie.
- ▶ Conceptul Von Neumann a dominat arhitecturile calculatoarelor.
- ▶ Acest concept a stat la baza sistemelor secentiale, monoprocesor.
- ▶ Reamintim ca o structura Von Neumann este caracterizata de:
  - ▶ un singur element de procesare CPU care contine: unitatea de aritmetică și logică (ALU), unitatea de comandă (CU) și registrele generale;
  - ▶ memoria este formata din locatii de dimensiune fixa cu organizare liniara si adresabila pe un singur nivel;
  - ▶ memoria principală utilizată pentru a stoca datele și instrucțiunile programului
  - ▶ reprezentarea internă a datelor și instrucțiunilor se face sub aceeași forma (binara).
  - ▶ executa operațiile elementare în mod secvential;
  - ▶ limbajul masinii este în general de nivel scazut avand instructiuni ce controleaza operatii simple si actioneaza asupra unor operanzi elementari;
  - ▶ capabilitatile de intrare iesire sunt reduse;



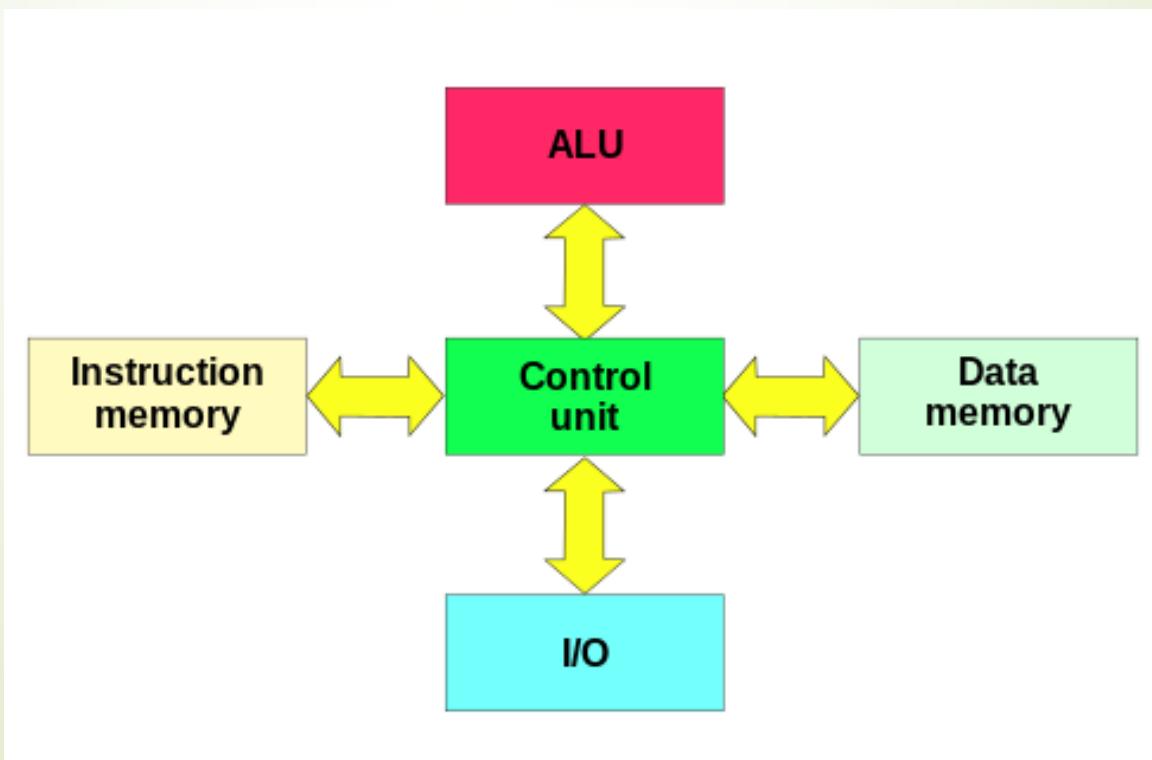
# Arhitectura Von Newmann

# Arhitectura Harvard

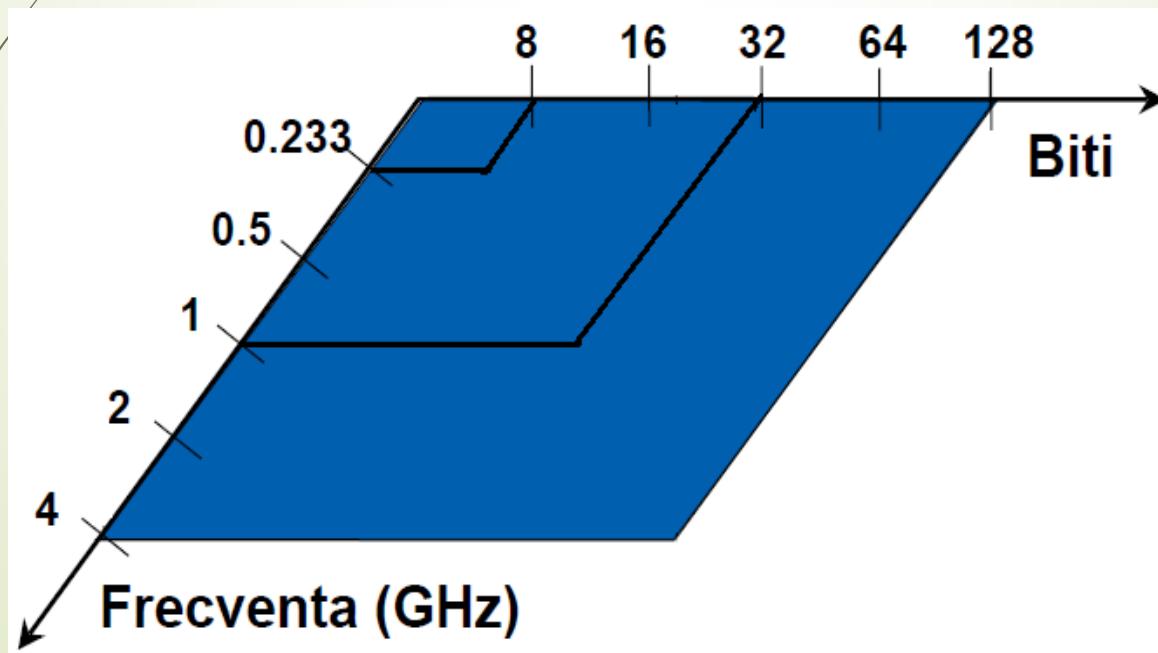
- ▶ Este o arhitectură a calculatoarelor cu stocare separată fizic și căi de semnal pentru datele și instrucțiunile de program.
- ▶ Spre deosebire de arhitectura Von Neumann care utilizează o singura magistrală atât pentru preluarea instrucțiunilor din memorie, cât și pentru transferul datelor, arhitectura Harvard are un spațiu de memorie separat pentru date și instrucțiuni.

# Arhitectura Harvard

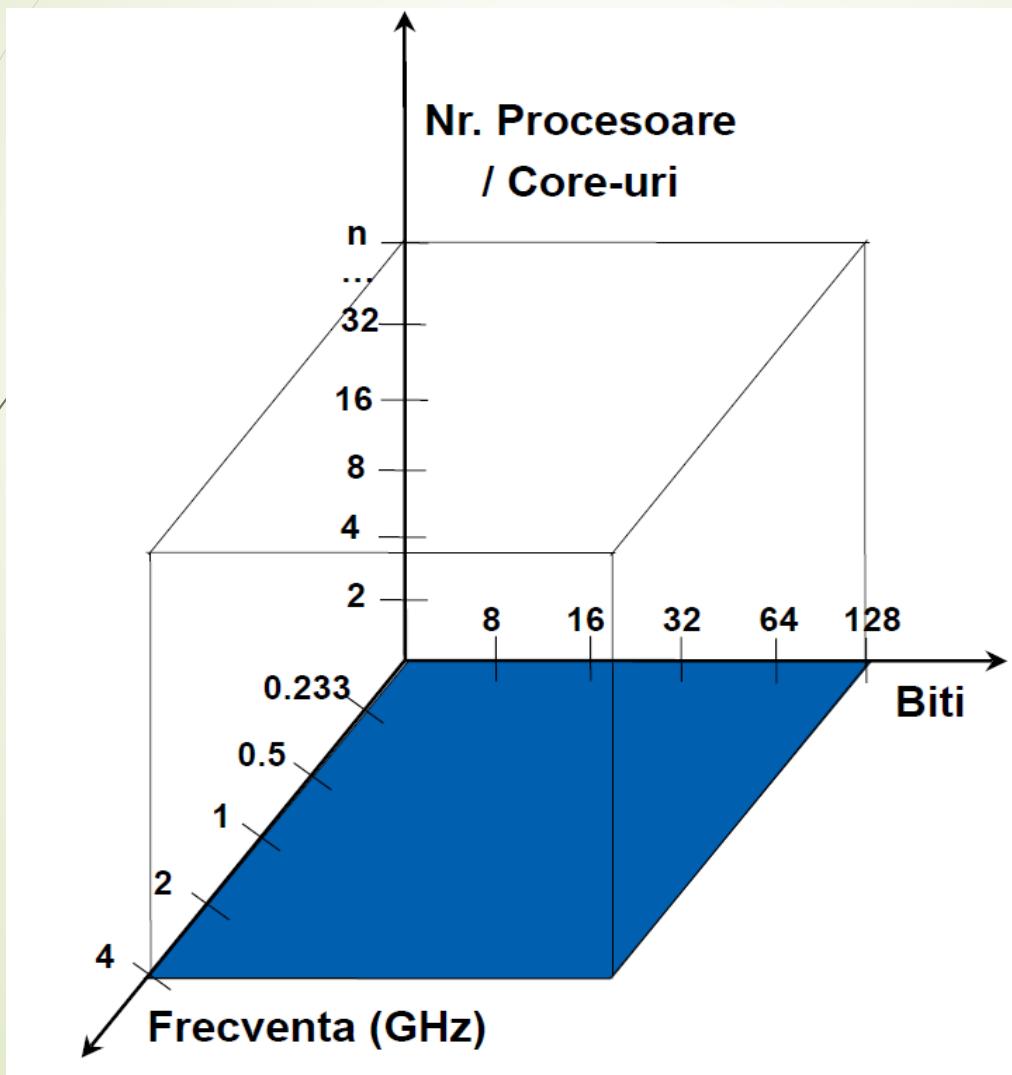
- Arhitectura modificată Harvard este utilizata în mod obișnuit în microcontrolere și DSP (Digital Signal Processing).
- În arhitectura Harvard, unitatea de procesare poate finaliza o instrucțiune într-un singur ciclu dacă există strategii de implementare adecvate.
- <https://ro.sawakinome.com/articles/technology/unassigned-2474.html>



# Evaluare primara a puterii de calcul

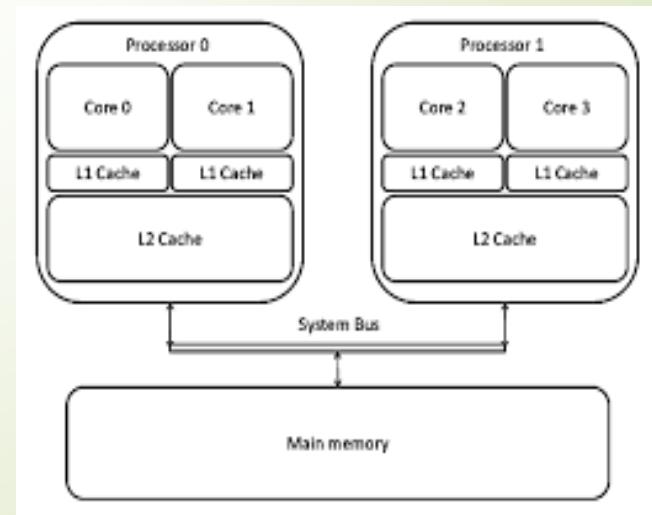
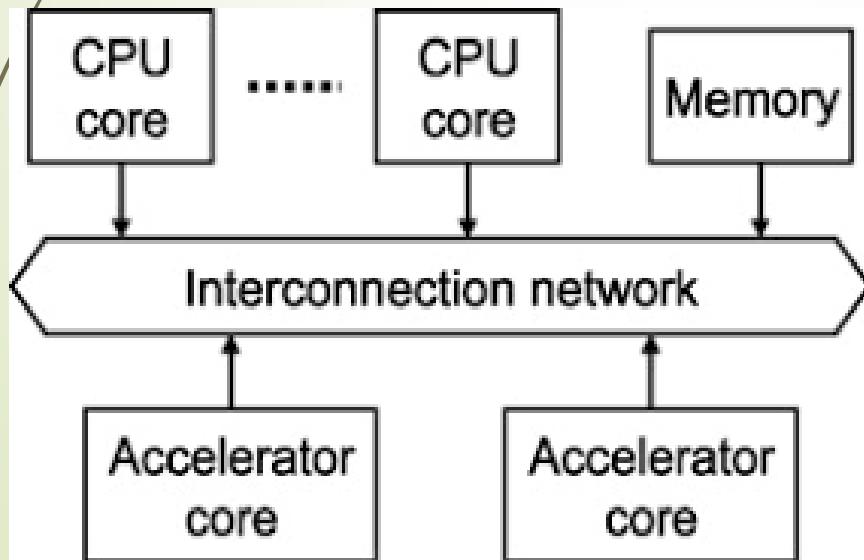
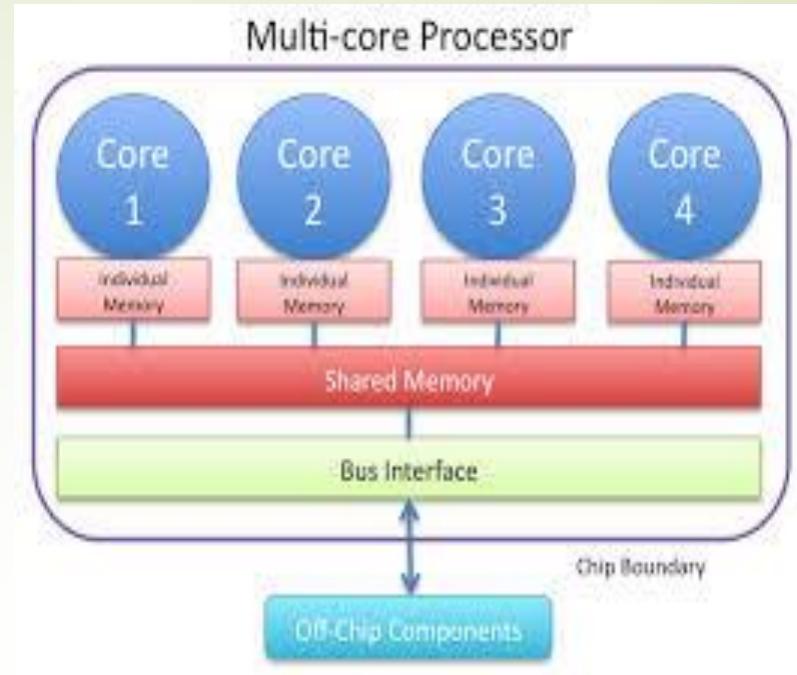


# Evaluare putere de calcul primara system multiprocessor

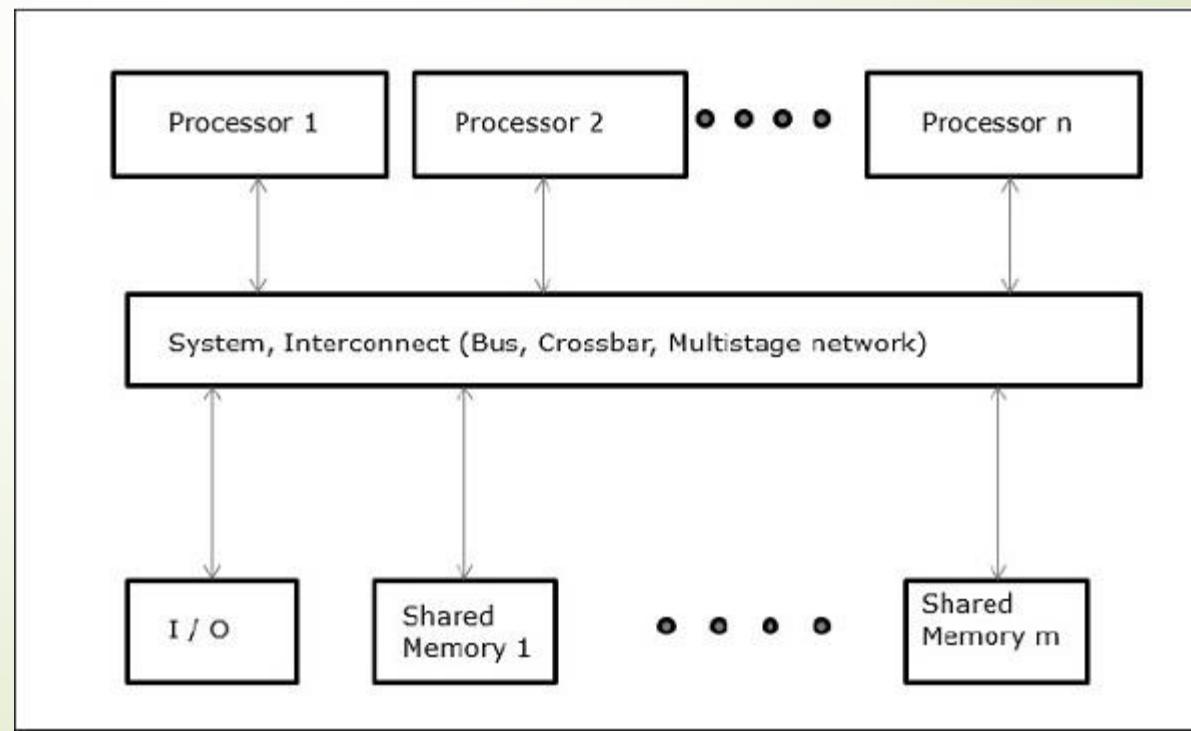
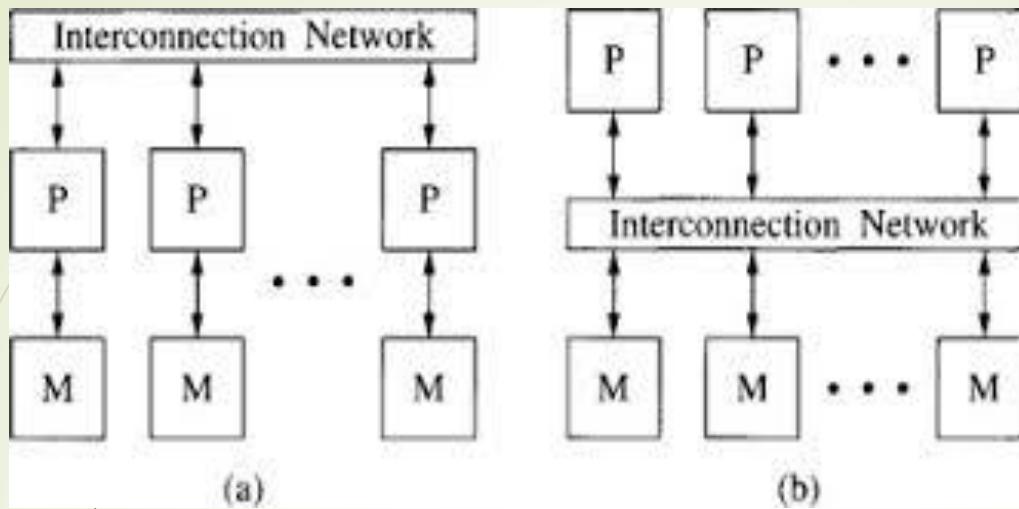




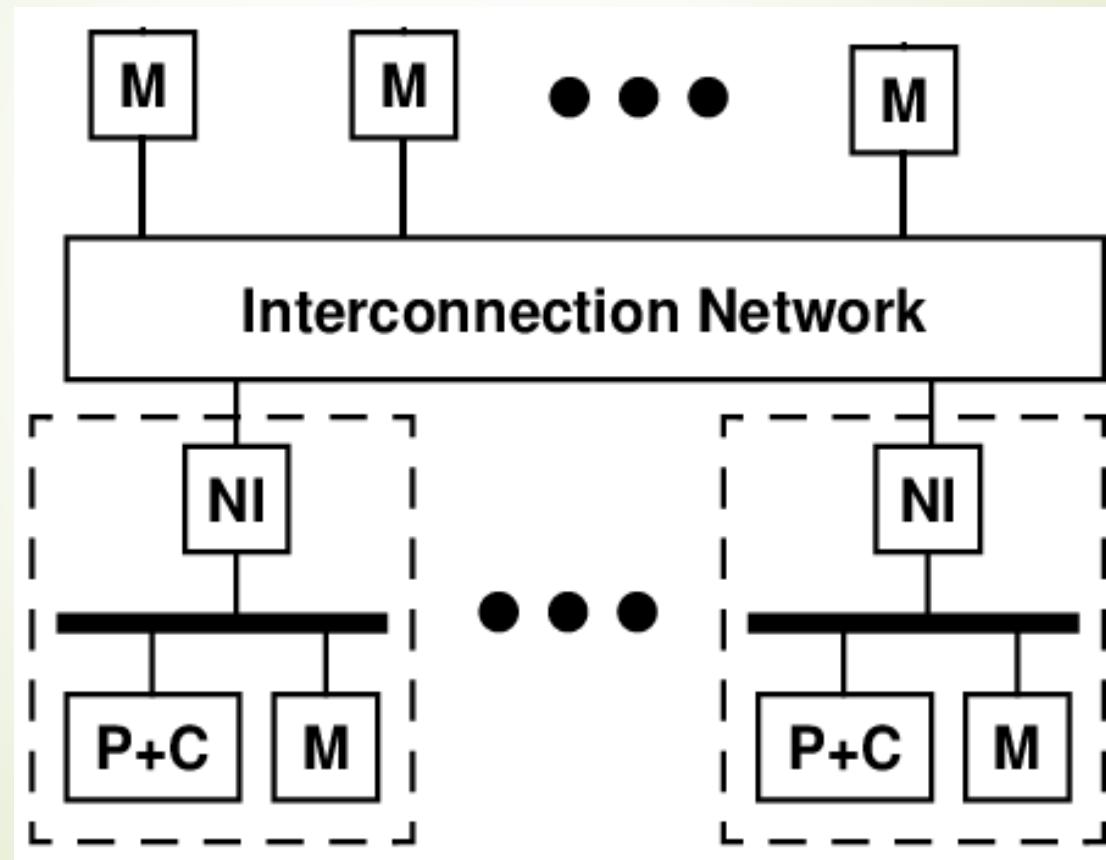
# Structuri multicore



# Sisteme UMA

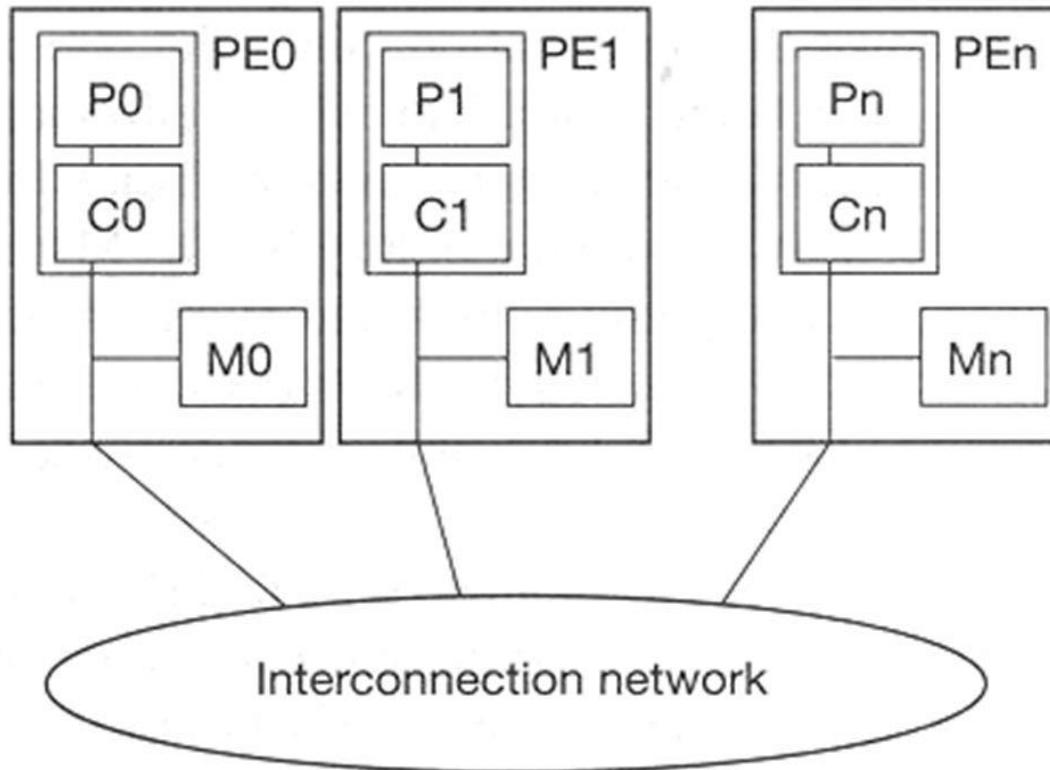


# Sisteme NUMA

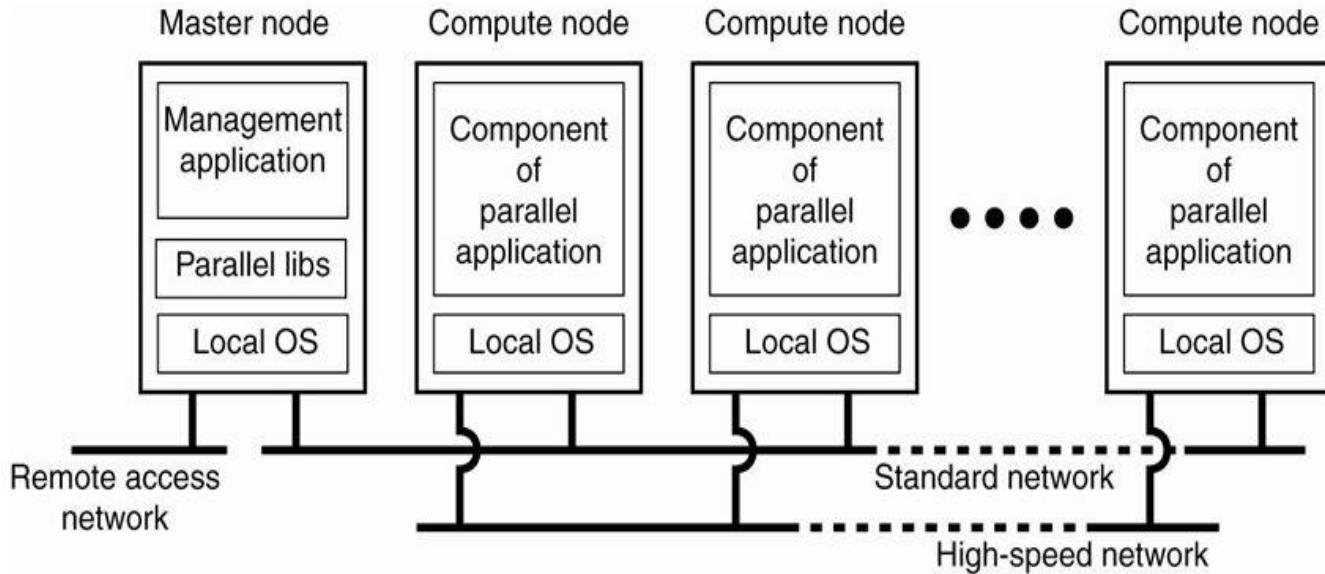


# CC NUMA

## Structure of CC-NUMA Architectures



# Cluster Computing Systems



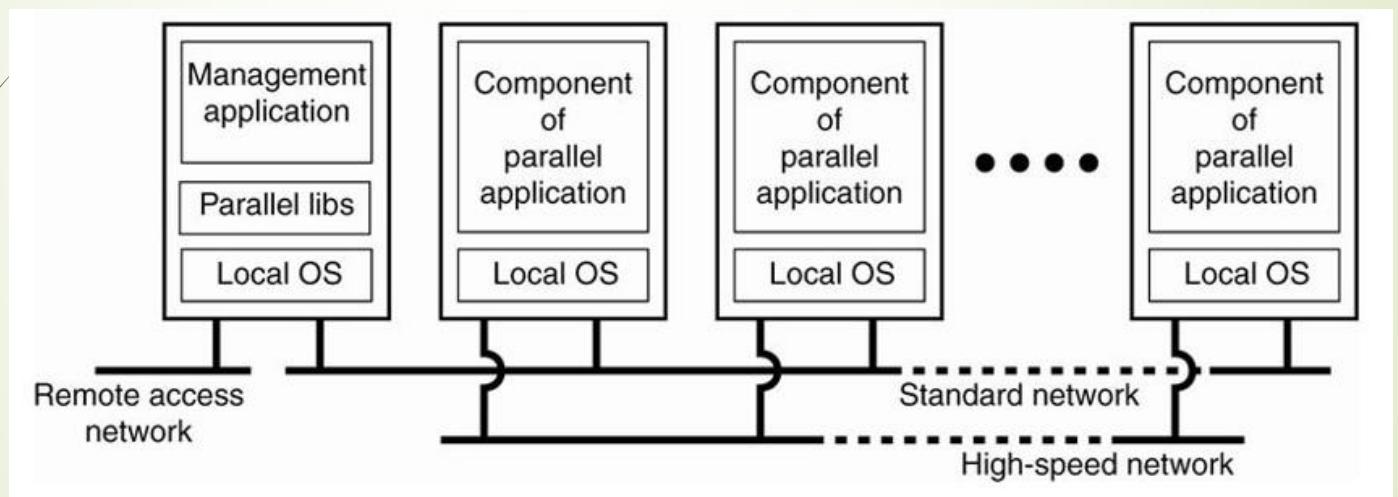
An example of a cluster computing system.

# Cluster Computing



# Tunnel Vision by Experts

- “I think there is a world market for maybe five computers.”
  - Thomas Watson, chairman of IBM, 1943.
- “There is no reason for any individual to have a computer in their home”
  - Ken Olson, president and founder of digital equipment corporation, 1977.
- “640K [of memory] ought to be enough for anybody.”
  - Bill Gates, chairman of Microsoft, 1981.



# Posibilitati executie sarcini

1

2

3

4

5

6

7

8

Paralel

1

2

3

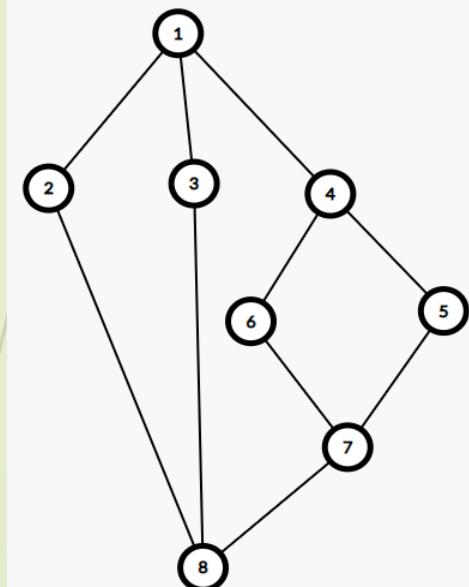
4

5

6

7

8



Dependenta de date intre Sarcini

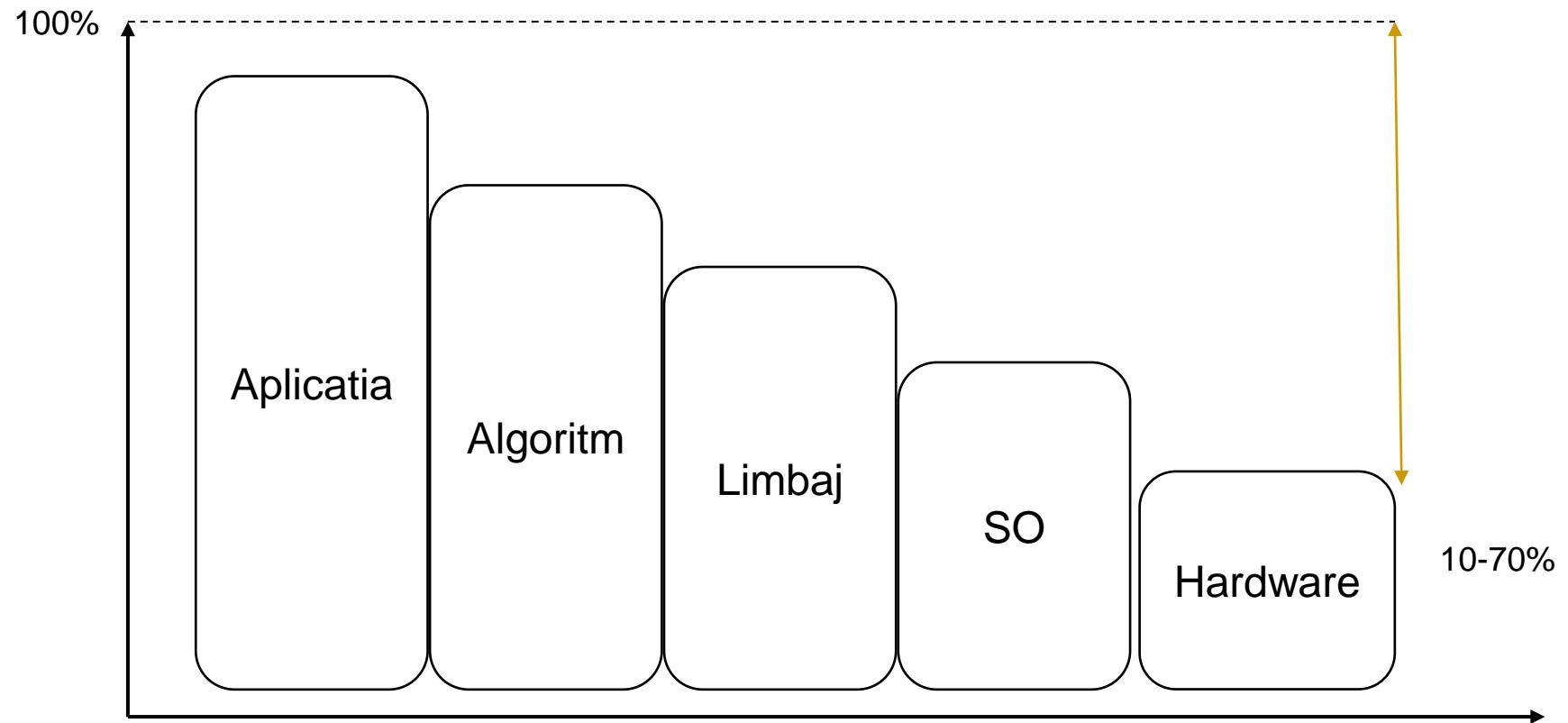
Serial

# Problematici asociate calculului paralel

# Aspecte ale prelucrarii paralele

- Problemele de baza in calculul parallel se refera la urmatoarele aspecte:
- Stabilirea granularitatii task-urilor
- Elaborarea algoritmilor cu paralelism intrinsec
- Proiectarea limbajelor de programare
- Proiectarea unor arhitecturi hardware adecvate
- Proiectarea unor sisteme de operare adecvate

# Grad de paralelism



# Nivelurile de paralelism

- Paralelismul poate fi examinat la multe niveluri în funcție de complexitatea dorită. Frecvent gasim descrierea paralelismului la nivelurile:

- \* job  $\text{JOB} = \{ \text{JOB}_1, \dots, \text{JOB}_i, \dots, \text{JOB}_m \}$
- \* task  $\text{JOB}_i = \{ \text{Ti}_1, \dots, \text{Ti}_j, \dots, \text{Ti}_n \}$
- \* proces  $\text{Ti}_j = \{ \text{Pi}_{j1}, \dots, \text{Pi}_{jk}, \dots, \text{Pi}_{jp} \}$
- \* thread  $\text{Pi}_{jk} = \{ \text{THi}_{jk1}, \dots, \text{THi}_{jkt} \}$
- \* variabilă
- \* instrucțiune
- \* bit

# Nivelurile de paralelism

## ***Nivel de job-uri***

Se executa doua sau mai multe programe independente pe resurse de procesare distincte.

- $\text{JOB} = \{ \text{JOB}_1, \dots, \text{JOB}_i, \dots, \text{JOB}_m \}$

## ***Nivel de task-uri***

- \*  $\text{JOB}_i = \{ T_{i1}, \dots, T_{ij}, \dots, T_{in} \}$

Fiecare  $T_i$  se executa pe cate un procesor distinct insa exista o relatie intre  $T_i$  si  $T_j$  in ceea ce priveste transferul de date sau completarea functiilor de prelucrare realizate in cadrul job-ului.

Exemplu:

Consideram un robot care are mai multe grade de libertate. Pentru fiecare grad de libertate exista un procesor. Programul de conducere a robotului este partitionat in taskuri care se ocupa de cite un grad de libertate al robotului.

Taskurile se executa in paralel, dar interactioneaza pentru miscarea robotului.

# Nivel de proces

- \*  $T_{ij} = \{P_{ij1}, \dots, P_{ijk}, \dots, P_{ijp}\}$

Task-urile sunt alcătuite din mai multe procese care în general sunt identificate de utilizator sau de către compilator dacă acesta este destinat pentru structuri multiprocesor.

Să considerăm task-ul

for i=1 to n

$$x(i) = x(i-2) + y(i-2)$$

$$y(i) = x(i-2) * y(i-1)$$

$$\text{suma}(i) = x(i) + y(i)$$

if  $s(i) > a$  then  $c=c+1$

else  $d=d+1$

end for

In cadrul acestui task identificam două procese:

P1(i)

$$x(i) = x(i-2) + y(i-2)$$

$$y(i) = x(i-2) * y(i-1)$$

P2(i)

$$\text{suma}(i) = x(i) + y(i)$$

if  $s(i) > a$  then  $c=c+1$

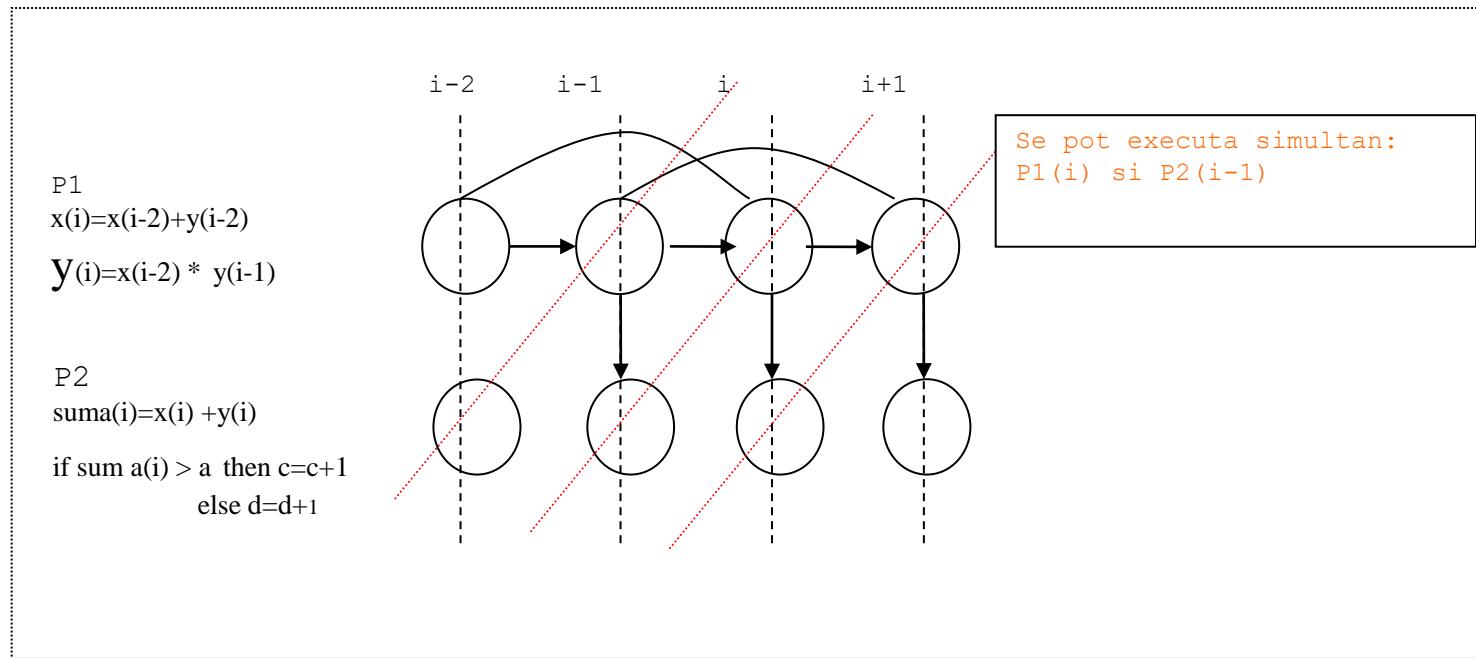
else  $d=d+1$

Intre P1(i) si P2(i) există o dependență de date, deci nu pot fi efectuate în paralel

Insa P1(i) si P2(i-1) pot fi efectuate in paralel

# Nivel de proces

- Interdependenta intre aceste procese este urmatoarea



# Nivel de variabila

$P_i = \{ I_{i1}, \dots, I_{ik} \}$

Fiecare proces este format dintr-un multime de instructiuni care pot calcula variabilele de iesire in functie de variabilele de intrare.

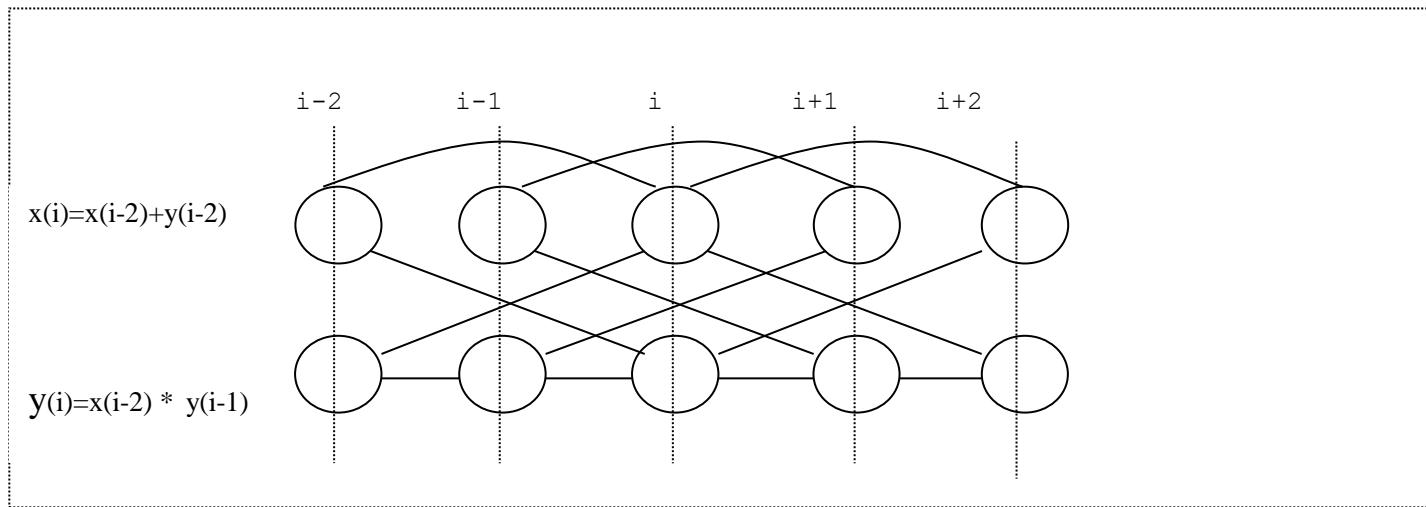
Paralelismul in cadrul unui proces se poate realiza prin calculul simultan a mai multe variabile de iesire.

In exemplul anterior putem considera ca in cadrul procesului P1 variabilele  $x(i)$  si  $y(i)$  se pot calcula in paralel

sau

$x(i)$  si  $y(i+1)$  se pot calcula in paralel

$$\begin{aligned} P_1 \\ x(i) &= x(i-2) + y(i-2) \\ y(i) &= x(i-2) * y(i-1) \end{aligned}$$



# Nivel de Instructiune

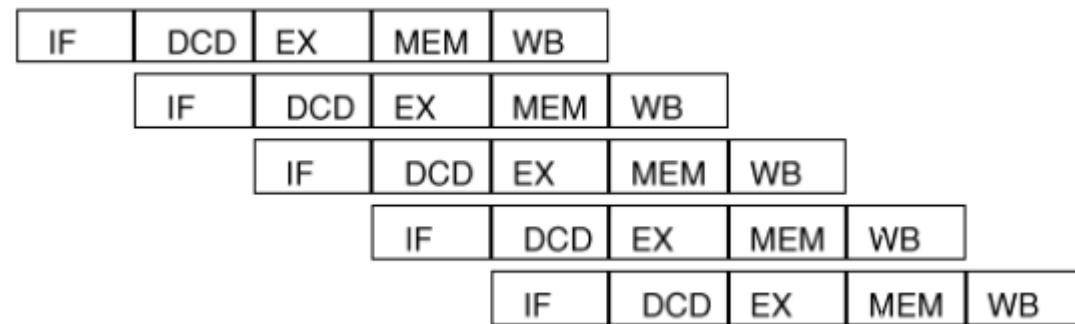
Generarea codului, pentru calculul expresiei  $z = x + y$ , in structura pipeline

Iw r1, x            IF ID EX MEM WB

Iw r2, y            IF ID EX MEM WB

add r3, r1, r2      IF ID stall EX MEM WB

sw z, r3            IF stall ID EX MEM WB



## Exemplu

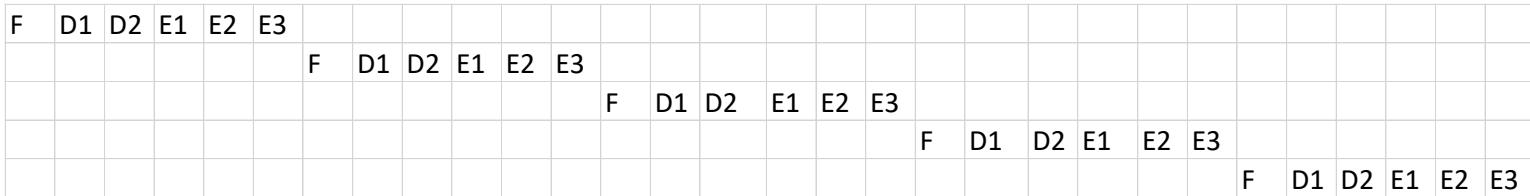
- Să luăm în considerare un procesor care implementează o structură paralelă pipeline de citire-interpretare-execuție pentru procesare suprascală.
- Arătați îmbunătățirea performanței față de procesarea scalară cu pipeline și procesarea fără pipeline, presupunând un ciclu de instrucțiuni care:
  - • citire care necesită o singura perioadă de ceas
  - • decodarea instrucțiunii necesită două perioade de ceas
  - • executia instrucțiunii necesită trei perioade de ceas
- și avem o secvență de 200 de instrucțiuni:

# Exemplu solutie

- O structura fara pipeline necesita 1200 de cicluri de ceas: ( $n \times 6$ )

$$200 * (1 + 2 + 3) = 1200$$

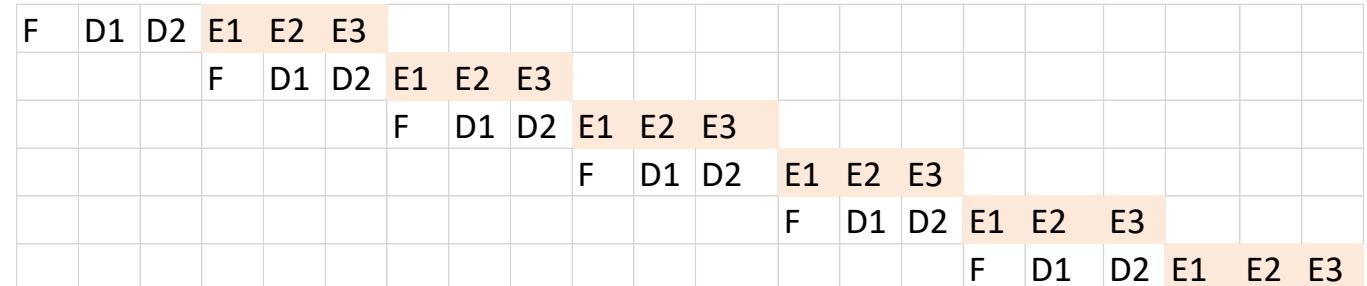
$$n * (1 + 2 + 3)$$



- O structura pipeline scalară necesita 603 cicluri de ceas:  $((n-1) * 3 + 6)$

$$1 + 2 + (200 * 3) = 603$$

$$1 + 2 + (n * 3)$$



# pipeline superscalară cu două unități

- O structura pipeline superscalară cu două unități paralele ar necesita  $303$  cicluri de ceas ( $n / 2 + 1) * 3$ )

$$1 + 2 + ((200/2) * 3) = 303$$

$$1 + 2 + ((n / 2) * 3)$$

F	D1	D2	E1	E2	E3						
F	D1	D2	E1	E2	E3						
	F	D1	D2	E1	E2	E3					
	F	D1	D2	E1	E2	E3					
		F	D1	D2	E1	E2	E3				
		F	D1	D2	E1	E2	E3				
			F	D1	D2	E1	E2	E3			
			F	D1	D2	E1	E2	E3			
				F	D1	D2	E1	E2	E3		
				F	D1	D2	E1	E2	E3		
					F	D1	D2	E1	E2	E3	
					F	D1	D2	E1	E2	E3	

Sau daca fetch-ul se realizeaza secvential

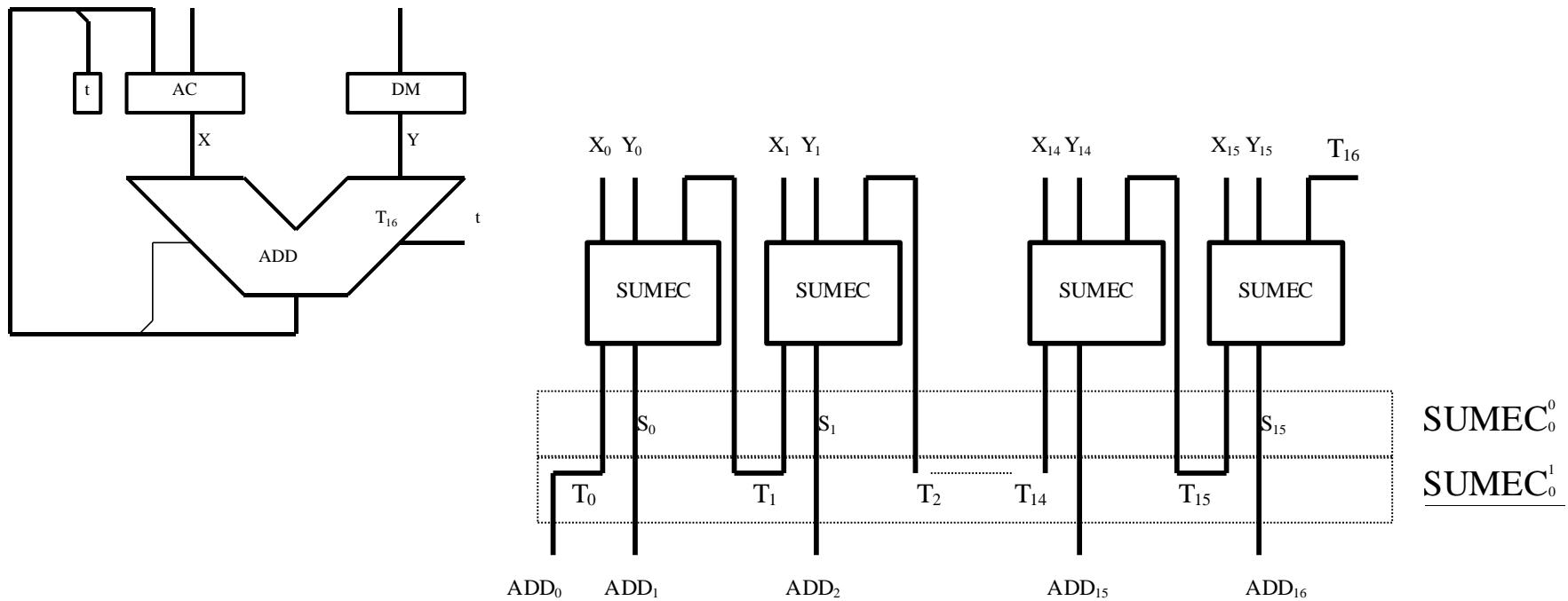
$$1+2+((200/2)*4= 403$$

$$1 + 2 + ((n / 2) * 4)$$

F	D1	D2	E1	E2	E3						
F	D1	D2	E1	E2	E3						
	F	D1	D2	E1	E2	E3					
	F	D1	D2	E1	E2	E3					
		F	D1	D2	E1	E2	E3				
		F	D1	D2	E1	E2	E3				
			F	D1	D2	E1	E2	E3			
			F	D1	D2	E1	E2	E3			
				F	D1	D2	E1	E2	E3		
				F	D1	D2	E1	E2	E3		
					F	D1	D2	E1	E2	E3	
					F	D1	D2	E1	E2	E3	

# Nivel bit

Toate calculatoarele, cu foarte putine exceptii, utilizeaza unitati aritmetice paralele cu sau fara anticiparea transportului si cu structura pipeline.



# Ce este performanța?

- În calcul, performanța este definită de 2 factori
  - Cerințe de calcul (ce trebuie făcut)
  - Resurse de calcul (cat costă să o faci)
- Problemele de calcul se traduc în cerințe
- Resurse de calcul interacțiune și compromise
- Performanța în sine este o măsură a cât de bine cerințele de calcul pot fi satisfăcute
- Evaluăm performanța pentru a înțelege relațiile dintre cerințe și resurse
- Decideți cum să schimbați „soluțiile” pentru a atinge obiectivele
- Măsurile de performanță reflectă deciziile despre cum și cât de bine „soluțiile” sunt capabile să satisfacă cerințe de calcul

# problemele de performanță

Aici ne preocupă problemele de performanță când folosind un mediu de calcul paralel

- Performanță în raport cu calculul paralel
- Performanța este rațiunea de a fi pentru paralelism

Performanță paralelă versus performanță secvențială

Dacă „performanța” nu este mai bună, paralelismul nu este necesar

Prelucrarea în paralel include tehnici și tehnologii necesare pentru a calcula în paralel

- Hardware, rețele, sisteme de operare, biblioteci paralele,
- Limbajele de programare, compilatoare, algoritmi, instrumente, ...

Paralelismul trebuie să ofere performanță

Cum? Cât de bine?

# Așteptarea performanței

- Dacă fiecare procesor este evaluat la  $k$  MFLOPS și există  $p$  procesoare, ar trebui să vedem performanța  $k * p$  MFLOPS?
  - Dacă durează 100 de secunde la un procesor, nu ar trebui să dureze 10 secunde pe 10 procesoare?
- Mai multi factori afectează performanța
- Fiecare trebuie înțelesi separat
- Dar ei interacționează între ei în moduri complexe
  - Soluția la o problemă poate crea alta
  - O problemă poate masca alta problema, etc.
- Scalarea (sistem, dimensiunea problemei) poate schimba condițiile
- Trebuie să înțelegeți limitele de performanță

# Calcule paralele

- Un calcul paralel “jenant” este unul care poate fi evident împărțit în task-uri independente complet care pot fi executate simultan
  - Într-un calcul paralel cu **adevărat jenant**, nu există interacțiunea între procese separate
  - Într-un calcul paralel **aproape jenant** rezultă calcule ce trebuie distribuite și colectate / combinate într-un fel
- Calculele paralel jenant au potențial pentru a atinge viteza maximă pe platforme paralele
  - Dacă este nevoie de timp  $T$  secvențial, există potențialul de a realiza timpul  $T / P$  care rulează în paralel cu  $P$  procesoare
  - De ce acest lucru nu este întotdeauna adevarat?

# Relatia intre algoritmi paraleli si arhitecturi paralele

- Prelucrarea paralela include atat algoritmi paraleli cat si arhitecturi paralele.
- Un algoritm paralel poate fi considerat ca o colectie de procese independente care se executa simultan, procesele comunicand in timpul executiei.
- Astfel un algoritm se executa pe unitati functionale hardware care in general constau din
  - elemente de procesare;
  - module de transfer date.
- Ceea ce intereseaza este cum se transpun procesele pe unitatile functionale hardware.

# Algoritmi paraleli / Arhitecturi paralele

- H.T.Kung a fost unul din primii care au studiat relatia intre algoritmi si arhitectura. A stabilit cateva caracteristici si a prezentat corelarea intre ele:

Algoritmi paraleli	Arhitecturi paralele
granularitate modul	complexitate procesor
control concurrent	mod de operare
mecanismul datelor	structura memoriei
geometria comunicatiei	retele de comutare
.complexitate algoritm	numar de procesoare; dimensiune memorie

# Algoritmi - Arhitectura

- **Granularitate modul (obiect)** - se refera la complexitatea modulului care poate fi job, task, proces sau instructiune.
    - De obicei exista posibilitati de paralelism la o granularitate mare dar care nu este exploataat deoarece implica o comunicatie intensa si o crestere a complexitatii software-ului.
    - La acest nivel se face o analiza intre granularitate si comunicatie pentru a stabili solutia cea mai buna.
  - **Control concurrent** se refera la schema de selectie a modulelor pentru executie.
    - Aceasta trebuie sa satisfaca dependenta de date si dependenta de control (asigurarea excluderii mutuale a accesului la aceeasi resursa).
- Citeva scheme de control sunt bazate pe:
- disponibilitatea datelor (data flow)
  - control centralizat (synchronized)
  - cereri (demand-driven).
- exemplu algoritmii care prelucreaza matrice se potrivesc foarte bine pe procesoarele sistolice sau pe masive de procesoare iar algoritmii care au transferuri conditionate si alte iregularitati se potrivesc foarte bine pe arhitecturi asincrone cum ar fi multiprocesoare si data-flow.

- **Mecanismul datelor** se refera la faptul cum sunt utilizati operanzii.
  - Datele furnizate de instructiuni pot fi utilizate ca "date pure" in structurile data-flow sau pot fi depuse in locatii adresabile in masinile Von Neumann.
- **Geometria comunicatiei** - se refera la sablonul de interactiune intre module.
  - Geometria comunicatiei poate fi regulata sau neregulata.
- **Complexitate algoritm** se refera la numarul de operatii necesare pentru implementarea algoritmului.
  - Are influenta asupra numarului de procesoare si asupra dimensiunii memoriei.

# Performanță și scalabilitate

- Evaluare
  - Runtime secvențial ( $T_{seq}$ ) este o funcție de
    - dimensiunea problemei și arhitectura
  - Runtime paralel ( $T_{par}$ ) este o funcție de
    - dimensiunea problemei și arhitectura paralelă
    - numarul de procesoare utilizate în execuție
- Performanță paralelă afectată de
  - algoritm + arhitectură
- Scalabilitate
  - Abilitatea algoritmului paralel de a atinge performanță câștigă proporțional cu numărul de procesoare și cu dimensiunea problemei

# Scalabilitate

- Un program se poate utiliza mai multe procesoare
  - Cum evaluezi scalabilitatea?
- Cum evaluezi performantele scalabilității?
- Evaluare comparativă
  - Dacă se dublează numărul de procesoare, la ce să ne așteptăm?
  - Scalabilitatea este liniară?
- Utilizați o măsură de eficiență paralelă
  - Este menținută eficiența pe măsură ce crește dimensiunea problemei?
- Aplicați valori de performanță

# Indicatori de performantele ai calculului paralel

Datorita complexitatii calculului paralel este greu de a stabili o masura a performantelor care sa stabileasca real si absolut performantele unui sistem cu arhitectura paralela. Totusi sunt utilizati cativa indicatori care masoara diferite aspecte globale.

## **Rata de executie**

masoara rata de producere a unor rezultate in unitatea de timp.

Uzual se folosesc :

- **MIPS / GIPS / TIPS** – milioane /giga/tera de instructiuni pe secunda care masoara numarul de instructiuni pe care le executa pe secunda o unitate de prelucrare mono sau multiprocesor. Un astfel de indicator este inadecvat pentru o masina SIMD care executa aceeasi instructiune pe mai multe fluxuri de date.
- **MOPS / GOPS / TOPS** - milioane /giga/tera de operatii pe secunda care masoara operatiile efectuate de unitatile de prelucrare. O astfel de unitate de masura nu tine seama de lungimea cuvintului si nici de natura operatiilor.
- **MFLOPS / GFLOPS / TFLOPS / PFLOPS**- milioane /giga/tera/peta de operatii cu virgula mobila pe secunda (Giga) care masoara operatiile cu virgula mobila efectuate (Tera) de unitatile de prelucrare.O astfel de masura este adecvata numai pentru aplicatii numerice dar nu reprezinta nici un fel de masura pentru prelucrari alfanumerice sau pentru aplicatii bazate pe inteligenta artificiala.
- **MLIPS / GLIPS/ TLIPS**– milioane /giga/tera de inferente logice pe secunda care masoara numarul de inferente logice realizate in aplicatii de IA

# Indicatori

Presupunem ca avem o structura cu  $p$  procesoare, care participă la rezolvarea unei probleme.

**Viteza de prelucrare -  $V_p$**

$$V_p = \frac{T_1}{T_p} \text{ unde}$$

$T_1$  - timpul necesar pentru prelucrare utilizând un singur procesor

$T_p$  - timpul necesar pentru prelucrare utilizând  $p$  procesoare

Cu alte cuvinte,  $V_p$  reprezintă raportul între prelucrarea secvențială și cea paralelă care scoate în evidență creșterea datorită paralelismului.

deoarece se consumă timp cu sincronizarea, comunicarea și "overhead" cerut de interacțiunea între procesoare.

$$1 \leq V_p < p$$

**Eficiența  $E_p$**

Este definită ca raportul între viteza de prelucrare și numărul de procesoare.

$$E_p = \frac{V_p}{p} = \frac{T_1}{p * T_p} < 1$$

Eficiența este o măsură a eficienței costului

Cost  $p * T_p$  este  $p * T_p >> T_1$

# Indicatori

## ***Redundanta Rp***

Este definita ca raportul intre numarul total al operatiilor Op necesar efectuarii calculului cu p procesoare si numarul de operatii O1 necesar efectuarii calculului cu un singur procesor.

$$R_p = \frac{O_p}{O_1} = \frac{\text{nt total de operatii efectuate pe cele } p \text{ procesoare}}{\text{nr de operatii necesare efectuarii pe 1 procesor}}$$

*reflecta timpul pierdut cu overhead-ul.*

## ***Utilizarea Up***

Este definita ca raportul dintre numarul de operatii  $O_p$  necesar efectuarii calculului cu p procesoare si numarul de operatii care ar fi putut fi efectuate cu p procesoare in timpul  $T_p$

$$U_p = \frac{O_p}{p*T_p} \leq 1$$

# Limite ale calculului paralel

Este foarte important a stabili limita calculului paralel.

Fie  $T_n$  timpul necesar executiei a  $n$  taskuri de  $k$  tipuri diferite.

Fiecare tip consta din  $n_i$  taskuri necesitind  $t_i$  secunde fiecare

$$T_n = \sum_{i=1}^k (n_i * t_i) \quad \text{iar} \quad n = \sum_{i=1}^k n_i \quad T_n \text{ timpul necesar executarii celor } n \text{ task-uri}$$

Prin definitie, rata de executie  $R$  este numarul de operatii efectuate in unitatea de timp

$$R_n = \frac{n}{T_n} = \frac{n}{\sum_{i=1}^k (n_i * t_i)}$$
$$\text{Fie } f_i = n_i/n \quad \sum_{i=1}^k f_i = 1 \quad R_i = \frac{1}{t_i}$$

In acest caz rezulta

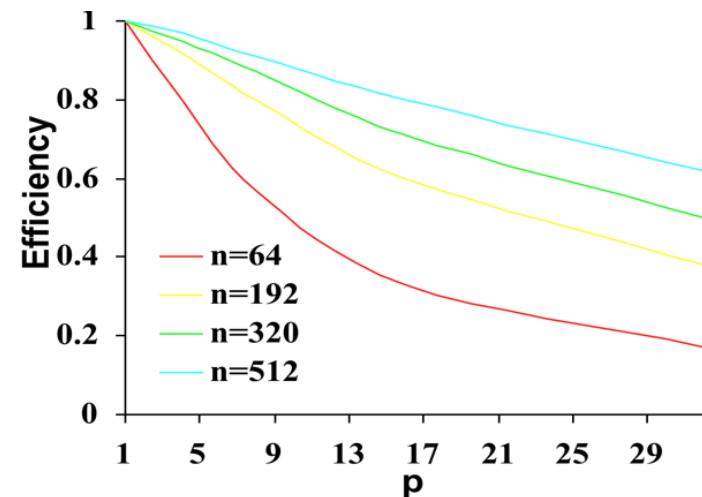
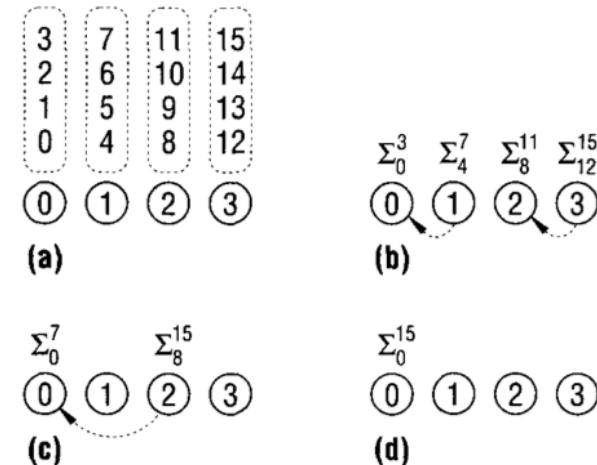
$$R_n = \frac{n}{\sum_{i=1}^k (n_i * t_i)} = \frac{1}{\sum_{i=1}^k (n_i/n * t_i)} = \frac{1}{\sum_{i=1}^k f_i} = \frac{1}{\Sigma (f_i/R_i)}$$

Aceasta relatie reprezinta "bottleneck" - limitarea in structurile paralele.

# Scalabilitatea în adunarea n numere

- Scalabilitatea unui sistem paralel este masura capacitatii de a creste viteza de prelucrare utilizand mai multe procesoare
- Adunarea a n numere utilizand p procesoare conduce la:

$$\begin{aligned} \text{■ } T_p &= \frac{n}{p} + 2 * \log p \\ \text{■ } V_p &= \frac{n}{\frac{n}{p} + 2 * \log p} \\ \text{■ } E_p &= \frac{V_p}{p} = \frac{T_1}{p * T_p} = \frac{n}{n + 2 * p * \log p} \end{aligned}$$



# Legea lui Amdahl

Sa consideram

$f = f_{seq}$  procentul de program care se executa sequential

$1-f = f_{par}$  procentul de program care poate fi paralelizat

Fie  $T_1$  timpul de executie pe o structura cu 1 procesor

Fie  $T_p$  timpul de executie pe o structura cu  $p$  procesoare

$V_p$  viteza de prelucrare

$$V_p = T_1 / T_p = T_1 / (f * T_1 + (1-f) * T_1 / p) = 1 / (f + (1-f)/p)$$

Sau

$$V_p = 1 / (f_{seq} + f_{par}/p)$$

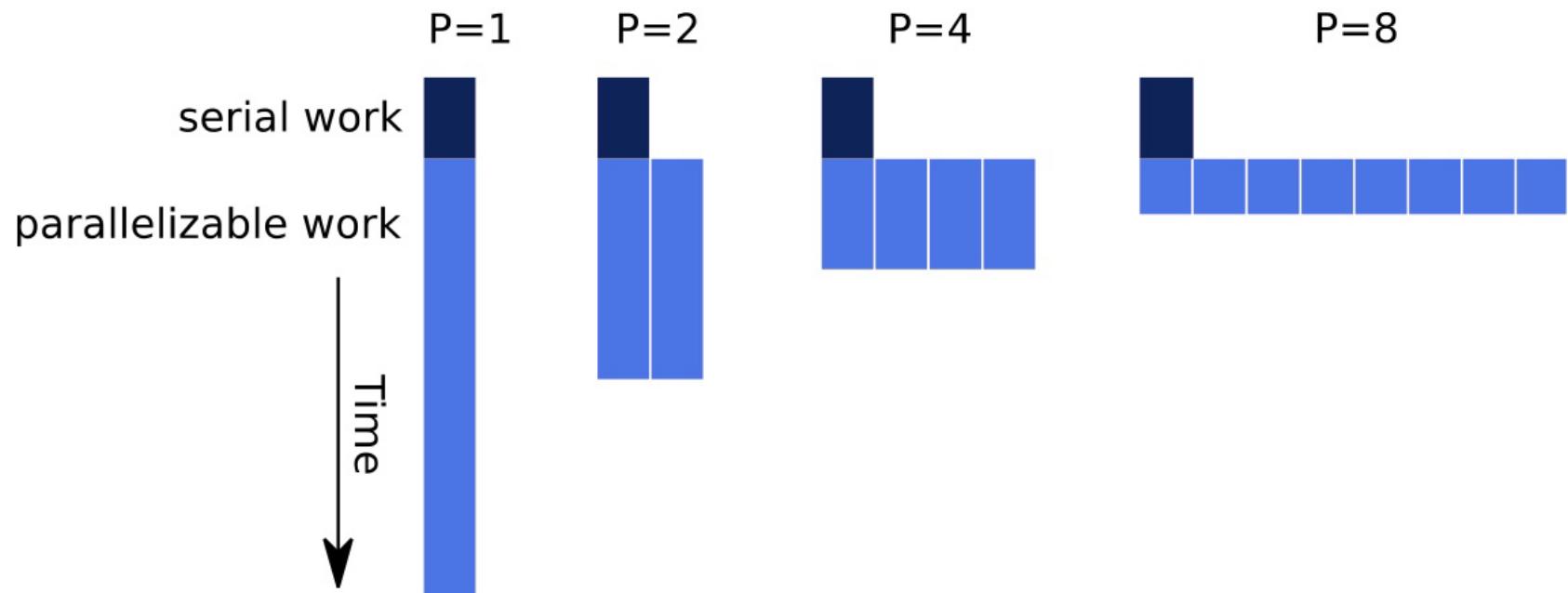
$$V_p = 1 / (f_{seq} + (1-f_{seq})/p)$$

Cand  $p \rightarrow \infty$

$$V_p = 1 / f \text{ sau } V_p = 1 / f_{seq}$$

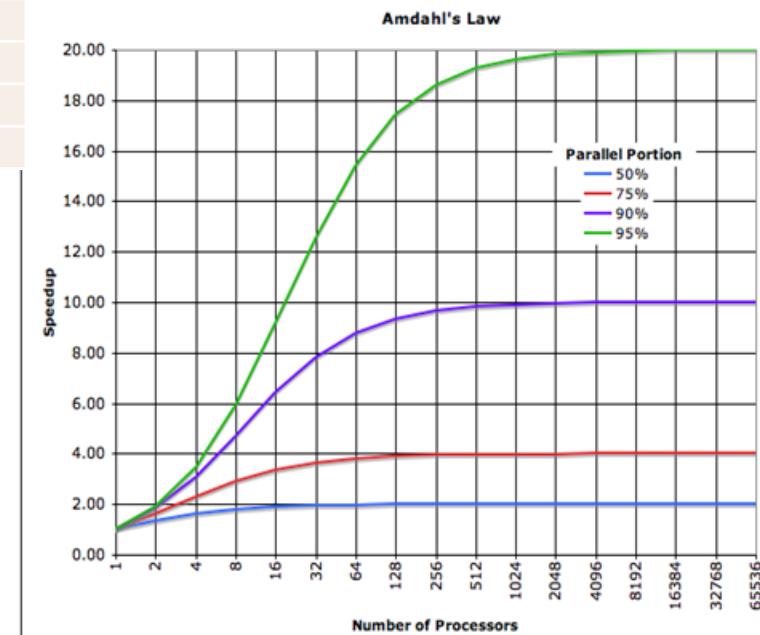
## Amdahl

### Amdahl



# Cresterea in Viteza

Numar procesoare	Crestere viteza			
	Paralelism			
	$f=100\%$ $(1-f)=0\%$	$f=50\%$ $(1-f)=50\%$	$f=10\%$ $(1-f)=90\%$	$f=1\%$ $(1-f)=99\%$
1	1	1.00	1.00	1.00
2	1	1.33	1.82	1.98
5	1	1.67	3.57	4.81
10	1	1.82	5.26	9.17
100	1	1.98	9.17	50.25
1,000	1	2.00	9.91	90.99
10,000	1	2.00	9.99	99.02
100,000	1	2.00	10.00	99.90
1,000,000	1	2.00	10.00	99.99
10,000,000	1	2.00	10.00	100.00



# Scalabilitate

- Capacitatea algoritmului paralel de a obține câștiguri de performanță proporțional cu numărul de procesoare și dimensiunea problemei

Când se aplică Legea lui Amdahl?

- Când dimensiunea problemei este fixă
- Scalare puternică ( $p \rightarrow \infty$ ,  $V_p = V_\infty \rightarrow 1/f$ )

Limita de accelerare este determinată de gradul de secvențialitate, nu de numarul de procesoare !!!

- Eficiența perfectă este greu de realizat

# Legea lui Gustafson-Barsis (accelerare scalată)

Un Data center este interesat , cand se pune problema scalarii, de probleme mai mari

- Cât de mare poate fi o problemă (HPC Linpack)
- Care este contrangerea problemei care se executa in paralel

Să presupunem că timpul de executie este menținut constant

- $T_p = C = (f + (1-f)) * C = (f_{seq} + f_{par}) * C$
- $f_{seq}$  este fractiunea de  $T_p$  cheltuită în execuția secvențială
- $f_{par}$  este fractiunea de  $T_p$  cheltuită în execuție paralelă

Scalara pe p procesoare conduce la

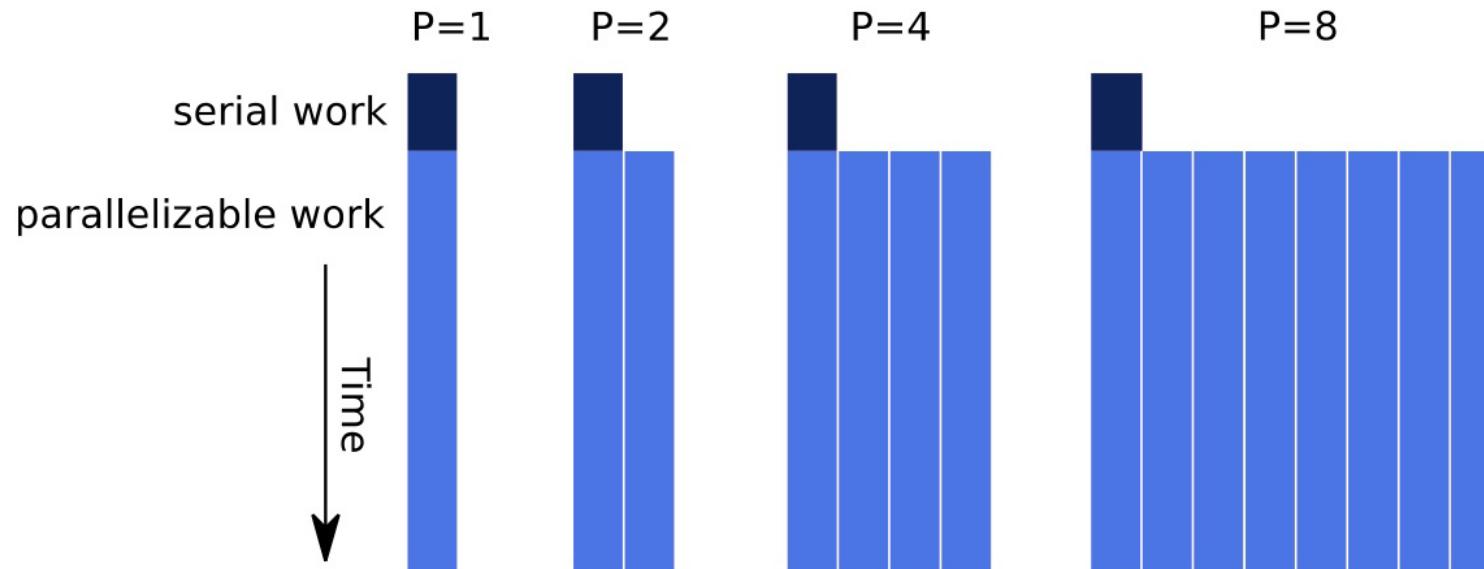
- $T_s = f_{seq} + p * (1 - f_{seq}) = f_{seq} + p * f_{par}$ , deoarece  $f_{par} = 1 - f_{seq}$
- $p * f_{par}$  ne arata cat putem incarca structura din centrul de date
- $T_s = 1 - f_{par} + p * f_{par} = 1 + (p-1)f_{par}$
- $T_n = f_{seq} + (1 - f_{seq}) * p / p = 1$

Care este viteza în acest caz?

$$V_p = T_s / T_p = T_s / 1 = f_{seq} + p(1 - f_{seq}) = 1 + (p-1)f_{par}$$

# Gustafson - Baris

## Gustafson-Baris



# *Gustafson-Barsis' / Scalabilitate*

## Scalabilitate

- Capacitatea algoritmului paralel de a atinge performanță proporțional cu numărul de procesoare și cu dimensiunea problemei

Când se aplică Legea lui Gustafson?

- Când dimensiunea problemei poate crește ca număr
- Scalare slabă  $V_p = 1 + (p-1)f_{par}$
- Funcția de accelerare include numărul de procesoare !!!
- Poate menține sau crește eficiența paralelă prin scalare problema / probleme

# Sun and Ni

- Introduce modelul bazat pe limita de memorie
- $T_s = f_{seq} + g(p)(1 - f_{seq})$ 
  - unde  $G(n)$  este creșterea de memorie a unui procesor
- $T_p = f_{seq} + g(p)(1 - f_{seq}) / p$
- $V_p = T_s / T_p = (f_{seq} + g(p)(1 - f_{seq})) / (f_{seq} + g(p)(1 - f_{seq}) / p)$
  
- Caz 1 :  $g(n) = 1$
- $V_p = (f_{seq} + 1 - f_{seq}) / (f_{seq} + (1 - f_{seq}) / p) = 1 / (f_{seq} + (1 - f_{seq}) / p) \sim Amdahl$
  
- Caz 2 :  $g(n) = p$
- $V_p = T_s / T_p = (f_{seq} + p(1 - f_{seq})) / (f_{seq} + p(1 - f_{seq}) / p) = (f_{seq} + p(1 - f_{seq})) / 1 = f_{seq} + p(1 - f_{seq}) \sim Gustafson$
  
- Caz 3:  $g(p) = m > p$
- $V_p = T_s / T_p = (f_{seq} + m(1 - f_{seq})) / (f_{seq} + m(1 - f_{seq}) / p) =$ 
  - $(f_{seq} + m(1 - f_{seq})) / (m/p + (1 - m/p)f_{seq})$
  
- Workload-ul crește mai repede decât cererea de memorie.

# Limita lui Wrolton

Jack Wrolton a studiat limitele calculului paralel utilizind un model care aproximeaza operarea unui multiprocesor.

El presupune ca un program paralel constă din :

- sectiuni de prelucrare distribuite pe diverse procesoare;
- sectiuni de sincronizare;
- sectiuni care se execută secvențial și constituie un "overhead".

Fie

$ts$  timpul de sincronizare;

$to$  secțiune de overhead

$t$  media timpului de execuție a unui task

$N$  numărul de task-uri (intre puncte de sincronizare)

$P$  numărul de procesoare

**Timpul de execuție secvențial al celor  $N$  task-uri este :**

$$T_1 = N*t$$

**Timpul de execuție a celor  $N$  task-uri executate pe  $p$  procesoare este :**

$$T_{n,p} = ts + (\lceil N / p \rceil) * (t+to)$$

**Viteza de prelucrare**

$$V_{n,p} = T_1 / T_{n,p} = \frac{N*t}{ts + (\lceil N / p \rceil) * (t+to)} = \frac{1}{ts / (N*t) + (1/N) * (\lceil N / p \rceil) * (1+(to/t))}$$

# Comentarii Worlton

Observatii:

Pentru a creste viteza de prelucrare trebuie sa avem in vedere reducerea efectului sincronizarii, overhead-ului si a numarul de pasi  $\lceil N / p \rceil$

- **efectul sincronizarii** cauzat de  $ts/(N*t)$  poate fi redus fie prin micsorarea timpului de sincronizare ts sau prin marirea intervalului intre sincronizari  $N*t$
- **efectul overhead-ului** cauzat de  $to/t$  poate fi redus prin reducerea timpului de overhead to sau prin cresterea granularitatii task-urilor.

*Cresterea granularitatii task-urilor ajuta la reducerea atat a efectului sincronizarii cat si a overhead-ului.*

- **numarul pasilor de calcul**  $\lceil N / p \rceil$  poate fi redus prin cresterea numarului de procesoare si avand un numar de task-uri multiplu de procesoare.

$$V_{n,p} = T_1 / T_{n,p} = \frac{N*t}{ts + (\lceil N / p \rceil)*(t+to)} = \frac{1}{ts / (N*t) + (1/N)*(\lceil N / p \rceil)*(1+(to/t))}$$

# Eficienta

$$V_{n,p} = T_1 / T_{n,p} = \frac{N^*t}{ts + (\lceil N/p \rceil)^*(t+to)} = \frac{1}{ts / (N^*t) + (1/N)^*(\lceil N/p \rceil)^*(1+(to/t))}$$

## Eficienta

$$E_{n,p} = \frac{V_{n,p}}{p}$$

Presupunind ca avem un numar mare de task-uri si ca overhead-ul este neglijabil relativ la granularitate avem :

$$p \ggg N \text{ fix}$$

$$V_{n,p} = \frac{ts}{t} \quad \text{si} \quad E_{n,p} = 0$$

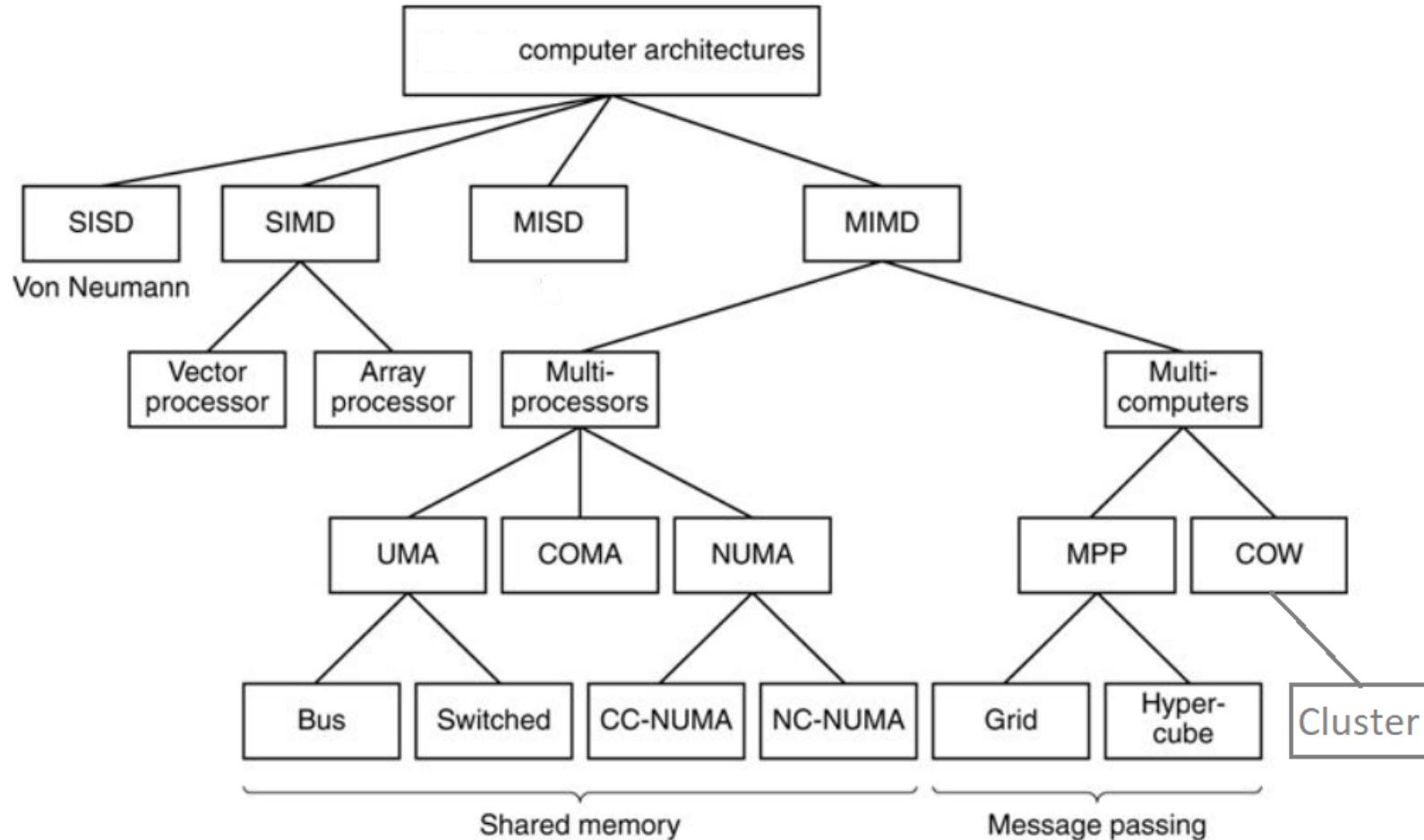
$$N \ggg p \text{ fix}$$

$$V_{n,p} = \frac{1}{1 + to/t} \quad \text{si} \quad E_{n,p} = \frac{1}{1 + to/t}$$

# Taxonomii

## Modele de calcul Paralel

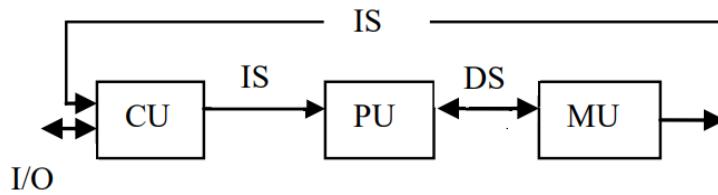
# Arhitecturi de calculatoare



# Taxonomii

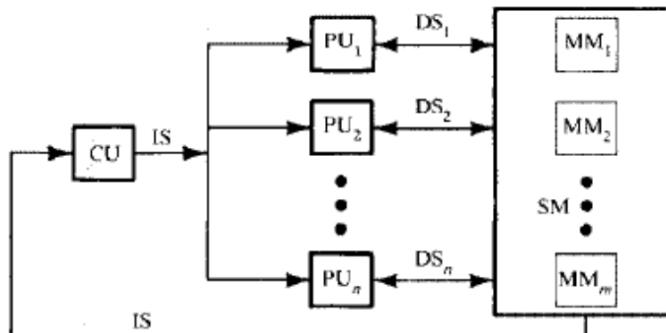
## 1. Taxonomia lui Flynn (bazata pe relatia dintre fluxul de instructiuni si fluxul de date):

- SISD      (single instruction single data flow)
- SIMD      (single instruction multiple data flow)
- MISD      (multiple instruction single data flow)
- MIMD      (multiple instruction multiple data flow)

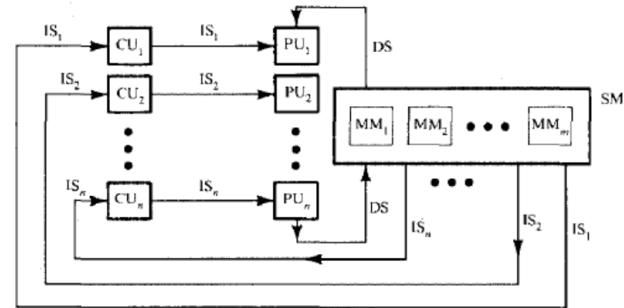


(a) SISD Uniprocessor Architecture

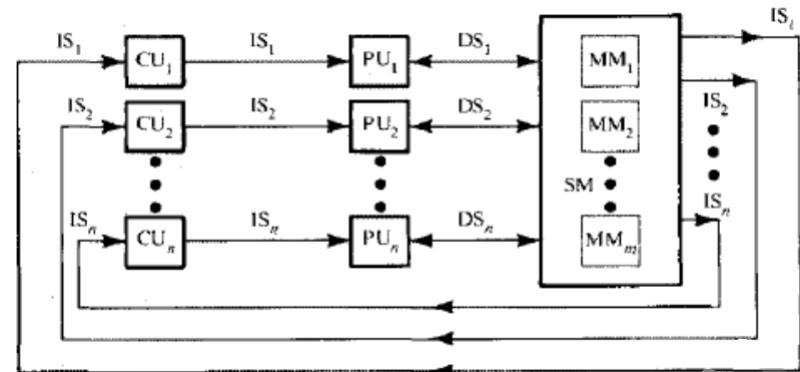
(b) SIMD Architecture



(c) MISD Architecture (the Systolic Array)



(d) MIMD architecture



# Taxonomii

## ***Taxonomia lui Shore:***

- masina seriala pe cuvant paralela pe biti
- masina paralela pe cuvant seriala pe biti
- masina ortogonală
- masina masiv neconectat
- masina masiv conectat

## ***Structurala***

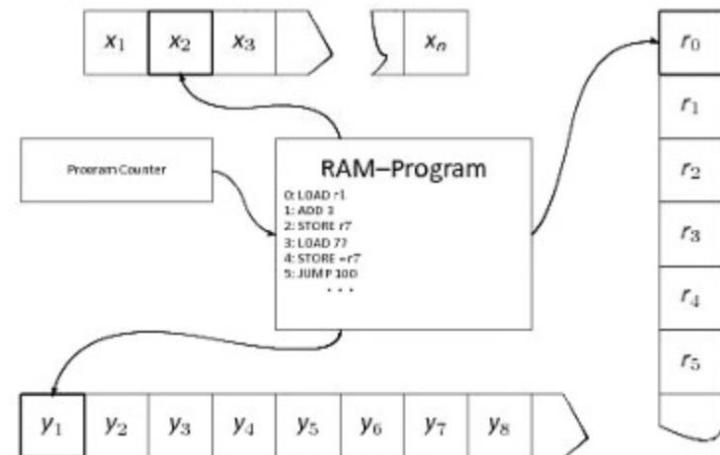
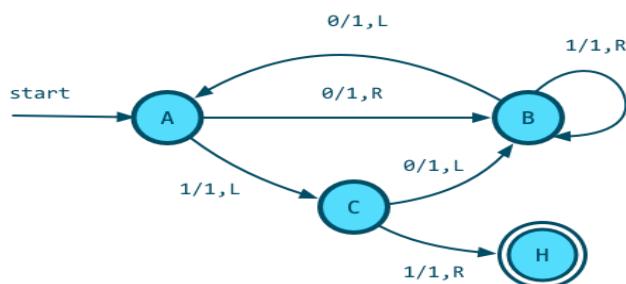
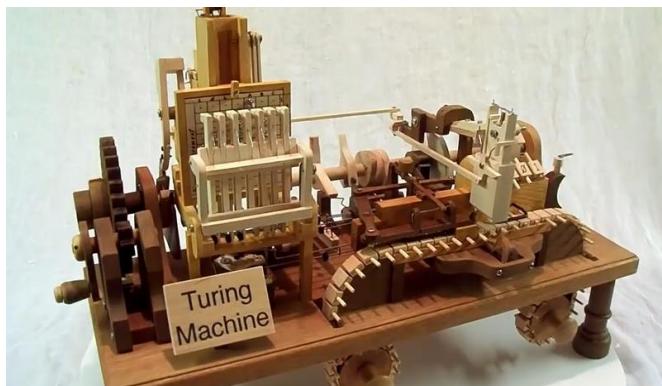
- uniprocesoare seriale
- unicCalculatoare paralele bazata pe paralelism functional si pipeline
- masive de procesoare

# *Modele ale calculului paralel*

- 1. Model RAM (Random Acces Machine)
  2. PIPELINE
  3. PROCESOARE DE VECTORI
  4. MULTIPROCESOARE cu memorie partajata
  5. MULTIPROCESOARE cu transfer prin mesaje
  6. PRAM (Parallel Random Acces Machine)
  7. PROCESARE SISTOLICA
  8. PROCESARE DATA FLOW

O mașină cu acces aleatoriu (RAM):

- o bandă de intrare infinită X cu poziția capului citit  $i \in N$ ,
- o bandă de ieșire infinită Y cu poziția capului de scriere  $o \in N$ ,
- un registru contor de program PC
- un program format dintr-un număr finit de instrucțiuni  $P[1], \dots, P[m]$ ,
- o memorie constând dintr-o succesiune infinită de registre  $r_0, r_1, r_2$



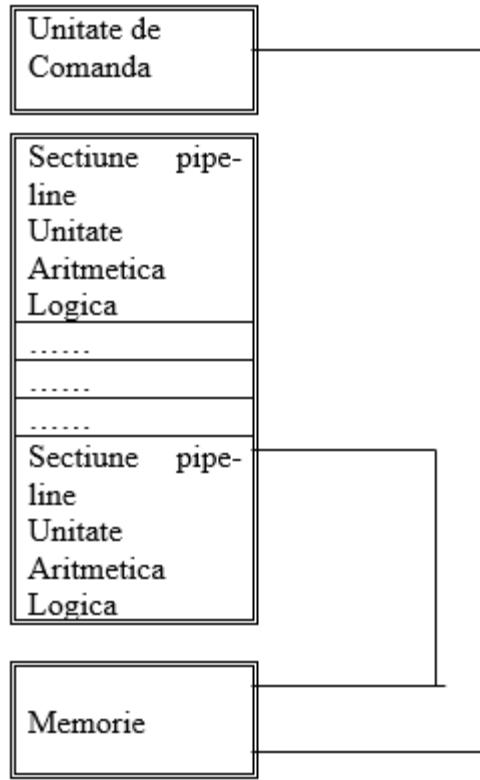
$$RAM = (X, Y, S, s_0, f, Z)$$

- S multimea finita a starilor automatului;
- $s_0$  starea initiala;
- X alfabetul finit al benzii de intrare;
- Y alfabetul finit al benzii de ieșire;
- f funcția de tranzitie, eventual parțial definite
- $f : S \times X \rightarrow S \times Y$
- Z inclus în S - multimea starilor finale

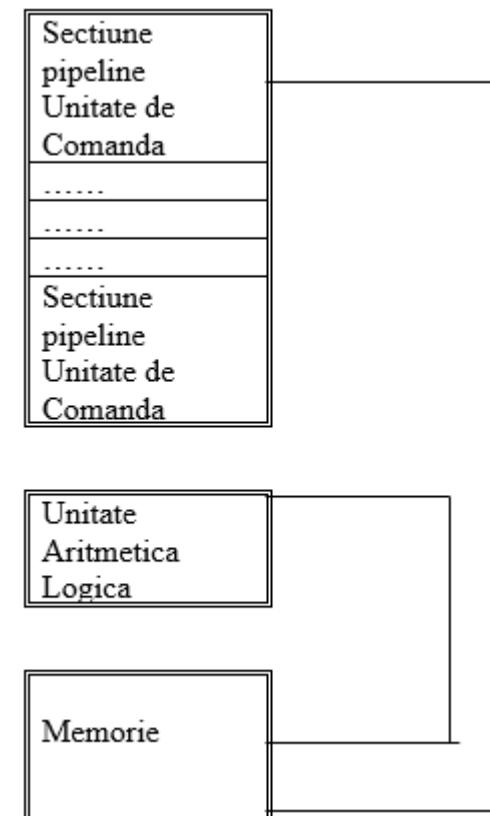
# *Structura pipeline.*

Conceptul de pipeline constă în a împărti un task T în subtask-uri  $T_1, T_2, \dots, T_k$  și de a le atribui unele elemente de procesare.  
Paralelismul se poate realiza fie la nivel de:

- \* prelucrare aritmetică;



- citire, interpretare, execuție instrucțiuni



# Procesoare de vectori.

## Multiprocesoare cu memorie partajata.

### Procesoare de vectori.

In cadrul acestui model, exista un set de instructiuni care trateaza vectorul ca operand simplu. (SIMD) sau

elemente de procesare vectoriala.(UAL de tip pipeline si registre de vectori).

O astfel de structura o gasim in sisteme monoprocesor care au ca extensie procesare de vectori.

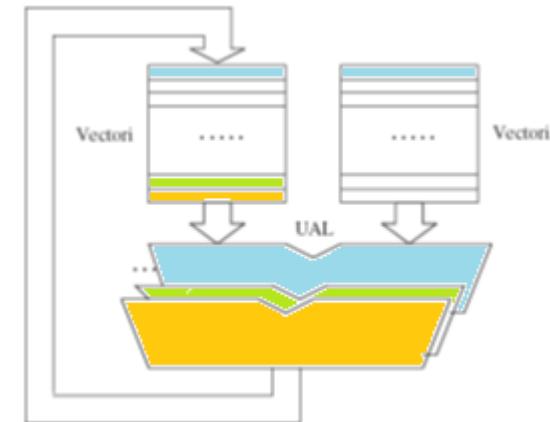
Acopera o gama redusa de probleme cu caracter vectorial.

### Multiprocesoare cu memorie partajata.

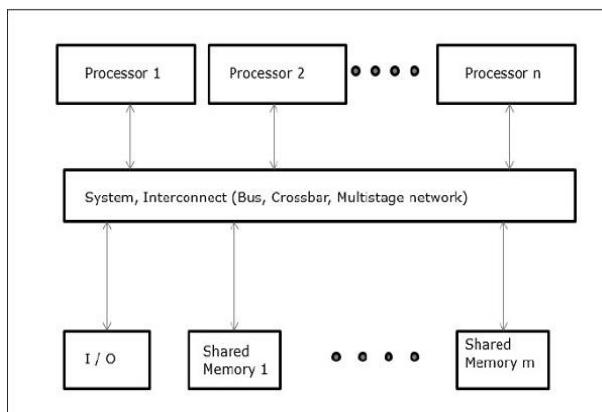
Acest model descrie structurile in care fiecare procesor contine propria unitate de prelucrare, propria memorie si propria unitate de comanda si comunica prin intermediul unei retele de comutare cu module de memorie partajate.

Fiecare procesor executa propriul set de instructiuni din memoria locala sau memoria partajata.

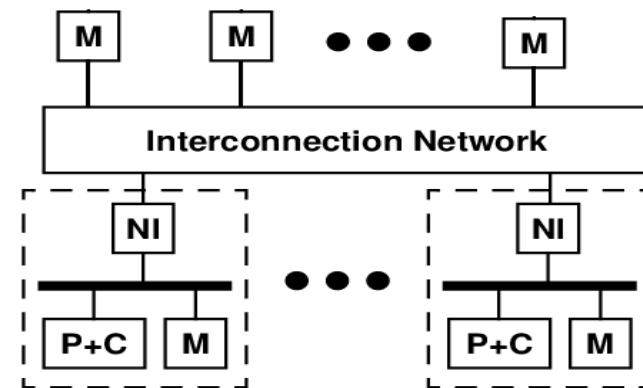
Reteaua de comutare permite schimbul de informatii intre procesoare si intre procesoare si memoria partajata.



UMA



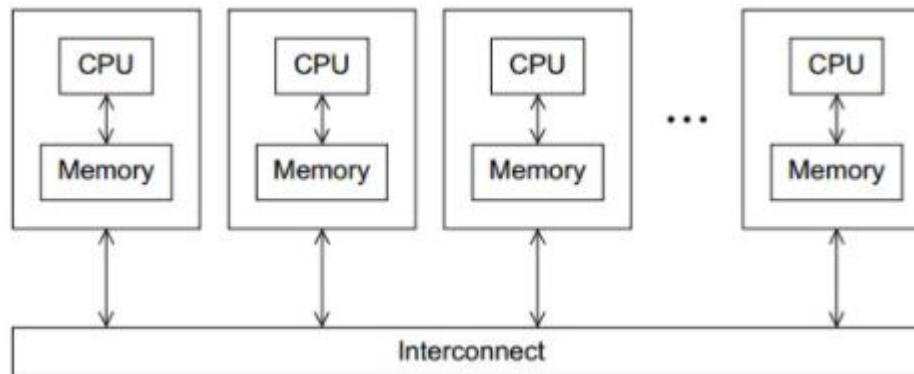
NUMA



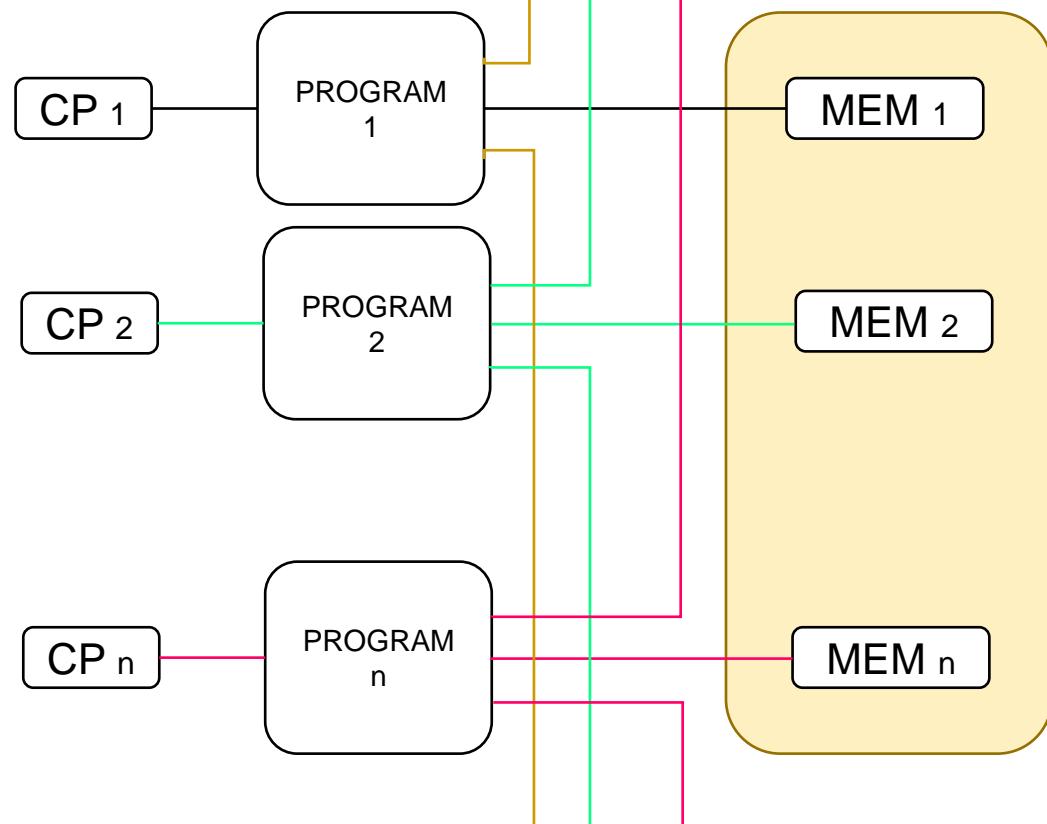
# **Multiprocesoare cu transfer prin mesaje**

## ***Multiprocesoare cu transfer prin mesaje.***

- In acest model memoria este distribuita fiecarui procesor astfel incit fiecare procesor dispune de propriul program si propria memorie de date.
- Comunicarea datelor partajate este realizata prin schimburi de mesaje prin intermediul unei retele de comutare mesaje.
- Acest model este diferit de modelul cu memorie partajata deoarece interconexiunea se realizeaza prin mesaje si nu prin acces direct (comutare de circuite).



# Modelul PRAM -parallel RAM

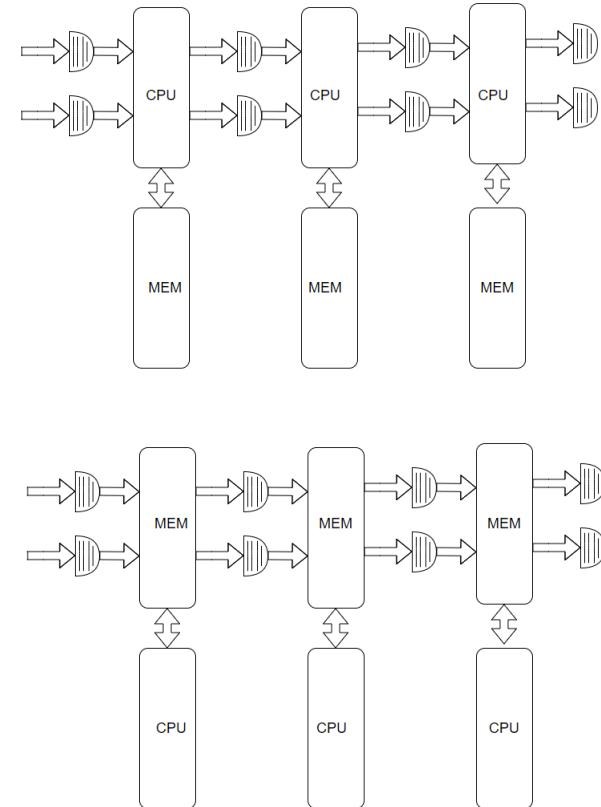
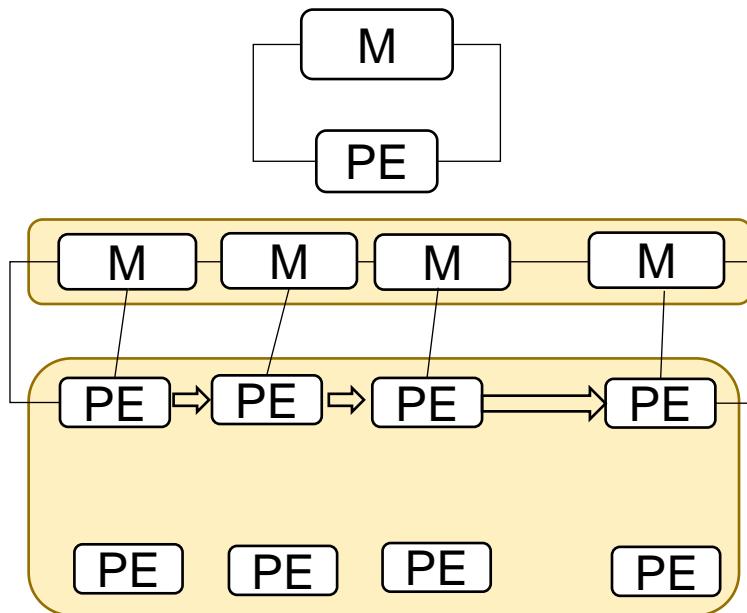


# Procesare sistolica

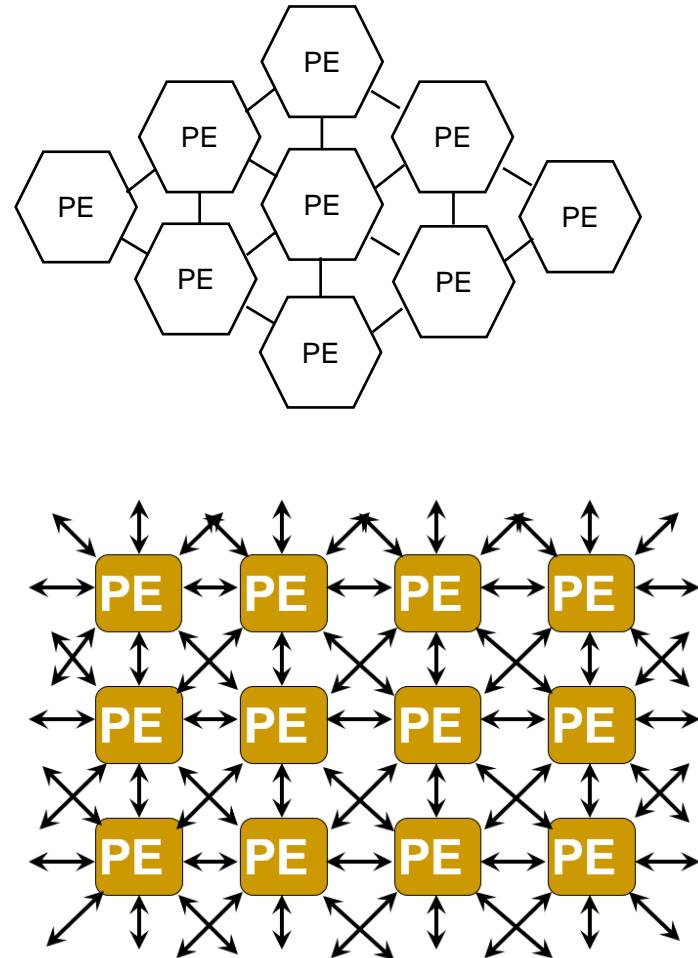
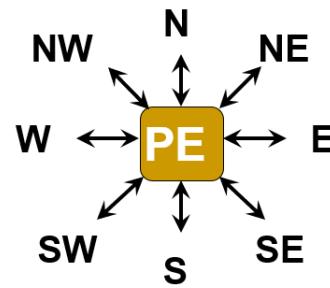
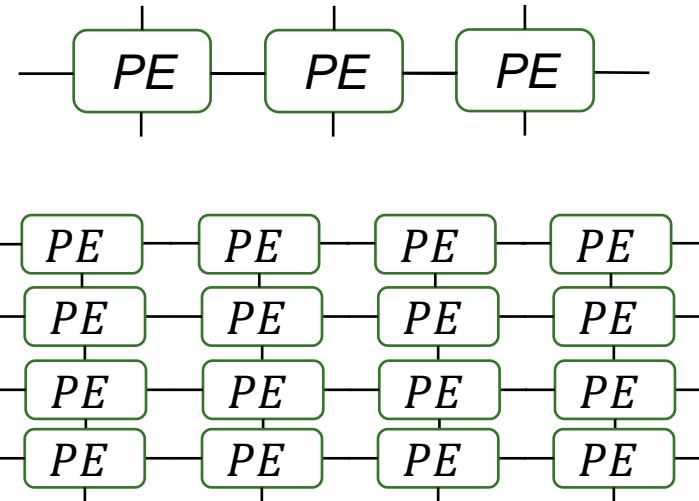
## Procesare sistolica.

Modelul procesarii sistolice constă din elemente de procesare identice aranjate într-o structură pipeline.

Procesarea sistolica este caracterizată prin interconexiuni simple și regulate ceea ce permite integrarea VLSI.



# Modalitati de conexiune Sistolica



# Exemplu

- Fie

- $A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$   $Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$   $X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$   $Y_0 = \begin{bmatrix} y_{01} \\ y_{02} \\ \vdots \\ y_{0n} \end{bmatrix}$

- Dorim sa calculam

- $Y = A * X + Y_0$

- Presupunem ca avem un element de procesare sistolica

$$y_{out} = a * x + y_{in}$$

**Y**

<b>y<sub>1</sub></b>				
<b>y<sub>2</sub></b>				
<b>y<sub>3</sub></b>				
<b>y<sub>n</sub></b>				

**A**

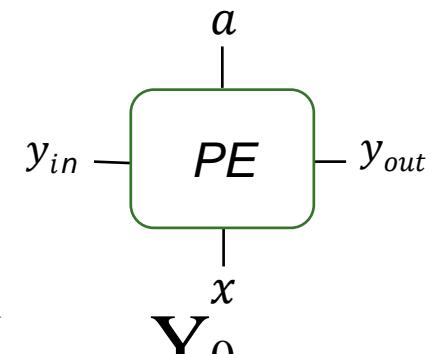
<b>a<sub>11</sub></b>	<b>a<sub>12</sub></b>	<b>a<sub>13</sub></b>		<b>a<sub>1n</sub></b>
<b>a<sub>21</sub></b>	<b>a<sub>22</sub></b>	<b>a<sub>23</sub></b>		<b>a<sub>2n</sub></b>
<b>a<sub>31</sub></b>	<b>a<sub>32</sub></b>	<b>a<sub>33</sub></b>		<b>a<sub>3n</sub></b>
<b>a<sub>n1</sub></b>	<b>a<sub>n2</sub></b>	<b>a<sub>n3</sub></b>		<b>a<sub>nn</sub></b>

**X**

<b>x<sub>1</sub></b>			
<b>x<sub>2</sub></b>			
<b>x<sub>3</sub></b>			
<b>x<sub>n</sub></b>			

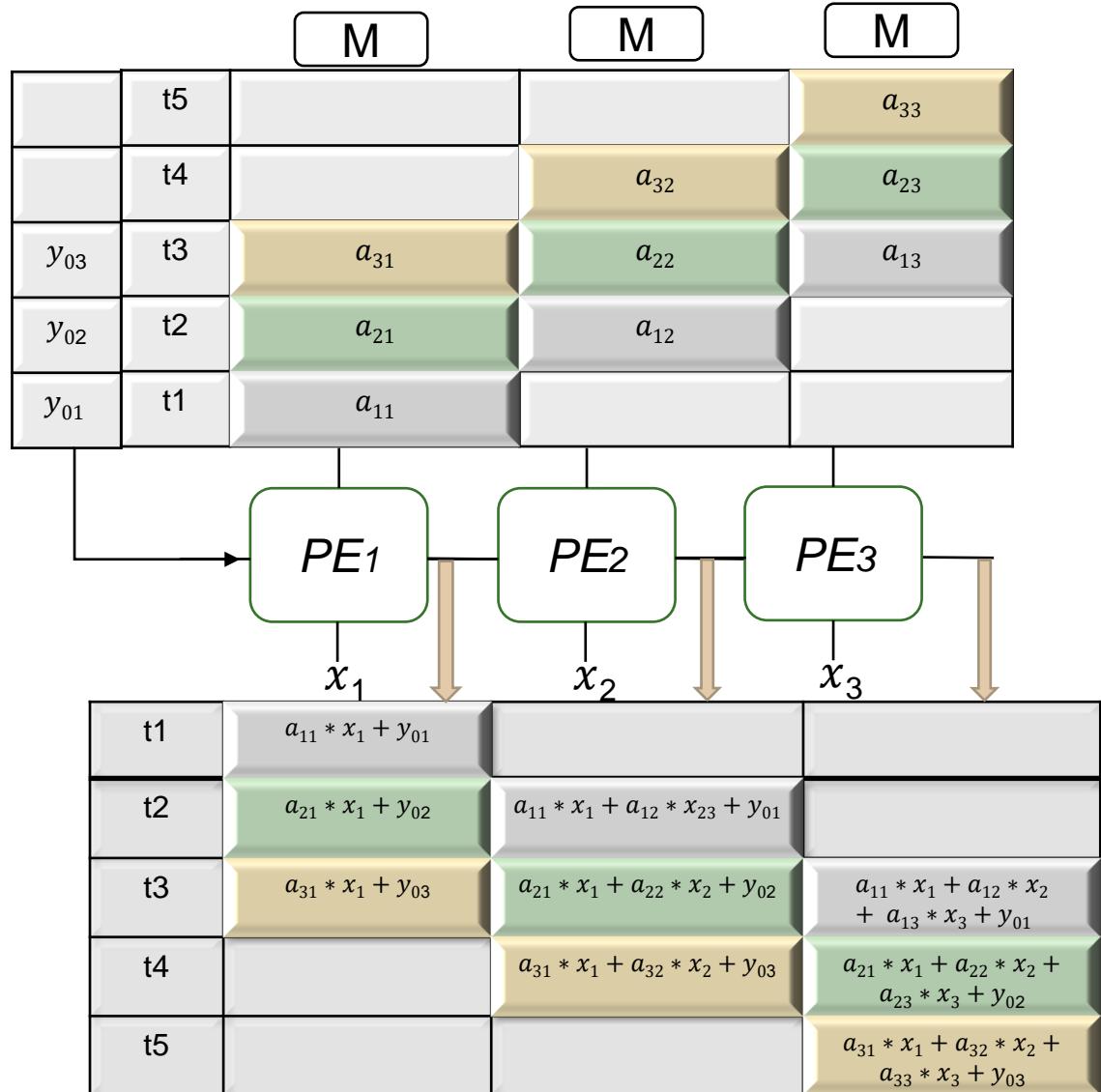
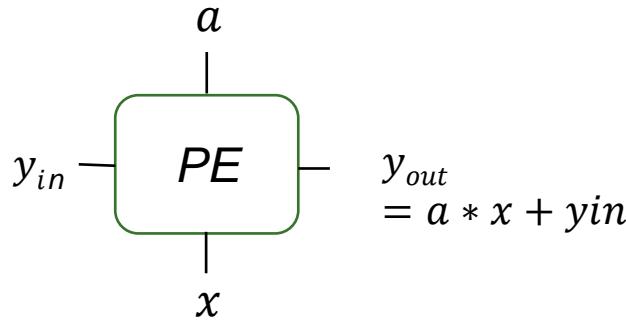
**Y<sub>0</sub>**

<b>y<sub>01</sub></b>			
<b>y<sub>02</sub></b>			
<b>y<sub>03</sub></b>			
<b>y<sub>0n</sub></b>			



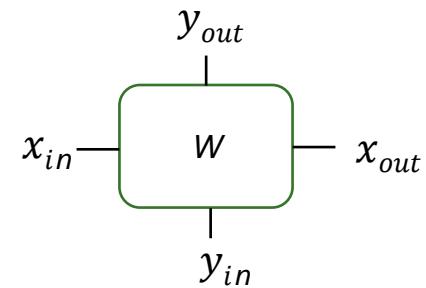
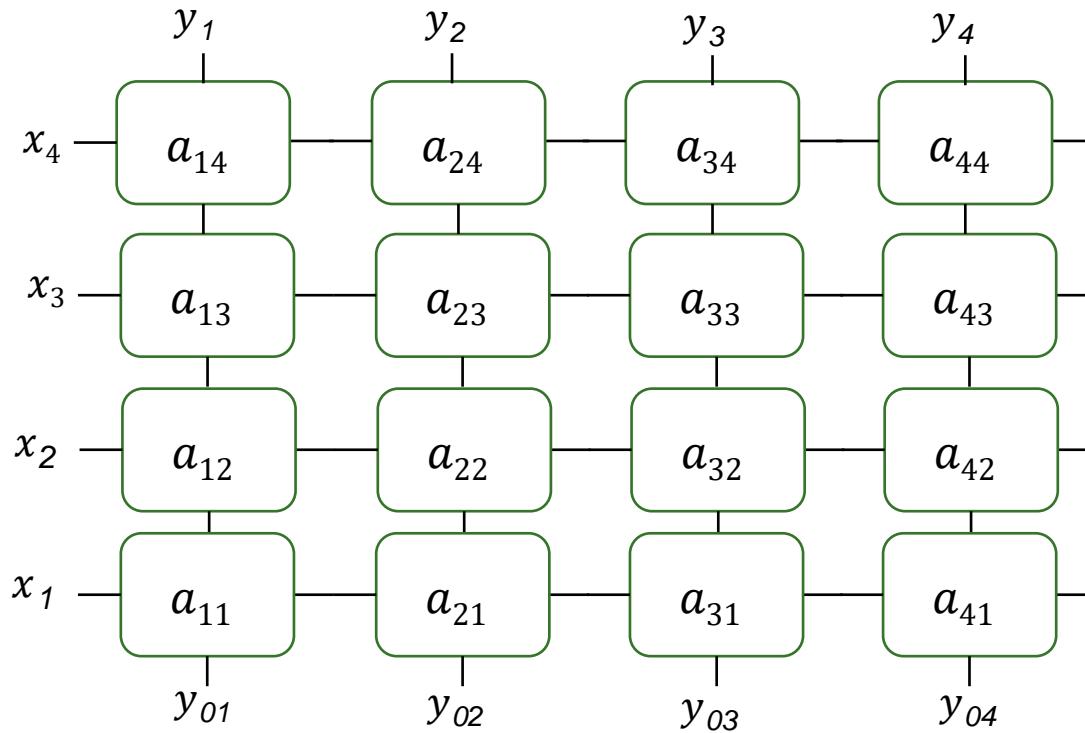
# Exemplu pentru n=3, structura unidimensională

- $y_i = \sum_{j=1}^n a_{ij} * x_j + y_{0i}$
- $y_1 = a_{11} * x_1 + a_{12} * x_2 + a_{13} * x_3 + y_{01}$
- $y_2 = a_{21} * x_1 + a_{22} * x_2 + a_{23} * x_3 + y_{02}$
- $y_3 = a_{31} * x_1 + a_{32} * x_2 + a_{33} * x_3 + y_{03}$



# Exemplu pentru n=4, structura bidimensională

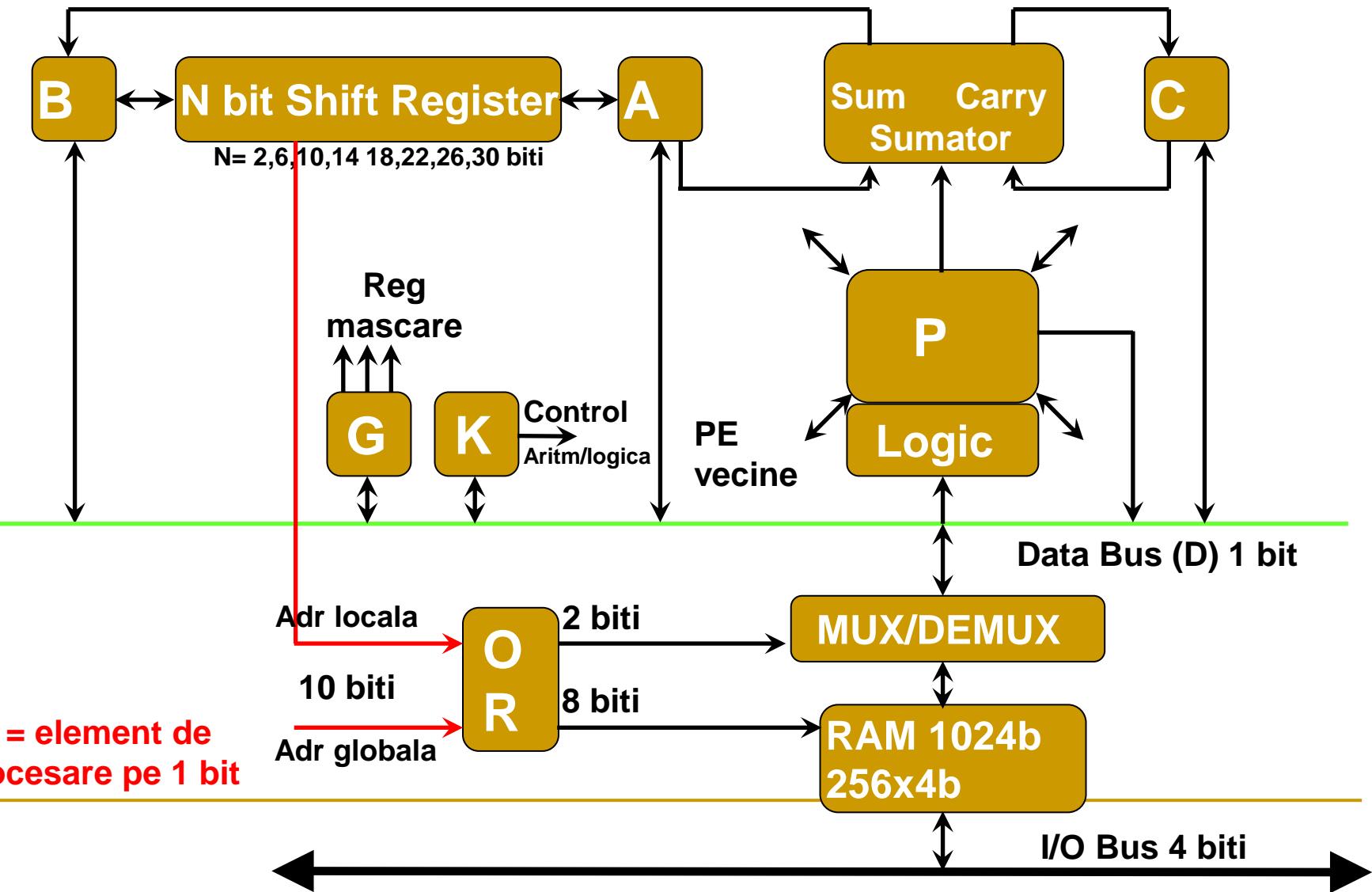
- $y_i = \sum_{j=1}^n a_{ij} * x_j + y_{0i}$
- $y_1 = a_{11} * x_1 + a_{12} * x_2 + a_{13} * x_3 + a_{14} * x_4 + y_{01}$
- $y_2 = a_{21} * x_1 + a_{22} * x_2 + a_{23} * x_3 + a_{24} * x_4 + y_{02}$
- $y_3 = a_{31} * x_1 + a_{32} * x_2 + a_{33} * x_3 + a_{34} * x_4 + y_{03}$
- $y_4 = a_{41} * x_1 + a_{42} * x_2 + a_{43} * x_3 + a_{44} * x_4 + y_{04}$



$$y_{out} = w * xin + yin$$

$$x_{out} = xin$$

# Exemplu de masina sistolica, Arhitectura Masinii Blitzen – PE



# Elementele PE

- P – registru pe 1 bit
  - Operanzi
  - Operatori in functiile aritmetice
  - Utilizat la rutare: primește ops de la PE-uri adiacente
- K – registru de control aritmetico-logic
  - Faciliteaza aparitia ops aritmetice inverse (+/-)
  - Algoritmi de impartiri (non-returning-division)
- G – registru de mascare
  - Se indica daca operatia se executa sau nu
- A & B – registru pe 1 bit
  - Preiau rezultatele de la sumator
  - Contribuie la suma

# Elementele PE

A	P	C	B	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

## ■ Sumator

- ADD:  $A + P + C \rightarrow B, C$
- HADD:  $A + C \rightarrow B, C$

## ■ Shift Register (SR) – Registrul de deplasare



- $N$  stagii = 2, 6, 10, 14, 18, 22, 26, 30
- 30 biti + A & B = 32 biti
- Bitul obtinut prin deplasare nu e neaparat salvat
- La fiecare pas SR se deplaceaza cu o pozitie R/L
- SR-ul poate fi sters

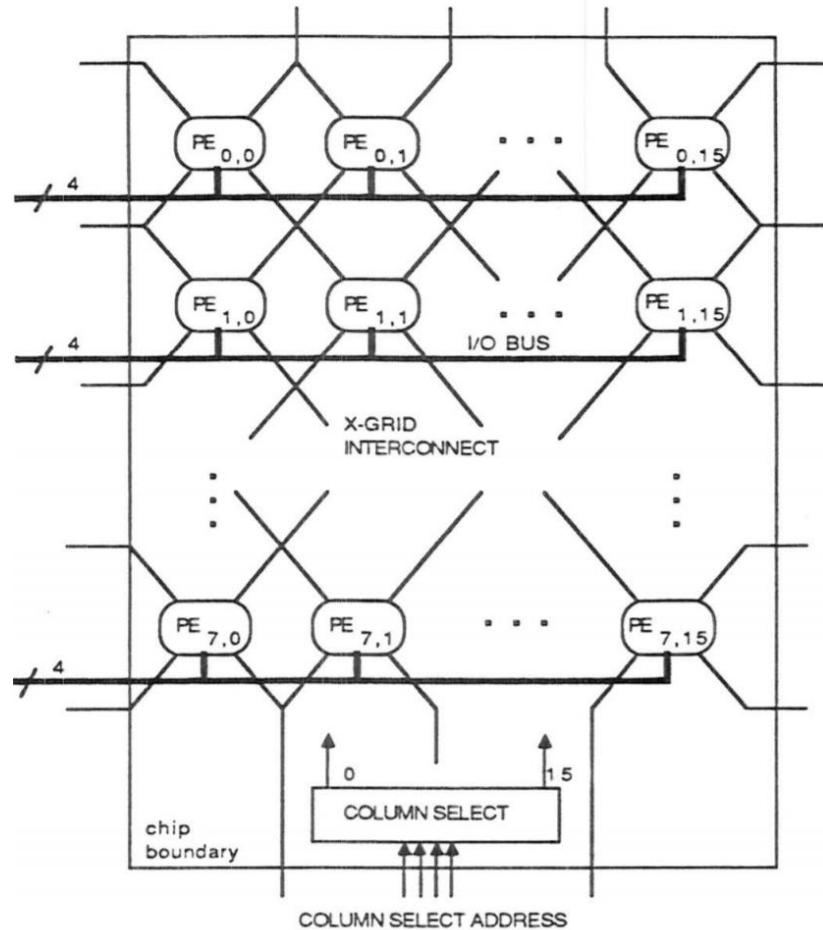
# Memoria & I/O Bus

- Accesul la memoria RAM
  - Cu o adresa globala de forma  $2^p$  (p e multiplu de 2)
  - Cu o adresa locala → specific masinii Blitzen (10 biti)
- I/O Bus
  - Formata din 4 biti si utilizata pentru conectarea PE-urilor la resurse externe de stocare a datelor
  - O magistrala I/O este folosita in comun de catre 16 PE-uri

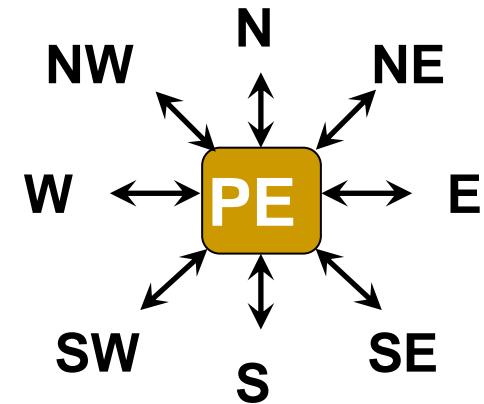
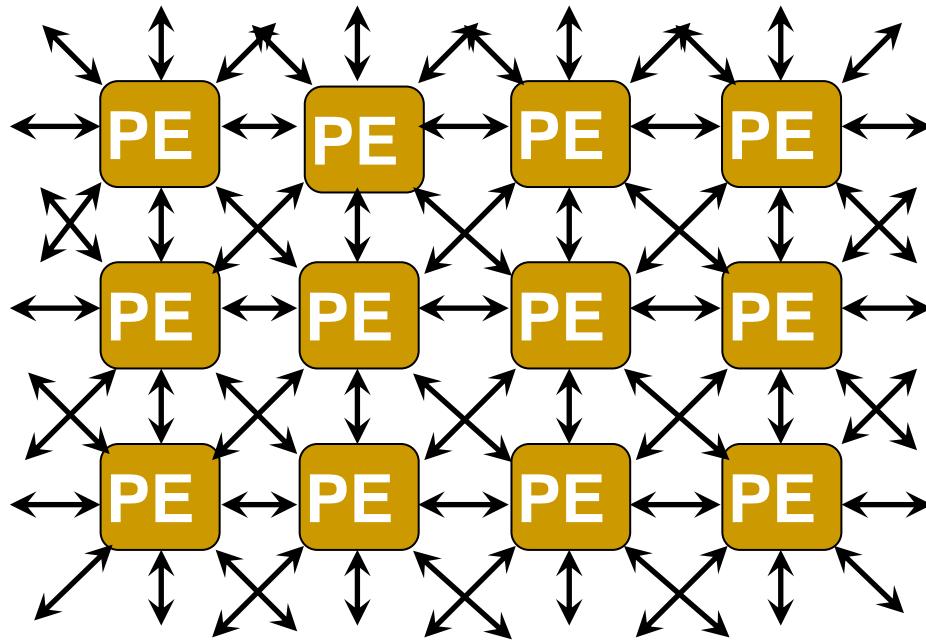


# Chip-ul Blitzen

- Un Chip Biltzen este format din
  - 8 magistrale
  - 8 linii a cate 16 procesoare fiecare  $\rightarrow$  128 PE/chip, fiecare cu cate 1K de RAM
  - Fiecare PE functioneaza la 20MHz cu  $8 \times 4 = 32$  biti/ciclu



# Un Sistem Blitzen

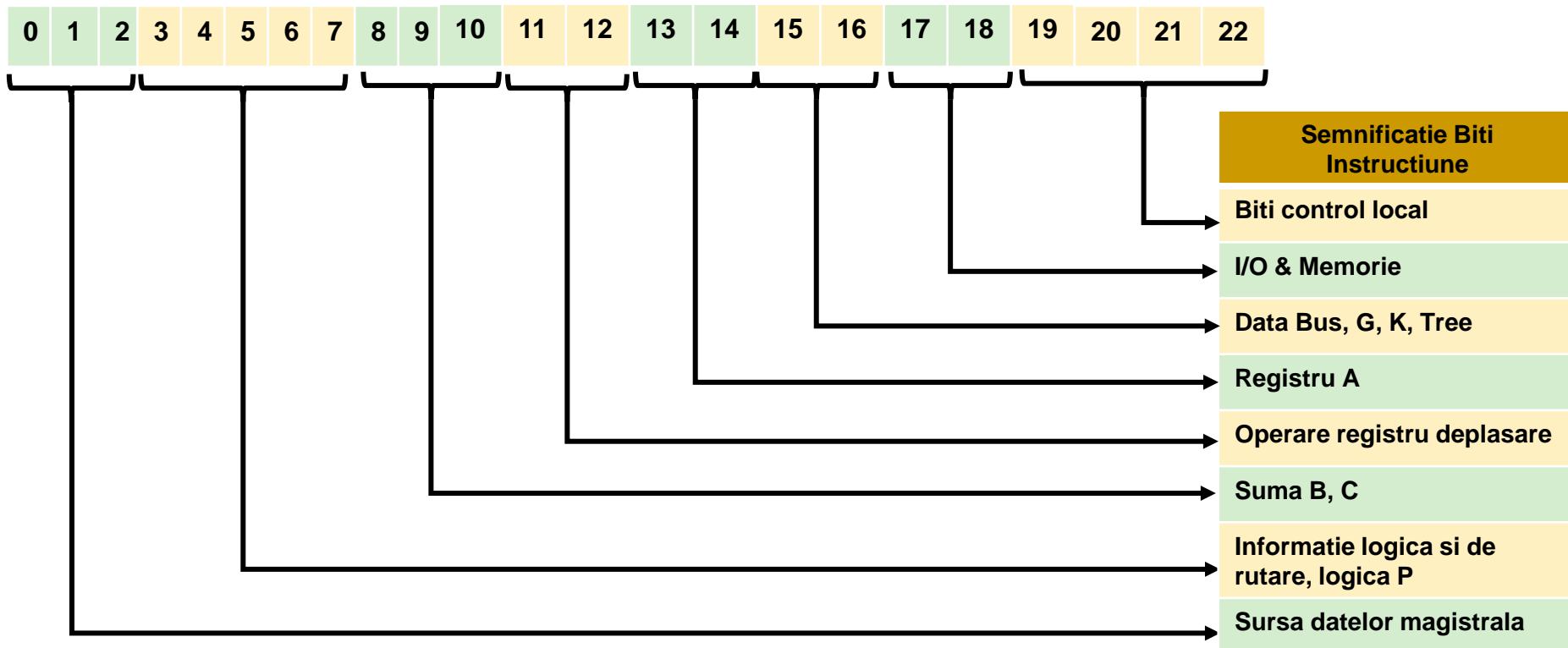


Structura X-Grid

- Sunt 8 directii de rutare
  - N, S, E, W, NE, SE, SW, NW
- Chip-urile Blitzen se grupeaza pe linii de 128 chip-uri → 128 x 128 PE = 16384PE (sistem Blitzen)

# Formatul Instructiunilor

- La fiecare ciclu se executa o instructiune intr-un pipeline cu 3 stagii: Decodare, Broadcast, Executie
- Instructiunile sunt pe 23 de biti organizati astfel:



# Semnificatia Bitilor Instructiune

- Bitii 0, 1, 2: sursa datelor de pe magistrala
  - Registrii A, B, C
  - Registrii G sau K
  - Registrul P
  - Memorie (cand este adresata memoria se executa un read 1b)
- Bitii 3, 4, 5, 6, 7:
  - Operatia logica ce trebuie operata in P
  - Operatia de rutare catre unul din cei 8 vecini
  - Configurarea registrului de deplasare (no assigns to P)
- Bitii 8, 9, 10: operatii execute de sumator
  - Suma & Carry din sumator catre registrii B si C
  - Pot fi utilizate si pentru incarcarea lui B & C

# Semnificatia Bitilor Instructiune

- Bitii 11,12: operatii de deplasare/shiftare
  - Stanga (L)/Dreapta (R)/Stergere (D)
- Bitii 13,14: operatii cu registrul A
  - Setare directa
  - Setare cu continutul registrului de deplasare/shiftare
- Bitii 15,16: rezultatul e transferat din magistrala de date
  - Catre registrii G, K
  - Catre OR-ul ierarhic (Tree)
- Bitii 17,18: prelucrarea informatiilor din
  - Memoria locala (R/W operations)
  - I/O Bus (I/O operations)

# Semnificatia Bitilor Instructiune

- Bitii 19,20,21,22: control local
- Bit 19: operatii mascate & nemascate ale registrului P
  - Bit 19 = 0: operatiile P nemascate
  - Bit 19 = 1: operatiile P mascate;
- Bit 20 : identifica operatii mascate efectuate de memorie, sumator, SR, & registrii A, B sau C
  - Bit 20 = 0: operatiile nemascate
  - Bit 20 = 1: operatiile mascate;
- Bit 21 : poate identifica daca operatiile sunt transmise sau daca a avut loc un broadcast. Este folosit pt:
  - Bit 21 = 0: operatiile curente
  - Bit 21 = 1: op complementate
- Bit 22 : operatii cu
  - Bit 22 = 0: adresa globala
  - Bit 22 = 1: cu cei 10 biti din registrul de deplasare (adr locala)

# Particularitati Blitzen

- PE-urile lucreaza la nivel de bit
- Registrul de Shiftare/Deplasare este bidirectional
- Blitzen-ul are memorie on-chip de 1k (1024 b)
- Este posibila mascarea separata a diverselor tipuri de operatii cu memoria sau cu SR (de deplasare)
- Exista un control local al adresei de memorie (Local Address Modifier)
- Bitul K de la Blitzen permite o executie a datelor de tip MIMD
- Masina Blitzen in anumite aplicatii este mult mai performanta ca MPP

# Exemple

- Mai multe instructiuni elementare distincte se pot executa in acelasi timp:
  - ADD + preluare date din memorie + preluare de pe magistrala de date
- Exemple de microinstructiuni:
  - SET\_C //  $C \leftarrow 1$
  - ADD //  $A + P + C \rightarrow B, C$
  - MOV\_BD // pune B pe magistrala de date D
  - MOV\_MD(ADDR) // pune de la ADDR pe magistrala D
  - MOV\_DP // pune in P continutul de pe D
  - ROUTE\_E // trimit P catre E si incarca in P din W

# Adunarea a doua Numere (8b)

```
#include "blitzen.h"      /*contine setul de instructiuni */
#define XADDR    100
#define YADDR    200
#define WADDR    300
#define NUMBITS  8
main (){
// suma W = X + Y;
// valori cum ar fi "X0" fac referinta la bitul 0 din X
set_route(GRID);
    load_file("in1.ism",XADDR,XADDR,8); //citesc X si Y
    load_file("in2.ism",YADDR,YADDR,8);
    CLR_C;                      // clear C,
for (int i=0; i< NUMBITS; i++){
    MOV_MD(XADDR+i); /* A <- X(i), data de la adresa XADDR pe mag da date D
    MOV_DA;           // pune in A continutul de pe D
END;
    MOV_MD(YADDR+i); /* P <- Y(i)
    MOV_DP;
END;
    ADD;                  /* adunare, rezultat in B
END;
    MOV_BD;              /* W(i) <- B
    MOV_DM(WADDR+i);
END;

}

save_file("sum.osm",WADDR,WADDR,8); //salvare rezultat
zyg_end();
}
```

# Categorii de Algoritmi

- Algoritmii ce se preteaza la utilizarea masinii Blitzen fac parte din urmatoarele categorii
- 1. Embarrassingly Parallel Algorithms (algoritmi “rusinosi” paraleli)
  - Fiecare PE actioneaza independent (parallelism maxim);
  - Exemplu: Valoarea de prag (apartenenta la domeniu)
- 2. Near Embarrassingly Parallel Algorithms
  - Un PE are nevoie de informatii pe care le va primi de la alte PE
  - Exemplu: tratarea imaginilor (filtre etc); detectia contururilor

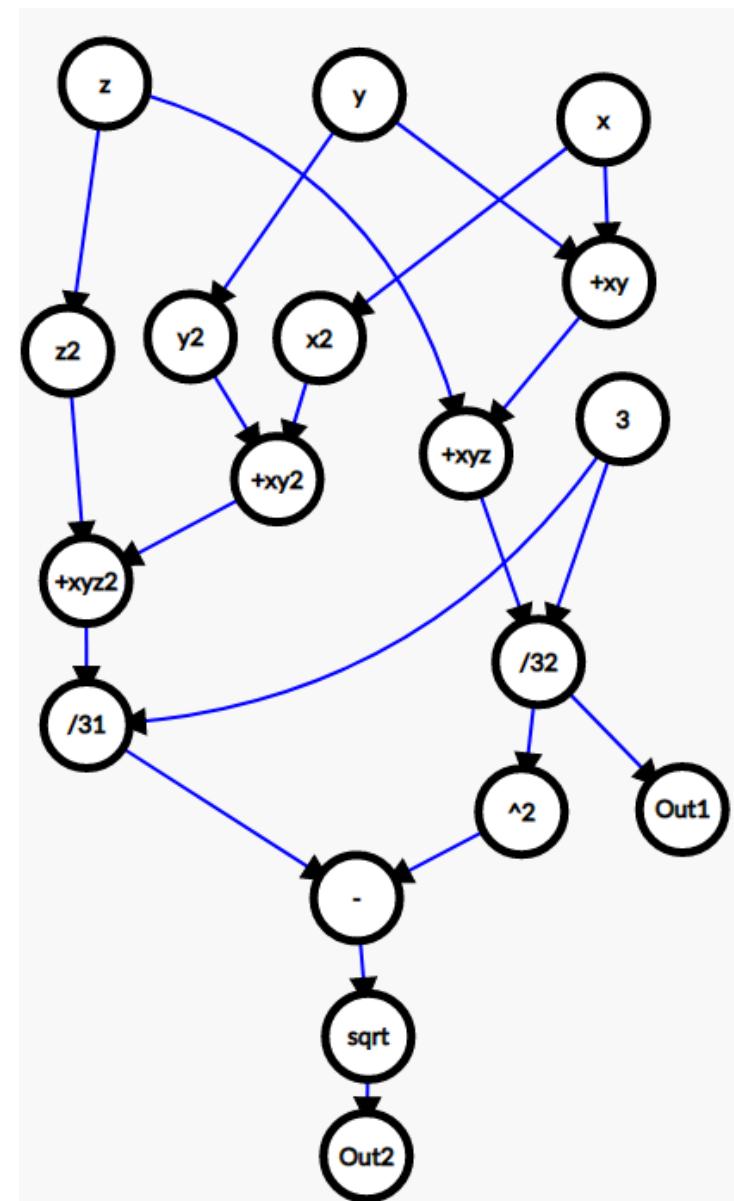
# Regula de activare:

- O instrucțiune este activată (adică executabilă) dacă sunt disponibili toți operanții.
- Observați că în modelul von Neumann, o instrucțiune este activată dacă este indicată de PC.
- Regula de calcul sau regula de declansare, specifică când se execută efectiv o instrucțiune activată.
- O instrucțiune este declanșată (adică executată) când devine activată.
- Efectul declanșării unei instrucțiuni este prelucrarea datelor sale de intrare (operanți) și generarea datelor de ieșire (rezultate).

# Exemplu

- function Med\_Dev:
- Calculeaza media si deviatia standard a trei variabile

```
function Med_Dev(x,y,z: real returns  
    real,real);  
  
let  
    Out1 := (x + y + z)/3;  
    Out2 := SQRT((x**2 + y**2 + z**2)/3 - Out1**2);  
  
in  
    Out1, Out2  
endlet  
endfun
```



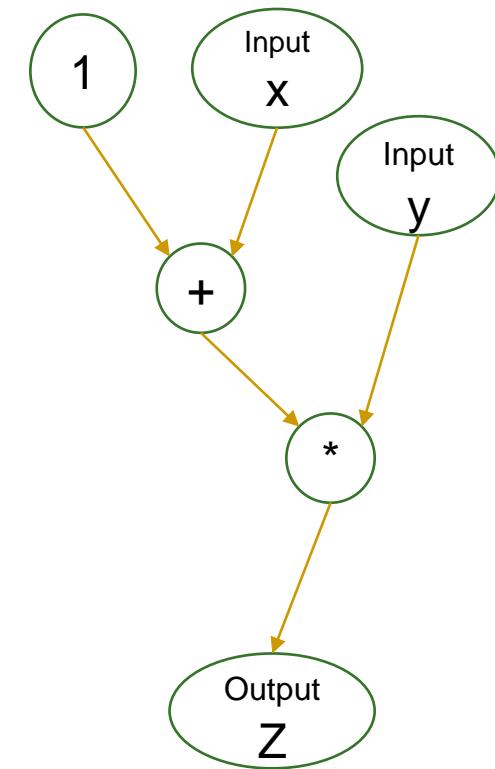
# Modelul Data Flow

- În Model bazat pe flux de date (data flow):
- execuția unei instrucțiuni de prelucrare este efectuată doar de disponibilitatea operandului!
- Nu dispune de PC
- lipsesc cele două caracteristici ale modelului von Neumann care devin blocaje în exploatarea paralelismului

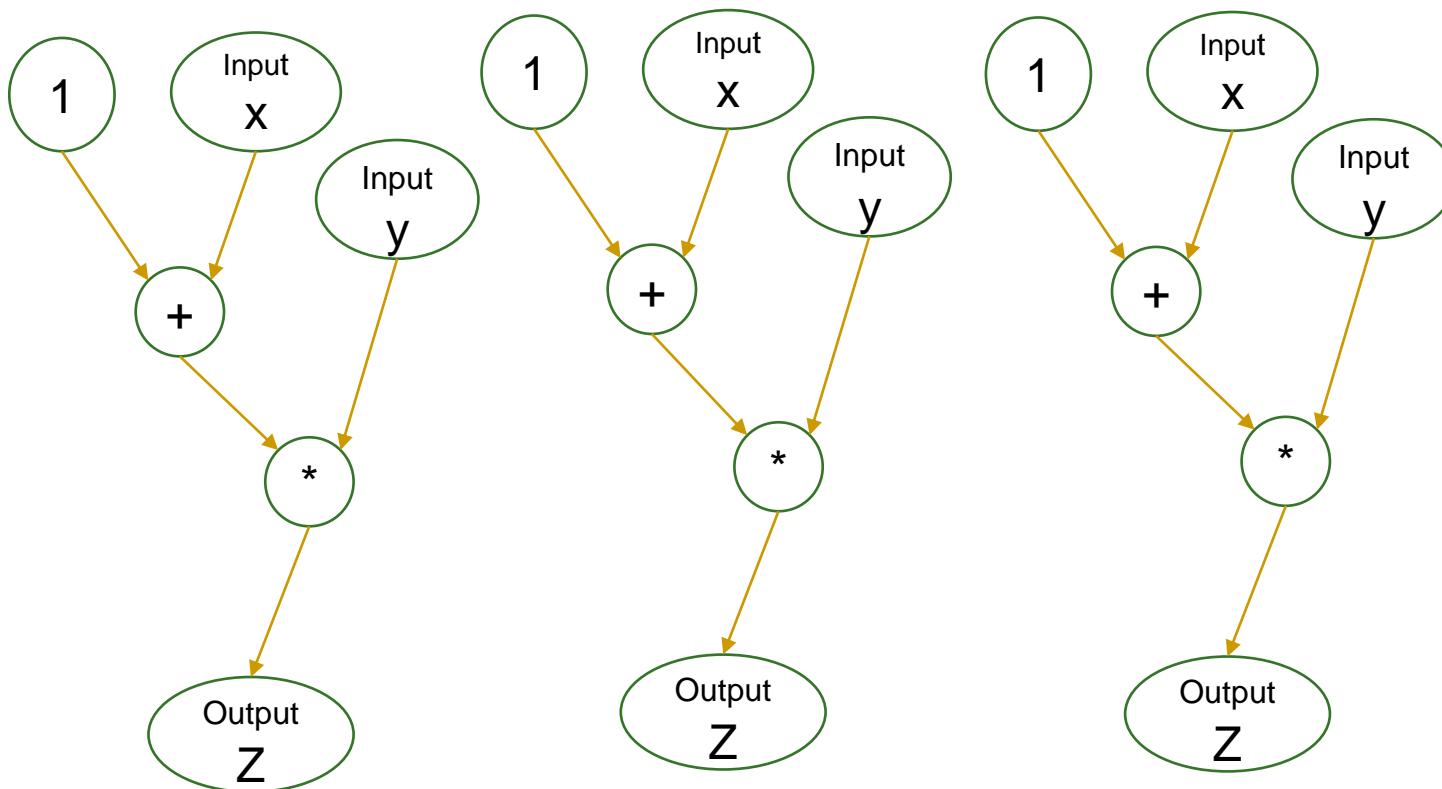
# Exemplu

- Un program de flux de date este compilat într-un graf de flux de date care este un graf direcționat format din noduri, care reprezintă instrucțiuni și arce, care reprezintă dependențe de date între instrucțiuni.
- Grafiul fluxului de date este similar cu un graf de dependență utilizat în reprezentările intermediare ale compilatoarelor.
- În timpul executării programului, datele se propagă de-a lungul arcurilor în pachete de date, numite jetoane (token-uri).
- Acest flux de jetoane permite unele dintre noduri (instrucțiuni) să inceapa execuția și le declanșează.

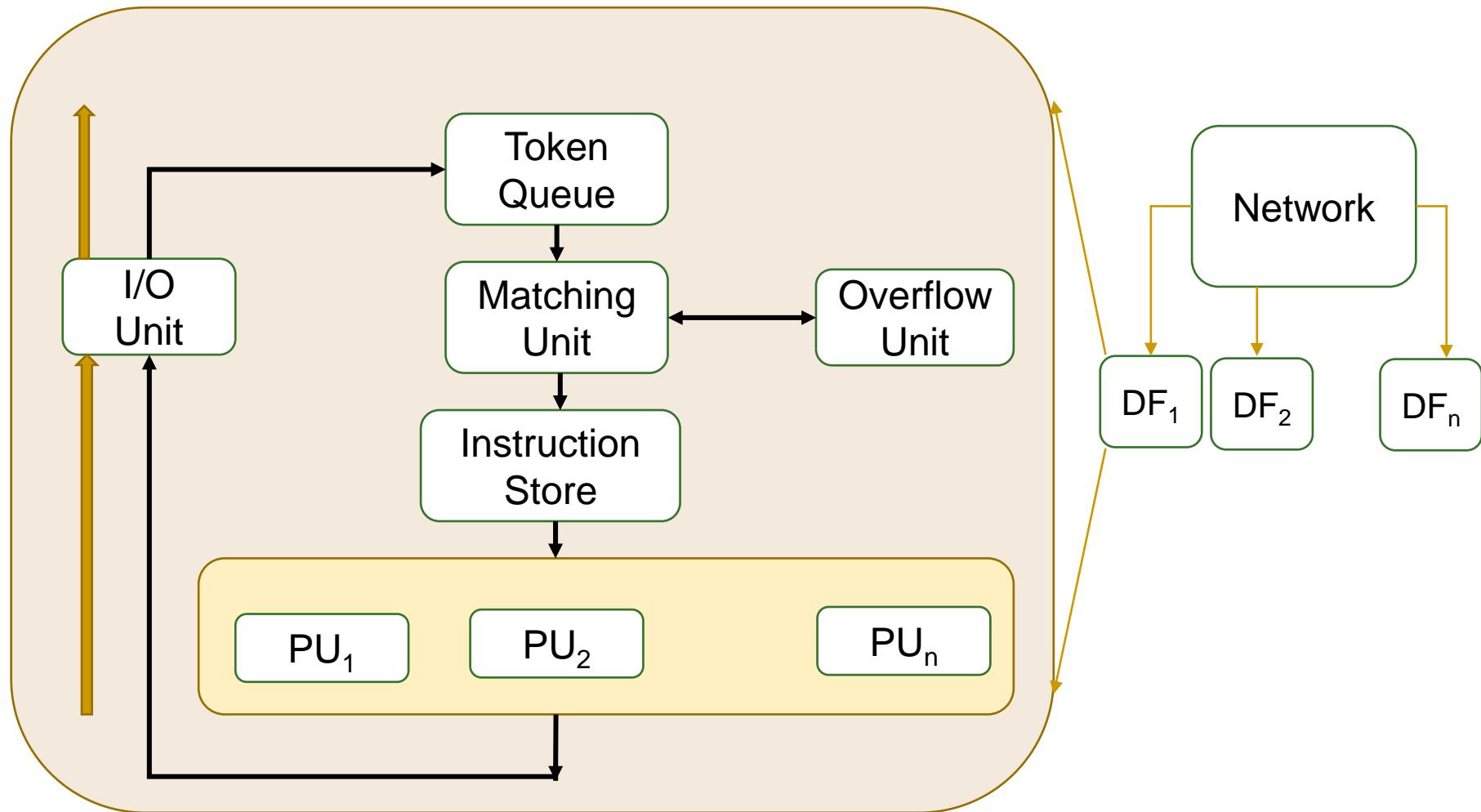
$$Z = (x+1)*y$$



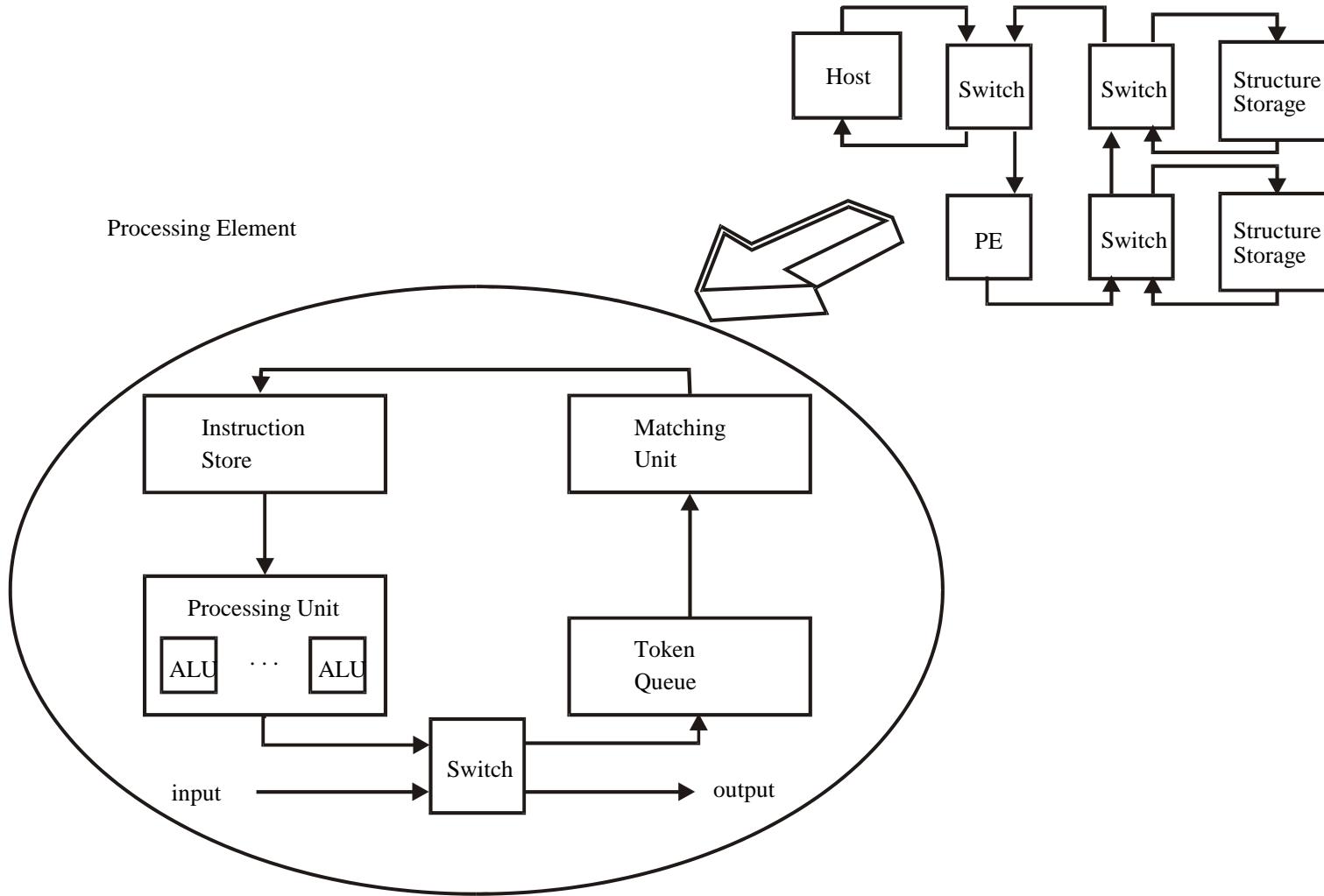
# Executie



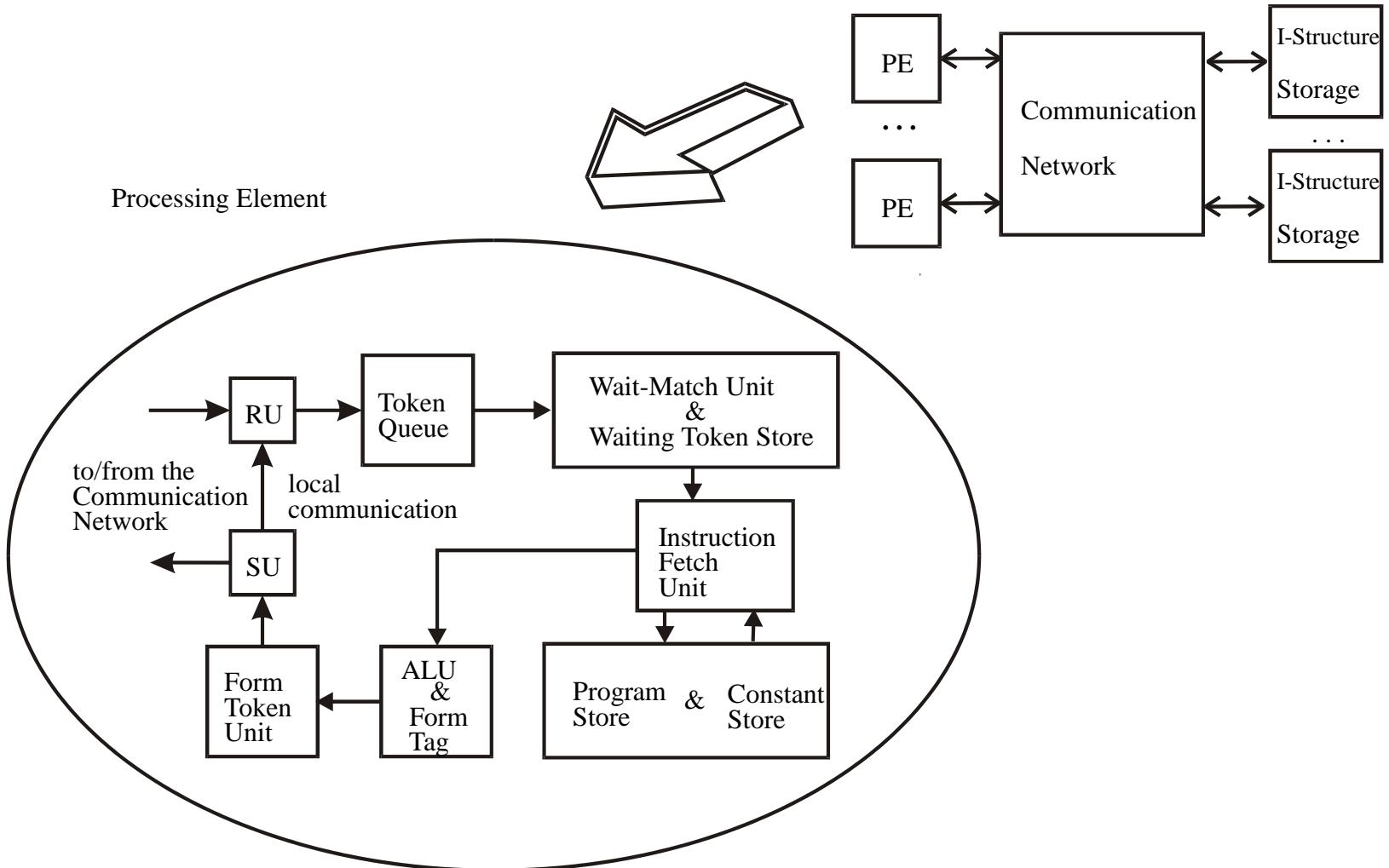
# Manchester Dataflow Computer



# Manchester Dataflow Machine



# MIT Tagged-Token Dataflow Architecture



# Caracteristici importante ale grafului asociat fluxului de date

## ■ Funcționalitate:

- *Evaluarea unui graf de flux de date este echivalentă cu evaluarea funcției matematice corespunzătoare.*

## ■ Compozibilitate:

- *Grafurile fluxului de date pot fi combinate pentru a forma grafuri noi.*

# Dataflow Static

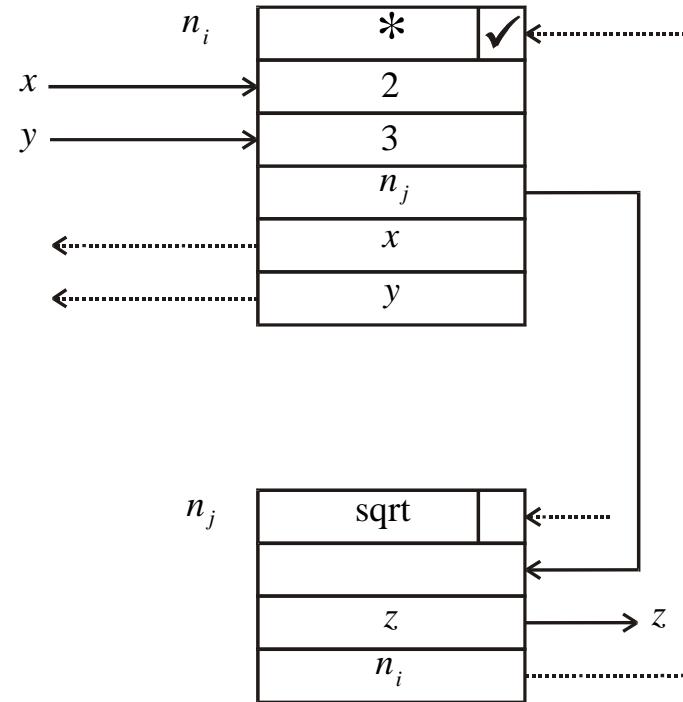
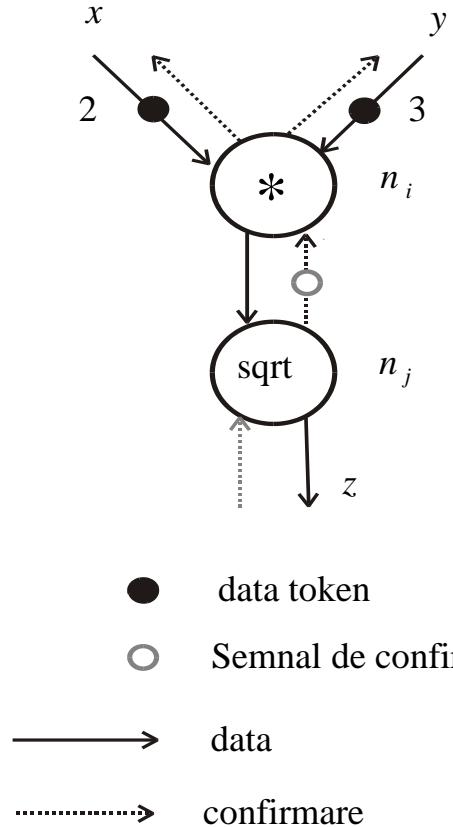
- Un graf de flux de date este reprezentat ca o colecție de şabloane de activitate, fiecare conținând:
  - codul instrucțiunii reprezentate,
  - sloturi de operand pentru păstrarea valorilor de operand,
  - câmpurile de adresă de destinație, referindu-se la sloturile de operand din şabloanele de activitate din secvențe care trebuie să primească valoarea rezultatului.

Deficiențe dataflow static:

- Iterațiile consecutive ale unei bucle pot fi doar executate sub forma de pipeline.
- Datorită token-urilor (jetoanelor) de confirmare, traficul de token-uri este dublat.
- Lipsa suportului de programare care este esențial pentru limbajul de programare modern
  - fără apeluri de procedură,
  - fără recursivitate.

Avantaj: model simplu de implementat

# Dataflow : Template-ul de activitate



# Semale de confirmare

- Abordarea statică a fluxului de date permite cel mult un simbol pe orice arc.
- Nu pot fi token-uri (jetoane) diferite destinate aceleiași destinații.
- Extinderea regulii de bază pentru declansare se face astfel:
  - Un nod activat este declanșat dacă nu există nici un indicativ pe oricare dintre arcurile sale de ieșire.
- Implementarea restricției prin semnale de confirmare (jetoane suplimentare), transmise de-a lungul arcurilor suplimentare de la nodurile consumatoare catre cele producătoare.
- Regula de declansare poate fi modificată la forma sa originală:
  - Un nod este declanșat în momentul în care devine activat.
- *Atentie: restricțiile structurale sunt ignorate presupunând resurse nelimitate!*

# Dataflow Dinamic

- Fiecare iterație de buclă sau invocare de subprogram ar trebui să poată executa în paralel ca o instanță separată a unui subgraf reentrant.
- Replicarea este doar conceptuală.
- Fiecare *token* (jeton) are un *tag* (o etichetă):
  - adresa instrucțiunii pentru care este destinată valoarea particulară a datelor
  - informații de context
- Fiecare arc poate fi văzut ca o *multime* care poate conține un număr arbitrar de Token-uri cu tag-uri diferite.
- Regula de activare și declanșare este:
  - Un nod este activat și declanșat de îndată ce jetoanele cu etichete identice sunt prezente pe toate arcele de intrare.
- Restrictiile structurale sunt ignore!

# Interpretare de tip U

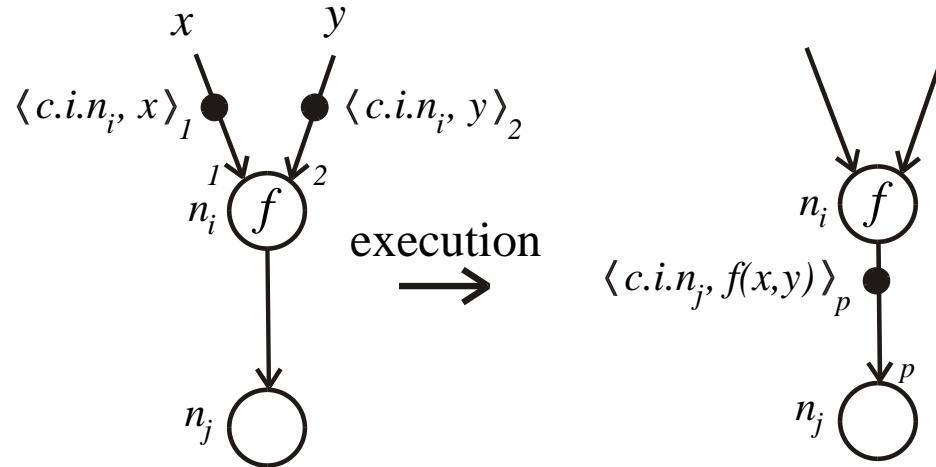
- Fiecare *token* constă dintr-un nume de activitate și date
  - numele activității cuprinde tag-ul.
- Tag (eticheta) are:
  - o adresă de instrucțiune **n**
  - câmpul contextual **c** care identifică în mod unic contextul în care urmează să fie invocată instrucțiunea,
  - numărul de inițiere **i** care identifică iterația de buclă în care are loc această activitate.
- De mentionat faptul ca **c** este el însuși un nume de activitate.
- Deoarece instrucțiunea de destinație poate necesita mai multe intrări, fiecare token poartă și numărul portului său de *destinație p*.
- Token-ul se poate reprezenta prin  $\langle c.i.n, data \rangle_p$   
 $< c\text{-context} . i\text{-iteratia} . n\text{-adresa} , data >$

## Interpretare de tip U

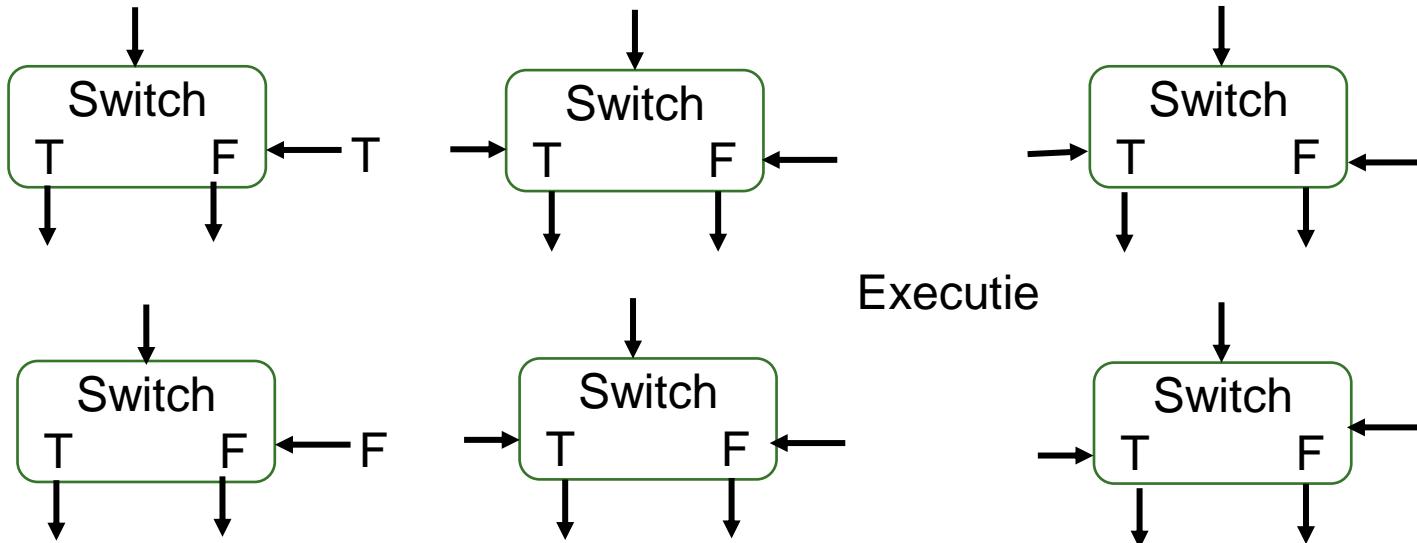
$\langle c.i.n, data \rangle_p < c\text{-context} . i\text{-iteratia} . n\text{-adresa} , data >$

- dacă nodul  $n_i$  îndeplinește o funcție  $f$  și
- dacă portul  $p$  al lui  $n_j$  este destinația lui  $n_i$ ,
- atunci avem

$$in : \{\langle c.i.n_i, x \rangle_1, \langle c.i.n_i, y \rangle_2\} \quad out: \{\langle c.i.n_j, f(x,y) \rangle_p\}$$



# Implementare decizie

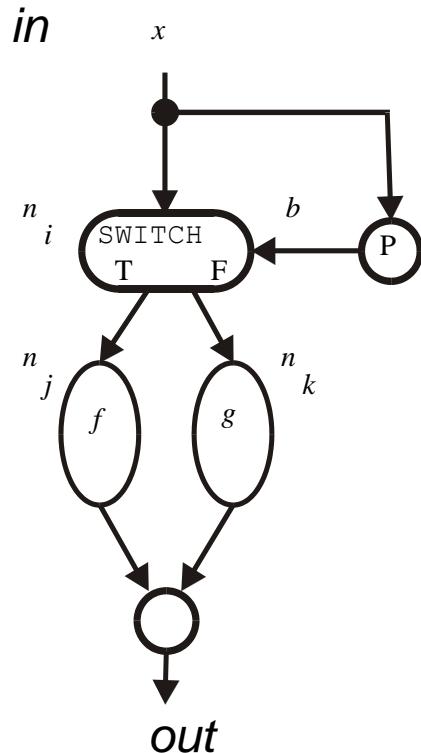


T/F sunt complementare  
0/1 sau 1/0

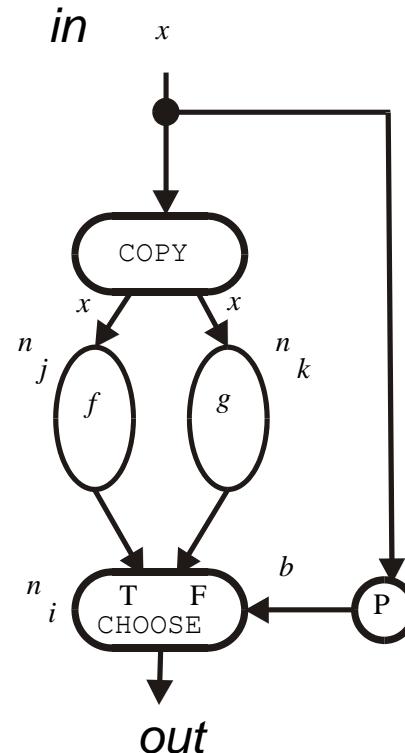
# Implementare ramificatie (decizie)

$\langle c.i.n, data \rangle_p < c\text{-context} . i\text{-iteratia} . n\text{-adresa , data} >$

$$in : \left\{ \langle c.i.n_i, x \rangle_{data}; \langle c.i.n_i; b \rangle_{control} \right\} \quad out : \begin{cases} \langle c.i.n_j, x \rangle if \quad b = T \\ \langle c.i.n_k, x \rangle if \quad b = F \end{cases}$$



Ramificatie  
clasica



Evaluare  
speculativa a  
ramificatiei

# L, L<sup>-1</sup>, D, and D<sup>-1</sup> Operatori pentru implementarea Bucelor

$\langle c.i.n, data \rangle_p < c\text{-context} . i\text{-iteratia} . n\text{-adresa , data} >$

$L: \quad in: \{ \langle c.i.n_i, x \rangle \} \quad out: \{ \langle c'.1.n_k, x \rangle \}, \quad unde \quad c' = \langle c.i.n_i \rangle$

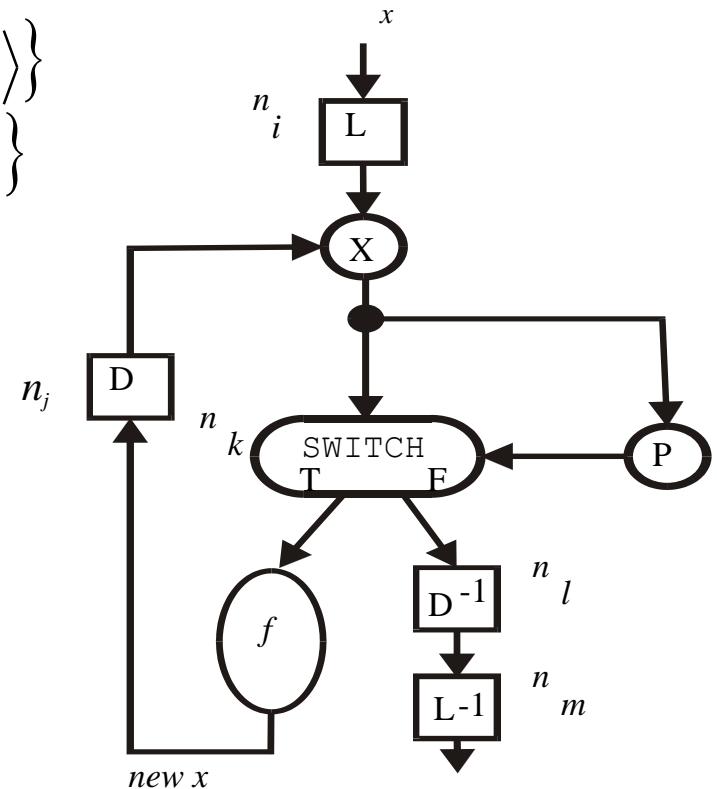
$D: \quad in: \{ \langle c'.j.n_j, x \rangle \} \quad out: \{ \langle c'.j+1.n_k, x \rangle \}$

$D^{-1}: \quad in: \{ \langle c'.k.n_l, x \rangle \} \quad out: \{ \langle c'.1.n_m, x \rangle \}$

$L^{-1}: \quad in: \{ \langle c'.1.n_m, x \rangle \} \quad out: \{ \langle c'.i.n_n, x \rangle \}$

- L: Initializare, context bucla
- D: incrementeaza contor bucla
- D<sup>-1</sup>: initializeaza contor la 1
- L<sup>-1</sup>: reface contextul original

Nota: **c** este el însuși un nume de activitate



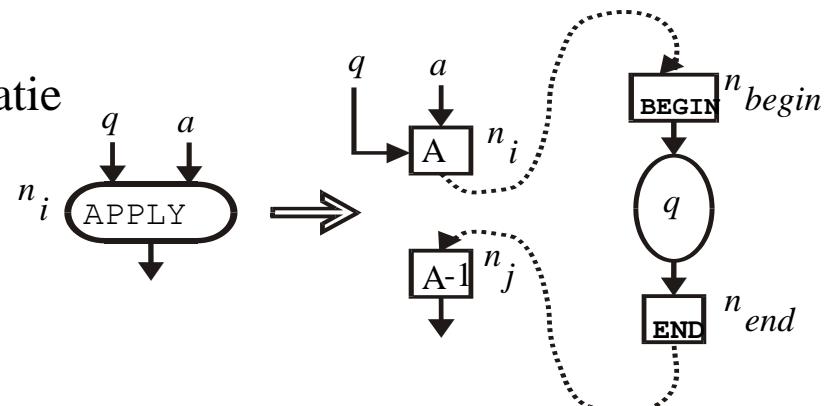
# A, A<sup>-1</sup>, BEGIN, and END Operatori pentru functii

■ A:  $in : \langle c.i.n_i, q \rangle_{func}, \langle c.i.n_i, a \rangle_{arg} \} \quad out : \langle c'.1.n_{begin}, a \rangle \}$

unde  $c' = \langle c.i.n_j \rangle$  si  $n_j$  este adresa operatorului  $A^{-1}$

■ END:  $in : \langle c'.i.n_{end}, q(a) \rangle \} \quad out : \langle c.i.n_j, q(a) \rangle \}$

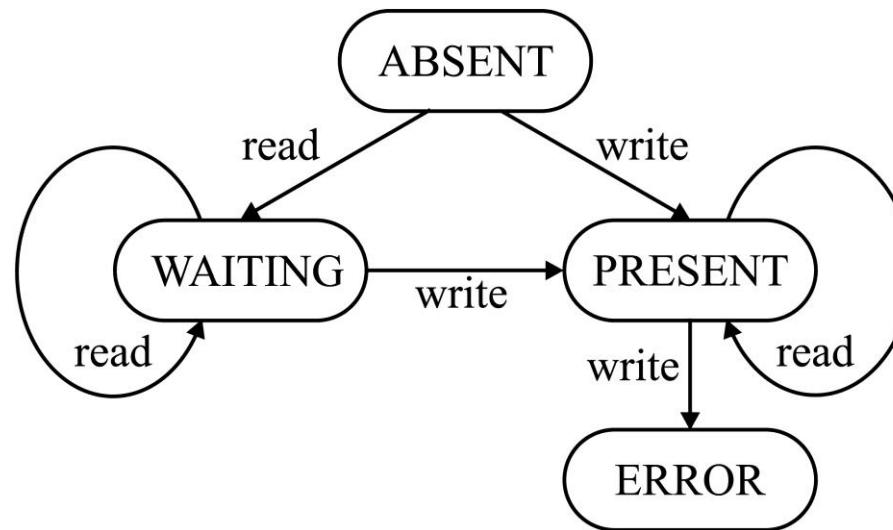
- A: creaza un nou context
- BEGIN: replica token-ul pentru fiecare ramificatie
- END: returneaza rezultatul
- $A^{-1}$ : replica iesirea pentru succesorii



## Structura I

■ Statutul fiecărui element al structurii I poate fi:

- **present**: elementul poate fi citit, dar nu scris,
- **absent**: o cerere de citire trebuie amânată, dar este permisă o operație de scriere în acest element,
- **așteptare**: cel puțin o cerere de citire a elementului a fost amânată.



# structura I

- Sunt definite trei operații elementare pe structurile I:
  - ***alocare***: rezervă un număr specificat de elemente pentru o nouă structură I,
  - ***I-fetch***: recuperează conținutul elementului structura - I specificat (dacă elementul nu a fost încă scris, atunci această operațiune este amânată automat),
  - ***I-store***: scrie o valoare în elementul structura - I specificat (dacă acel element nu este gol, se raportează o condiție de eroare).
- Aceste operații elementare sunt utilizate pentru a construi noduri SELECT și ASSIGN.
- ***I-fetch*** este implementată ca operație de memorie în fază divizată:
  - o cerere de citire emisă unei structuri I este independentă în timp de răspunsul primit și, prin urmare, nu provoacă o așteptare de către PE emitent.

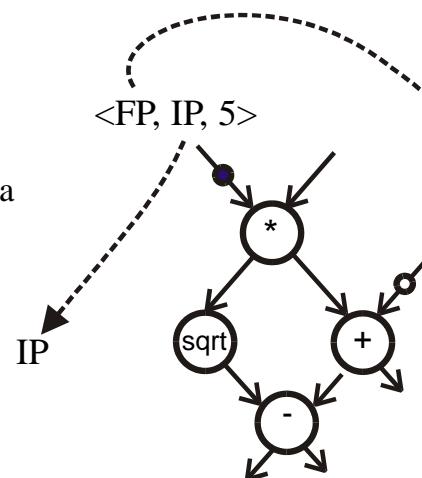
## Implementare cu token explicit

- **Scop:** implementarea eficientă a potrivirii token-urilor.
- **Ideea de bază:** se aloca un frame separat în memoria de frame-uri pentru fiecare iterație de buclă activă sau invocare de subprogram.
- Un **frame** este format din sloturi în care fiecare slot deține un operand care este utilizat în activitatea corespunzătoare.
- Deoarece accesul la sloturi este direct nu este necesară nici o căutare asociativă.

# Explicit Token

Memoria de Instructiuni

Cod operatie	Offset de frame	destinatie stanga	destinatie dreapta
*	2	+1	+2
sqrt			+2
+	3	+1	+5
-	5	+3	+2

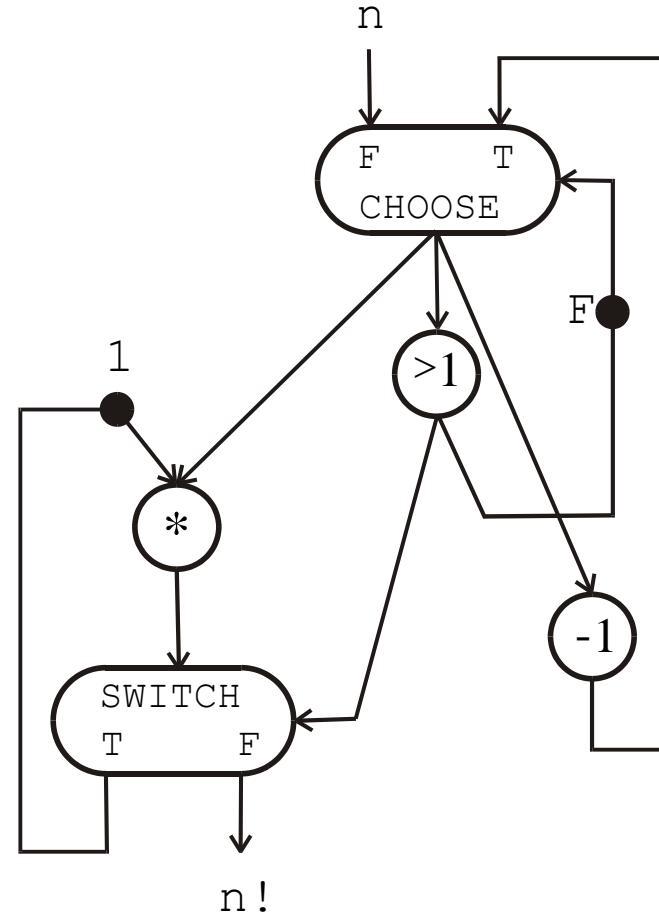


Memoria de frame-uri

bit de prezență	valuare
✓	7.25

## Exemplu calcul factorial n!

```
initial j = n; k = 1
while j > 1 do
    j = j - 1;
    k = k * j;
return k
```



# Exemplu de prelucrare vectori

Input d,e,f

//a,b,c,d,e,f

//vectori de date

$$C_0=0$$

For i=1 to n

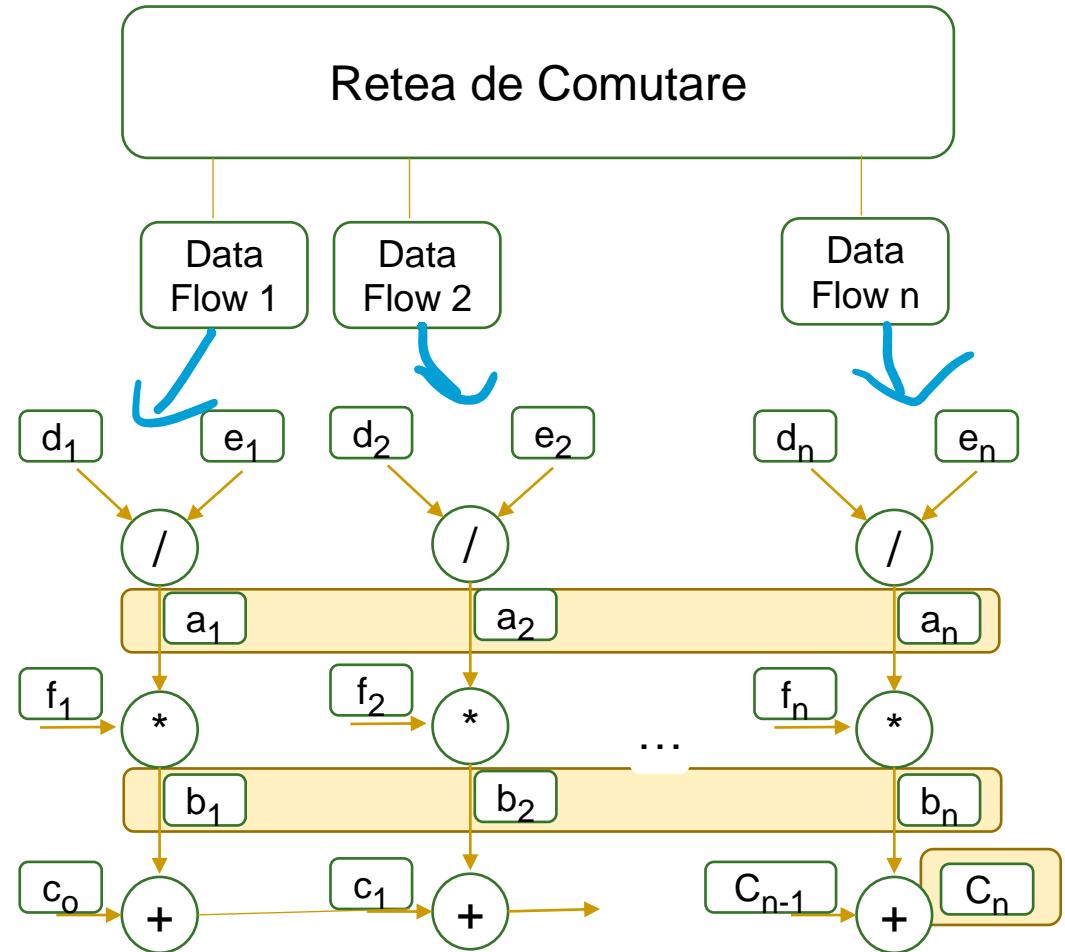
$$a_i=d_i / e_i$$

$$b_i=a_i * f_i$$

$$c_i=b_i+c_{i-1}$$

End for

Output a,b,c<sub>n</sub>



# Ordinea instructiunilor irelevanta

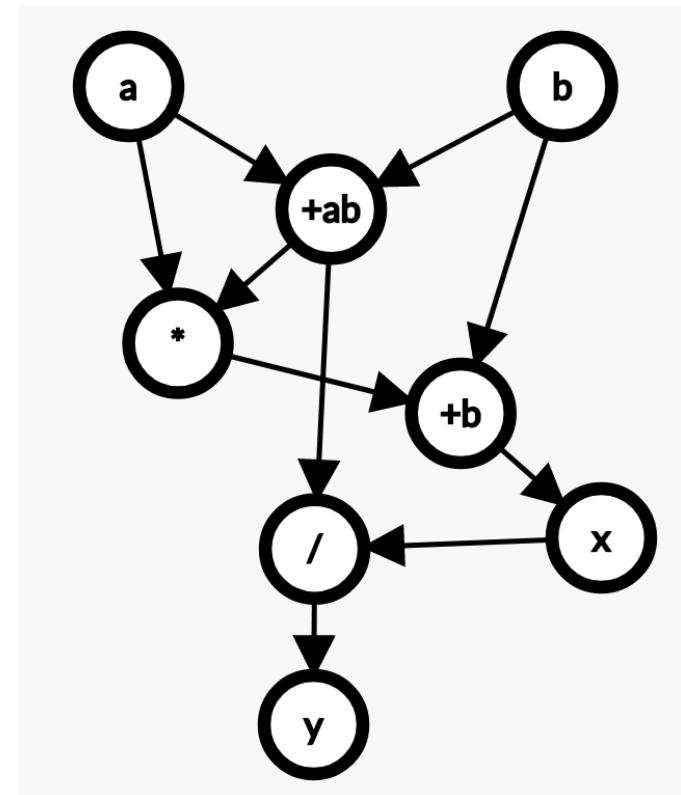
Exemplu:

input a, b

$$y = (a+b) / x$$

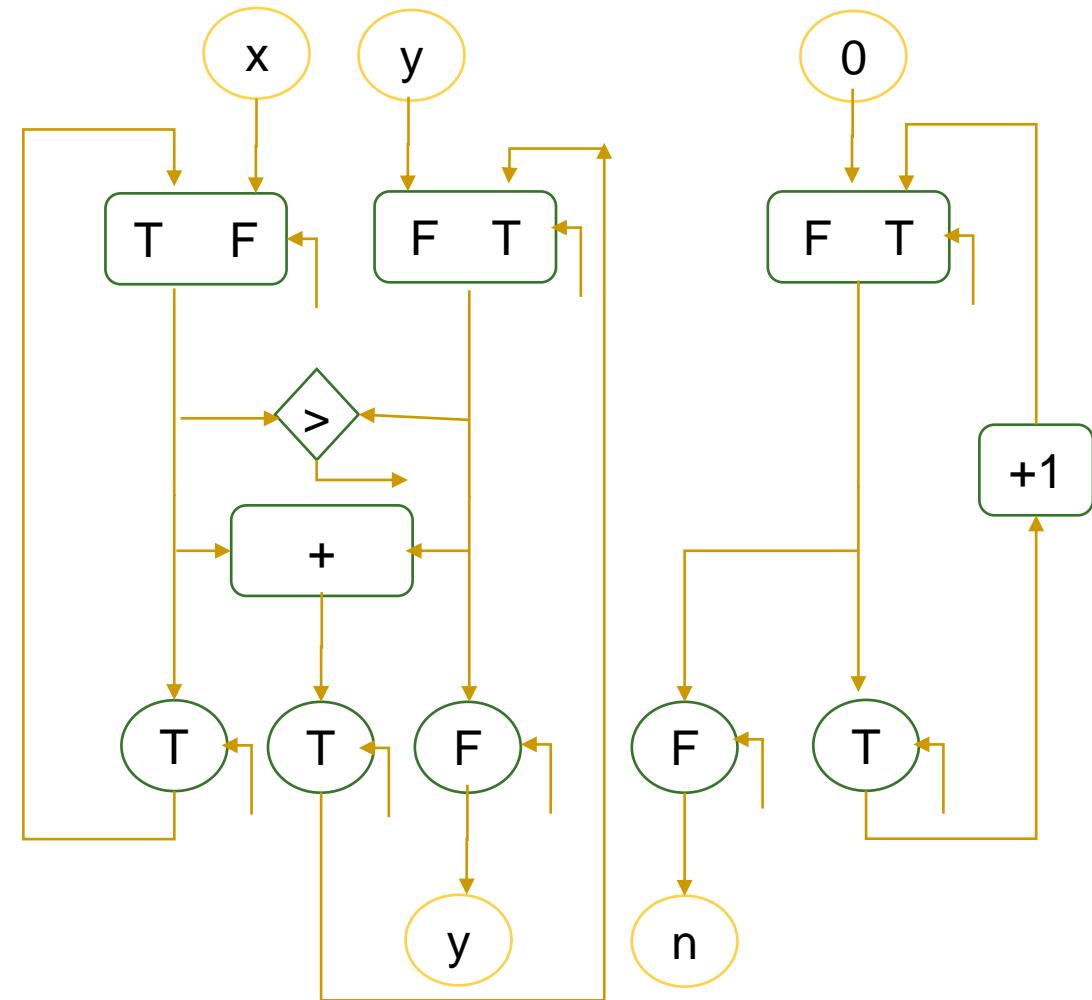
$$x = (a * (a+b)) + b$$

output y, x



# Exemplu de bucla

```
input y, x  
n = 0  
while y < x do  
    y = y + x  
    n = n + 1  
end  
output y, n  
Nu s-a folosit array
```



# Caracteristici de baza

- Executia instructiunilor este bazata pe disponibilitatea datelor
- Datele sunt pastrate in cadrul instructiunii
- Disponibilitatea datelor este verificata de o unitate specializata
- Tokenul asociat datelor este verificat de o unitate specializata
- Executia instructiunilor este asincrona

# Avantaje/ Dezavantaje

## Avantaje

- Potential ridicat de parallelism
- Viteza crescuta de executie
- Usor de integrat comunicatia si sincronizarea

## Dezavantaje

- Overhead-ul mare pentru control
- Manipularea dificila a structurilor de date

# Unde se foloseste data flow in microprocesoarele actuale

Cea mai recentă generație de microprocesoare superscalare afișează o execuție dinamică în afara ordinii, denumită flux de date local sau flux de date micro.

Colwell și Steck 1995, în prima lucrare despre PentiumPro:

Fluxul instrucțiunilor în Arhitectura Intel este prevăzut cu mecanismul prin care instrucțiunile sunt decodificate în micro-operații ( $\mu$ ops), sau serie de  $\mu$ ops, iar aceste  $\mu$ ops sunt plasate într-un set de microoperatii executabile speculative, care se vor executa în afara ordinii de operații (out of order), Vor fi executate pe principiul data flow (când operanții sunt gata) și utilizate în ordinea programului sursă.

Fiecare instrucțiune este gata să fie executată imediat ce toți operanții sunt disponibili.

# Exemplu care evidențiază diferite structuri elementare

Considerăm adunarea a doi vectori

$$Z = X + Y$$

$$z_i = x_i + y_i$$

Fiecare componentă are mantisa și exponent (caracteristica)

$$x_i = x_{ic} \ x_{im}$$

Operatia de adunare are 4 operatii elementare organizate in pipeline

- Compararea caracteristicilor
- Deplasarea mantisei (daca este cazul)
- Adunarea mantiselor
- Normalizarea rezultatului
- Aceste operatii sunt independente – structura pipeline
- Consideram aceste operatii dureaza  $\sim t$  (pentru simplificare)

# **CONTROLUL PROCESELOR – SARCINILOR CONCURENTE**

# \*Obiective

- Acest capitol este dedicat tratării separate a celor mai reprezentative aspecte legate de controlul și gestiunea sarcinilor concurente.
- Se pune un accent deosebit pe prezentarea
  - unor algoritmi de excludere mutuală
    - context centralizat
    - context distribuit
  - analiza situațiilor de blocare
  - Sincronizarea sarcinilor.
- Problemele tratate în acest capitol asigură baza teoretică pentru analiza sau dezvoltarea unor sisteme software care să lucreze
  - în regim de multiprogramare
  - multiprelucrare
- atât pentru sisteme monoprocesor, cât și pentru sisteme cu prelucrare paralelă

# Definiții și concepte de bază

- Activitățile paralele se pot desfășura atât în sisteme de tip **MIMD**, **SIMD** cât și în cadrul sistemelor convenționale, de tip **SISD**.
- Un exemplu de activități paralele într-un sistem de calcul conventional de tip **SISD**, Von Neumann îl constituie efectuarea simultană a operațiilor din **UCP**, cu cele din subsistemul de intrări/ieșiri prin **DMA** sau canal de I/E.
- De asemenea, activități paralele se întâlnesc la nivelul microoperațiilor care constituie o microinstructiune în cadrul unei unități de comandă microprogramată.

În general pentru creșterea performanțelor unui sistem de calcul se prevăd mai multe procesoare de diferite tipuri, cum ar fi:

- procesoare centrale de prelucrare;
- procesoare de intrări/ieșiri;
- procesoare specializate (coprocesoare matematice, coprocesoare neurale, etc).
- Pe de altă parte mai multe programe sau părți ale unui aceluiași program pot fi executate în paralel.
- Comunicația între acestea poate fi asigurată printr-un schimb de mesaje.
- Chiar dacă există un singur procesor (nu o structură **MIMD**), care este partajat între mai multe programe, vom admite că, din punct de vedere logic aceste programe se execută în paralel disputându-și accesul la diferite resurse fizice ale sistemului.

## Sarcina -proces secvențial

**Sarcina - proces secvențial** este o activitate care constă din execuția, pe un procesor, a unui grup de instrucțiuni (de obicei indivizibil) asociat unui set de date specificat.

- Deși pare că fiecare sarcina are procesorul și datele sale, în realitate mai multe sarcini pot utiliza împreună:
  - un procesor
  - o secvență de cod
  - o structură de date.

O **sarcină** poate fi specificată prin relația sa cu exteriorul:

- intrările necesare;
- funcția executată;
- ieșirile generate;
- starea la un moment dat;
- timpul de execuție.

- Comportarea intrinsecă, internă, nu va fi specificată decât în situații excepționale, pentru a facilita înțelegerea interacțiunii cu celelalte sarcini.
- Une sarcini i se asociază două evenimente:
  - **SI** - inițiere sarcină;
  - **SF** - terminare sarcină.
- Dacă notăm cu  $t(\cdot)$  timpul de apariție a unui eveniment, vom considera că:  
 $t(SF) - t(SI) \neq 0$  și finit

cu condiția ca toate resursele necesare execuției lui S să fie disponibile în momentul  $t(SI)$

Vom considera sarcinile **neinterpretate**, ceea ce este echivalent cu faptul că pentru o mulțime dată de sarcini să ne intereseze secvențele evenimentelor de inițiere și terminare fără să ne preocupe particularitățile de execuție ale sarcinilor.

În ceea ce privește granularitatea unei sarcini, aceasta poate fi:

- program -JOB;
- modul de program - task;
- procedură - sarcina;
- secvența - thread
- instrucție;
- microinstrucție;
- microoperare.

Sistemul de calcul în care se execută sarcinile constă dintr-o mulțime de resurse:

- procesoare;
- dispozitive de I/E;
- biblioteci de programe;
- proceduri;
- variabile;
- fișiere de date, etc.

și este caracterizat de o mulțime  $\Sigma$  de stări.

În funcție de granularitatea sarcinilor, procesorul poate fi:

- calculator universal;
- UCP într-un sistem multiprocesor;
- UAL;
- unitate de comandă;
- microsevențiator într-o unitate de comandă.

Atât posibilitățile sistemului, cât și funcțiile sarcinilor date sunt reprezentate prin tranziții permise în  $\Sigma$ :  $s_0 s_1 s_2 \dots s_n$  – spațiul stării, de forma  $s_i \rightarrow s_j$ ,  $(s_i, s_j \in \Sigma)$  definite pentru inițierea și terminarea sarcinilor.

**Inițierea** unui sarcina corespunde unei tranziții de stări care reflectă:

- preluarea resurselor necesare;
- inițializarea stării sistemului (a resurselor);
- citirea parametrilor de intrare.

Terminarea unei sarcini corespunde unei tranziții de stări care reflectă:

- eliberarea sau suspendarea temporară a utilizării resurselor;
- scrierea parametrilor de ieșire;
- dacă valori ale parametrilor sau stări interne sunt asociate cu resursele, terminarea unei sarcini trebuie însoțită de valoarea stării resurselor.

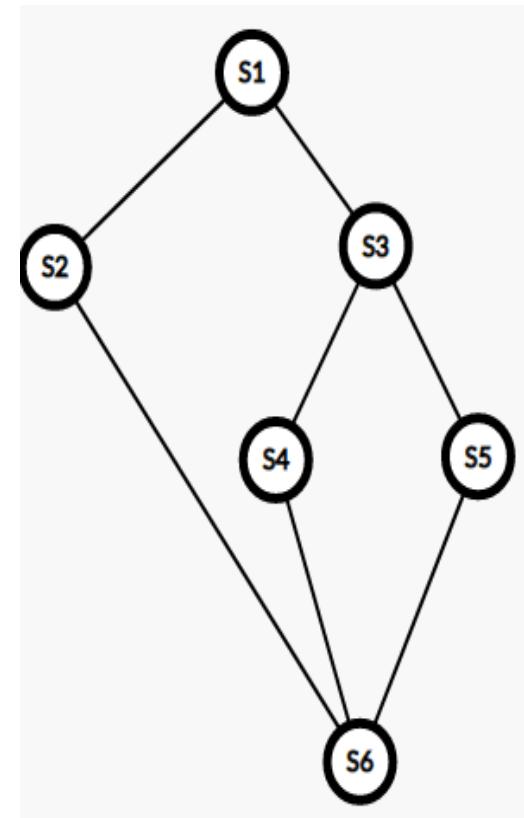
Fie  $S = \{S_1, S_2, \dots, S_n\}$  o mulțime de sarcini iar  $\prec$  o relație de ordine parțială, nereflexivă, pe  $S$  (denumită și relație de precedență pe mulțimea de sarcini  $S$ ).

Perechea  $C = (S, \prec)$  este denumită **sistem de sarcini**.

- Relația  $\prec$  reprezintă precedențele în execuția sarcinilor.

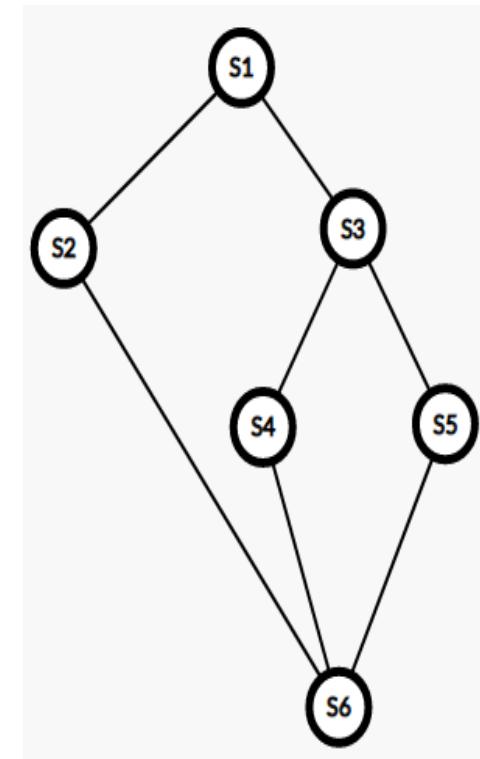
Astfel,  $S_i \prec S_j$

- Înseamnă că  $S_i$  trebuie terminată înainte de inițierea lui  $S_j$ .



Sarcinile din  $S$  sunt independente dacă pentru  $C = (S, \prec)$  avem  $\prec = \Phi$ ; unde  $\Phi$  este mulțimea vidă.

- O reprezentare grafică convenabilă a unui sistem de sarcini se obține utilizând grafuri aciclice direcționate, GAD.
- Sracinile sunt reprezentate prin noduri iar relația de precedență prin mulțimea arcelor.
- Relația de precedență dată de mulțimea arcelor poate fi definită ca fiind cea mai mică relație în  $S$  a cărui închidere prin tranzitivitate este  $\prec$ .
- Deoarece  $\prec$  reprezintă o ordonare în timp a sarcinilor, vom spune că două sarcini  $S_i$  și  $S_j$ , pot fi concurente dacă și numai dacă ele sunt independente, adică între ele nu există o relație de precedență.



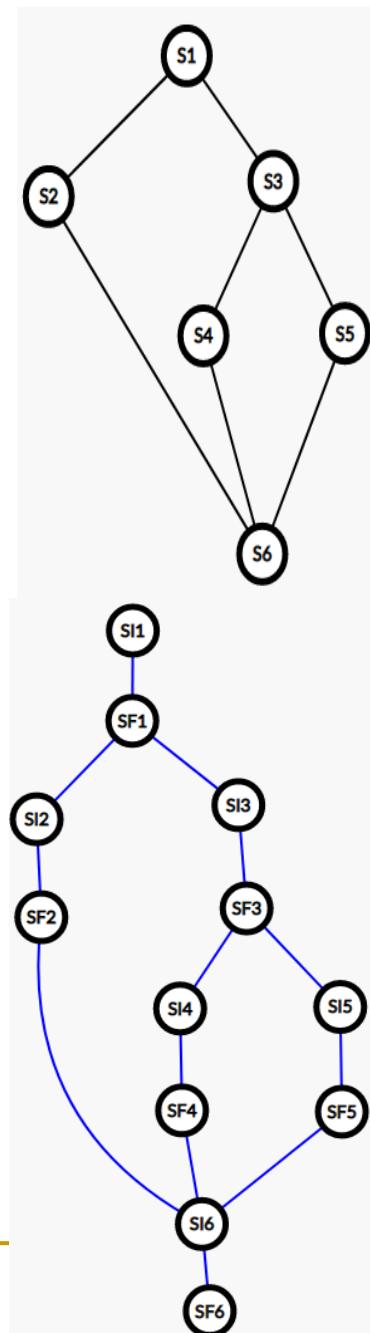
## \*Exemplu:

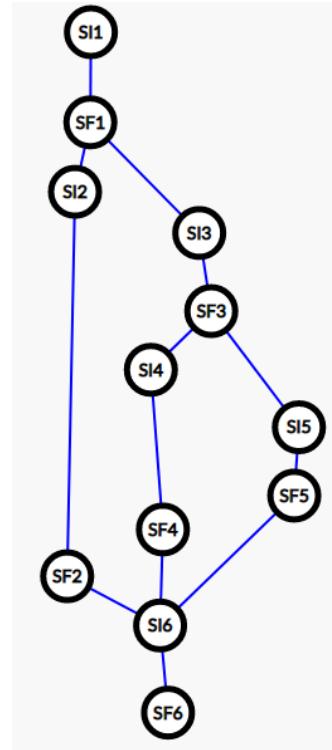
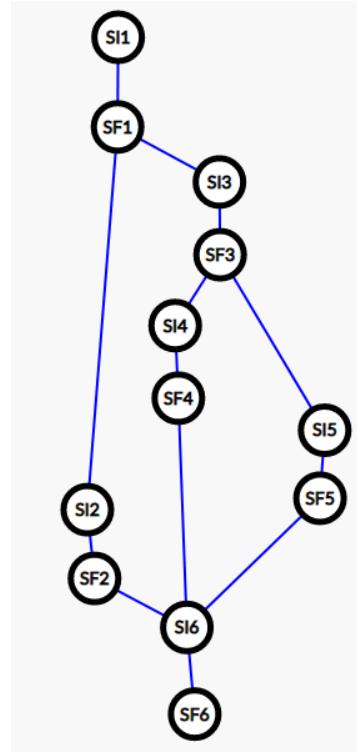
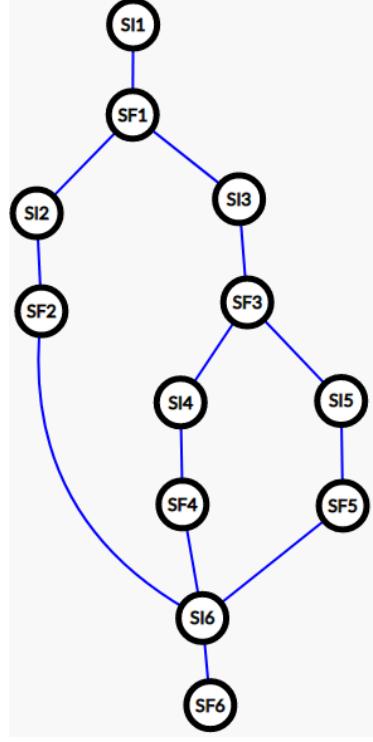
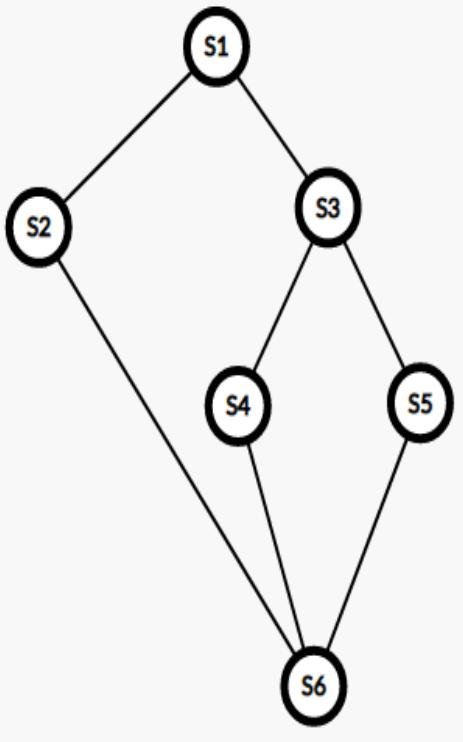
- Două sarcini  $S_i$  și  $S_j$  sunt **independente** dacă  $S_i$  nu este nici predecesor, nici succesor al lui  $S_j$ .
- O sarcină  $S$  din sistemul de sarcini  $C$  este pe **nivelul k**, sau are nivelul k, dacă cea mai lungă cale de la  $S$  la un nod terminal are lungimea k.
- Un nod terminal are nivelul 1.

O **secvență de execuție** a unui sistem de n sarcini,  $C = (S, \prec)$  este orice sir  $\alpha = a_1 a_2 \dots a_{2n}$  de evenimente de inițiere și terminare a sarcinilor cu respectarea relațiilor de precedență impuse de  $\prec$ .

## Altfel spus:

- pentru orice  $S \in S$ ,  $SI - S$  initiat și  $SF - S$  terminat apar o singură dată în  $\alpha$  ;
- dacă  $a_i = SI$  și  $a_j = SF$ , atunci  $i < j$  ;
- dacă  $a_i = SF_k$  și  $a_j = SI_l$ , cu  $S_k \prec S_l$  atunci  $i < j$  ;





Sistem de sarcini  
reprezentat prin  
graf

Exemple de secvențe de execuție pentru sistemul de sarcini

- $C = (S, \prec)$
- $S = \{S_1, S_2, S_3, S_4, S_5, S_6\}$
- reprezentat prin graful
- $\alpha_1 = S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6$
- $\alpha_2 = S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6$
- $\alpha_3 =$

# secvență de execuție parțială

O secvență de execuție parțială este orice prefix al unei secvențe de execuție.

Mulțimea secvențelor de execuție reprezintă mulțimea tuturor secvențelor de evenimente (de inițiere și terminare) care conduc la finalizarea lui **C** respectându-se relația  $\leq$ .

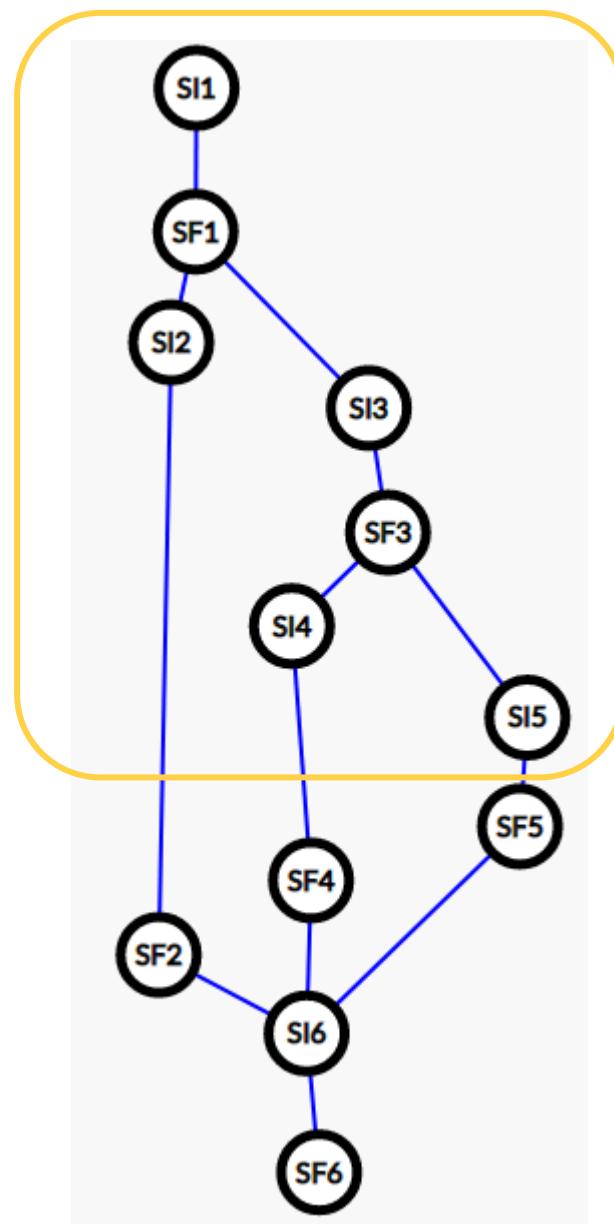
Un sarcină **S** este activă după o secvență de execuție parțială

$\alpha_p = a_1 a_2 \dots a_k$ , dacă există  $i \leq k$  astfel că  $a_i = SI$ , dar pentru orice  $j \leq k$ ,  $a_j \neq SF$

Ex

$\alpha_p = SI_1 SF_1 SI_2 SI_3 SF_3 SI_4 SI_5$

- $\alpha_2 = SI_1 SF_1 SI_2 SI_3 SF_3 SI_4 SI_5 SF_5 SF_4 SF_2 SI_6 SF_6$



Fie  $\Sigma$  spațiul stărilor pentru un sistem de sarcini.

- Secvența stărilor corespunzătoare secvenței de execuție

$$\alpha = a_1 \ a_2 \dots \ a_{2n}$$

este dată prin:

$$\sigma = s_0 s_1 s_2 \dots \ s_{2n}, \quad s_j \in \Sigma, \quad 0 \leq j \leq 2n,$$

iar  $s_0$  este starea inițială dată.

- Tranziția stării definită de evenimentul  $a_i$  este reprezentată de:

$$s_{i-1} \xrightarrow{} s_i$$

- O tranziție pentru o pereche de stări-eveniment  $(s, a)$  va fi definită numai dacă evenimentul a poate să apară când sistemul este în starea  $s$ .

Un sistem de sarcini reprezentat printr-un graf cu un singur nod inițial și un singur nod terminal se numește **închis**.

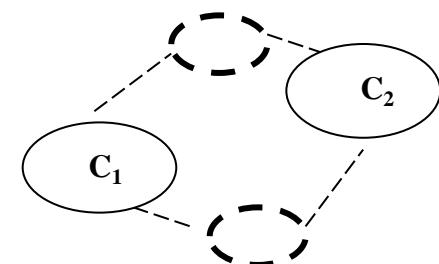
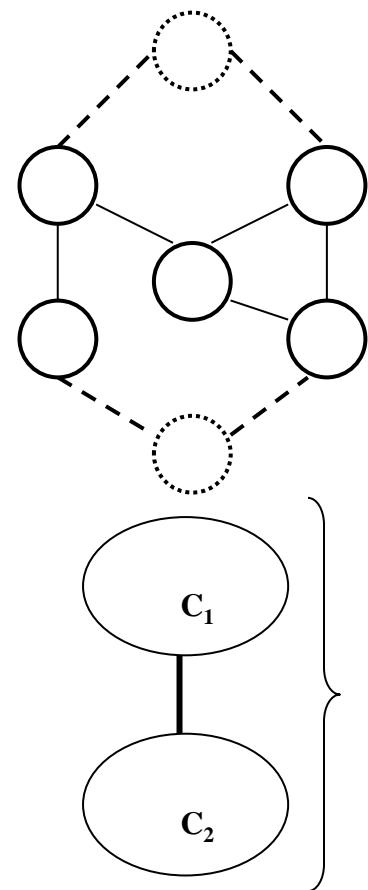
- Dacă sistemul de  $n$  sarcini nu este închis se pot prevedea sarcinile  $S_0$  și  $S_{n+1}$  inoperante pentru a obține un sistem închis.
- Două sisteme de sarcini închise  $\mathbf{C}_1$  și  $\mathbf{C}_2$  pot fi concatenated ( $\mathbf{C}_1 \cdot \mathbf{C}_2$ ), graful asociat obținându-se prin introducerea unui arc de la nodul terminal al lui  $\mathbf{C}_1$  la nodul inițial al lui  $\mathbf{C}_2$ .
- O secvență de execuție a lui  $(\mathbf{C}_1 \cdot \mathbf{C}_2)$  este orice sir

$$\alpha = \alpha_1 \cdot \alpha_2$$

de secvențe de execuție  $\alpha_1$  a lui  $\mathbf{C}_1$ ,  $\alpha_2$  a lui  $\mathbf{C}_2$ .

**Combinarea paralelă** a două sisteme de sarcini  $\mathbf{C}_1$  și  $\mathbf{C}_2$ ,  $\mathbf{C}_1 \parallel \mathbf{C}_2$ , care nu au sarcini în comun, ( $\forall S_i \in \mathbf{C}_1$  și  $\forall S_j \in \mathbf{C}_2$ ,  $S_i \neq S_j$ ) constă în simpla lor reuniune.

- Graful lui  $\mathbf{C}_1 \parallel \mathbf{C}_2$  are ca subgrafuri disjuncte grafurile lui  $\mathbf{C}_1$  și  $\mathbf{C}_2$ .



Orice sarcină din  $C_1$  este independentă de orice sarcina din  $C_2$ .

Dacă  $a = a_1 a_2 \dots a_{2m}$  și  $b = b_1 b_2 \dots b_{2n}$  sunt secvențe de execuție pentru  $C_1$  respectiv pentru  $C_2$ , o secvență de execuție a lui  $C_1 \mid C_2$  se formează astfel:

$$C = C_1 C_2 \dots C_{2(m+n)}$$

unde :

$$c_1 = a_1 \sim b_1 \quad (\text{fie } a_1 \text{ fie } b_1)$$

dacă  $a_1 a_2 \dots a_i$  și  $b_1 b_2 \dots b_j$  sunt elemente ale lui

$$c_1 c_2 \dots c_{i+j} \text{ cu } i+j < 2(m+n)$$

atunci :

$$c_{i+j+1} = a_{i+1} \sim b_{j+1} \quad (\text{fie } a_{i+1} \text{ fie } b_{j+1})$$

Combinarea paralelă a secvențelor de execuție apare frecvent în proiectarea sistemelor, în special a sistemelor cu prelucrare paralelă, uneori sub o formă diferită de cea prezentată anterior în sensul că unele sisteme implică secvențe repetitive.

- Sistemele care conțin secvențe repetitive pot fi modelate fie prin grafuri aciclice fie prin grafuri ciclice.
- Astfel, se pot modela  $k$  cicli ai unui sistem **C** prin concatenarea  
$$\mathbf{C}^k = \mathbf{C}_1 \cdot \mathbf{C}_2 \cdot \dots \cdot \mathbf{C}_k$$
- O secvență de execuție  $\alpha$  a lui  $\mathbf{C}^k$  este de forma:

$$\alpha = \alpha_1 \alpha_2 \dots \alpha_k$$

unde  $\alpha_i$  este o secvență de execuție a lui **C** pentru iteratăia  $i$ .

Un sistem de sarcini se numește **ciclic** dacă este de forma  $\mathbf{C}^k$  pentru  $k > 1$ , sau dacă este o combinare paralelă de sisteme închise în care cel puțin unul este ciclic.

# Proprietatea de determinare a sistemelor de sarcini

## Definiție:

Sistem determinat (sistem de sarcini funcțional) este un sistem format din sarcini care se execută în paralel și cooperează pentru realizarea unor operații logice și de calcul, care produce același rezultat indiferent de durată de execuție a fiecărei sarcini independente, sau de ordinea în care acestea se executa.

## Definiție:

Un sistem de sarcini este nedeterminat dacă rezultatele produse de sarcini independente depind de ordinea în care acestea se executa.

- Exemplu de sistem nedeterminat:

$S_1: R_1 \leftarrow \text{BUSFN} (M; \text{DCD (ADR)})$

$S_2: M * \text{DCD (ADR)} \leftarrow R_2$

$S_1$  - citește din locația de memorie de la adresa ADR;

$S_2$  - scrie în locația de memorie de la adresa ADR.

$\alpha_1 = SI_1 SF_1 SI_2 SF_2$

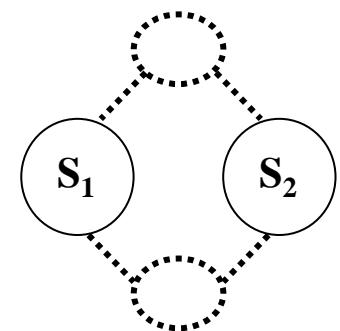
**se va citi valoarea  $R_1$**  și în celula de memorie va ramane valoarea  $R_2$

$\alpha_2 = SI_2 SF_2 SI_1 SF_1$

în celula de memorie va ramane valoarea  $R_2$  și se **va citi valoarea  $R_2$**

Produc rezultate diferite

- Nedeterminarea se poate rezolva introducând o relație de precedență adecvată între sarcinile care erau independente.



- Pentru a stabili condițiile necesare și suficiente ca un sistem să fie determinat, vom considera un model simplificat în care sistemul fizic este privit ca o mulțime ordonată de locații de memorie:

$$\mathbf{M} = ( M_1, M_2, \dots, M_m )$$

ce conțin orice valoare dintr-o mulțime de valori  $V$ .

- Stările sistemului vor fi definite de valorile care se găsesc în memorie la un moment dat:

De exemplu dacă:

$$a = a_1 a_2 \dots a_{2n} \text{ și}$$

$$\sigma = \sigma_0 \sigma_1 \sigma_2 \dots \sigma_{2n}$$

reprezintă o secvență de execuție, respectiv secvența de stări corespunzătoare, iar  $M_i(k)$  reprezintă valoarea din celula  $M_i$  imediat după evenimentul  $a_k$  starea sistemului va fi:

$$\mathbf{s}_k = [ M_1(k), M_2(k), \dots, M_m(k) ]$$

Mulțimea tuturor stăriilor poate fi definită astfel:

$$\Sigma = V^m \quad V^m = [V \times V \times \dots \times V], \text{ produs cartezian.}$$

- Pentru a formaliza efectul execuției unei sarcini asupra celulelor de memorie vom considera că fiecărei sarcini  $S$  i se asociază funcția:

$$f_S : V^d \longrightarrow V^r$$

unde:

$d = |D_S|$  cardinalul domeniului valorilor de intrare,  $D_S$ ;

$r = |R_S|$  cardinalul domeniului valorilor de ieșire,  $R_S$ .

- Pentru o stare inițială  $s_0$  și o secvență de execuție  $\alpha$ , secvența corespunzătoare de stări:  $\sigma = s_0 s_1 \dots s_{2n}$  este definită în felul următor :

Fie  $D_S = (M_{x1}, M_{x2}, \dots, M_{xd})$  domeniul valorilor de intrare;

$R_S = (M_{y1}, M_{y2}, \dots, M_{yr})$  domeniul valorilor de ieșire;

dacă  $a_{k+1} = SI$ , atunci  $M_i(k+1) = M_i(k)$ ,  $1 \leq i \leq m$ ;

dacă  $a_{k+1} = SF$ , și  $a_l = SI$ ,  $l \leq k$ , atunci stările locațiilor de memorie din domeniul de valori al lui  $S$  la momentul  $k+1$  sunt:

$$[M_{y1}(k+1), M_{y2}(k+1), \dots, M_{yr}(k+1)] = f_S(M_{x1}(l), M_{x2}(l), \dots, M_{xd}(l)) \quad \text{și} \\ M_i(k+1) = M_i(k) \quad \text{pentru } (\forall) M_i \notin R_S$$

- Altfel spus, dacă  $a_{k+1}$  este un eveniment de inițiere nu apare o schimbare a stării, adică  $\delta_{k+1} = \delta_k$ .
- Dacă însă  $a_{k+1}$  este un eveniment de terminare a lui S,  $\delta_{k+1} \neq \delta_k$  doar în domeniul  $R_s$ , noile valori din  $R_s$  fiind determinate de  $f_S$  pe baza valorilor din domeniul de definiție din momentul imediat precedent inițierii lui S .
- Secvența de stări  $\sigma = \delta_0 \delta_1 \dots \delta_{2n}$  ce rezultă dintr-o secvență de execuție, poate fi reprezentată sub forma unui tablou de dimensiuni  $m * (2n+1)$  având
  - pe linii celulele de **memorie M**, iar
  - pe coloane **stările**.

# Sistem de sarcini nedeterminat

\*Exemplu:

Să considerăm un sistem format din două sarcini, iar relația de precedență fiind mulțimea vidă

$$C = (\{S_1, S_2\}, \emptyset)$$

$$M = (M_1, M_2)$$

$$D_{S1} = D_{S2} = R_{S1} = R_{S2} = M$$

Sistemului i se atribuie două interpretări:

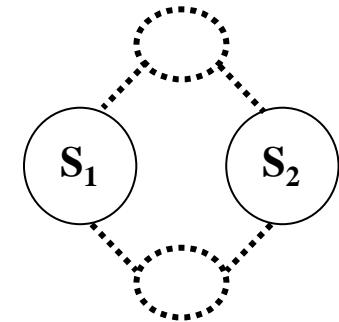
Sarcini	Interpretare 1	Interpretare 2
$S_1$	$(M_1, M_2) \leftarrow (1, M_2)$	$(M_1, M_2) \leftarrow (M_1 + M_2, M_2)$
$S_2$	$(M_1, M_2) \leftarrow (2, M_2)$	$(M_1, M_2) \leftarrow (M_1, M_1 + M_2)$

Valorile inițiale corespunzătoare stării inițiale  $s_0$ :  $M \leftarrow (1,2)$

Fie două secvențe de execuție  $\alpha_1$  și  $\alpha_2$

$$\alpha_1 = SI_1 SF_1 SI_2 SF_2 \text{ și}$$

$$\alpha_2 = SI_2 SF_2 SI_1 SF_1$$



# Comportarea celor două secvențe de execuție

Să analizăm comportarea celor două secvențe de execuție conform cu cele două interpretări ale sistemului:

	$\alpha_1 = \text{SI}_1 \text{SF}_1 \text{SI}_2 \text{SF}_2$	$\alpha_2 = \text{SI}_2 \text{SF}_2 \text{SI}_1 \text{SF}_1$
	$\sigma = \delta_0 \delta_1 \delta_2 \delta_3 \delta_4$	$\sigma = \delta_0 \delta_1 \delta_2 \delta_3 \delta_4$
Interpretare 1	$M_1 \ 1 \ 1 \ 1 \ 1 \ 2$	$M_1 \ 1 \ 1 \ 2 \ 2 \ 1$
	$M_2 \ 2 \ 2 \ 2 \ 2 \ 2$	$M_2 \ 2 \ 2 \ 2 \ 2 \ 2$
Interpretare 2	$M_1 \ 1 \ 1 \ 3 \ 3 \ 3$	$M_1 \ 1 \ 1 \ 1 \ 1 \ 4$
	$M_2 \ 2 \ 2 \ 2 \ 2 \ 5$	$M_2 \ 2 \ 2 \ 3 \ 3 \ 3$

Se observă că sistemul de sarcini C nu este determinat pentru nici una din cele două interpretări .

# Reprezentare prin secvență de valori

- O reprezentare mai convenabilă constă din secvență de valori pe care un sistem de sarcini dat C o înscrie într-o celulă de memorie  $M_i$ , în timpul unei secvențe de execuție  $\alpha$ .
- Această reprezentare se notează sub formă vectorială:

$$V(M_i, \alpha) = (V_1, V_2, \dots, V_p)$$

Considerând

$$\alpha = a_1 a_2 \dots a_{2n} \quad \text{secvență de execuție}$$

$$\sigma = \delta_0 \delta_1 \dots \delta_{2n}, \quad \text{secvență de stări corespunzătoare,}$$

$\varepsilon = \Phi$  și  $\Phi$  este sirul vid, secvența de valori corespunzătoare se definește astfel:

$$V(M_i, \varepsilon) = M_i(\Phi) \quad \text{în lipsa unei secvențe de execuție, iar}$$

$V(M_i, a_1 a_2 \dots a_k) = \underline{\text{dacă}} \quad a_k = SF \text{ și } M_i \in R_S \text{ (domeniul unde } S \text{ produce valori)}$

$$\underline{\text{Atunci}} = (V(M_i, a_1 a_2 \dots a_{k-1}), M_i(k))$$

Altfel nu se modifică deci este  $V(M_i, a_1 \dots a_{k-1})$   
nu se mai adaugă o nouă valoare

- Se iau în considerare numai valorile înscrise în celulele de memorie din domeniul de valori și numai la terminarea S.

**Obs.**  $V(M_i, \alpha) = \dots$   $V(M_j, \alpha) = \dots$  pot avea dimensiuni diferite

- Trebuie notat că  $V(M_i, \alpha)$  și  $V(M_j, \alpha)$  nu trebuie să fie de aceeași lungime deoarece  $M_i$  poate fi în  $R$  pentru anumite sarcini  
■ iar  $M_i$  poate fi în  $R$  pentru altele (în număr diferit).

Pentru secvența de valori:  $V(M_i, \alpha) = (v_1, v_2, \dots, v_p)$  se definește  
*valoarea finală*  $F(M_i, \alpha) = v_p$ .

Rezultă că pentru sevența de execuție parțială

$$\alpha = a_1 a_2 \dots a_k$$

$$\mathcal{A}_k = [F(M_1, \alpha), F(M_2, \alpha), \dots, F(M_m, \alpha)]$$

# Exemplu: Să reconsiderăm exemplul precedent:

- \* Secvența de stări poate fi reprezentată astfel printr-un tablou:

	$\alpha_1 = \text{SI}_1 \text{SF}_1 \text{SI}_2 \text{SF}_2$			$\alpha_2 = \text{SI}_2 \text{SF}_2 \text{SI}_1 \text{SF}_1$			
	$V(M, \varepsilon)$	$V(M, a_1 a_2)$	$V(M, a_1..a_4)$	$V(M, \varepsilon)$	$V(M, a_1 a_2)$	$V(M, a_1..a_4)$	
Interpretare 1	$M_1$	1	1	2	$M_1$	1	2
	$M_2$	2	2	2	$M_2$	2	2
Interpretare 2	$M_1$	1	3	3	$M_1$	1	1
	$M_2$	2	2	5	$M_2$	2	3

Interpretarea 1 este nedeterminată deoarece  $S_1$  și  $S_2$  sunt în dispută pentru a scrie în  $M_1$ , iar

în interpretarea 2 o sarcină scrie într-o celulă citită de alta sarcină.

	$\alpha_1 = \text{SI}_1 \text{SF}_1 \text{SI}_2 \text{SF}_2$					$\alpha_2 = \text{SI}_2 \text{SF}_2 \text{SI}_1 \text{SF}_1$				
	$\sigma = \delta_0 \delta_1 \delta_2 \delta_3 \delta_4$					$\sigma = \delta_0 \delta_1 \delta_2 \delta_3 \delta_4$				
Interpretare 1	$M_1$ 1 1 1 1 2					$M_1$ 1 1 2 2 1				
	$M_2$ 2 2 2 2 2					$M_2$ 2 2 2 2 2				
Interpretare 2	$M_1$ 1 1 3 3 3					$M_1$ 1 1 1 1 4				
	$M_2$ 2 2 2 2 5					$M_2$ 2 2 3 3 3				

- Un sistem de sarcini  $\mathbf{C} = (\mathbf{S}, \prec)$  este **neinterpretat** dacă se cunosc doar relația de precedență  $\prec$ , domeniile de definiție  $D$  și de valori  $R$  pentru fiecare sarcină.
- O **interpretare** pentru  $\mathbf{C}$  constă din specificarea funcției  $f_S$  pentru fiecare  $S_i \in \mathbf{S}$ .

În continuare vom considera sisteme de sarcini **neinterpretate**.

**Definiție:**

Un **sistem de sarcini**  $\mathbf{C}$  este *determinat* dacă pentru orice stare inițială  $s_0$  dată,  $V(M_i, \alpha) = V(M_i, \alpha')$ ,  $1 \leq i \leq m$ , pentru toate secvențele de execuție  $\alpha, \dots, \alpha'$  din  $\mathbf{C}$ .

(Secvențele de valori depind numai de valorile inițiale în  $s_0$ .)

### Definiție:

Sarcinile  $S$  și  $S'$  sunt **neinterferente** dacă:

$S$  este succesor sau predecesor a lui  $S'$ , sau

$$R_S \cap R_{S'} = R_S \cap D_{S'} = D_S \cap R_{S'} = \emptyset$$

- Mulțimea  $S = \{ S_1, S_2, \dots, S_n \}$  se spune că este formată din sarcini **mutual neinterferente** dacă pentru  $\forall i, j$  ( $i \neq j$ )  $S_i$  și  $S_j$  sunt neinterferente.
- Pentru a arăta că un sistem de sarcini mutual neinterferente este determinat vom prezenta **teorema de suficiență și necesitate**.
- Înainte de a prezenta această teoremă vom stabili o lemă utilizată în cadrul teoremei de necesitate și suficiență.

### Lemă:

Fie  $\mathbf{C} = (\prec, \alpha)$  un sistem de  $n$  sarcini mutual neinterferente iar  $S$  o sarcina terminală a lui  $\mathbf{C}$ . ( $R_S \cap R_{S'} = R_S \cap D_{S'} = D_S \cap R_{S'} = \emptyset$ )

Dacă  $\alpha = \beta_1 \text{ SI } \beta_2 \text{ SF } \beta_3$  este o secvență de execuție validă a lui  $\mathbf{C}$ , atunci  $\alpha' = \beta_1 \beta_2 \beta_3 \text{ SI SF}$  este de asemenea o secvență de execuție a lui  $\mathbf{C}$  pentru  $\beta_0$  dată,

$$V(M_i, \alpha) = V(M_i, \alpha'), \forall 1 \leq i \leq m$$

### Justificare :

- S nu are succesiuni în  $\mathbf{C}$  conform cu  $(\prec, \alpha')$  satisfac relațiile de precedență din  $\mathbf{C}$  și deci trebuie să fie o secvență de execuție validă.
- S scrie numai în celulele ce aparțin lui  $R_S$  și deoarece pentru orice  $S'$  inițiată în  $\beta_3$ , după terminarea lui  $S$  în  $\alpha$ ,  $R_S \cap D_{S'} = \emptyset$  fiind neinterferente, orice astfel de sarcini  $S'$  găsesc aceleași valori în  $D_{S'}$  atât pentru  $\alpha$  cât și pentru  $\alpha'$ .
- Deci pentru  $M_i \notin R_S$  rezultă  $V(M_i, \alpha) = V(M_i, \alpha')$

- Pentru orice  $M_j \in D_S$ ,  $V(M_j, \beta_1) = V(M_j, \beta_1\beta_2\beta_3)$  deoarece nici o sarcina  $S'$  din  $\beta_1\beta_2\beta_3$  nu scrie în  $D_S$  deoarece  $R_{S'} \cap D_S = \emptyset$  fiind mutual neinterferente.
- Deci  $F(M_j, \beta_1) = F(M_j, \beta_1\beta_2\beta_3)$  pentru  $\forall M_j \in D_S$ , iar  $S$  scrie aceeași valoare în  $\forall M_i \in R_S$  atât pentru  $\alpha$  cât și pentru  $\alpha'$ .
- Considerând că  $v$  este valoarea înscrisă de  $S$  în  $M_i \in R_S$  pentru  $\alpha$ , rezultă că:

$$V(M_i, \alpha) = V(M_i, \beta_1 \text{ SI } \beta_2 \text{ SF } \beta_3)$$

$$V(M_i, \alpha) = V(M_i, \beta_1 \text{ SI } \beta_2 \text{ SF}) \quad \text{nu există } S' \in \beta_3 \text{ care scrie în } R_S$$

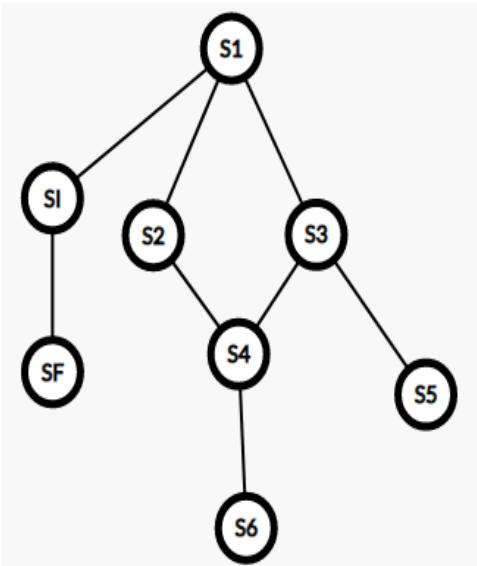
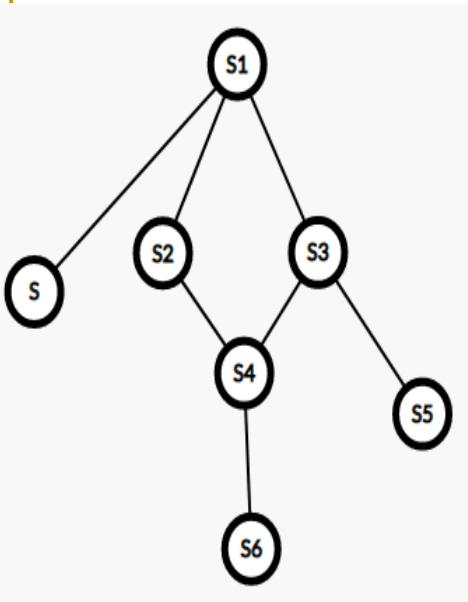
$$V(M_i, \alpha) = (V(M_i, \beta_1 \text{ SI } \beta_2), v) \quad S \text{ scrie valoarea } v \text{ în}$$

$$V(M_i, \alpha) = (V(M_i, \beta_1), v) \quad \text{nu există } S' \in \beta_2 \text{ care scrie în } R_S$$

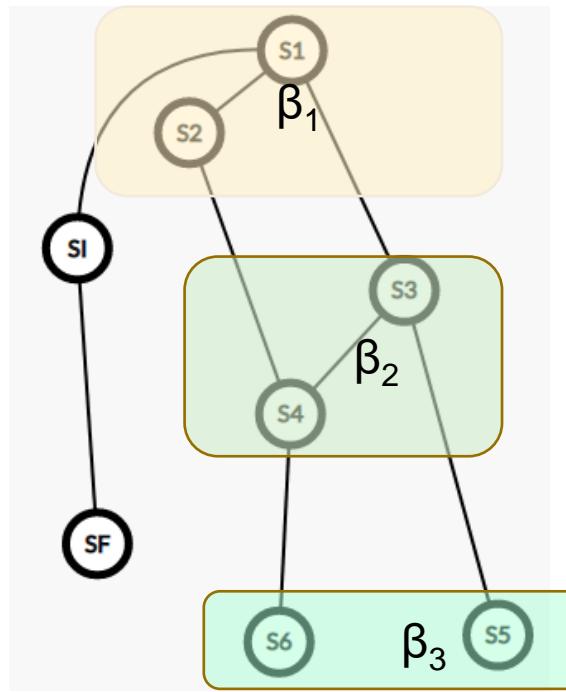
$$V(M_i, \alpha) = (V(M_i, \beta_1 \beta_2 \beta_3), v) \quad \text{nu există } S' \in \beta_2 \beta_3 \text{ care scrie în } R_S$$

$$V(M_i, \alpha) = (V(M_i, \beta_1 \beta_2 \beta_3 \text{ SI SF}) \quad S \text{ scrie } v \text{ în } M_i$$

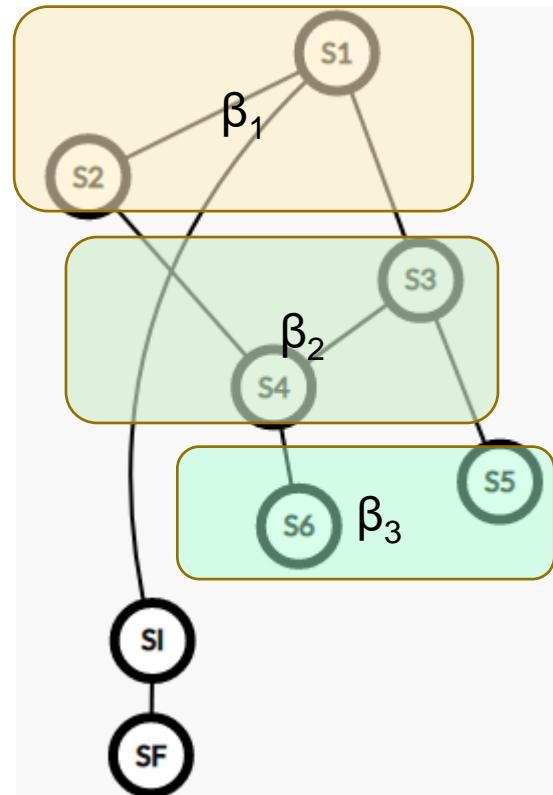
$$V(M_i, \alpha) = V(M_i, \alpha')$$



$$\alpha = \beta_1 \text{ SI } \beta_2 \text{ SF } \beta_3$$



$$\alpha' = \beta_1 \beta_2 \beta_3 \text{ SI } \text{SF}$$



Lema

## Teorema de suficiență:

Sistemele de sarcini formate din sarcini mutual neinterferente sunt determinate.

Justificare (prin inducție)

- Pentru un sistem de sarcini format dintr-o singura sarcina este evident.
  - Presupunem că afirmația este adevărată pentru sisteme cu  $n-1$  sarcini, ( $\alpha'_1$  și  $\alpha'_2$  sunt secvențe de execuție pentru un sistem cu  $n-1$  sarcini)
  - Presupunem sistemul  $\mathbf{C} = (\mathbf{S}, \prec)$  format din  $n$  sarcini.
- 
- Dacă sistemul  $\mathbf{C}$  are o singură secvență de execuție validă, el este determinat.
  - Presupunem 2 secvențe de execuție  $\alpha_1, \alpha_2$  în  $\mathbf{C}$  și  $S$  sarcina terminală în  $\mathbf{S}$ .

Conform lemei putem forma 2 secvențe de execuție  $\alpha'_1$  și  $\alpha'_2$  astfel:

$$\alpha_1 = \alpha'_1 \text{ SI } SF; \quad \alpha_2 = \alpha'_2 \text{ SI } SF,$$

în condițiile în care

$$V(M_i, \alpha_1) = V(M_i, \alpha'_1), \quad 1 \leq i \leq m$$

$$V(M_i, \alpha_2) = V(M_i, \alpha'_2), \quad 1 \leq i \leq m$$

$\alpha'_1$  și  $\alpha'_2$  sunt secvențe de execuție pentru un sistem cu  $n-1$  sarcini:

$C' = (S \setminus S, \prec')$ , relația de precedență  $\prec'$  fiind obținută din  $\prec$  eliminând relația de precedență ce implică pe  $S$ .

$$V(M_i, \alpha'_1) = V(M_i, \alpha'_2) \text{ din ipoteza de inducție.}$$

- Deci se poate presupune că valorile din  $D_S$  sunt aceleași, atât pentru  $\alpha'_1$  cât și pentru  $\alpha'_2$ ,
- respectiv  $F(M_i, \alpha'_1) = F(M_i, \alpha'_2)$  pentru  $M_i \in D_S$ .
- Rezultă astfel că pentru  $\alpha'_1$  și  $\alpha'_2$ , sarcina  $S$  scrie aceeași valoare  $v$  pentru orice celulă  $M_i \in R_S$ .

Pentru  $M_i \notin R_S$ :

$$\begin{aligned} V(M_i, \alpha_1) &= V(M_i, \alpha'_1 \text{ SI } SF) && \text{Lemă} \\ &= V(M_i, \alpha'_1) && M_i \notin R_S \text{ și ip. de inducție} \\ &= V(M_i, \alpha'_2) && M_i \notin R_S \\ &= V(M_i, \alpha'_2 \text{ SI } SF) && \text{Lemă} \\ &= V(M_i, \alpha_2) \end{aligned}$$

Pentru  $M_i \notin R_S$   $V(M_i, \alpha_1) = V(M_i, \alpha_2)$

Pentru  $M_i \in R_S$ :

$$\begin{aligned} V(M_i, \alpha_1) &= V(M_i, \alpha'_1 \text{ SI } SF) && \text{Lemă} \\ &= (V(M_i, \alpha'_1), v) && \text{ip. de inducție} \\ &= (V(M_i, \alpha'_2), v) \\ &= V(M_i, \alpha'_2 \text{ SI } SF) && \text{Lemă} \\ &= V(M_i, \alpha_2) \end{aligned}$$

Pentru  $M_i \in R_S$   $V(M_i, \alpha_1) = V(M_i, \alpha_2)$

- *Sistemul C este determinat dacă sarcinile sunt mutual neinterferente.*

### Teorema de necesitate:

Fie  $C$  un sistem de sarcini astfel că pentru fiecare  $S \in S$ ,  $f_S$  nu este specificată, dar  $D_S$  și  $R_S \neq \emptyset$  sunt date.

$C$  este determinat pentru orice interpretare a sarcinilor sale numai dacă acestea sunt mutual neinterferente.

### Justificare (reducere la absurd)

- Presupunem că  $S$ ,  $S'$  sunt interferente, independente (adică nu există nici o relație de ordine între ele).
- Atunci există două secvențe de execuție valide:

$$\alpha = \beta_1 \text{ SI } SF \text{ SI' } SF' \beta_2$$

$$\alpha' = \beta_1 \text{ SI' } SF' \text{ SI } SF \beta_2$$

- Presupunem că există o celulă de memorie  $M_i \in R_S \cap R_{S'}$  (sunt considerate interferente) și putem alege  $f_S$  și  $f_{S'}$  (două interpretări, pentru  $S$  și  $S'$ ), astfel încât pentru:

$f_S$ :  $S$  să scrie în  $M_i$  valoarea  $u$ ;

$f_{S'}$ :  $S'$  să scrie în  $M_i$  valoarea  $v$ ;     $u \neq v$

În ceea ce privește valorile din  $R_S$  și  $R_{S'}$  pentru secvențele  $\alpha$  și  $\alpha'$  putem spune că:

$$V(M_i, \alpha) = V(M_i, \beta_1 \text{ SI } SF \text{ SI'} SF') = (V(M_i, \beta_1), u, v); \\ u \neq v$$

$$V(M_i, \alpha') = V(M_i, \beta_1 \text{ SI'} SF' \text{ SI } SF) = (V(M_i, \beta_1), v, u);$$

Deci în celula  $M_i$ , secvențele  $\alpha$  și  $\alpha'$  înscriu secvențe de valori diferite

$$(V(M_i, \beta_1), u, v); \text{ respectiv } (V(M_i, \beta_1), v, u); \\ V(M_i, \alpha) \neq V(M_i, \alpha')$$

ceea ce înseamnă că sistemul de sarcini **C** nu este determinat.

Deci este necesar ca  $R_S \cap R_{S'} = \emptyset$  pentru că altfel rezultă că **C** este nedeterminat.

- Presupunem că există o celulă  $M_j \in D_S \cap R_{S'}$ .
- Presupunem că există o celulă  $M_i \in R_S$ .
- Atunci, putem alege o interpretare a lui  $S'$ , o funcție  $f_{S'}$  astfel încât  $F(M_j, \beta_1) \neq F(M_j, \beta_1 \text{ SI' SF'})$ .
- Rezultă în acest caz că  $S$  citește valori diferite pentru  $\alpha$  și  $\alpha'$  (având în vedere că  $M_j \in D_S \cap R_{S'}$ ).
- Putem alege în acest caz o funcție  $f_S$  astfel încât sarcina  $S$  să scrie în  $M_i$ , valoarea  $u$  pentru secvența  $\alpha$  și valoarea  $v$  pentru secvențe  $\alpha'$ , cu  $u \neq v$ .

În acest caz putem spune că:

$$\begin{aligned} V(M_i, \alpha) &= V(M_i, \beta_1 \text{ SI' SF'} \text{ SI' SF'}) \\ &= V(M_i, \beta_1 \text{ SI' SF'}) & R_S \cap R_{S'} = \emptyset \\ &= (V(M_i, \beta_1), u) & S \text{ scrie } u \text{ pentru } \alpha \end{aligned}$$

$$\begin{aligned}
 V(M_i, \alpha') &= V(M_i, \beta_1 \textcolor{blue}{SI}' \textcolor{blue}{SF}' \textcolor{blue}{SI} \textcolor{blue}{SF}) \\
 &= (V(M_i, \beta_1 \textcolor{blue}{SI}' \textcolor{blue}{SF}') , v) \\
 &= (V(M_i, \beta_1), v) \quad S \text{ scrie } v \text{ pentru } \alpha'
 \end{aligned}$$

deci  $V(M_i, \alpha) \neq V(M_i, \alpha')$  deci **C** nu este determinat.

Rezultă că trebuie ca :  $D_S \cap R_{S'} = \emptyset$

Similar se poate arăta că trebuie ca :  $D_{S'} \cap R_S = \emptyset$

O aplicație foarte importantă a teoremelor arătate anterior se referă la **paralelismul maxim** în sisteme de sarcini în care constrângerile de precedență sunt impuse numai de cerințele de determinare.

- Din definiția dată pentru **determinare** rezultă că fiecare sarcina produce o singură secvență de valori pentru o celulă de memorie  $M_i$  în condițiile unei stări inițiale date.

- Două sisteme cu aceeași mulțime de sarcini  $S$  sunt **echivalente** dacă sunt determinate și dacă pentru aceeași stare initială produc aceeași secvență de valori.

Un sistem de sarcini **C** și graful asociat  $G$  (definit de relația de precedență  $<$ ) implică un **paralelism maxim** dacă **C** este determinat și dacă **eliminarea** unui arc oarecare  $(S, S')$  din  $G$ , face ca sarcinile  $S$  și  $S'$  să devină **interferente**.

- Astfel dacă  $(S, S')$  este un arc într-un graf cu maximum de paralelism atunci:

$$(R_S \cap R_{S'}) \cup (R_S \cap D_{S'}) \cup (D_S \cap R_{S'}) \neq \emptyset$$

- Având un sistem de sarcini **C**, determinat, este util să construim sistemul echivalent **C'** cu maximum de paralelism.

# Sistem de sarcini cu maximum de paralelism

**Teoremă :**

Avand un sistem de sarcini  $C = (S, \prec)$  se poate construi un sistem  $C' = (S, \prec')$  unde

relația  $\prec'$  este închiderea prin tranzitivitate a relației:

$$Z = \{(S, S') \in \prec \mid (R_S \cap R_{S'}) \cup (R_S \cap D_{S'}) \cup (D_S \cap R_{S'}) \neq \emptyset\}$$

$C'$  este unicul sistem echivalent cu  $C$  care implică **maximum de paralelism**.

**Exemplu :**

Fie sistemul de sarcini  $C = (S, \prec)$  unde

$$S = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8\}$$

Să se construiască sistemul de sarcini  $C' = (S, \prec')$  echivalent, care realizează maxim de paralelism tinând seama de **cerințele de determinare** (care are la baza conflictul de resurse)

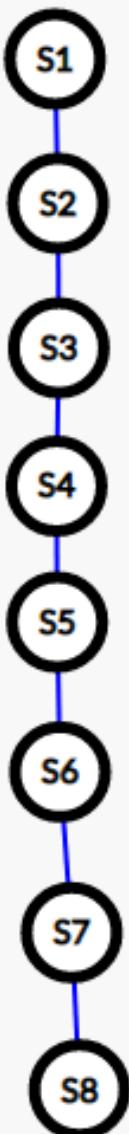
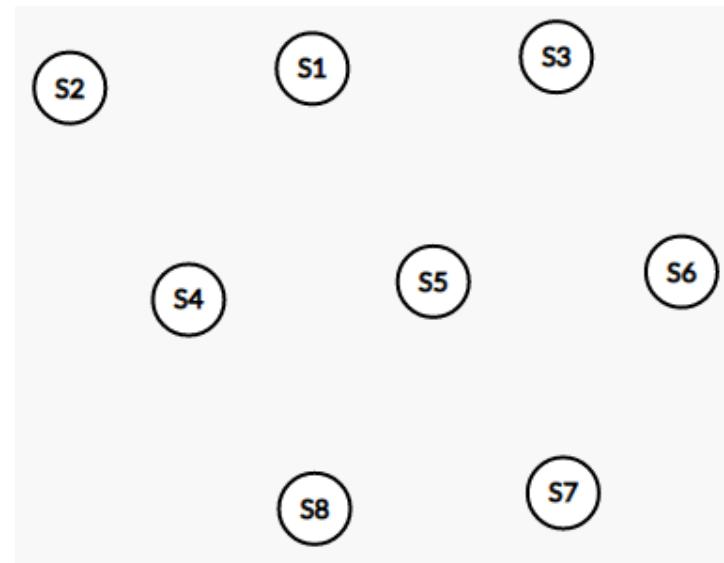
- Sistemul este caracterizat prin:
  - $M = \{ M_1, M_2, M_3, M_4, M_5 \}$
- iar domeniile  $D_S$  și  $R_S$  sunt specificate prin tabelul :

M	Domeniu de definiție $D_S$	Domeniu de valori $R_S$
M1	S1,S2,S7,S8	S3
M2	S1,S7	S5
M3	S3,S4,S8	S1
M4	S3,S4,S5,S7	S2,7
M5	S6	S4,S6,S8

# Posibilitati de executie a sistemului de sarcini



maxim de paralelism tinand seama de cerințele de determinare bazate pe eliminarea conflictului de resurse



$$Z = \{(S, S') \in \lessdot \mid (R_S \cap R_{S'}) \cup (R_S \cap D_{S'}) \cup (D_S \cap R_{S'}) \neq \emptyset\}$$

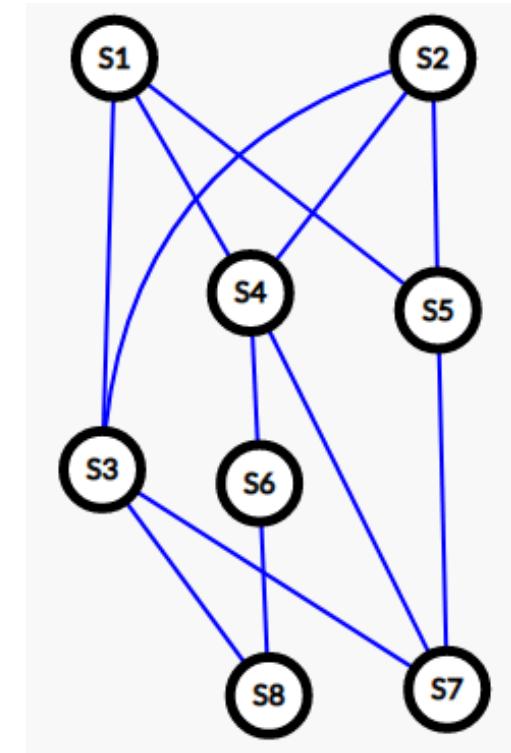
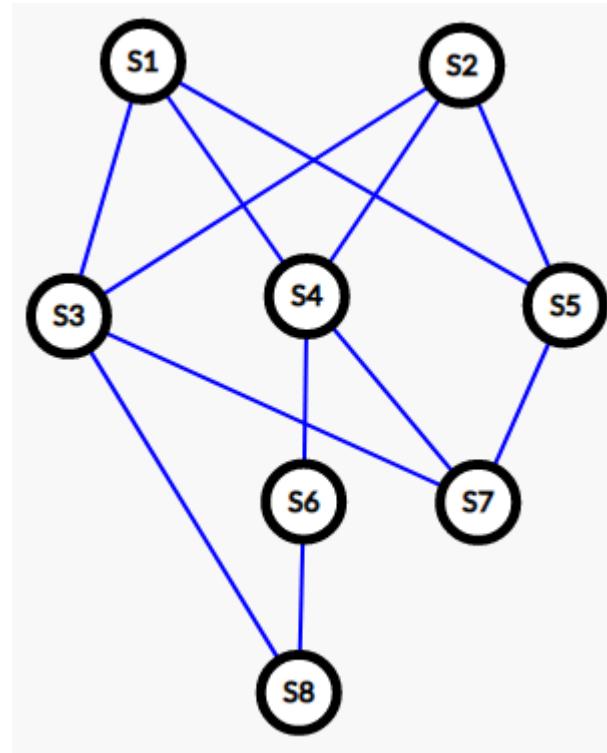
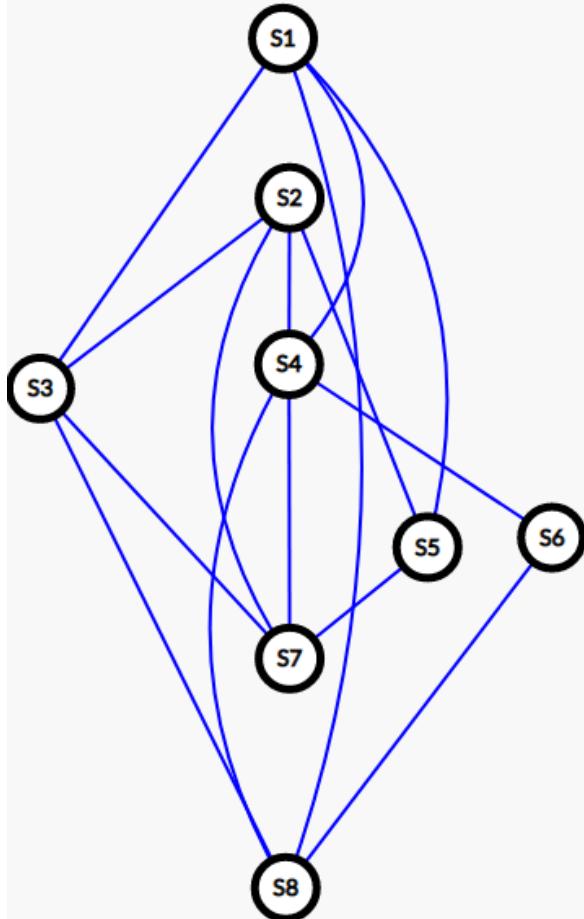
Relația  $Z$ , conform definiției de mai sus va fi dată de mulțimea arcelor:

$$Z : \{ \quad (S_1, S_3) (S_1, S_4) (S_1, S_5) (S_1, S_8) \\ (S_2, S_3) (S_2, S_4) (S_2, S_5) (S_2, S_7) \\ (S_3, S_7) (S_3, S_8) \\ (S_4, S_6) (S_4, S_7) (S_4, S_8) \\ (S_5, S_7) \\ (S_6, S_8) \}$$

M	Domeniu de definiție $D_S$	Domeniu de valori $R_S$
M1	S1, S2, S7, S8	S3
M2	S1, S7	S5
M3	S3, S4, S8	S1
M4	S3, S4, S5, S7	S2, 7
M5	S6	S4, S6, S8

Rezultă graful asociat  $G'$ , care este organizat pe niveluri eliminând arcele excluse prin tranzitivitate

# graful asociat



Sistemul C' echivalent se execută în 4 perioade de timp considerând că avem un număr suficient de procesoare (3 procesoare) sau in 4 perioade avand la dispozitie numai 2 procesoare

# Analiza situațiilor de blocare

- Un sistem care este format din sisteme de sarcini combinate paralel  $C=C_1 \mid C_2 \mid \dots \mid C_n$   
este *blocat* dacă evoluția sa este suspendată indefinitely din cauză că mai multe sisteme de sarcini  $C_i$  sunt active simultan, fiecare utilizând, fără a putea fi întrerupte, resurse necesare altor sisteme din  $C$ .
- O resursă deținută de o sarcina **neinterruptibila** poate fi eliberată numai de sistemul de sarcini care o utilizează la un moment dat.

- Pentru studiul situațiilor de blocare vom privi sistemul fizic ca o colecție de resurse R de tip:

$R = R_1 \ R_2 \dots R_m$ , fiecare tip conținând:

$W = W_1 \ W_2 \dots W_m$  exemplare

Vectorul  $W=(W_1, W_2, \dots, W_m)$  reprezintă capacitatea sistemului.

### Exemplu de resurse $R_i$ :

- unități centrale de prelucrare;
- coprocesoare matematice;
- pagini de memorie;
- magistrale de interconectare;
- înregistrări fizice (sectoare, piste, cilindri) la discuri magnetice;
- fișiere de date;
- zone de mesaje;
- tabele de descriere a diferitelor sarcini;
- variabile globale, etc.

■ Să considerăm un exemplu simplu de blocare:

Fie două sisteme  $C_1$  și  $C_2$  și câte o resursă de tip  $R_1$  și  $R_2$ . Evoluția întregului sistem poate fi reprezentată în spațiul cu două dimensiuni definit **spațiul de evoluție**.

Evoluția sistemului  $C=C_1 \cup C_2$  este reprezentată de succesiunea de puncte  $(x_1y_1) (x_2y_2)\dots(x_ky_k)\dots$  în acest spațiu

Presupunem că originea este în punctul  $(0,0)$ .

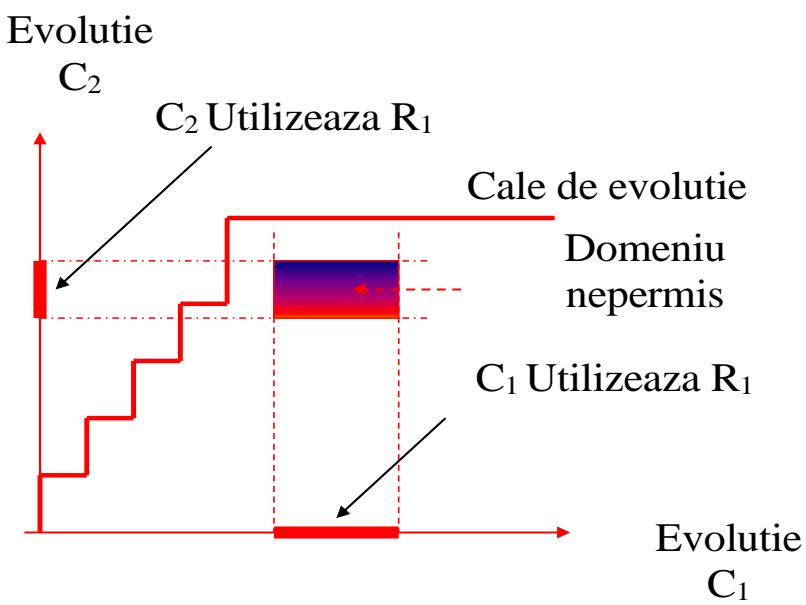
Evoluția este ireversibilă deci

$x_k \leq x_{k+1}$ ,  $y_k \leq y_{k+1}$  și

Nu orice cale de evoluție, este realizabilă.

Evoluția în care  $C_1$  și  $C_2$  astfel să utilizeze simultan aceeași resursă  $R_1$  este nepermisă

Zona hasurată reprezintă un domeniu nepermis, care conduce la blocarea sistemului.



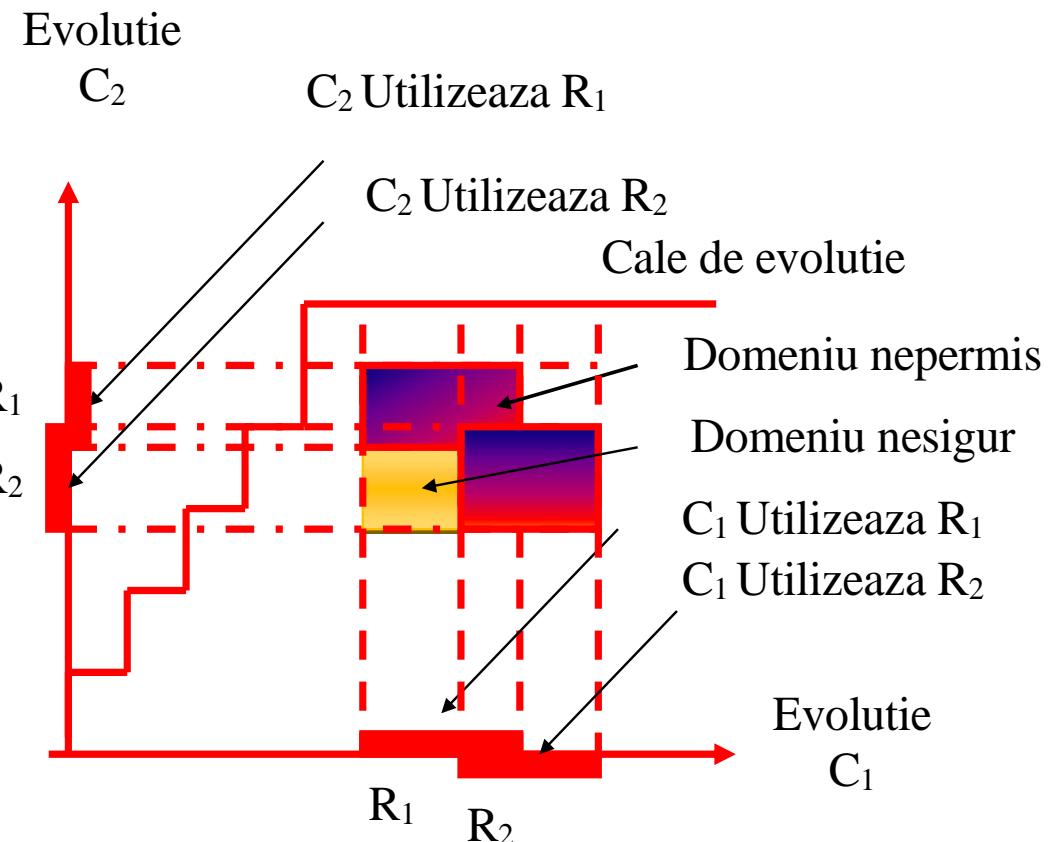
# Exemplu de utilizare a două resurse

Fie utilizarea resurselor  $R_1$  și  $R_2$  de către  $C_1$  și  $C_2$ .

Zona hașurată reprezintă un domeniu nepermis, care conduce la blocarea sistemului.

Zona portocalie reprezintă un domeniu nesigur, care permite încă continuare evoluția sistemului, însă pentru un timp limitat.

Datorită ireversibilității procesului de evoluție se va ajunge în mod sigur la blocarea sistemului.



- **Blocarea** este definită ca o stare de *așteptare circulară* nerezolvabilă din cauza condițiilor de excludere mutuală și neintreruptibilitate.
- Pentru a exista o cale de evoluție care conduce la blocare sunt necesare următoarele condiții:
  - **Utilizare exclusivă**: Fiecare sistem de sarcini dorește controlul exclusiv asupra resursei pe care o utilizează;
  - **Neîntreruptibilitate**: O resursă nu este eliberată până la terminarea completă a utilizării acesteia;
  - **Așteptare circulară**: Fiecare sistem de sarcini ține ocupate resurse în timp ce așteaptă ca altele să elibereze resurse pe care să le preia.

- Relativ la blocare se pun trei probleme:

### **1. Prevenirea**

Una sau mai multe din condițiile care conduc la blocare sunt înălăturate printr-o proiectare adecvată sau prin restricții în utilizarea resurselor.

### **2. Detectarea și înălăturarea**

Sistemul de operare și planificatorul supraveghează evoluția tuturor sistemelor de sarcini luând măsuri speciale de corecție a evolutiei la detectarea situațiilor de blocare.

### **3. Evitarea**

Pe baza unor informații preliminare despre cerințele de resurse, sistemul de operare / planificatorul dirijează evoluția pentru evitarea zonelor nesigure.

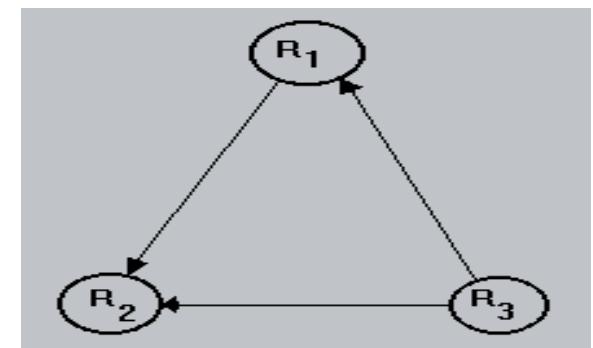
Considerând că fiecare tip de resursă conține un singur element  $w_1=w_2=\dots=w_n=1$ , putem alege o reprezentare sub forma de grafuri, fiecare nod reprezentând o resursă iar arcele reprezintă cererea de resurse.

Arcele pot fi etichetate cu necesarul de resurse.

- Arcele sunt orientate și sunt definite astfel:  
dacă la un moment dat, după evenimentul  $a_k$ , sistemul **C** ține ocupată resursa  $R_i$  în timp ce aşteaptă preluarea lui  $R_j$ , graful conține un arc orientat de la nodul  $R_i$  la nodul  $R_j$ .
- În exemplul urmator numărul de tipuri de resurse este  $n=3$ ;  
Un proces utilizează  $R_1$  și o cere pe  $R_2$ ,

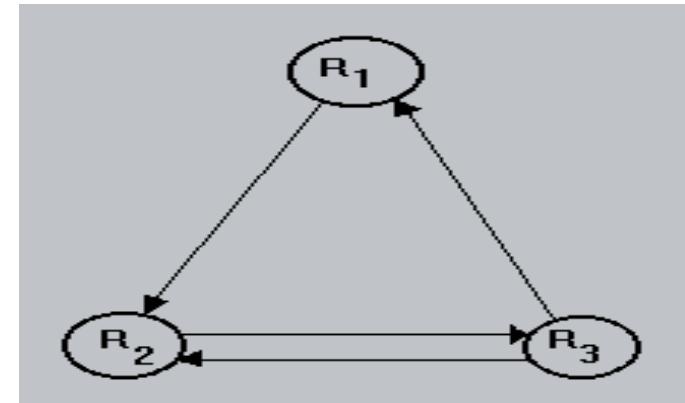
Alt proces utilizează pe  $R_3$  și cere pe  $R_1$  și  $R_2$ .

Graf de cereri fără bucle



Dacă cele trei condiții de blocare sunt posibile (operative) condiția necesară și suficientă pentru blocare este existența unei bucle (circuit, ciclu) în graful de cereri.

Exemplu , de graf de cereri în care se realizează condiția de așteptare circulară cu resurse ocupate.



Graf de cereri cu bucle

- Pentru cazul general în care există mai multe resurse de un anumit tip, fiecare nod reprezintă un tip de resursă.
- Cererile pentru resurse R<sub>i</sub> trebuie să nu depășească, ca număr, w<sub>i</sub>.
- Un arc de la R<sub>i</sub> la R<sub>j</sub> arată că există cel puțin un proces care cere una sau mai multe resurse de tip R<sub>j</sub> și deține una sau mai multe resurse de tip R<sub>i</sub>.
- În cazul acesta o buclă în graful de cereri reprezintă o condiție necesară dar nu și suficientă pentru existența situației de blocare.

# Prevenirea blocării

- Pentru a proiecta un sistem în care posibilitatea blocării este exclusă trebuie să existe certitudinea că în orice moment de timp cel puțin una din condițiile necesare nu este îndeplinită.
- Prima din cele trei condiții (excluderea mutuală) este uneori impusă de restricții fizice privind accesul la resurse nepartajabile și deci nu poate fi eliminată pentru orice fel de resurse.

Pentru celelalte condiții vom enumera câteva posibilități:

- Dacă o sarcina care utilizează o resursă, cere o altă resursă *trebuie* să o elibereze pe prima și dacă este necesară în continuare să o ceară din nou împreună cu resursele adiționale de care are nevoie (**interruptibilitate**).
- Fiecare proces să ceară toate resursele de care are nevoie, putând continua numai după ce le are pe toate (**așteptare cu resurse**).
- **Ordonarea liniară** a tipurilor de resurse pentru toate sarcinile. Dacă un proces utilizează resurse de tip  $R_i$ , la un moment dat, el poate cere numai resurse din tipurile următoare lui  $R_i$  în acea ordonare,
- exemplu:  $R_j$  cu  $j > i$ . În acest caz, graful de cereri nu are circuite.

\*Observație:

- **Prima soluție** se pretează numai pentru resurse interruptibile, a căror stare poate fi ușor salvată și restaurată. De exemplu UCP.
- **A doua soluție** poate deveni costisitoare dacă unele resurse sunt alocate unor sarcini și sunt neutilizate o perioadă lungă de timp.
- **A treia soluție** poate fi utilizată numai pentru anumite componente, de exemplu lansarea job-urilor într-un sistem cu multiprogramare prin alocarea memoriei și a dispozitivelor de I/E într-o ordine fixă.

## Detectarea și înlăturarea blocării

- Pentru cazul în care există o singură resursă de fiecare tip, mecanismul de detectare a blocării constă:
- dintr-o *procedură* care actualizează graful de cereri ori de câte ori apare o cerere, o preluare sau eliberare de resurse și
- o *procedură* de detectare a circuitelor (bucelelor) în acest graf.
- Pentru cazul general, la fiecare moment de timp  $t$ , corespunzător secvenței de execuție  $\alpha = a_1a_2...a_k$ , se definesc
- $x_{ij}[k]$  reprezintă numărul de resurse de tip  $R_j$  alocate, respectiv și
- $y_{ij}[k]$  cerute de  $C_i$ , după evenimentul  $a_k$  din  $\alpha$ , unde:  
$$C=C_1 | C_2 | \dots | C_n, \quad 1 \leq i \leq n \quad \text{iar} \quad 1 \leq j \leq m,$$
 $m$  fiind numărul de tipuri de resurse

Se obțin astfel:

$$X[k] = \begin{bmatrix} X_1(k) \\ X_2(k) \\ \vdots \\ \vdots \\ X_n(k) \end{bmatrix}$$

matrice de alocare

$$Y[k] = \begin{bmatrix} Y_1(k) \\ Y_2(k) \\ \vdots \\ \vdots \\ Y_n(k) \end{bmatrix}$$

matrice de cereri

cu

$$X_i[k] = (x_{i1}[k], x_{i2}[k], \dots, x_{im}[k])$$

$$Y_i[k] = (y_{i1}[k], y_{i2}[k], \dots, y_{im}[k]), \quad 1 \leq i \leq n$$

- Starea  $s_k$  corespunzătoare secvenței a după evenimentul  $a_k$  este dată de perechea  $[X[k], Y[k]]$ .

Să notăm cu:

$V[k] = (v_1[k], v_2[k], \dots, v_m[k])$  vectorul resurselor disponibile.

Elementul  $v_j[k] \leq w_j$ ,  $1 \leq j \leq m$ , numărul de resurse de tip  $R_j$  disponibile la un moment dat după evenimentul  $a_k$  este:

$$V_j(k) = W_j - \sum_{i=1}^n X_{ij}[k], \quad 1 \leq j \leq m$$

- Pentru doi vectori  $X$  și  $Y$ , relația  $X \leq Y$  este adevarată, dacă și numai dacă relația este adevărată pentru fiecare pereche de elemente corespondente din  $X$  și  $Y$ .
- Utilizând informația conținută în starea  $s_k$  dată de perechea de matrici  $[X[k], Y[k]]$  vom elabora un algoritm de detectare a blocării.

## Definiție:

Fie  $\mathbf{C} = \mathbf{C}_1 | \mathbf{C}_2 | \dots | \mathbf{C}_n$  unde  $\mathbf{C}_i$ ,  $1 \leq i \leq n$ , este de forma

$$\mathbf{C}_i = S_i[1] \ S_i[2] \dots \ S_i[l_i], \quad l_i \geq 1.$$

Fie

$\alpha = a_1 a_2 \dots a_k$  o secvență de execuție parțială a lui  $\mathbf{C}$  iar

$\sigma = s_0 \ s_1 \dots s_k$  secvența parțială de stări, corespunzătoare.

Dacă există o mulțime nevidă  $D$  de indici ai lui  $\mathbf{C}$  astfel că pentru orice  $i$  din  $D$  **nu este** adevărată relația:

$$Y_i[k] \leq V[k] + \sum_{j \notin D} X_j[k]$$

vector de cereri

vector de resurse

vector de resurse  
deținute de sarcini ce  
nu aparțin lui  $D$

atunci  $\sigma$  este blocată (sau  $s_k$  este o stare de blocare).

De asemenea putem spune că orice sistem de sarcini  $\mathbf{C}_i$  cu  $i \in D$  este blocat.

## Exemplu:

Fie  $C_1$  și  $C_2$  două sisteme de sarcini:

$$C_1 = S_1[1], S_1[2], S_1[3] \text{ și } C_2 = S_2[1], S_2[2]$$

care utilizează resursele  $R=(R_1, R_2, R_3, R_4)$  cu  $w = (3, 3, 3, 3)$ , în felul următor :

	C 1		C 2	
	resurse cerute	resurse eliberate	resurse cerute	resurse eliberate
$S_1[1]$	(0,1,2,0)	(0,0,2,0)	$S_2[1]$	(2,0,2,2)
$S_1[2]$	(1,0,0,1)	(0,0,0,1)	$S_2[2]$	(1,0,1,0)
$S_1[3]$	(0,0,2,0)	(1,1,2,0)		(3,0,3,0)

$\alpha = SI_1[1] SF_1[1] SI_2[1] SF_2[1] SI_1[2] SF_1[2] SI_2[2] \dots$  secvența de execuție.

	X	Initial	SI <sub>1</sub> [1]	SF <sub>1</sub> [1]	SI <sub>2</sub> [1]	SF <sub>2</sub> [1]	SI <sub>1</sub> [2]	SF <sub>1</sub> [2]	SI <sub>2</sub> [2]
		$R_1, R_2, R_3, R_4$							
Alocare	$X_1$	0 0 0 0	0 1 2 0	0 1 0 0	0 1 0 0	0 1 0 0	1 1 0 1	1 1 0 0	Blocare
$X[k]$	$X_2$	0 0 0 0	0 0 0 0	0 0 0 0	2 0 2 2	2 0 2 2	2 0 2 0	2 0 2 0	
Cereri	$Y_1$	0 1 2 0	0 0 0 0	1 0 0 1	1 0 0 1	1 0 0 1	0 0 0 0	0 0 2 0	
$Y[k]$	$Y_2$	2 0 2 2	2 0 2 2	2 0 2 2	0 0 0 0	1 0 1 0	1 0 1 0	1 0 1 0	
$V[k]$		3 3 3 3	3 2 1 3	3 2 3 3	1 2 1 1	1 2 1 3	0 2 1 2	0 2 1 3	

După evenimentul  $S1[2]$ , atât  $Y1[k] \leq V[k]$  cât și  $Y2[k] \leq V[k]$  astfel că  $P1[3]$  și  $P2[2]$  nu pot avea loc cu valorile obținute pentru  $V[k]$  înaintea acestor evenimente.

Sistemul s-a blocat deoarece nici  $C_1$  nici  $C_2$  nu pot continua fără ca unul din ele să fie întrerupt pentru a elibera resursele deținute.

## Algoritm pentru detectarea blocării

Etapele algoritmului sunt:

- 1°. Inițializare :  $D \leftarrow \{1, 2, \dots, n\}$  ;  $V \leftarrow V(k)$
- 2°. Caută indicii  $i$  din  $D$  pentru care  $Y_i(k) \leq V$  și execută pasul 3.  
Dacă nu există nici un indice, algoritmul se termină.
- 3°.  $D \leftarrow D \setminus \{i\}$  ;  $V \leftarrow V + X_i(k)$  și salt la pasul 2.

- Algoritmul se bazează direct pe definiția de mai sus și constă din **simularea** execuției sarcinilor până când rămâne un set de sisteme de sarcini  $C_i$  pentru care, în final, resursele disponibile nu sunt suficiente sau mulțimea de indici  $D$  devine vidă.
- În cel mai defavorabil caz algoritmul dat analizează la pasul  $i$ , întreaga mulțime de sisteme de sarcini din  $D$ , în număr de  $(n-i+1)$ .

Timpul de execuție al algoritmului este în acest caz proporțional cu

$$m \cdot \sum_{i=1}^n (n-i+1) = O(m \cdot n^2)$$

- Printr-o aranjare a structurilor de date și o reprezentare mai elaborată a stărilor resurselor se poate obține un algoritm cu timpul de execuție proporțional cu  $O(m \cdot n)$ .
- Complexitatea acestui algoritm face ca implementarea lui să fie justificată numai în sisteme în care apariția unei blocări este intolerabilă.
- În multe sisteme se utilizează o soluție simplă care se bazează pe limitarea timpului în care sistemul poate sta într-o anumită stare.
- Dacă s-a ajuns la această limită se consideră că s-a realizat o situație de blocare și sistemul este scos din blocare și trecut într-o anumită stare, predeterminată din care continuă evoluția.
- Implementarea metodelor de detectare a situațiilor de "time out" se face atât la nivelul mașinii de bază cât și la nivelul sistemului de operare.

## Evitarea blocării

- Pentru a analiza problema evitării blocării se presupune că o sarcină S constă dintr-o succesiune de pași.
- Pe parcursul execuției unui pas utilizarea resurselor rămâne constantă și aceasta implică achiziționarea resurselor necesare pentru acest pas în plus față de cele primite la pasul precedent.
- Urmează apoi o perioadă de execuție în care cerințele de resurse nu se schimbă.
- În final când se termină execuția, toate resursele care nu sunt cerute la pasul următor sunt eliberate și trecute într-o colecție de resurse disponibile.

- Fie  $\sigma = s_0s_1\dots s_k$  secvența de stări corespunzătoare secvenței de execuție  $a = a_1\dots a_k$ .
- Dacă există cel puțin o secvență de execuție completă, validă, care are ca prefix pe  $a$  atunci  $s_k$  este o stare **sigură**, altfel starea  $s_k$  este o stare **nesigură** cu pericol de blocare în evoluția următoare.
- Cu alte cuvinte  $s_k$  este *sigură* dacă utilizând resursele disponibile  $V[k]$  și cele care vor fi eliberate de pașii care se execută la momentul respectiv, este posibil să găsim o secvență validă de pași încă neinitiați, pentru sarcinile inițiate dar neterminate astfel încât toate sarcinile din sistem să poată continua execuția spre finalizare.
- **Starea inițială** în care încă nu s-au alocat resurse, toate fiind disponibile, este întotdeauna o stare sigură.
- Deci există posibilitatea finalizării tuturor sarcinilor.

- Să considerăm în continuare problemele pe care trebuie să le rezolve un **supervizor** pentru evitarea blocării.
- Să presupunem că sistemul se găsește în starea  $s_k$  sigură și că există cererile de resurse date de  $Q[k]$ .
- **Supervizorul** trebuie să determine dacă orice cerere din  $Q[k]$  poate fi satisfăcută,  $y_{ij}[k] \leq v_j[k]$ .
- În general pot fi mai multe astfel de cereri și deci se pot prevedea diferite *criterii de alocare* pentru a stabili care cerere să fie satisfăcută mai întâi, de exemplu:

**FIFO** - (First In First Out)-primul venit primul servit

**SJF** - (Shortest Job First)- sarcina cea mai scurtă, prima servită;

**ROUND ROBIN** - rulare prin rotație;

Priorități atribuite sarcinilor;

etc.

- Pentru a determina dacă o stare  $s_k$  este sigură ca urmare a satisfacerii unor cereri de resurse trebuie căutată o secvență adecvată de pași.
- Pentru aceasta se presupune că resursele disponibile sunt complete cu cele alocate sarcinilor în curs de execuție.
- Aceasta este echivalentă cu presupunerea că primul pas într-o secvență analizată nu începe până când toți pașii în curs de execuție au toate condițiile pentru a se finaliza.

# Excluderea mutuală

## Definiții. Concepțe de bază.

- Într-un sistem de sarcini concurente, acestea pot interacționa în două moduri:
  - indirect, prin disputa pentru aceleași resurse;
  - direct, prin transmitere / recepție de mesaje sau informații de stare.
- Când se execută sarcini independente evoluția acestora poate fi controlată de către supervisor aşa cum am văzut la analiza situațiilor de blocare.
- Totuși când sarcinile interacționează în mod direct mecanismul de control al evoluției lor trebuie să apară în mod explicit în programele care le definesc.
- Interacțiunea directă se efectuează prin mecanisme specifice contextului în care evoluează sistemul de sarcini concurente.

Contextul în care evoluează un sistem de sarcini concurente poate fi:

- centralizat, dacă există o memorie comună accesibilă pentru citire scriere tuturor sarcinilor, sau
- distribuit dacă nu există memorie comună, ci doar memorie locală accesibilă unei sarcini și inaccesibilă celorlalte sarcini.

- Excluderea mutuală se ocupă de problemele legate de controlul resurselor **reutilizabile** (acele resurse a căror stare internă este modificată în timpul utilizării dar care poate fi initializată la atribuirea lor altor sarcini) astfel ca la un moment dat să fie utilizate de o singură sarcină.

- Deoarece sarcinile modifică starea resursei în timpul utilizării, este necesar ca o resursă să fie alocată unei singure sarcini la un moment dat pentru a asigura o execuție corectă a sarcinilor.
- Pe de altă parte, deoarece o sarcină initializează starea unei resurse înainte de a o utiliza, ordinea în care mai multe sarcini o utilizează este indiferentă.
- Astfel orice soluție pentru excluderea mutuală care implică a priori constrângeri privind ordinea de execuție a sarcinilor este neadecvată.
- O soluție corectă trebuie să se aplique unui sistem de sarcini independente.
- Dacă utilizarea resurselor este sub controlul unui sistem de operare centralizat, la construirea unui sistem de sarcini trebuie să se țină seama numai de convențiile prin care se comunică sistemului de operare cererile și eliberările de resurse.

Vom considera că excluderea mutuală trebuie rezolvată de însuși sistemul de sarcini, utilizând *variabile globale* sau variabile de stare și mesaje pentru comunicația între sarcini în funcție de context: centralizat sau distribuit.

- Sarcinile care trebuie să se execute mutual exclusiv constituie secțiuni critice.
- O soluție corectă pentru excluderea mutuală trebuie să eliminate apariția unor rezultate diferite sau blocari în tranzițiile sistemului în condițiile unor timpi de execuție a sarcinilor diferenți, pentru diferite procesoare.
- De exemplu: Să considerăm posibilitatea ca două sarcini  $S_1$  și  $S_2$  care se execută pe procesoare diferenți să utilizeze în comun o resursă  $R$ .
- $S_1$  și  $S_2$  utilizează o variabilă globală care specifică disponibilitatea resursei  $R$ .

- $S_1$  testează variabila, găsește R disponibilă și înainte ca  $S_1$  să schimbe valoarea variabilei pentru a arăta că R este ocupată,  $S_2$  testează și ea variabilă și găsește R disponibilă.
- Mai concret  $S_1$  și  $S_2$  care evoluează într-un context centralizat au primit aceeași interpretare, citirea variabilei Semafor, incrementarea valorii citite și scrierea noii valori.

Fie, de exemplu, secvența  $SI_1 SI_2 SF_1 SF_2$ , deci

- 1°.  $S_1$  citește variabila Semafor;
- 2°.  $S_2$  citește variabila Semafor;
- 3°.  $S_1$  incrementeză și scrie noua valoare;
- 4°.  $S_2$  incrementeză și scrie noua valoare.

În urma acestei secvențe de execuție, Semafor primește valoarea Semafor + 1 în loc de Semafor + 2.

- Rezolvarea cererilor de acces la memorie pentru **UCP** și **DMA** reprezintă un exemplu de excludere mutuală la nivel hardware.
- Scrierea unor texte la o imprimantă comună într-o rețea locală de calculatoare constituie un exemplu de excludere mutuală, la nivel de aplicații.
- Accesul la o înregistrare dintr-un fișier pentru a fi citită și modificată de mai multe sarcini constituie deasemenea un exemplu de excludere mutuală în context distribuit.
- Pentru simplitate vom considera în continuare o singură resursă R care poate fi utilizată de o singură sarcină la un moment dat.

## Definiție:

Fie  $\alpha = a_1, a_2, \dots, a_k$  o secvență de execuție parțială a sistemului de sarcini  $C=(S,<)$ . Sarcinile  $S$  și  $S'$  sunt **mutual exclusive** în  $\alpha$  dacă și numai dacă cel mult una din ele este activă după orice prefix a lui  $\alpha$ .

Pentru a implementa excluderea mutuală vom considera că o soluție este corectă dacă cel mult o sarcină este în secțiunea critică la un moment dat și sunt satisfacute următoarele condiții adiționale:

- Nu se face nici o presupunere relativă la activitatea și numărul de procesoare pe care se execută sistemul de sarcini.

Desigur, aşa cum s-a arătat și mai sus, se consideră ca citirea și scrierea unei variabile de memorie sunt operații atomice, indivizibile. Lansarea simultană a unor astfel de operații va avea ca efect execuția lor secvențială, într-o ordine oarecare, necunoscută apriori.

- Soluția implementată nu trebuie să fie afectată de vitezele relative ale sarcinilor care se exclud mutual, considerând că aceste viteze sunt diferite de zero.
- O sarcină care operează în afara secțiunii critice nu poate bloca altă sarcină de la intrarea în secțiunea critică.
- Două sarcini care doresc să intre în secțiunea critică nu pot fi într-o buclă de aşteptare reciprocă.

Deci dacă se elibereză secțiunea critică și există sarcini în aşteptare, este exclusă posibilitatea ca nici un proces să nu intre în secțiunea critică. În plus, o sarcină care dorește să intre în secțiunea critică nu poate fi în stare de aşteptare un timp nedefinit.

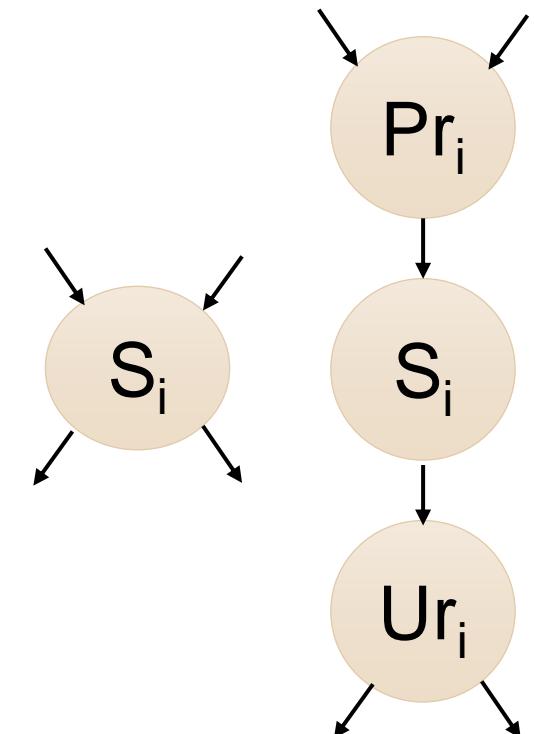
- Toate sarcinile trebuie tratate la fel, fără priorități sau alte privilegii.

# Algoritmi de excludere mutuală în mediu centralizat

- Complexitatea soluției pentru excluderea mutuală depinde de tipurile de operații indivizibile care se pot efectua asupra **variabilelor globale**.
- Astfel dacă se lasă pe seama proiectantului hardware responsabilitatea implementării excluderii mutuale prin operații indivizibile mai elaborate, se obține o soluție mai simplă pentru excluderea mutuală decât dacă se consideră ca operații indivizibile simple citiri și scrieri în memorie.
- Astfel, sunt calculatoare care au instrucțiuni de tipul Test and Set (Read Modify Write), sau chiar primitive de sincronizare indivizibile, implementate ca operații atomice.

- Dacă cel puțin două sarcini independente ale unui sistem **C** trebuie să utilizeze mutual exclusiv resursa R vom modifica **C** pentru a asigura această cerință.
  - Pentru fiecare  $S_i$  care utilizează resursa R vom introduce sarcinile  $Pr_i$  și  $Ur_i$  astfel că:
    - $Pr_i$  precede imediat sarcina  $S_i$ , iar
    - $Ur_i$  urmează imediat sarcinii  $S_i$ .
  - Vom nota cu **C'** sistemul de sarcini extins. Cu toate că  $S_i$  rămâne neinterpretat  $Pr_i$  și  $Ur_i$  vor fi complet specificate.
  - $Pr_i$  va fi astfel proiectat încât să poată fi terminat și deci să permită inițierea lui  $S_i$  dacă și numai dacă pentru  $\forall j \neq i$ ,  $Pr_j$  nu poate fi terminat.
  - Funcția principală a lui  $Ur_i$  este de a comunica celorlalte sarcini că  $S_i$  a terminat utilizarea lui R.
- ```

do{
    Entry Section
    Critical Section
    Exit Section
} while (TRUE);
  
```



- Pentru a projecța pe  $Pr_i$  și  $Ur_i$  astfel ca să nu existe subsecvențe de execuție de forma:

$PrF_i \quad PrF_j \quad UrF_i$

vom considera ca variabile globale:

$p$  - o variabilă de tip întreg.

$Q = (Q[0], Q[1], \dots, Q[n])$  - un vector de  $n+1$  variabile de tip întreg.

Inițial  $p=0$  și  $Q[i]=0$ ,  $0 \leq i \leq n$ .

- **Funcția vectorului  $Q$**  este de a memora într-o ordine **FIFO** lista sarcinilor ce așteaptă utilizarea lui  $R$ . Acest lucru se realizează prin memorarea indicelui sarcinei care utilizează pe  $R$  în  $Q$ .
- Astfel, coada de așteptare la resursa  $R$  este organizată ca o simplă listă înlănțuită, memorată în  $Q$ .
- *Variabila  $p$*  conține indicele ultimului element din lista (ultimul element intrat în coada  $Q$ ).
- Condiția de coadă vidă, considerată ca și condiție inițială este dată de  $p=0$  și  $Q[0] = 0$ .

# Definirea sarcinilor auxiliare $Pr_i$ și $Ur_i$

Procesele auxiliare  $Pr_i$  și  $Ur_i$  pot fi specificate prin organigramele următoare

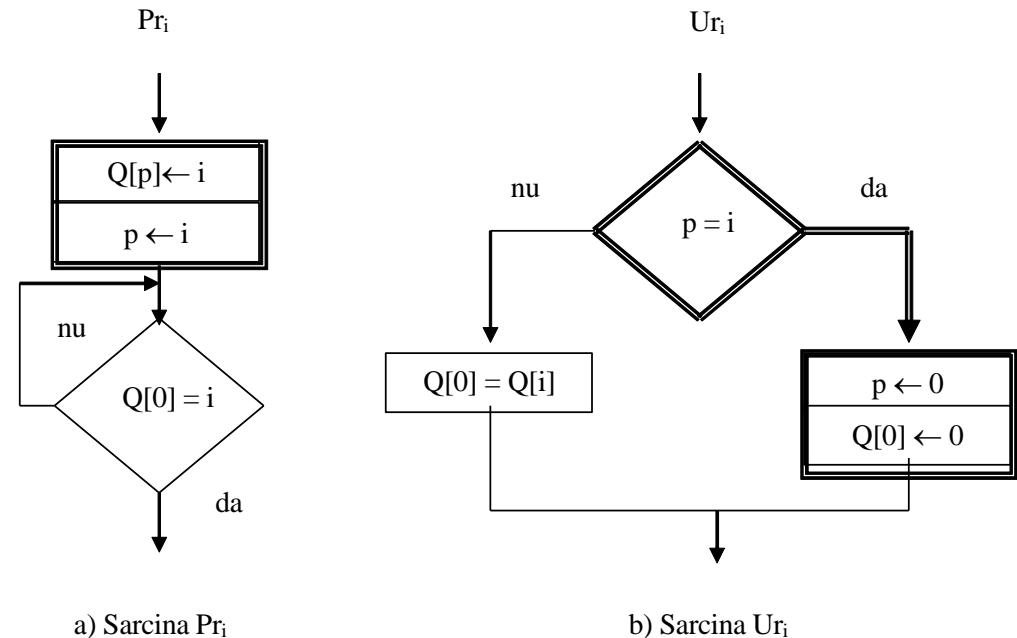
**Sarcina  $Pr_i$**  arată intenția lui  $S_i$  de a utiliza resursa  $R$ , introduce pe  $S_i$  în coada  $Q$  și așteaptă într-o buclă până când  $S_i$  ajunge în capul listei  $Q[0]$ .

**Sarcina  $Ur_i$**  înlătură pe  $S_i$  din coadă și trece următoarea sarcină pentru execuție în capul listei.

Dacă  $S_i$  a fost ultima sarcină introdusă în coadă

( $p=i$  când se execută  $Ur_i$ )

atunci coada este vidă și se trec în zero  $p$  și  $Q[0]$ .



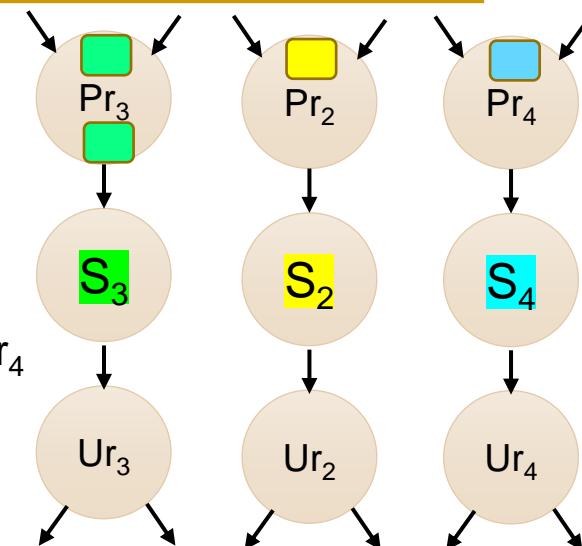
Pentru a putea elimina posibilitatea interferențelor între  $Pr_i, Pr_j$  sau  $Pr_i, Ur_j$  ce se execută concurrent trebuie ca anumite operații asupra variabilelor globale să fie **indivizibile**, cele în chenar dublu.

# Fie un sistem de sarcini C=(S,<)

$S=\{S_1, S_2, S_3, S_4, S_5, \dots, S_n\}$

- Se arată modul de reprezentare în coada Q considerând solicitările de a intra în secțiunea critică în ordinea  $S_3 S_2 S_4$ .

$\alpha = \dots [Prl_3 \ Prl_2 \ Prl_4 \ PrF_3 \ SI_3 \ SF_3] Ur_3 \ PrF_2 \ SI_2 \ SF_2 [Ur_2 \ PrF_4 \ SI_4 \ SF_4] Ur_4 \dots$



|      |   |  |  |  |  |
|------|---|--|--|--|--|
| p    | 0 |  |  |  |  |
| Q[0] | 0 |  |  |  |  |
| Q[1] |   |  |  |  |  |
| Q[2] |   |  |  |  |  |
| Q[3] |   |  |  |  |  |
| Q[4] |   |  |  |  |  |

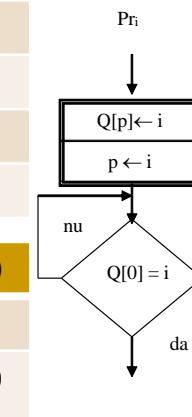
|      |   |   |  |  |  |
|------|---|---|--|--|--|
| p    | 0 | 3 |  |  |  |
| Q[0] | 0 | 3 |  |  |  |
| Q[1] |   |   |  |  |  |
| Q[2] |   |   |  |  |  |
| Q[3] |   |   |  |  |  |
| Q[4] |   |   |  |  |  |

|      |   |   |   |  |  |
|------|---|---|---|--|--|
| p    | 0 | 3 | 2 |  |  |
| Q[0] | 0 | 3 | 2 |  |  |
| Q[1] |   |   |   |  |  |
| Q[2] |   |   |   |  |  |
| Q[3] |   |   |   |  |  |
| Q[4] |   |   |   |  |  |

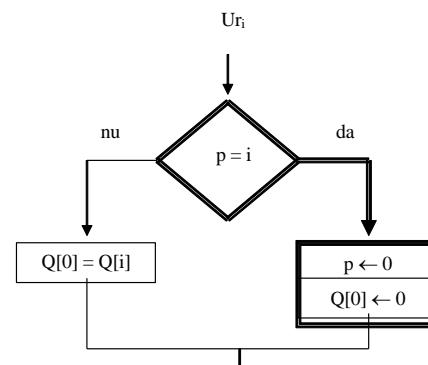
|      |   |   |   |   |  |
|------|---|---|---|---|--|
| p    | 0 | 3 | 2 | 4 |  |
| Q[0] | 0 | 3 | 2 |   |  |
| Q[1] |   |   |   |   |  |
| Q[2] |   |   |   |   |  |
| Q[3] |   |   |   |   |  |
| Q[4] |   |   |   |   |  |

|      |   |   |   |   |  |
|------|---|---|---|---|--|
| p    | 0 | 3 | 2 | 4 |  |
| Q[0] | 0 | 3 | 2 |   |  |
| Q[1] |   |   |   |   |  |
| Q[2] |   |   |   |   |  |
| Q[3] |   |   |   |   |  |
| Q[4] |   |   |   |   |  |

|      |   |   |   |   |   |
|------|---|---|---|---|---|
| p    | 0 | 3 | 2 | 4 | 0 |
| Q[0] | 0 | 3 | 2 |   |   |
| Q[1] |   |   |   |   |   |
| Q[2] |   |   |   |   |   |
| Q[3] |   |   |   |   |   |
| Q[4] |   |   |   |   |   |



a) Sarcina  $Pr_i$



b) Sarcina  $Ur_i$

Initial  $p = 0$  și  $Q[0] = 0$ .

**Exemplu** de interferență care conduce la o funcționare incorectă:

Să presupunem că  $p=Q[0]=0$  și că  $Pr_j$  începe execuția.

- **Situația 1.** După ce  $Pr_j$  se execută  $Q[0] \leftarrow j$  și înainte de a executa  $p \leftarrow j$ , o sarcină  $Pr_k$  cu viteza mai mare în execuție (pe un procesor mai rapid) execută ambele operații  $Q[p] \leftarrow k ; p \leftarrow k$

$S_k$  se va executa imediat, dar cererea pentru  $S_j$  s-a pierdut deoarece  $Q[0]$  a fost rescris de  $Pr_k$ .

Considerând *cele două operații indivizibile*  $Q[p] \leftarrow i ; p \leftarrow i$

se elimină astfel de interferențe.

- **Situația 2.** Să presupunem că  $p$  a fost testat de  $Ur_i$  și a fost găsit egal cu  $i$  ( $p=i$ ).

Inainte ca  $Ur_i$  să steargă  $Q[0]$  și  $p$   $Q[0] \leftarrow 0 ; p \leftarrow 0$  se execută sarcina  $Pr_k$ . Înainte de a se testa egalitatea  $Q[0]=k$  în cadrul sarcinii  $Pr_k$ ,  $Ur_i$  se execută mai departe trecând pe  $Q[0]$  și  $p$  în zero.

Este evident că  $Pr_k$  nu se va executa niciodată, *deci operațiile implicate de  $Ur_i$  trebuie să fie indivizibile*.

- În procedura de realizare a excluderii mutuale arătate anterior **primitiva de aşteptare** Pr<sub>i</sub> ocupă procesorul, ceea ce duce la utilizarea ineficientă a acestuia în sisteme în care un număr mare de sarcini intră în competiție pentru resursa R.
- În același timp apare o puternică interferență la memorie prin accesul la variabilele globale; operațiile indivizibile fiind destul de lungi.
- Soluția de excludere mutuală prezentată are marea dezavantaj că necesită execuția unor operații indivizibile complicate, ceea ce de cele mai multe ori nu este posibil prin arhitectura procesoarelor sau chiar dacă este posibil, duce la scăderea performanțelor sistemului.

În continuare vom prezenta o sută de algoritmi, în ordinea dezvoltării lor, care presupun că singurele operații indivizibile sunt citirea și scrierea unei locații de memorie.

- Dacă există mai multe cereri de citire sau scriere, ele se vor executa serial, într-o ordine oarecare, nespecificată.
- Prezentarea acestor algoritmi se va face, pe cât posibil, în forma în care au fost publicați, desprinderea sarcinilor Pr<sub>i</sub> și Ur<sub>i</sub> fiind banală.

# Algoritmi de excludere mutuală în mediu centralizat

- **Algoritmul lui Dekker** Dekker si Peterson  
IMPORTANT
- Algoritmul lui Dijkstra
- Algoritmul incorect al lui Hyman
- Algoritmul lui Knuth
- Algoritmul lui De Bruijn
- Algoritmul lui Eisenberg și Mac Quire
- **Algoritmul lui Peterson**
- Algoritmul lui Burns

## Algoritmul lui Dekker

- Algoritmul lui Dekker se aplică pentru 2 sarcini  $S_0$  și  $S_1$  care partajază variabilele:

```
wants_to_enter : array of 2 booleans  
turn : integer  
wants_to_enter[0] ← false  
wants_to_enter[1] ← false  
turn ← 0 // or 1
```

Semnificația acestor variabile este următoarea:

wants\_to\_enter[] ← false  $S_i$  nu dorește să intre în secțiunea critică.

wants\_to\_enter[] ← true  $S_i$  dorește să intre în secțiunea critică.

turn = i; Este rândul lui  $S_i$  să intre în secțiunea critică.

- Prin variabilele wants\_to\_enter sarcinile își dispută accesul la secțiunea critică.
- Dacă ele doresc simultan să intre în secțiunea critică, vor utiliza variabila comună turn pentru a arbitra această situație de simultaneitate.

```
variables
wants_to_enter : array of 2 booleans
turn : integer
wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0 // or 1
```

### S0:

```
wants_to_enter[0] ← true
while wants_to_enter[1] {
    if turn ≠ 0 {
        wants_to_enter[0] ← false
        while turn ≠ 0 {
            // busy wait
        }
        wants_to_enter[0] ← true
    }
}
// critical section
...
turn ← 1
wants_to_enter[0] ← false
// remainder section
```

### S1:

```
wants_to_enter[1] ← true
while wants_to_enter[0] {
    if turn ≠ 1 {
        wants_to_enter[1] ← false
        while turn ≠ 1 {
            // busy wait
        }
        wants_to_enter[1] ← true
    }
}
// critical section
...
turn ← 0
wants_to_enter[1] ← false
// remainder section
```

# Comentarii

- Sarcinile indică intenția de a intra în secțiunea critică care este testată de bucla exterioară while.
- Dacă celălalt proces nu a marcat intenția, secțiunea critică poate fi introdusă în siguranță, indiferent de tura curentă.
- Excluderea reciprocă va fi în continuare garantată, deoarece niciuna dintre sarcini nu poate intra în zona critică înainte de a-și seta indicatorul (ceea ce implică cel puțin o sarcina va intra în bucla while).
- Alternativ, dacă variabila celuilalt proces a fost setată, bucla while este activă și variabila turn va stabili cine are voie să devină critic.
- Sarcinile fără prioritate își vor retrage intenția de a intra în secțiunea critică până când li se va acorda din nou prioritate (bucla interioară while).
- Sarcinile cu prioritate vor intra în secțiunea lor critică.

# Comentariu

- Un avantaj al acestui algoritm este că nu necesită instrucțiuni speciale de testare și setare (citire / modificare / scriere atomică) și, prin urmare, este extrem de portabil pe arhitecturi de mașini.
- Un dezavantaj este că este limitat la două sarcini și folosește așteptarea ocupată în loc de suspendarea procesului. (Utilizarea așteptării ocupate sugerează că sarcinile ar trebui să petreacă o cantitate minimă de timp în secțiunea critică.)
- Sistemele de operare moderne oferă primitive de excludere reciprocă, care sunt mai generale și mai flexibile decât algoritmul lui Dekker.
- Cu toate acestea, în absența unei dispute reale între cele două sarcini, intrarea și ieșirea din secțiunea critică sunt extrem de eficiente atunci când este utilizat algoritmul lui Dekker.
- Multe procesoare moderne își execută instrucțiunile "out of order". Acest algoritm nu va funcționa pe mașinile cu procesoare "out of order".

# Algoritmul lui Peterson

- Algoritmul lui Peterson oferă o descriere algoritmică bună a rezolvării problemei secțiunilor critice și ilustrează unele dintre complexitățile implicate în proiectarea software-ului care răspunde cerințelor de excludere reciprocă, progres și aşteptare limitată.
- Într-un articol cu titlu destul de sugestiv pentru ce o să urmeze, "Myths about the Mutual Exclusion Problem" Peterson a elaborat un algoritm extrem de simplu și elegant.
- Sunt utilizate două variabile globale,  
flag - care arată poziția fiecărui sarcini față de secțiunea critică și  
turn - care rezolvă situațiile de simultaneitate.

## Algoritmul lui Peterson este:

```
bool flag[2] = {false, false};
```

```
int turn;
```

```
S0:    flag[0] = true;
```

```
S0_gate: turn = 1;
```

```
while (flag[1] == true && turn == 1)
```

```
{
```

```
    // busy wait
```

```
}
```

```
// critical section
```

```
...
```

```
// end of critical section
```

```
flag[0] = false;
```

```
S1:    flag[1] = true;
```

```
S1_gate: turn = 0;
```

```
while (flag[0] == true && turn == 0)
```

```
{
```

```
    // busy wait
```

```
}
```

```
// critical section
```

```
...
```

```
// end of critical section
```

```
flag[1] = false;
```

# Comentariu

- Algoritmul îndeplinește cele trei criterii esențiale pentru a rezolva problema secțiunii critice, cu condiția ca modificările variabilelor turn, flag [0] și flag [1] să se propage imediat și atomic.
- Cele trei criterii sunt excluderea reciprocă, progresul și așteptarea limitată.

## Excludere mutuală

- S0 și S1 nu pot fi niciodată în secțiunea critică în același timp:
- Dacă S0 se află în secțiunea critică, atunci flag [0] este adevărat.
  - În plus, fie flag [1] este fals (adică S1 și-a părăsit secțiunea critică), fie turn=0 (adică S1 încearcă acum să intre în secțiunea critică, dar așteaptă),
- Dacă ambele sarcini sunt în secțiunile lor critice, atunci starea trebuie să îndeplinească flag [0] și flag [1] și turn = 0 și turn = 1.
- Nici o stare nu poate satisface atât turn = 0, cât și turn = 1, deci nu poate exista o stare în care ambele sarcini se află în secțiunile lor critice.
- Deoarece turn poate lua una din cele două valori, poate fi înlocuit cu un singur bit, ceea ce înseamnă că algoritmul necesită doar trei biți de memorie.

**Progresul:** o sarcină nu poate intra imediat din nou în secțiunea critică dacă celălaltă sarcină și-a setat flag-ul pentru a spune că ar dori să intre în secțiunea sa critică.

**Așteptarea limitată :** în algoritmul lui Peterson, o sarcină nu va aștepta niciodată mai mult de o tură pentru intrarea în secțiunea critică.

# Algoritmul lui Peterson pentru mai multe sarcini

- Algoritmul de filtrare generalizează algoritmul lui Peterson la  $N > 2$  procese.
- În loc de un flag boolean, necesită o variabilă întreagă pe sarcină, stocată într-un singur registru atomic cu un scrisă de o singura sarcină și citită de mai multe sarcini (SWMR) și  $N-1$  variabile suplimentare în registre similare.
- Registrele pot fi reprezentate în pseudocod sub formă de tablouri:
- `level : array of N integers`
- `last_to_enter : array of N-1 integers`
- Variabilele de nivel iau valori de până la  $N - 1$ , fiecare reprezentând o „variabilă de așteptare” distinctă înainte de secțiunea critică.
- Sarcinile avansează de la o variabilă la alta, terminând cu variabila în celula  $N - 1$ , care este secțiunea critică. Mai exact, pentru a obține o blocare, sarcina Si va executa

$i \leftarrow Si$

for  $\ell$  from 0 to  $N-1$  exclusive

`level[i] ← ℓ`

`last_to_enter[ℓ] ← i`

    while `last_to_enter[ℓ] = i` and there exists  $k \neq i$ , such that `level[k] ≥ ℓ`

`wait`

Pentru a elibera blocarea la ieșirea din secțiunea critică, sarcina Si stabilește nivelul [i] la -1.

# Soluții hardware de excludere mutuală în mediu centralizat pentru sisteme monoprocesor

- În vederea realizării excluderii mutuale se pot prevedea diferite instrucțiuni sau primitive implementate la nivelul arhitecturii mașinii.
- Există o mare diversitate de astfel de soluții, iar în continuare vor fi considerate doar câteva exemple mai sugestive.
- Într-un sistem monoprocesor pot exista sarcini concurente care își dispută accesul la resurse.
- Apare și un paralelism efectiv între Unitatea Centrală de prelucrare și subsistemul de I/E care poate antrena și alte subansamble, cum ar fi:
  - DMA, canale de I/E, module de memorie,
- Relația între UCP și celealte subansamble este de tip "master-salve".

În special datorită sistemului de întreruperi, într-un sistem monoprocesor apare o întrețesere a execuției sarcinilor, iar anumite secțiuni ale acestora fiind regiuni critice care trebuie executate în mod exclusiv, fără a fi întrerupte.

- Pentru a realiza aceasta trebuie ca sistemul de întreprere SIntr să poată fi dezactivat / activat sau anumite nivele să fie mascate / demascate.

Astfel avem:

<început sarcina>;

*dezactivare\_SIntr;*

<secțiune critică>;

*activare\_SIntr;*

<rest sarcina>;

<început sarcina>;

*mascare\_într;*

<secțiune critică>;

*demascare\_într;*

<rest sarcina>;

- Desigur că ar exista o soluție pentru excluderea mutuală prin evitarea acesteia, prin eliminarea concurenței, dar aceasta nu este acceptabilă datorită degradării performanțelor.

# Soluții hardware de excludere mutuală în mediu centralizat pentru sisteme multiprocesor

Pentru sistemele multiprocesor realizarea excluderii mutuale depinde și mai mult de arhitectura sistemului. În principiu, trebuie realizată execuția indivizibilă a unor secvențe de acces la memoria comună.

Dintre soluțiile posibile amintim:

- instrucțiuni atomice de tip XCHG care schimbă între ele două locații de memorie sau un registru general și o locație de memorie;
- instrucțiuni atomice de tip TAS (Test And Set) sau RMW (Read Modify Write) care citesc o celulă de memorie, o modifică și actualizează noua valoare;
- instrucțiuni atomice de tip Lock, Unlock;
- instrucțiuni atomice de incrementare, decrementare;
- instrucțiuni atomice de înlocuire și adunare.

# Excludere mutuală cu instrucțiuni XCHG

- Instrucțiunea XCHG(r,m) interschimbă, ca o operație atomică, conținutul registrului r cu celula de memorie m (semafor).
  - deci  $r_i$  este o variabilă locală a sarcinii  $S_i$  și este inițializată cu 0, iar
  - m este o variabilă globală, comună tuturor sarcinilor care este inițializată cu 1.
- Ambele variabile pot lua valorile 0, 1.

Algoritmul pentru  $S_i$  va fi:

<început sarcina>;

*repetă*

    XCHG( $r_i, m$ );

*până\_când*  $r_i$ ;

    <secțiune critică>;

    XCHG( $r_i, m$ );

    <rest sarcina>;

- Numai procesul care găsește  $m=1$  va intra în secțiunea critică și va trece în zero. La ieșire din secțiunea critică va returna pe  $m$  la valoarea 1 pentru a permite celorlalte sarcini să intre în secțiunea critică. ( $m=1$  este o convenție, în unele sisteme se consideră  $m=0$ )
  - Când o sarcină  $S_i$  intră în secțiunea critică,  $r_i=1$ .
  - Deci pentru a realiza excluderea mutuală trebuie ca în orice moment să se verifice relația:
- $$m + \sum_{i=0}^{n-1} r_i = 1,$$

#### LOCK\_Si

|      |             |       |
|------|-------------|-------|
| AST: | MOV         | AL,0  |
|      | LOCK        |       |
|      | <b>XCHG</b> | AL,m  |
|      | TEST        | AL,AL |
|      | JZ          | AST   |

se intră în execuția sarcinii , zona critică

#### UNLOCK\_Si

|     |       |
|-----|-------|
| MOV | $m,1$ |
|-----|-------|

#### LOCK\_Sj

|      |             |       |
|------|-------------|-------|
| AST: | MOV         | AL,0  |
|      | LOCK        |       |
|      | <b>XCHG</b> | AL,m  |
|      | TEST        | AL,AL |
|      | JZ          | AST   |

se intră în execuția sarcinii , zona critică

#### UNLOCK\_Sj

|     |       |
|-----|-------|
| MOV | $m,1$ |
|-----|-------|

# Excluderea mutuală cu instrucțiuni TAS

- Instrucțiunea TAS(m) testează conținutul lui m și
  - dacă este 0,  $m \leftarrow 1$  și reîntoarce rezultat adevărat.
  - dacă m este 1, lasă conținutul nemodificat și întoarce rezultat fals.
- Variabila globală m este inițializată cu 0.

Pentru  $S_i$ , algoritmul este:

```
<început sarcina>;  
repetă  
până_când TAS(m);  
<secțiune critică>;  
 $m \leftarrow 0;$   
<rest sarcina>
```

- Având în vedere că instrucțiunea TAS(m) se execută strict serial, numai sarcina care găsește  $m=0$  va intra în secțiunea critică, și va face atribuirea  $m \leftarrow 1$ .
- La ieșirea din secțiunea critică va face  $m \leftarrow 0$  pentru a permite celorlalte sarcini să intre în secțiunea critică.
- Microprocesorul x68000 include în setul de instrucțiuni o instrucțiune TAS indivizibilă, pentru excludere mutuală în structuri multiprocesor.

# Excludere mutuală cu instrucțiuni LOCK, UNLOCK

Primitiva **LOCK(m)**  
este echivalentă cu:

{

*repetă nimic cât timp m=0;*

$m \leftarrow 0;$

}

Primitiva **UNLOCK(m)**  
este echivalentă cu:

{

$m \leftarrow 1;$

}

Pentru familia x86

LOCK\_Si

| AST: | MOV  | AL,0  |
|------|------|-------|
| LOCK | XCHG | AL,m  |
| TEST | TEST | AL,AL |
| JZ   | JZ   | AST   |

se intra in executia sarcinii , zona critica

UNLOCK\_Si

MOV      m,1

- Initializând variabila m cu 1, algoritmul de excludere mutuală este:

<început sarcina>

LOCK(m);

<secțiune critică>;

UNLOCK(m);

<rest sarcini>;

LOCK\_Si

|      |      |       |
|------|------|-------|
| AST: | MOV  | AL,0  |
|      | LOCK |       |
|      | XCHG | AL,m  |
|      | TEST | AL,AL |
|      | JZ   | AST   |

se intra in executia sarcinii , zona critica

UNLOCK\_Si

|     |     |
|-----|-----|
| MOV | m,1 |
|-----|-----|

- La unele procesoare, această funcție este realizată de instrucțiuni denumite TSB (Test and Switch Branch).

# MICROPROGRAMARE

## Noțiuni și concepte de bază

Organizarea memoriei de control

Suportul fizic pentru păstrarea micropogramelor

Organizarea logică a instrucțiunilor

Codificare verticală

Codificarea orizontală

Codificarea minimală

Codificarea cu control rezidual

Codificarea cu control prin adrese

Codificare mixtă

Implementarea microinstrucțiunilor

# NOTIUNI SI CONCEPTE DE BAZĂ



Din punct de vedere structural unităile de comandă sunt de două tipuri:  
-convenționale în sensul propus de Von Newmann ;  
-microprogramate conform conceptului introdus de M.Wilkes.

Microprogramarea este o tehnică de proiectare și implementare a funcțiilor de control a sistemelor de prelucrare a datelor numerice, ca o secvență de semnale de control ce interpretează static sau dinamic funcțiile de prelucrare a datelor. Semnalele de comandă, necesare la un moment dat pentru controlul primitivelor funcționale sunt organizate într-un cuvânt de control, memorat într-o memorie PROM sau RAM. Structura cuvântului de control este influențată de semnificația atribuită noțiunii de microoperărie .

Microoperăria ( $\mu O$ ) este o primitivă a funcțiilor de prelucrare a datelor, care reprezintă o operație elementară asupra unei primitive funcționale (transfer, acțiune de înscriere sau de incrementare, activare pe magistrală, etc. ), ce se desfășoară de obicei într-o perioadă de timp (perioada de fact sau de ceas).

Microoperăria implică existența unui operator, care corespunde unei unități funcționale și a operanzilor asupra căror acționează.

Cuvântul de control este pus în corespondență cu noțiunea de microinstructiune.

Microinstructiunea ( $\mu I$ ) este un set de microoperări independente de date, fără conflict de resurse, care se pot executa simultan (pe perioada unei perioade de sincronizare).



## TIPURI DE MICROUNSTRUCTIUNI

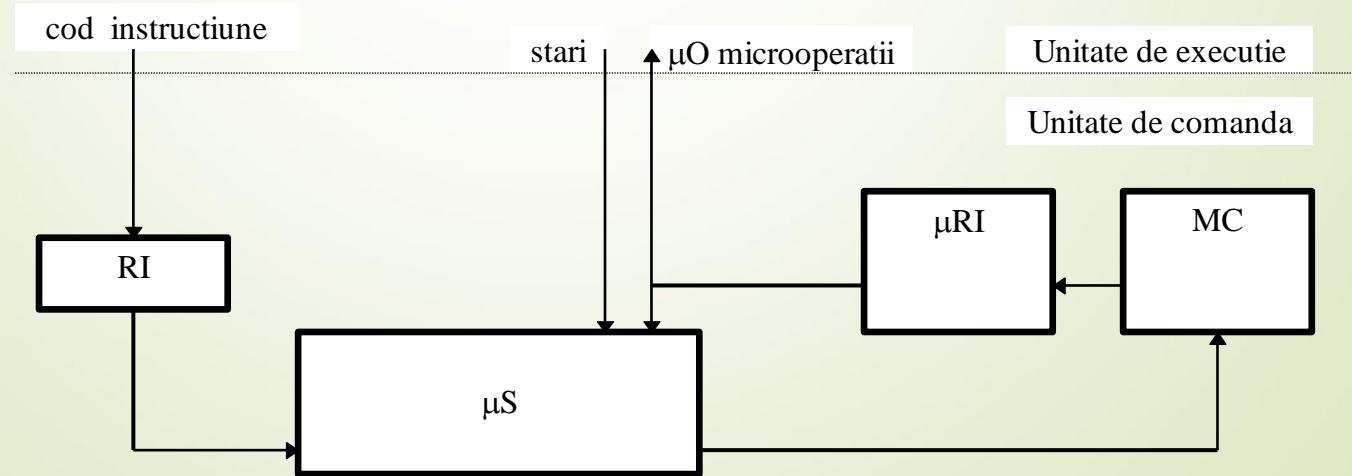
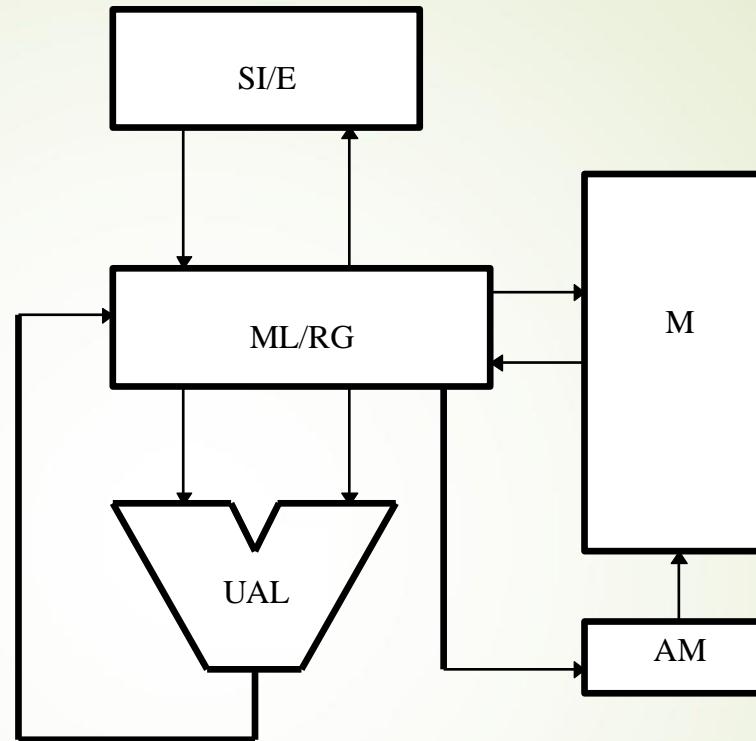
- microinstructiuni operationale care controlează primitivele functionale ale unității de execuție a sistemului numeric, asigurând fluxul de informație și acțiunile asupra resurselor;
- microinstructiune de ramificație (de salt) care inspectează starea primitivelor functionale și asigură ramificația în algoritmul de control, constituind suportul pentru implementarea deciziilor.



Prin microprogram se înțelege o secvență de microinstrucțiuni ce implementează un algoritm care descrie :

- citirea interpretarea și execuția unui set de instrucțiuni mașină;
- primitive ale sistemului de operare;
- primitive ale limbajelor de programare;
- etc.

# Structura generală a unui sistem de calcul cu unitate de comandă microprogramată



# PRIMITIVE FUNCTIONALE



- **M** - memoria principală a sistemului, în care se păstrează programele ca secvență de instrucțiuni mașină și datele care se prelucrează ;
- **ML** - memoria locală a sistemului, reprezentată de registrele generale de lucru accesibile sau nu utilizatorului;
- **UAL** - unitatea aritmetică logică ;
- **S I/E** - subsistemul de intrări / ieșiri ;
- **MC** - memoria de control în care se păstrează microprogramul ca secvență de microinstrucțiuni ;
- **$\mu$ RI** - registrul de microinstrucțiuni, care păstrează microinstrucțiunea curentă ce se execută. Conținutul său specifică toate microoperațiile care se execută în acel moment în unitatea de execuție ;
- **$\mu$ S** - microsecvențiatorul, unitatea de comandă convențională, elementară, care asigură citirea interpretarea și execuția microinstrucțiunilor din memoria de control precum și înlănțuirea acestora pe baza registrului de instrucțiuni mașină RI și a stării primitivelor funcționale.



- O structură microprogramată se caracterizează prin:
- organizarea memoriei de control ;
  - suportul fizic pentru păstrarea microprogramelor
  - organizarea logică a microinstructiunilor ;
  - implementarea microinstructiunilor



## 8.1.1. ORGANIZAREA MEMORIEI DE CONTROL



Dupa criteriu de analiză relația MC față de M:

- memorie de control separată de memoria principală, atât din punct de vedere fizic cât și din punct de vedere al adresării logice,
- Raportul dintre viteza de lucru a memoriei de control și cea a memoriei principale este de circa 10 în favoarea memoriei de control,  $V_{MC} \gg V_M$   
- memoria de control este implementată în același spațiu fizic și de adresare cu cel al memoriei principale. Ambele au același ciclu de memorie și sunt adresațe prin aceeași logică. Divizarea în memorie de control și memorie principală se poate face la nivel fizic sau la nivel logic.

Acest tip de organizare cere ca memoria să fie suficient de rapidă pentru a fi folosită ca MC și destul de ieftină pentru a fi folosită ca memorie principală.

-memoria de control este implementată separat de memoria principală însă este încărcată din aceasta. Se utilizează un sistem de memorie ierarhică

Memoria MC este încărcată din memoria principală prin intermediul memoriei tampon MT.



Organizarea MC în cadrul MP

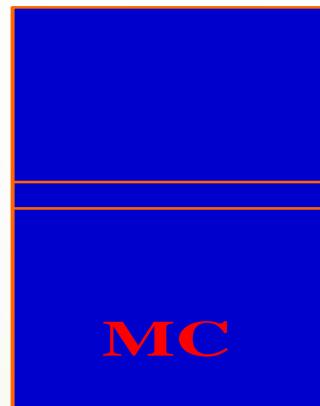
Organizarea memoriei de control

Organizarea MC separată de MP  
dar încarcabilă din aceasta



# CLASIFICAREA MEMORIEI DE CONTROL MC

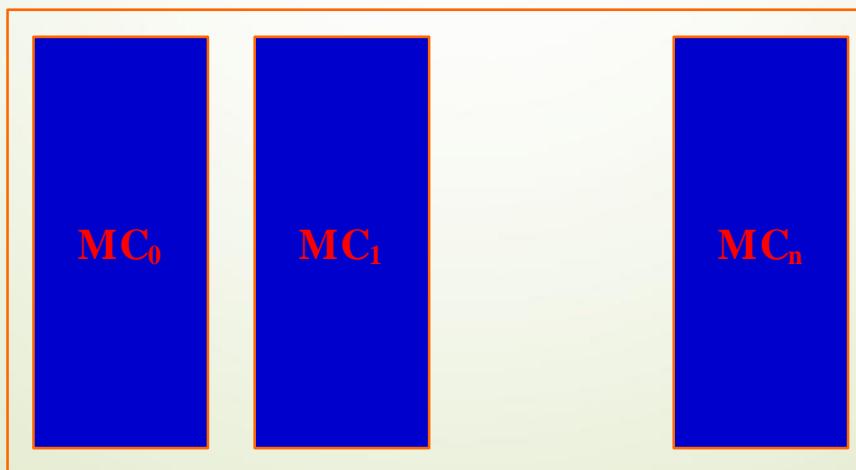
1. **memorie de control cu o microinstructiune pe cuvânt**, în care fiecărui cuvânt de memorie îi corespunde o singură microinstructiune. Citirea unei microinstructiuni presupune un singur acces la MC.



## 2. memorie de control cu organizare pe pagini

Unei adrese din memoria de control i se asociază mai multe microinstructiuni, din pagini diferite. În acest fel se asigură la nivel de microprogram execuția unei microinstructiuni de tip CASE, care introduce facilitatea de decizii multiple.

O adresă de MC va adresa aceeași locație în toate paginile mémoriei de control iar vectorul de condiții de test va activa pagina ce specifică microinstructiunea următoare care se va executa.



### **3. memorie de control cu organizare pe blocuri**

Pentru acest tip de organizare există două feluri de adrese:

- adrese de microinstructiuni din același bloc cu microinstructiunea curentă ;
- adrese de blocuri.

Împărțirea memoriei de microprograme în blocuri se face ținând seama atât de structura microprogramului cât și de resursele fizice disponibile. O organizare de acest fel conduce, în general, la micșorarea lungimii microinstructiunii însă introduce timp suplimentar cu comutarea adreselor de blocuri.



#### **4. memorie de control divizată**

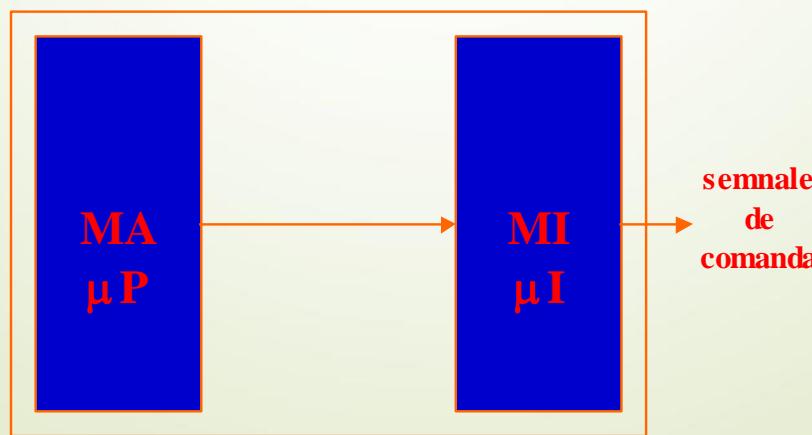
Memoria de microprograme divizată este alcătuită din două unități de memorie distincte :

MI - memorie de microinstructiuni, care păstrează toate microinstructiunile distincte posibile necesare pentru controlul resurselor fizice ;

MA -memorie de adrese de microinstructiuni, care păstrează programul specificat, nu prin microinstructiuni ce controlează resursele fizice, ci prin adrese de microinstructiuni (adrese pentru memoria MI).

În general numărul de tipuri de microinstructiuni distincte este mult mai redus decât micropogramul în sine, ceea ce implică ca numărul de biți necesari pentru adresarea memoriei MI să fie și el redus. În acest fel, lungimea cuvântului din memoria MA este mult mai mic decât al memoriei MI, ceea ce conduce la o reducere substanțială a memoriei de control.

În schimb, pentru a executa o microinstructiune trebuie făcute două adresări, una la memoria MA și una la memoria MI ceea ce conduce la un ciclu mai mare de microinstructiune.



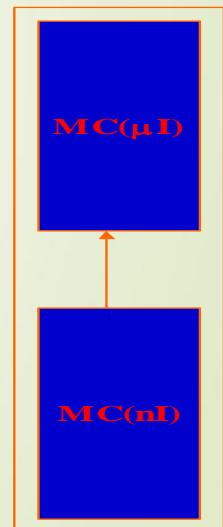
## **5. memorie de control structurată pe două niveluri**

La o astfel de organizare mecanismul citirii interpretării și execuției unei instrucțiuni mașină este următorul :

- o instrucțiune mașină este interpretată de un set de microinstrucțiuni rezident în memoria de control MC ( $\mu$ I) ;
- la rândul ei, o microinstrucțiune este interpretată de o secvență de nanoinstrucțiuni rezidentă în memoria MC (nI) .

Această tehnică a nanoprogramării este conceptual echivalentă cu microprogramarea, structurarea pe două niveluri oferă o flexibilitate mai mare și posibilitatea implementării unor structuri de control foarte complexe.

O astfel de organizare întâlnim la calculatorul NANODATA QM1.





# SUPORTUL FIZIC PENTRU PASTRAREA MICROPROGRAMELOR

Memoria de microprograme MC este o unitate de memorie de mare viteză care păstrează microprogramele ce se execută.

În ceea ce privește suportul fizic, pentru păstrarea microprogramelor, acesta poate fi realizat cu memorii PROM sau cu memorii RAM.

Realizarea memoriei de control :

-cu componente de tip RAM conferă o caracteristică statică

- componentelor de tip RAM oferă o caracteristică dinamică ce permite utilizatorului ca printr-un ansamblu de mijloace software să aibă acces la microprogramul din memoria de control..

Este foarte important în a face deosebire între :

- mașini microprogramate, și
- mașini microprogramabile

Prima categorie se referă la modalitatea de implementare a unității de comandă în sensul conceptului introdus de Wilkes fără a oferi resursele hardware și suportul de programe pentru accesul utilizatorului la nivelul microprogramului.

Cea de a două categorie oferă atât resursele hardware cât și facilitățile software pentru accesul utilizatorului la nivelul microinstructiunilor.



# ORGANIZAREA LOGICA A INSTRUCTIUNILOR

Un cuvânt din memoria de control specifică un set de microoperații ce

constituie componentele primitive ale controlului resurselor sistemului.  
Organizarea logică a microinstructiunilor este influențată de :

- gradul de paralelism între microoperații, ce se dorește realizat;
- structura mașinii de bază ;
- gradul de codificare sau de flexibilitate dorit ;
- gradul de optimizare al lungimii cuvântului de control .

Presupunând că micropogramele sunt specificate ca o secvență de

seturi disjuncte de microoperații, se pot distinge mai multe modalități de codificare a acestora și anume:

- codificare verticală sau maximală ;
- codificare orizontală sau cu control direct ;
- codificare minimală ;
- codificare cu control rezidual ;
- codificare cu control prin adrese ;
- codificare mixtă.



# CODIFICARE VERTICALA

Fiecare microinstructiune operatiunala specifica o singura microoperatie. Setul de microoperatii (MO) necesar pentru controlul primitivelor functionale se codifica in  $\lceil \log_2 |(MO)| \rceil$  biti, ce constituie lungimea cuvantului din MC.

Aceasta codificare reprezinta un caz extrem, deoarece elimină orice posibilitate de desfășurare paralelă a operațiilor elementare.

Din punctul de vedere al minimizării cuvantului de control, codificarea verticală

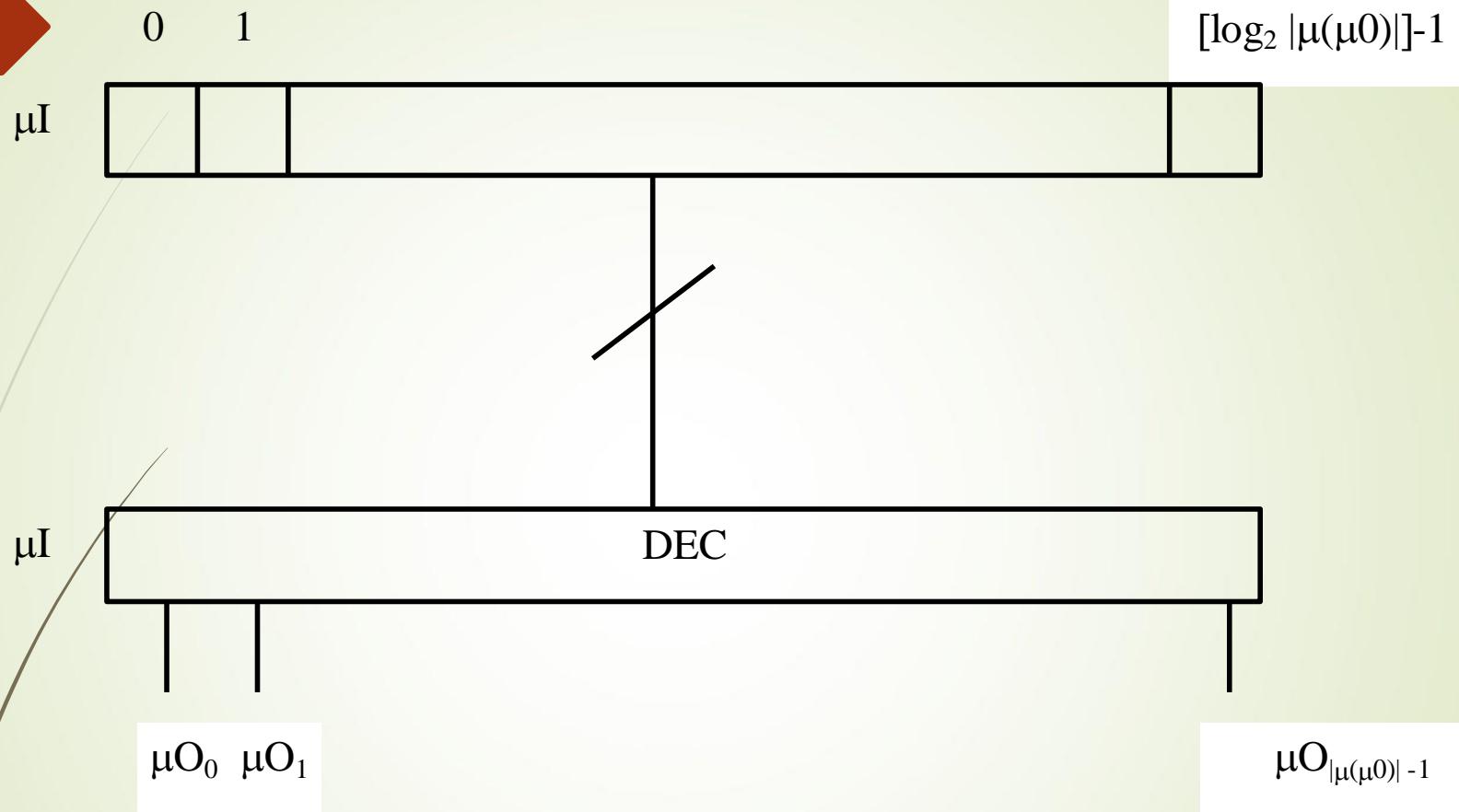
implică numărul cel mai mic de biti. Dimensiunea mare a decodificatorului face ca

realizarea fizică a acestuia să aibe loc pe mai multe niveluri, ceea ce conduce la

introducerea de întârzieri

Un dezavantaj major al codificării maximale îl reprezintă eliminarea controlului

paralel asupra resurselor precum și inflexibilitatea dezvoltării sau completării sistemului în ceea ce privește introducerea de noi microoperatii. Este aplicabilă numai în sisteme dedicate care au o structură specifică.



**Codificarea verticală a microoperațiilor**



# CODIFICAREA ORIZONTALA

În cadrul acestei codificări, fiecare microoperătie din setul (MO) este pusă în corespondență cu un bit din cadrul cuvântului de control. Controlul microoperațiilor se face în mod direct.



## Codificarea orizontală

Codificarea orizontală realizează controlul tuturor microoperațiilor paralele, posibile, ce se pot desfășura în sistem.

Deși oferă o flexibilitate mare și asigură paralelismul maxim, utilizarea acestei codificări este un caz extrem datorită folosirii ineficiente a memoriei de control.

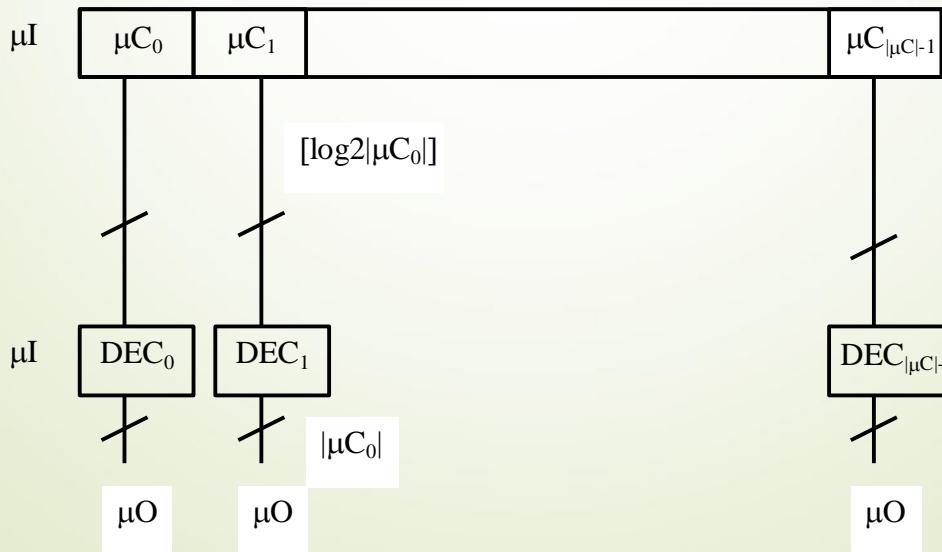


# CODIFICAREA MINIMALA

Combină flexibilitatea și paralelismul potențial oferite de codificarea orizontală cu eficiența codificării verticale.

Ideea de bază este de a grupa în clase de compatibilitate setul de microoperații care se exclud reciproc ( $\mu O$  dintr-o clasă de compatibilitate nu se vor efectua niciodată simultan).

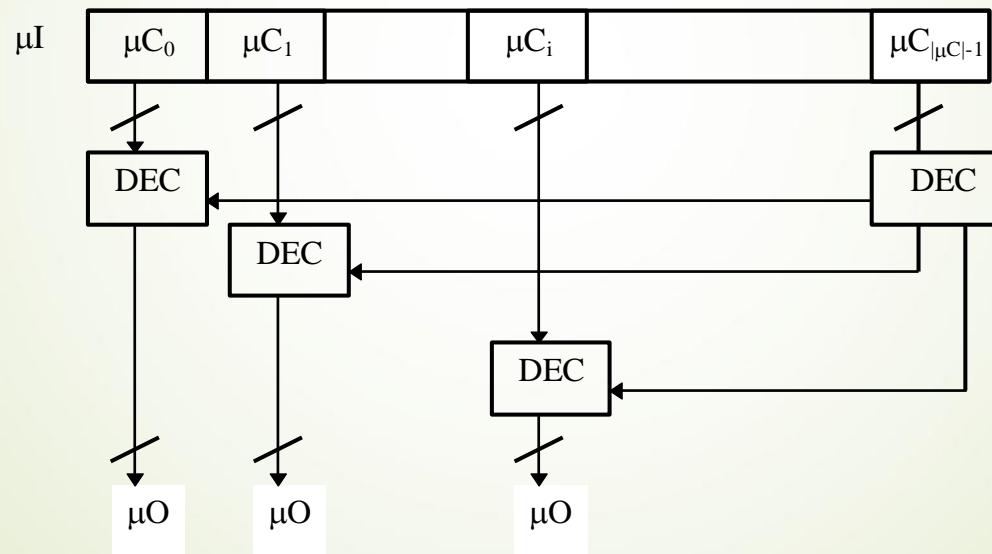
Microinstrucțiunea este împărțită în câmpuri. Un câmp corespunde unei clase de compatibilitate. La nivel de câmpuri se realizează o codificare orizontală iar în cadrul câmpurilor se realizează o codificare verticală.



Pentru un câmp  $\mu C_j$  care codifică  $|\mu C_j|$  microoperații sunt necesari  $\lceil \log_2(|\mu C_j|+1) \rceil$  biți, deoarece trebuie să se prevadă și posibilitatea de a nu specifica nici o microoperație din cadrul câmpului.

O variantă a acestei codificări o reprezintă codificarea pe două niveluri sau indirectă.

În codificarea pe două niveluri, validarea unor câmpuri depinde de valoarea altui câmp de control din microinstrucțiune.

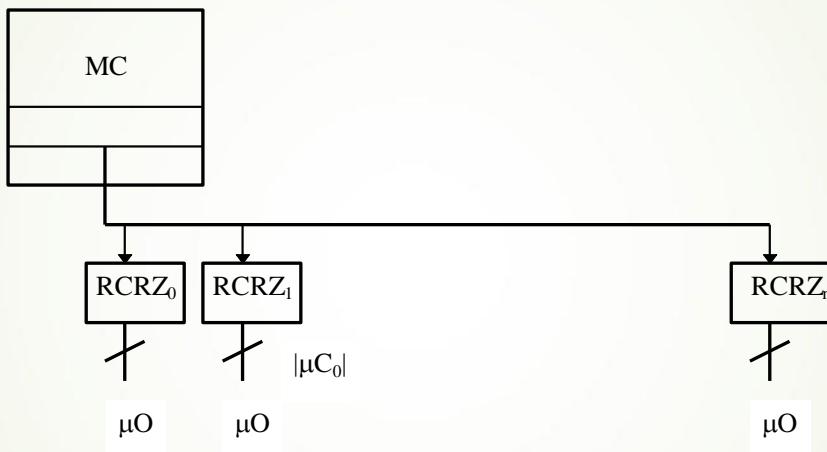


Codificare minimală pe două niveluri



# CODIFICAREA CU CONTROL REZIDUAL

Această metodă de codificare folosește pentru controlul primitivelor funcționale registre de control rezidual. Cuvântul de control nu controlează direct resursele ci prin intermediul registrelor de control rezidual încărcate sub acțiunea microinstructiunilor



## Codificare cu control rezidual

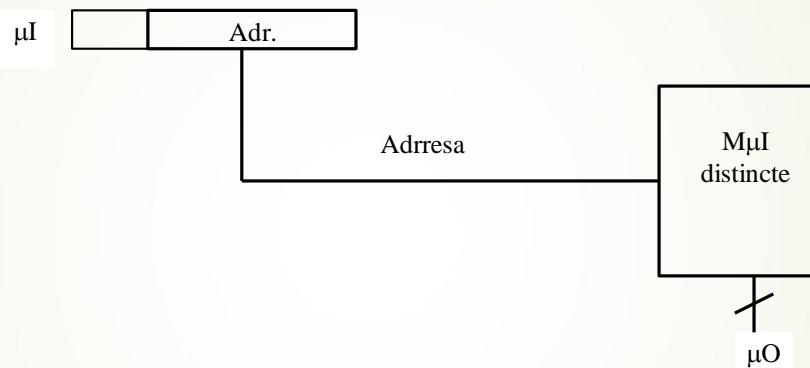
Microinstructiunile pot înlocui sau modifica valoarea unuia sau mai multor registre de control. Astfel, se asigură o economie de memorie de control atunci când unele primitive funcționale realizează aceeași operație în mod repetat sau când un set de microoperații este activ o perioadă mare de timp, iar alte seturi de microoperații se modifică.

Registrele de control rezidual RCRZ<sub>j</sub>, care specifică microoperațiile de control al resurselor hardware pot fi manevrate cu ajutorul unor microinstructiuni de dimensiuni reduse.



# CODIFICAREA CU CONTROL PRIN ADRESE

O modalitate de implementare a microinstructiunilor operationale este aceea în care se specifică o adresă în cadrul unei memorii, unde sunt memorate toate microinstructiunile distincte posibile ce controlează sistemul.



## Codificare cu control prin adrese

Numărul de microinstructiuni distincte nu depinde de numărul de resurse controlate ci de numărul de  $\mu O$  disticte, de lungimea  $\mu P$  și de numărul de variabile de stare testate.

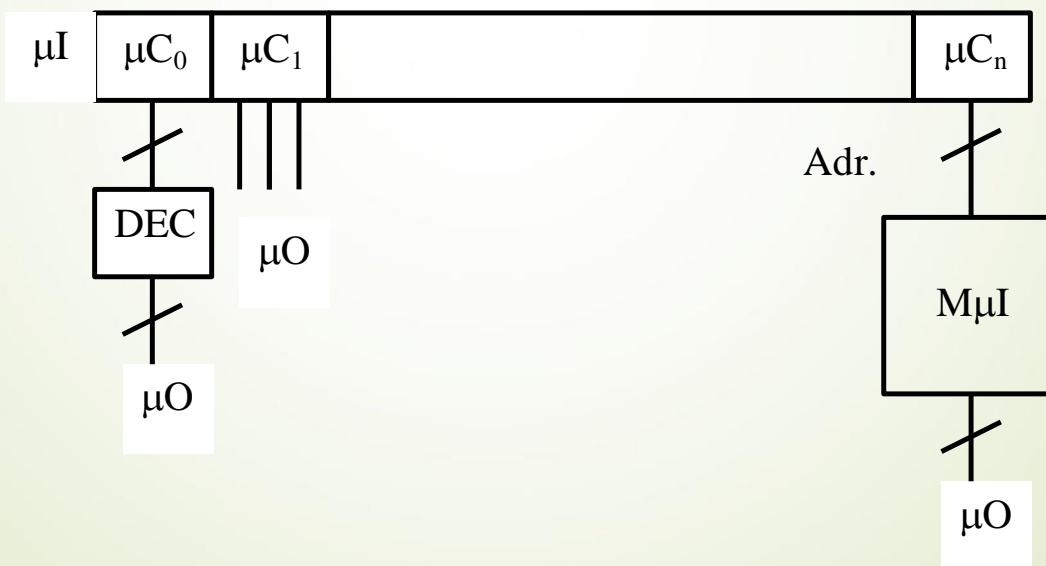
Memoria care păstrează microinstructiunile distincte va avea lungimea cuvântului suficient de mare pentru a controla toate microoperațiile care se pot efectua simultan. Trebuie notat că fiecare  $\mu I$  este memorată o singură dată.

O astfel de implementare, face ca microprogramul să fie format dintr-o secvență de adrese care apelează  $\mu I$  păstrate în memoria de  $\mu I$ .



# CODIFICAREA MIXTA

O variantă utilizată mult în practică este aceea în care microinstructiunea este împărțită în câmpuri. Unele câmpuri controlează direct microoperațiile (sub formă codificată sau directă) iar altele specifică adrese de memorie ce conține un subset de microinstructiuni distincte.





# IMPLEMENTAREA INSTRUCTIUNILOR

Microinstructiunile sunt citite, interpretate și executate de către microsecentiator ( $\mu S$ ) în același fel în care o unitate de comandă convențională execută instrucțiuni mașină.

Se poate defini o caracteristică serie-paralel care măsoară cantitatea de suprapunere între faza de execuție a unui curent și citirea, interpretarea următoare.

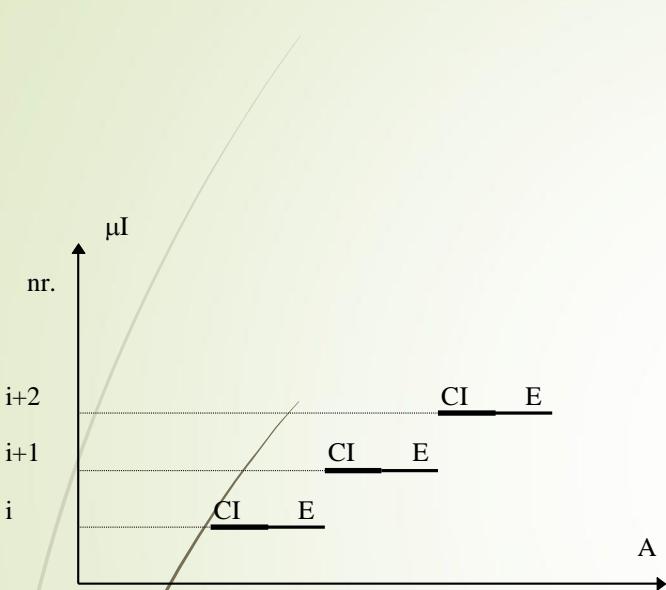
Din punctul de vedere al caracteristicii serie-paralel distingem trei tipuri de implementări :

- implementare serie;
- implementare serie-paralelă ;
- implementare paralelă.

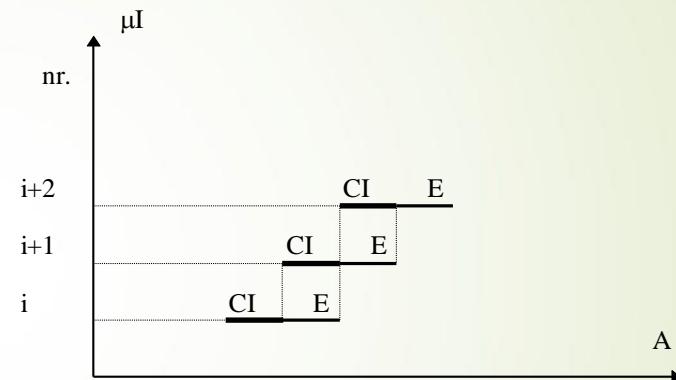
Fie CI-faza de citire interpretare și E-faza de execuție a unei  $\mu l$ .

În implementarea serie, citirea microinstructiunii următoare nu începe decât după ce s-a terminat execuția microinstructiunii curente.

În implementarea paralelă, Fig. 9.17, faza de citire a  $\mu l$  următoare se desfășoară în același timp cu execuția  $\mu l$  curente.

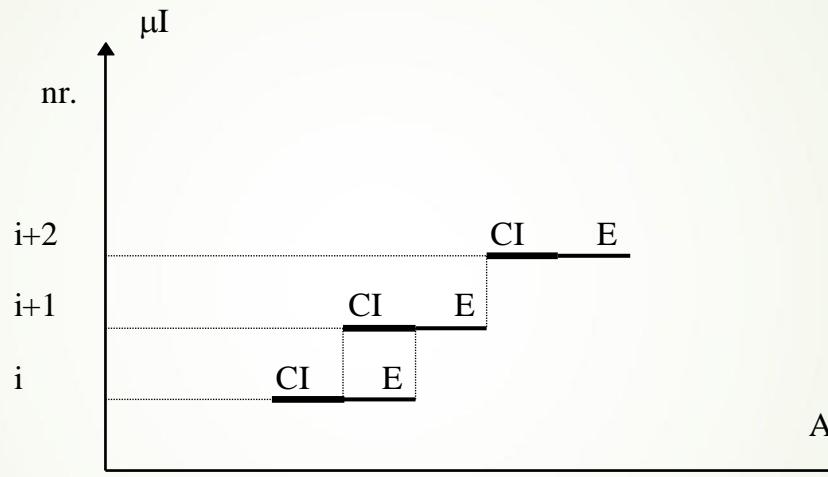


Implementare serie



Implementare parallel

O combinație a performanțelor implementării paralele și a costului redus al implementării serie este realizată de implementarea serie-paralelă a microinstructiunilor din figura urmatoare.



**Implementare serie-paralel**

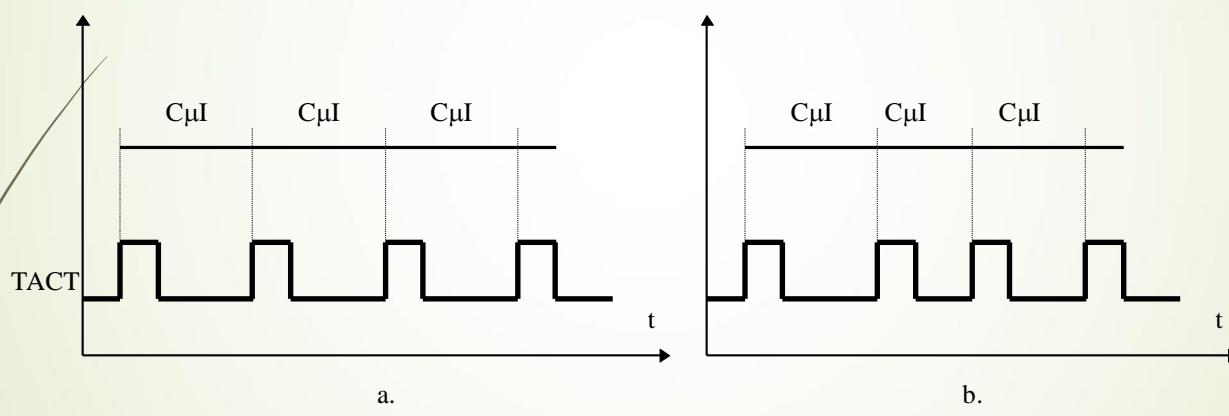
Faza de citire a µl următoare se desfășoară fie în timpul execuției µl curente, fie după terminarea ei, în funcție de tipul microinstrucțiunii curente.

Succesiunea de microinstrucțiuni operaționale se desfășoară prin suprapunerea fazelor de citire și execuție, iar cele imediat următoare unor ramificații se citesc după terminarea execuției µl curente (de ramificație).

Implementarea serie-paralelă este caracteristică arhitecturilor "pipe line", și este cea mai folosită.

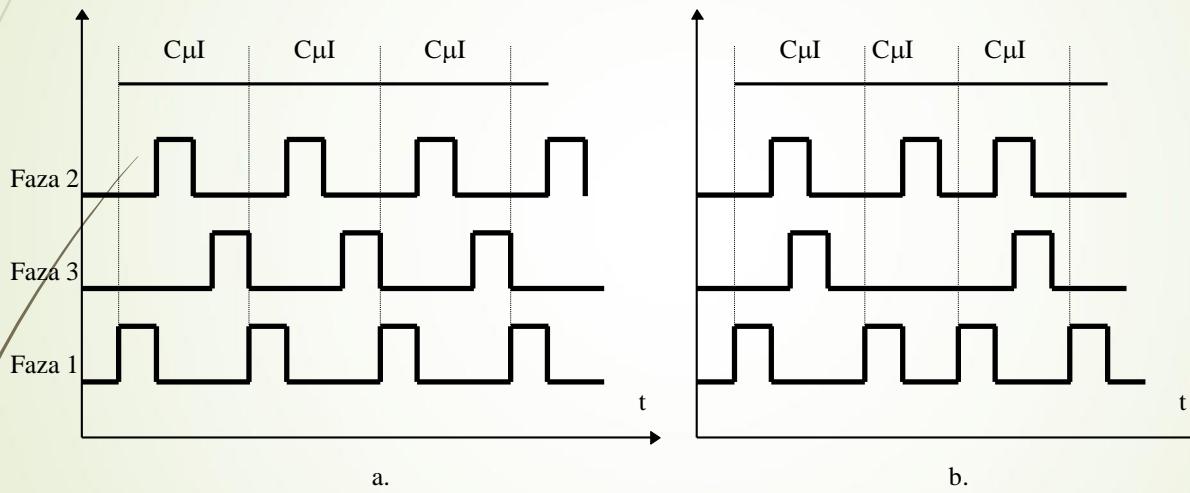
Referitor la implementarea microinstrucțiunilor se poate defini și o caracteristică monofază-polifază care se referă la numărul de faze utilizate într-un ciclu de microinstrucțiune.

Într-o implementare monofază, microoperațiile sunt generate simultan, toate semnalele de control specificate de microinstrucțiune fiind active în același timp (pentru cele de tip impuls se ține seama de front).



Implementare monofază

- a. ciclul macroinstructiunii  $C\mu I$  constant
- b. ciclul macroinstructiunii  $C\mu I$  variabil



## Implementare polifază

- a. ciclul microinstructiunii constant
- b. ciclul microinstructiunii variabil

## Determinarea paralelismului între operații elementare în cadrul structurilor numerice.

- ▶ La nivelul unității de comandă a structurii numerice, e necesar să se implementeze controlul concurrent al op. elementare.
- ▶ Obiectivul problemei de determinare a microoperațiilor paralele constă în a stabili, pe baza:
  - ▶ grafului de dependențe de date și
  - ▶ a cererilor de resurse,seturile de microoperații ce pot controla resurse diferite ale sistemului simultan.
- ▶ Obs: Conceptele si algoritmii prezentati pot fi aplicati și la execuția task-urilor.

# Definii

- ▶ **Def.1** Se numește microsubbloc **μSB** un subset maxim al unui set de microoperații ce constituie o secvență indivizibilă cu un singur punct de intrare și un singur punct de ieșire și conține maximum o microoperație de salt, ca ultimă operație din secvență.
- ▶  $\mu SB(\mu O) = (\mu B(\mu O), < )$  unde
  - ▶  $\mu B(\mu O) = \{\mu O_1, \mu O_2, \mu O_3, \dots, \mu O_{|\mu B|}\}$
  - ▶  $<$  relatie de ordine nereflexiva
- ▶ **Def.2 μOd** – microoperatie disponibila- acea microoperație pentru care toate microoperațiile de care ea era dependenta de date au fost asignate unor microinstrucțiuni.
$$\mu O_d = \{\mu O_j / \forall \mu O_i < \mu O_j \Rightarrow \mu O_i \in \mu I\}$$
- ▶ **Def. 3** Prin nivel într-un subbloc **μON<sub>k</sub>** înțelegem toate microoperațiile independente situate la aceeași distanță (adâncime) față de nodul inițial.

**Def.4** Setul de operatii disponibile la nivelul k,  $\mu OD_k$ , este construit din multimea de microoperatii disponibile din cadrul asociat microoperatiilor la nivelul k de generare de seturi paralele de microoperatii.

**PROP.1. Microoperatiile din seturile de microoperatii disponibile sunt independente.**

Presupunem ca nu ar fi independente. Daca avem

$$\mu O_i \in \mu OD_k \quad \mu O_j \in \mu OD_k$$

si nu sunt independente, atunci fie  $\mu O_i < \mu O_j$  sau  $\mu O_j < \mu O_i$ .

Conform Def 2  $\forall \mu O_l < \mu O_j \quad \mu O_j \in \mu OD_k$

rezulta ca  $\mu O_l$  a fost asignata (atribuita) unei microinstructiuni. Rezulta ca  $\mu O_l$  nu poate face parte din  $\mu OD_k$  si singura solutie este ca ele sa fie independente.

$|\mu ON|$  - numarul de niveluri dintr-un microsubloc egale cu timpul minim de executie pentru microsublocul respectiv

Fiecare nivel va specifica toate microoperatiile paralele care se desfasoara la un anumit moment de timp.

$n\mu l$ - numarul minim de microinstructiuni ce ar putea fi generate intr-un microsubloc

Evaluare estimativa:

$$nM\mu I = \frac{|\mu oB(\mu O)|}{nR} - \text{multimea de microoperatii din bloc}$$

► unde:

$nR$  – numarul de resurse ale sistemului

$\mu CR = \emptyset$  conflictul intre resurse

**nM<sub>μI</sub>** – numarul maxim de microinstructiuni

$$nM\mu I = |\mu B(\mu O)|$$

- toate microoperatiile se executa secvential

**nM<sub>μI</sub>, nM<sub>μO</sub>** sunt cazuri extreme.

$nO\mu I$  - numarul optim de microinstructiuni

$$\mu ON = |\mu ON| = \text{cardinalul nivelelor pe care le stabilim}$$

$$\forall \mu ON_j \in \{\mu ON\} \Rightarrow \mu CR(\mu ON_j) = \emptyset$$

## $\mu PT_I$ – partitia cea mai timpurie de executie a unei $\mu$ operatii.

Aceasta este constituita din multimea de niveluri care contin toate  $\mu$ operatiile independente situate la aceeasi distanta fata de un nod initial in graful de dependente de date.

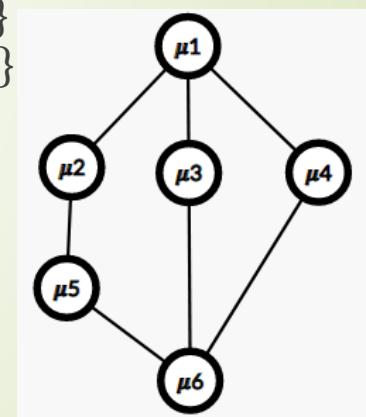
- ▶  $\mu PT_I = \{\mu ON | \forall \mu O_i, \mu O_j \in \mu ON \text{ avem } DI(\mu O_i) = DI(\mu O_j)\}$
- ▶ unde prin DI am notat adancimea fata de noul initial in graful de dependente de date.
- ▶  $\mu PT_L$  – partitia cea mai tarzie de executie a unei  $\mu$ operatii
- ▶  $\mu PT_L = \{\mu ON | \forall \mu O_i, \mu O_j \in \mu CM \text{ avem } DF(\mu O_i) = DF(\mu O_j)\}$
- ▶ unde prin DF am notat adancimea fata de nivelul final (nodul final)

$$\begin{aligned}\mu PT_I &= \{<\mu O_1>; <\mu O_2, \mu O_3, \mu O_4>; <\mu O_5>; <\mu O_6>\} \\ \mu PT_L &= \{<\mu O_1>; <\mu O_2>; <\mu O_3, \mu O_4, \mu O_5>; <\mu O_6>\}\end{aligned}$$

Plecand de la cele doua definitii , se numeste microoperatie critica o microoperatie care apartine aceluiasi subnivel in ambele partitii ( $\mu PT_I$ ,  $\mu PT_L$ ).

Ex: in cazul anterior avem 2 si 5

Orice modificare (mutare pe alt nivel) pentru o microoperatie critica conduce la un numar suplimentar de momente de timp pentru executie. Pentru celelalte (necritice), le putem pune oriunde intre nivelele date din partitia cea mai timpurie si cea mai tarzie, fara ca pentru executie sa avem necesar un timp suplimentar.



## PARTITIA OPERATIILOR ELEMENTARE PE NIVELURI DE EXECUTIE

Partitiile stabilite numai pe baza dependentei de date nu sunt optime, intrucat intervine si conflict de resurse. Pentru a pune in evidenta acest conflict, in cadrul partitiilor se face o grupare dupa tipul de resurse utilizate de fiecare microoperatie.

O micropartitie este egala cu multimea de niveluri.

$$\mu PT = \{\mu ON_1, \mu ON_2, \dots, \mu ON_{|\mu PT|}\}$$

*care reprezinta multimea de niveluri in care*

$$\mu ON_j = \{\mu ON_{j1}, \dots, \mu ON_{j|nR|}\}$$

$$N_{ji} = \{\mu O_h \mid \forall \mu O_h \in \mu ON_j \rightarrow \mu O_h \text{ utilizeaza aceeasi } RES_i\}; \quad 1 \leq i \leq NR$$

$RES_i$  – resursa i

$\mu ON_j$ - reprezinta nivelul j in cadrul partitiei, care este format din subniveluri care utilizeaza aceeasi resursa

Este bine ca un nivel sa-l impartim in subniveluri . Pentru cele care folosesc aceeasi resursa, intre ele avem conflicte.

Trebuie execute secential.

- DEF Cererea de resurse QRES este o aplicatie definita pe  $\mu B$  (multimea de resurse din bloc):  $\mu oB \times RES \rightarrow N$

unde  $RES = RESS \cup RESD \cup RESB \cup RESULC \cup RESSH$ .

RESS – resursa de tip sursa

RESD – resursa de tip destinati

RESB – resursa de tip bus

ULC – unitati logice combinationale

SH – shuffler

$DRES_i$  – resurse de tipul i disponibile

$QRES_i$  – resurse de tipul i cerute

$$DRES = \{DRES_1, DRES_2, \dots, DRES_{nR}\}$$

$$QRES_j = \{QRES_{j1}, QRES_{j2}, \dots, QRES_{j,nR}\}$$

– vectorul de cereri de resurse necesare microoperatiilor

care apartin nivelului j ( $\mu O N_j$ )

$$QRES_{ji} = \sum \mu O_h \text{ sau } QRES_{ji} = \sum QRES_{\mu O_h},$$

unde  $\mu O_h \in \mu O N_j$  care utilizeaza resursa i

- Daca cererea de resurse este mai mare decat disponibilul de resurse, atunci pe nivelul j care foloseste resursa i apare conflict de resurse.
- $QRES_{ji} > DRES_i \Rightarrow \mu CR(\mu O N_{ji}) = 1$



Se numeste microinstructiune completa  $\mu IC$  acea microinstructiune la care nu se poate adauga nici o operatie din setul de microoperatii disponibile, fara a se crea conflict de interese.

$$\mu IC: \forall \mu I (\forall \mu O_i \in \mu OD, \mu O_i \notin \mu I \Rightarrow \mu CR(\mu O_i, \mu I) = 1)$$

In cazul in care cererea de resurse:

$$QRES_{ji} > DRES_i \Rightarrow \mu IC \subset \mu ON_j; \mu ON_j \setminus \mu IC \neq \emptyset$$

Acest lucru inseamna ca exista microoperatii din nivelul j care vor fi trecute pe nivelele urmatoare datorita conflictului de resurse.

$$QRES_{ji} > DRES_i \Rightarrow \mu IC \subset \mu ON_j; \mu ON_j \setminus \mu IC \neq \emptyset$$

## DEFINITII

Intarziera in ceea ce priveste numarul de niveluri cu care se va muta o microoperatie din nodul curent j spre nodul final din graful de precedenta datorita conflictului de resurse i:

$$nN_{ji} = \begin{cases} 0 & , \text{daca } QRES_{ji} \leq DRES_i \\ \left[ \frac{QRES_{ji} - DRES_i}{DRES_i} \right] & , \text{daca } QRES_{ji} > DRES_i \end{cases}$$

Intarziera maxima pe nivelul j:

$$nN_j = \max(nN_{ji}) , \text{ unde } i = 1,2,\dots,nR$$

Stabilirea unei metodologii de determinare a partitionarilor unui microsubloc in microinstructiuni complete (care sunt actiunile pe care le fac la un moment dat).

Stabilirea efectului setului de microoperatii disponibile asupra generarii partitiilor.

Pentru a genera o singura microinstructiune completa din  $\mu$ OD,  
**ar trebui sa existe un criteriu care sa fie dependent numai de celelalte**  
microoperatii din setul disponibil.

Aceasta alegere ar trebui sa fie independenta de microoperatiile ce vor deveni disponibile dupa asignarea microoperatiei respective.

PROPOZITIE: Pentru a alege o microoperatie care apartine setului de microoperatii disponibile  $k$ , pentru a o introduce intr-o instructiune  $\mu I_C$  (completa, astfel incat numai o partitie, si anume  $\mu PI$  optima, sa fie generata), trebuie sa existe un criteriu de selectie dependent numai de microoperatiile care apartine setului disponibil dat, si independent de orice microoperatie din setul ce va deveni disponibil. Aceasta implica existenta unui criteriu de ordonare a microoperatiilor din setul disponibil.

JUSTIFICARE: Presupunem ca  $\mu CR(\mu OD_k) = 1$ , atunci avem conflict de resurse.  
Nu toate microoperatiile pot fi plasate in aceeasi microinstructiune.

Fie  $\mu IC$  generata de  $\mu OD_k$ .

Faptul ca generam aceasta microinstructiune inseamna ca toate operatiile dependente de aceasta devin disponibile. Vom genera un nou set de microoperatii disponibile:

- $\mu OD_{k+1} = \{\{\mu OD_k \setminus \mu IC\} \cup \{\mu O_d\}_k\}$ 
  - microoperatii care devin disponibile datorita asignarii acelei instructiuni complete
- $\{\mu O_d\}_k = \{\mu O | \forall \mu O_i \in \mu IC \Rightarrow \mu O_i < \mu O\}$

Fara examinarea efectului unor microinstructiuni suplimentare asupra dinamicii lui  $\mu OD_{k+1}$ , alegerea unei microoperatii trebuie sa fie independenta de setul  $\{\mu O_d\}_k$  rezultat. Trebuie sa existe acest criteriu de ordonare (care nici nu exista).

Orice metoda de ordonare a microoperatiilor din setul  $\mu OD_k$ , bazata pe resursele folosite de fiecare microoperatie si independenta de resursele folosite de celelalte microoperatiile produce o ordonare arbitrara.

Masurarea efectului se poate face pe baza urmatoarelor informatii:

- ▶ resursele solicitate:
  - de microoperatia respectiva
  - de microoperatiile dependente de microoperatia analizata
- ▶ gradul de dependenta a celoralte microoperatii fata de cea analizata

Microoperatiile din setul disponibil sunt independente, deci nu avem acces la cele 3 aspecte (influenta uneia asupra celeilalte).

Orice metoda de ordonare va conduce la o alegere arbitrara.

Fiecare microinstructiune generata produce un singur set de microoperatii disponibile. **Daca doua microinstructiuni complete genereaza acelasi set de microoperatii disponibile, atunci cele doua microinstructiuni complete sunt identice.**

- Fie microinstructiunile  $\mu IC_i$ ,  $\mu IC_j$  generate din setul disponibil la nivelul k ( $\mu OD_k$ ). Presupunem ca  $\mu IC_i \neq \mu IC_j$ . Fie  $\mu OD_{k+1}^i$ ,  $\mu OD_{k+1}^j$  seturile de microoperatii disponibile rezultate la momentul urmator.

$$\mu OD_{k+1}^i = (\mu OD_k \setminus \mu IC_i) \cup \{\mu O_d\}_{ki}$$

$$\mu OD_{k+1}^j = (\mu OD_k \setminus \mu IC_j) \cup \{\mu O_d\}_{kj}$$

$$\mu IC_i \setminus (\mu IC_i \cap \mu IC_j) \subset \mu OD_{k+1}^i$$

$$\mu IC_j \setminus (\mu IC_i \cap \mu IC_j) \subset \mu OD_{k+1}^j$$

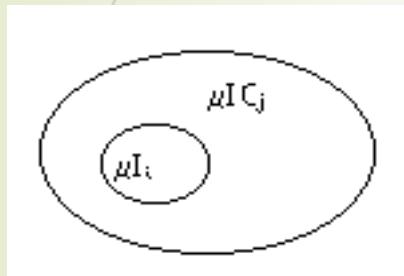
- Daca  $\mu OD_{k+1}^i \equiv \mu OD_{k+1}^j$  ar trebui ca:

$$\mu IC_j \setminus (\mu IC_i \cap \mu IC_j) \equiv \mu IC_i \setminus (\mu IC_i \cap \mu IC_j) \Rightarrow \mu IC_i \equiv \mu IC_j$$

**PROPOZITIE:** Partitia unui microsubloc se obtine numai prin generarea de  $\mu IC$  (nu pot lua combinatii mai reduse de microoperatii).

Presupunem ca din setul  $\mu OD_k$  putem genera  $\mu I_i$  si  $\mu IC_j$ .

Alegerea  $\mu I_i$  va produce un set de operatii disponibile  $\mu OD_{k+1}^i$



$$\mu IC_j \setminus \mu I_i \neq \emptyset$$

$$\mu OD_{k+1}^i = \mu OD_{k+1}^j \cup (\mu IC_j \setminus \mu I_i)$$

Deoarece  $\mu I_i \subset \mu IC_j$  atunci  $\mu IC_j \setminus \mu I_i \neq \emptyset$ .

$$\mu OD_{k+1}^i \subset \mu OD_{k+1}^j ,$$

avem eventual un numar mai mare de subniveluri (nu conduce la solutia optima).

Pe baza observatiei rezulta 2 categorii de algoritmi.



Partitie minima: - prin generarea din setul de microoperatii  $\mu OD_k$  a tuturor  $\mu IC$  posibile si pentru fiecare dintre ele sa se analizeze influenta ce o are asupra generarii setului de microoperatii disponibile urmator,  $\mu OD_{k+1}$ . Acest algoritm are complexitate MP completa.

Partitie euristică: - bazat pe ordonare a microoperatiilor din setul disponibil  $\mu OD_k$  in functie de succesorii in graful de dependente de date. Aceasta metoda nu va genera o partitie optima, insa va fi mult mai rapida in situatii acceptabile.

# Calculul limitei inferioare pentru nivelurile de alocare

Presupunem că avem  $\mu SB = (\mu B(\mu o), \angle)$  care are graful asociat  $G$

$\mu B(\mu o) = \{ \mu o_1, \mu o_2, \dots, \mu o_{|\mu B|} \}$  microoperatiile din subblock

$P = \{ P_1, P_2, \dots, P_{|P|} \}$  resursa, elemente de executie

În procesul de calcul al limitei inferioare apar 4 situații care trebuie analizate.

**CAZUL 1**. Prima valoare a limitei inferioare a înălțimea  $h$  a grafului  $G$ , unde  $h =$  calea cea mai lungă de la un nod rădăcină la un nod terminal.

Valoarea  $h =$  valoarea minimă a limitei inferioare (nu s-a ținut seama de numarul de resurse și de conflictul de resurse).

**CAZUL al 2-lea – estimare mai corectă –** se analizează și distribuția microoperatiilor pe setul de resurse pe care le controleaza.

Fie  $\rho$  – această valoare limită.

Pentru calculul lui  $\rho$ , graful  $G$  va fi împărțit într-un subsistem de grafuri  $G_i, 1 \leq i \leq |P|$

Resursele pot fi omogene sau nu.

Împărțirea se face în stabilirea – unor subseturi de sarcini alocate pe resurse, definite astfel:

$$G_i : \mu B(\mu o)_{|P_i|} = \{ \mu o \mid \mu o \text{ controleaza resursa } i \}$$

# Subgraf maximal

## Microoperatie terminală

**Def.** O componentă a unui graf orientat aciclic  $G_i$  este un subgraf maximal  $SGM_{ij} \subset G_i$  aî pentru orice microoperatie  $\mu_{o_k}$  care aparține lui  $SGM_{ij}$

$(\mu_{o_k} \in SGM_{ij})$  există  $\mu_{o_m} \in SGM_{ij}$  aî

$\mu_{o_k} < \mu_{o_m}$  sau  $\mu_{o_m} < \mu_{o_k}$

(deci, trebuie să existe o legătură între cele două noduri)

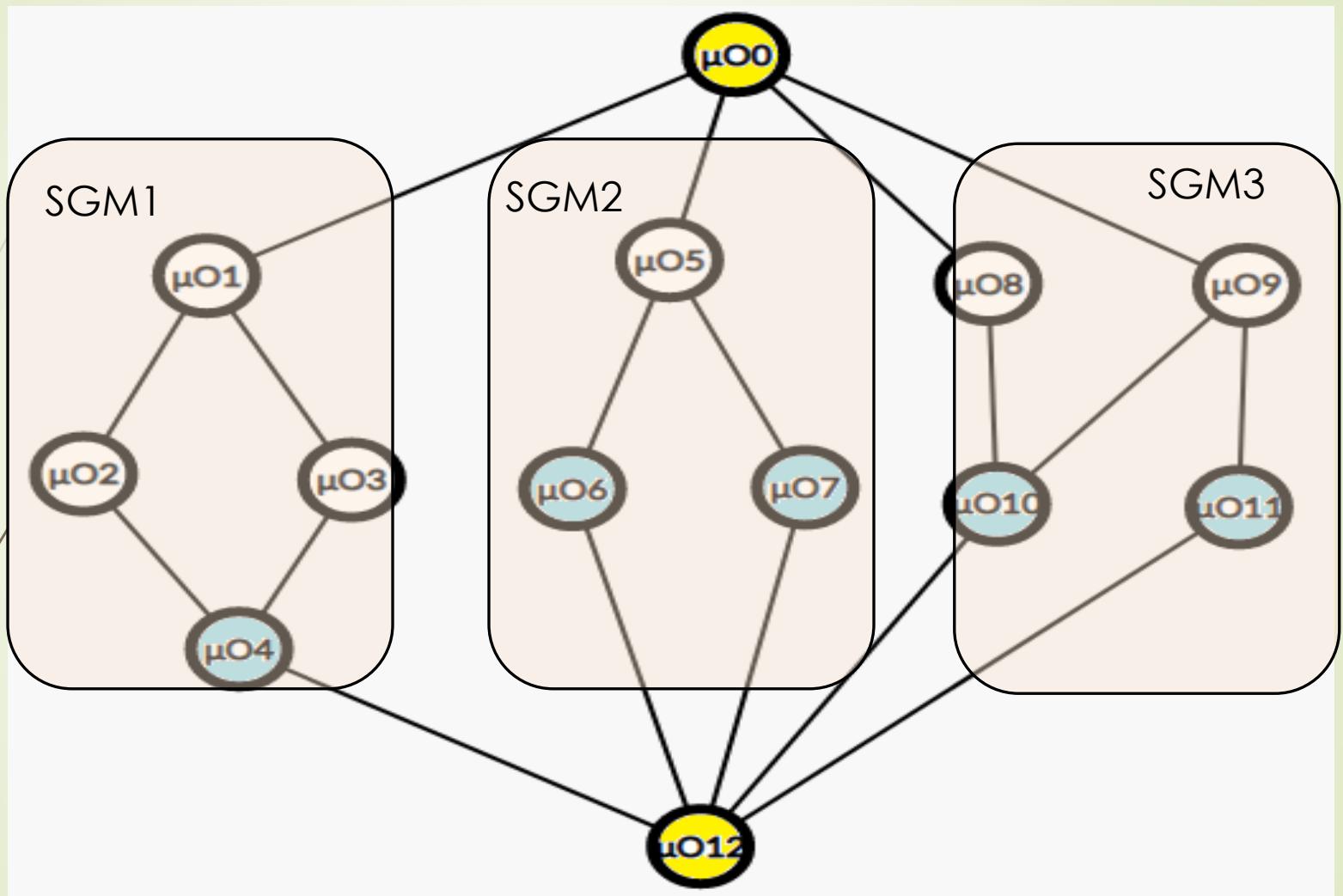
**Obs.**  $G_i$  e considerat ca o combinare paralelă a subgrafurilor maximale.,

$$G_i = || SGM_{ij}$$

**Def.** Microoperatie terminală. O microoperatie  $\mu_o$  a unui graf orientat aciclic e o microoperatie terminală  $\mu_{o_T}$  dacă nu mai există nici o sarcină  $\mu_{o_j}$  aî  $\mu_{o_T} < \mu_{o_j}$

Nivelul unei microoperatii  $\mu_{o_i}$  într-un graf  $G_i$  orientat e dat de nr. de arce obținute într-o parcurgere pe drumul maxim între  $\mu_{o_i}$  și  $\mu_{o_T}$

# Exemplu



Considerand ca avem trei resurse  $P = \{P1, P2, P3\}$   
Sarcini terminale:  $\mu O4, \mu O6, \mu O7, \mu O10, \mu O11$   
Nivelul  $\mu O1= 2$  (sarcinile terminale au nivelul 0)

# Calcul $\rho$

Pentru fiecare microoperatie terminală din  $SGM_{ij}$  obținută prin împărțirea pe resurse a microoperatiilor, vom calcula nivelul din cadrul lui G și alegem microoperatia terminală cu nivel maxim.

Fie  $\mu o_{Tj}$  aceste microoperatii, iar cu  $\| \mu o_{Tj} \|$  notăm nivelul unei astfel de sarcini în , graful G.

Pentru fiecare subgraf  $G_i$ ,  $i=1, ..., |P|$  calculăm limita inferioară.

$$\rho_i = \max \{ \rho_{ij} \} \text{ unde } 1 \leq i \leq |P|, \quad 1 \leq j \leq |G_i|$$

$$\rho_{ij} = |SGM_{ij}| + \| \mu o_{Tj} \|$$

$$\| \mu o_{Tj} \| = \min_k \{ \| \mu o_{Tjk} \| \} \text{ cu } 1 \leq k \leq N_{\mu o_T}$$

$|SMG_{ij}|$ = nr. de microoperatii din cadrul componentei j a subgrafului  $G_i$

$\|S_{Tj}\|$ - nivelul minim al sarcinilor terminale din cadrul componentei j a subgrafului  $G_i$  calculat față de poziția în  $G$  asociat sistemului de microoperatii asociat  $\mu SB$ .

$|G_i|$ =nr. de componente ale subgrafului  $G_i$

$$\rho = \max_i \left\{ \max_j \left\{ |SGM_{ij}| + \min_k \{ \| \mu o_{Tjk} \| \} \right\} \right\} \text{ unde, } \begin{array}{l} 1 \leq k \leq N_{\mu o_T} \\ 1 \leq i \leq |P| \\ 1 \leq j \leq |G_i| \end{array}$$

$\mu B(\mu_0) = \{\mu_0_1, \mu_0_2, \mu_0_3, \mu_0_4, \mu_0_5, \mu_0_6, \mu_0_7, \mu_0_8, \mu_0_9, \mu_0_{10}, \mu_0_{11}, \mu_0_{12}\}$   
 $P = \{P_1, P_2, P_3\}$

Presupunem că avem următoarea repartiție pe resurse.

$$\mu_0_{P1} = \{\mu_0_1, \mu_0_4, \mu_0_5, \mu_0_7, \mu_0_9\} \quad G1$$

$$\mu_0_{P2} = \{\mu_0_2, \mu_0_3, \mu_0_6, \mu_0_8, \mu_0_{10}\} \quad G2$$

$$\mu_0_{P3} = \{\mu_0_{11}, \mu_0_{12}\} \quad G3$$

G1:

$$SGM_{11} = \{\mu_01, \mu_04, \mu_05, \mu_07, \mu_09\}$$

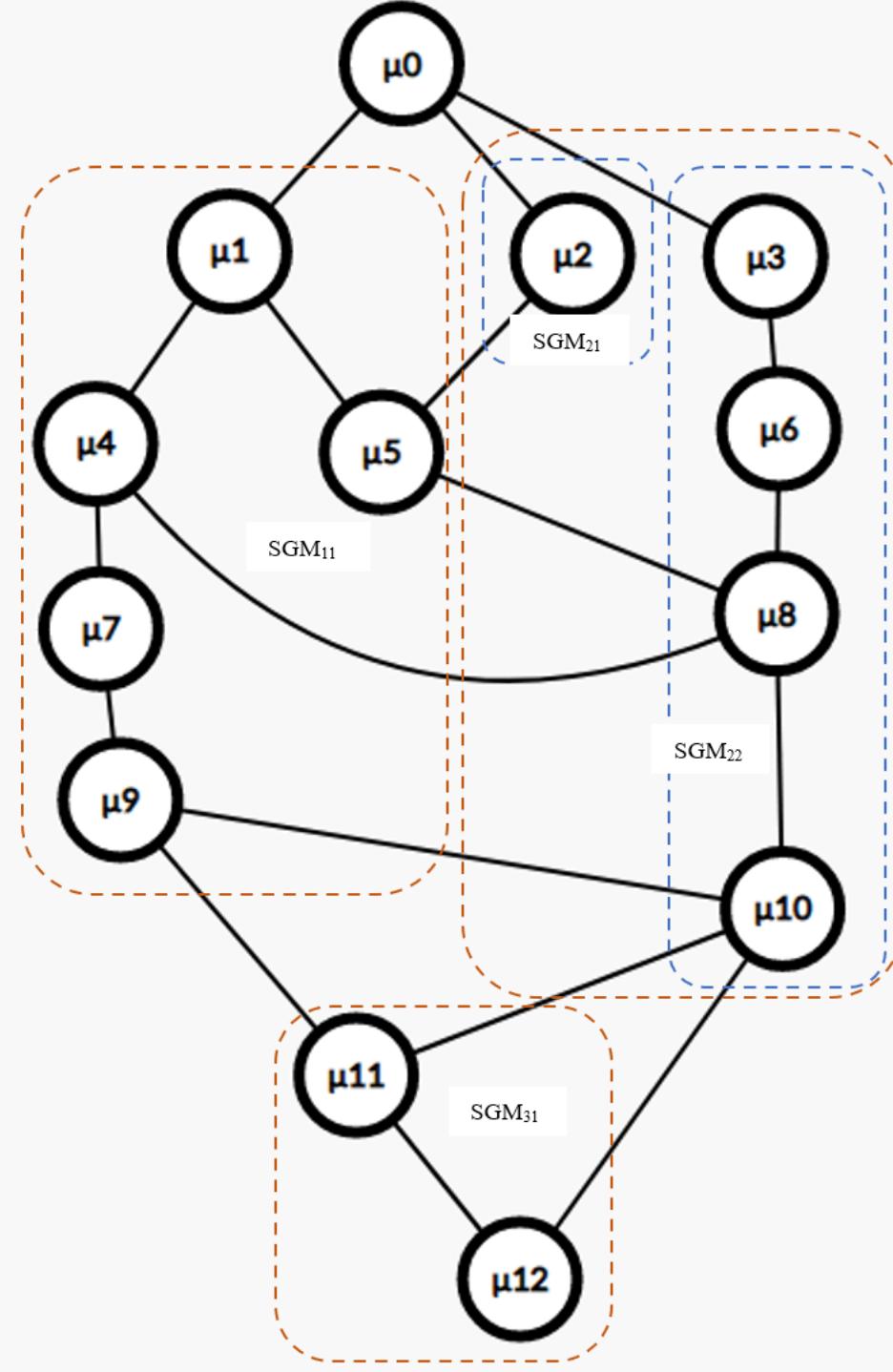
G2:

$$SGM_{21} = \{\mu_02\};$$

$$SGM_{22} = \{\mu_03, \mu_06, \mu_08, \mu_0{10}\}$$

G3:

$$SGM_{31} = \{\mu_0{11}, \mu_0{12}\}$$



**G<sub>1</sub>:** 2 sarcini terminale μ<sub>05</sub>, μ<sub>09</sub>, nivelul lui G<sub>5</sub>, calculat în G, pe calea cea mai lungă  
 $\|\mu_{05}\|=4$   
 $\|\mu_{09}\|=3$   
 $|SGM_{11}|=5$

$$\rho_1 = |SGM_{11}| + \min \{\|\mu_{05}\|, \|\mu_{09}\|\} = 5 + 3 = 8$$

**G<sub>2</sub>:** pentru SGM<sub>21</sub>, o singură sarcină care este și terminală

$$\|\mu_{02}\| = 5$$

$$\rho_{21} = \left\{ \underbrace{|SGM_{21}|}_1 + \min \|\mu_{02}\| = 1 + 5 = 6 \right\}$$

pentru SGM<sub>22</sub>,

$$\begin{aligned} \rho_{22} &= \left\{ \underbrace{|SGM_{22}|}_4 + \min \|\mu_{010}\| = 4 + 2 = 6 \right\} \\ \Rightarrow \rho_2 &= \max(\rho_{21}, \rho_{22}) = \max(6, 6) = 6 \end{aligned}$$

**G<sub>3</sub>**  
SGM<sub>31</sub> și μ<sub>0T</sub>=12  
 $|SGM_{31}|=2$   
 $|\mu_{012}|=0$   
ρ<sub>3</sub>= 2+0=2

$$\rho = \max_i \left\{ \max_j \left\{ |SGM_{ij}| + \min_k \{\|\mu_{0Tjk}\|\} \right\} \right\} \text{ unde, } \begin{array}{l} 1 \leq k \leq N_{\mu_{0T}} \\ 1 \leq i \leq |P| \\ 1 \leq j \leq |G_i| \end{array}$$

Am obținut 2, 6, 8 ⇒ ρ = 8

Pe același exemplu, h=6 (înălțimea maximă a grafului).  
Deci, le luăm pe cele cu 8 niveluri sau mai mult.

Justificarea alegerii, în cadrul componentei SGM<sub>ij</sub>, a sarcinii terminale cu nivel minim în G e aceea că orice parcurgere a unei componente trebuie să se termine într-un astfel de nod.

Nu s-a ținut cont de conflictul de resurse.

În general, resursele nu sunt distincte,  $\rho$  va fi mai mare decât în realitate.

Vom defini pentru fiecare resursă un  $\rho_i$

$$\rho_i = \max_j \left\{ \max \left[ \left( \lceil |SGM_{ij}| / |P_i| \rceil \right), \left( h(SGM_{ij}) + \|\mu o_{Tj}\| \right) \right] \right\}$$

unde  $|P_i|$  = număr de resurse de tip i

$|SGM_{ij}|$  = nr. de microoperări din cadrul componentei j,  
nr. minim de momente de timp.

Trebuie respectată dependența de date.

$$h(SGM_{ij}) + \|\mu o_{Tj}\|$$
 dat de dependența de date

$$\left[ \frac{|SGM_{ij}|}{|P_i|} \right]$$
 - e dat de nr. de resurse pe care le avem

$h(SGM_{ij})$  - nr. de momente de timp necesare pentru execuția sarcinilor, având  
în vedere dependența de date

## CAZUL 3

Limita inferioară =  $\mu$ , limita inferioară pentru situația în care resursele sunt distințe.

►  $\mu = \max_i \{|\mu oP_i|\} \quad 1 \leq i \leq |P|$

Fiind o singur resursă, se va fi controlată secvențial, deci avem  $|\mu oP_i|$ , fiind  $|\mu oP_i|$  microoperării.

Când resursele nu ar fi fost diferite, am putea considera că

►  $\mu = \max_i \{[|\mu oP_i| / |P_i|]\}$

În calculul acestei limite, s-a presupus că nu există dependență de date între microoperările care controlează diferite resurse.

## CAZUL 4

Se consideră că o microoperatie nu se desfășoară numai într-o perioadă de timp ca în cazurile anterioare, ci pe durata a mai mulți cicli.

Notăm cu  $\mu M_c$  setul de microoperatii care se desfășoară pe  $n_c$  cicli.

$\Rightarrow$  limita inferioară e influențată de nr. de cicli și o posibilitate de calcul ar fi

$z = |\mu M_c| \cdot n_c$  - e o variantă foarte simplistă, numai pentru punerea problemei.

În general, ajungem la  $LI = \max\{h, \rho, \mu, z\}$

$C_{|AD|}^K \Rightarrow Q$ ; în funcție de LI, eliminăm f. mult.

Algoritmul de partitie va merge până la limita inferioară.

Nu se vor gasi soluții cu valori mai mici decat LI.

Dacă am găsit o soluție egală cu LI, rezultă că am găsit o soluție.

## ALGORITM CARE CONDUCE LA PARTITIE MINIMA:

- Algoritm de partitionare pe niveluri a setului de micro-operații dintr-un micro-subbloc

$\mu SB = (\mu B(\mu O), <)$  - sistemul de microoperatii pe care trebuie sa-l partitionam

$\mu PT$  – partitia ce se va genera

$\mu PTA$  – partitia de microinstructiunea anterioara, considerata cea mai buna pana in momentul respectiv

$\mu OD$  – setul de microoperatii disponibile

$\mu IC_i$  – microinstructiunea completa generata din setul de microoperatii disponibil curent

$n_{\mu l}$  – numarul minim de microinstructiuni ce se poate obtine prin codificarea microoperatiilor din subloc

$\{\mu O_d\}_i$  – multimea de microoperatii ce devin disponibile prin generarea microinstructiunii complete curente

Algoritmul genereaza o partitie optima de  $\mu IC$ , tinand cont de conflictul de resurse si dependenta de date impusa de relatia impusa de ordine partiala.

# Algoritm

## P1. Initializare

- Se initializeaza partitia  $\mu PT = \emptyset$ ,  $\mu PTA = \mu B(\mu O)$  - tot microsublocul (deci, daca le luam si le facem sevential)
- $\mu OD = \{\mu O_j | \forall \mu O_j \in \mu B(\mu O) \text{ nu exista } \mu O_i < \mu O_j\}$  - deci pentru care nu exista predecesori
- $\mu OND = \{\mu O_i | \mu O_i \in \mu B(\mu O) \setminus \mu OD\}$  - operatii nedisponibile

## P2. Daca $\mu OND = \emptyset$ si $|\mu OD| \leq 3$ atunci salt la pas 5

## P3. Se genereaza $\{\mu IC\}$ – multimea de microinstructiuni complete

- $\{\mu IC\} = \{\mu IC_j, \dots, \mu IC_k\}$  - din  $\mu OD$  curent
  - daca  $j \neq k$  (s-au generat mai multe  $\mu IC$ )  
atunci salvare  $\mu PT_j = \mu PT$   
 $\mu OD_j = \text{setul curent } \mu OD$   
 $\{\mu IC\}_j = \{\mu IC\} \setminus \mu IC_j$

## P4. $\mu PT = \mu PT \cup \mu IC_j$

$$\mu OD = \mu OD \setminus \mu IC_j \cup \{\mu O_d\}_j$$

$$\mu OND = \mu OND \setminus \{\mu O_d\}_j$$

- daca  $|\mu OD| \neq 0$  si  $|\mu PT| \leq |\mu PT_j| - 1$  atunci salt pas 2

P5. daca  $\mu OD = \Phi$  atunci salt pas 7

P6. Se genereaza  $\mu IC$  din  $\mu OD$  curent.

$$\mu PT = \mu PT \cup \mu IC;$$

$$\mu OD = \mu OD \setminus \mu IC;$$

daca  $\mu OD \neq \Phi$  si  $|\mu PT| < |\mu PTA| - 1$  atunci pas 4

altfel pas 2

P7. daca  $|\mu PT| < |\mu PTA|$  atunci  $\mu PTA = \mu PT$

altfel daca  $|\mu PTA| \leq nm\mu I$  atunci  $\mu PTA$  este optima  
STOP.

P8. daca  $\{\mu IC\}_j \neq 0$  (cel care a fost salvat la pasul 3)

atunci reface  $\mu PT = \mu PT_j$ ;

$$\mu OD = \mu OD_j;$$

$j \leftarrow j + 1$ ; (trec la urmatorul, selecteaza urmatoarea  
 $\mu IC$  din setul generat)

$$\mu IC = \mu IC_j$$

$$\mu OND = \{\mu O_j | \mu O_j \in \mu B(\mu O) \setminus (\mu PT \cup \mu OD)\}$$

salt pas 4

altfel  $\mu PTA$  este optima.

GATA.

► Obs  $nm\mu I$  – Limita inferioara

# Algoritm heuristic pentru partitionarea unui microsubloc

Pentru a evita generarea tuturor partitiilor de microinstructiuni complete, folosim un criteriu de ordonare al operatiilor din setul de microoperatii disponibile pa baza numarului de succesi.

Nu duce la solutia optima, dar duce la o solutie acceptabila.

$$psucc(\mu O_i) = \{\mu O_j \mid \forall \mu O_i, \mu O_j \in \mu B(\mu O), \text{ cu } i \neq j, \text{ avem } \mu O_i < \mu O_j\}$$

$$psucc(\mu I_i) = \sum_{j=1}^n psucc(\mu O_{ij})$$

n = numarul de microoperatii din  $\mu I_i$

# Etapele algoritmului

## Pas 1

Se initializeaza partitia  $\mu PT = \Phi$  ,  $\mu PTA = \mu B(\mu O)$

$\mu OD = \{\mu O_j | \forall \mu O_j \in \mu B(\mu O) \text{ nu exista } \mu O_i < \mu O_j\}$  – deci pentru care nu exista predecesori

$\mu OND = \{\mu O_i | \mu O_i \in \mu B(\mu O) \setminus \mu OD\}$  – operatii nedisponibile

## pas 2

facem o noua partitie, numai in anumite conditii

daca  $\mu OND = \Phi$  (am avansat cu acest test pana la sfarsit) atunci pas 4

## pas 3.

genereaza  $\mu IC = \{\mu IC_j, \dots, \mu IC_k\}$  microinstructiuni complete, ordonam acest set in functie de succesori; ordonam astfel incat

$psucc(\mu IC_j) \geq psucc(\mu IC_{j+1}) \geq \dots \geq psucc(\mu IC_k)$

Dintre doua microinstructiuni , pastram pe nivel pe cea cu cei mai multi succesori, intrucat mutarea ei pe nivelul urmator ar implica mutarea pe nivelul urmator a tuturor succesorilor ei. Deci, ea ar apartine micropartitiei curente.

$\mu PT = \mu PT \cup \mu IC_j$

Reactualizam operatiile disponibile, nedisponibile la momentul respectiv.

$\mu OD = \mu OD \setminus \{\mu O | \mu O \in \mu IC_j\} \cup \{\mu O_d\}_j$

$\mu OND = \mu OND \setminus \{\mu O_d\}$

transfer **pas 2**

#### Pas 4

spatiul disponibil nu exista; gasesc ultima structura de microinstructiuni complete.

Genereaza  $\mu\text{IC} = \{\mu\text{IC}_h, \dots, \mu\text{IC}_l\}$ .

Ultima data: iau toate  $\mu\text{IC}$  care se genereaza aici (intrucat aici nu mai pot merge pe o cale sau alta).

$$\mu\text{PT} = \mu\text{PT} \cup \mu\text{IC}_h$$

$\mu\text{OD} = \mu\text{OD} \setminus \{\mu\text{O} \mid \mu\text{O} \in \mu\text{IC}_h\}$  - nu mai adaug nimic, nemaiavand succesorii (e ultima data).

$h = h+1$  – le iau pe toate, dar din cele generate din spatiul disponibil. Continuam acest pas pana ce nu mai am nici o  $\mu\text{IC}$ .

daca  $\mu\text{OD} \neq \Phi$  atunci pas 4

altfel  $\mu\text{PT}$  – partitia generata STOP





# **Specificarea paralelismului operatiilor elementare**

**Codificarea minimă a unui set de microinstrucțiuni**

# Punerea problemei

- ▶ În proiectarea unităților de comandă microprogramate, după ce s-a determinat setul de microinstructiuni complete ce realizează execuția unui microsubbloc în minimum de pași prin controlul tuturor operațiilor paralele ce se pot executa în sistem, se pune **problema codificării minime a acestui set de microinstructiuni**.
- ▶ Problema optimizării numărului de biți **constă în a stabili lungimea cuvântului memoriei de control ce specifică microinstructiunile, astfel încât aceasta să fie minimă**, având în vedere asigurarea controlului tuturor microoperațiilor paralele specificate de secvența de microinstructiuni ce descrie unitatea de comandă.
- ▶ Problema generală a **optimizării numărului de biți este o problemă din clasa "NP complete"**. Apartenență la clasa "NP complete" a fost demonstrată de Robertson

# µo compatibile

# µo incompatibile

- Fie
  - $\mu B(\mu o) = \{\mu o_1, \mu o_2, \dots, \mu o_{[\mu B]}\}$  setul de microoperații distincte din cadrul microsubblocului.
  - $\mu PT = \{\mu IC_1, \mu IC_2, \dots, \mu IC_{[\mu PT]}\}$  partiția unui microsubbloc în microinstructiuni complete și
- **Def. 1**
  - Două microoperații  $\mu o_i$  și  $\mu o_j$  sunt **compatibile** dacă pentru orice  $k$ ,  $1 \leq k \leq [\mu PT]$ , dacă  $\mu o_i \in \mu IC_k$  atunci  $\mu o_j \notin \mu IC_k$ .
- **Def. 2**
  - Două microoperații  $\mu o_i \in \mu B(\mu o)$ ,  $\mu o_j \in \mu B(\mu o)$  sunt **incompatibile** dacă există cel puțin o microinstructiune completă  $\mu IC$  astfel încât  $\mu o_i \in \mu IC_k$  și  $\mu o_j \in \mu IC_k$ .

# clasă de compatibilitate

- ▶ **Def. 3**
  - ▶ O **clasă de compatibilitate  $CC(\mu_o)$**  este un set (subset) al multimii  $MB(\mu_o)$  în care oricare două microoperații sunt compatibile între ele.
  - ▶  $CC(\mu_o) = \{\mu_o \mid \text{pt orice } \mu_o_i, \mu_o_j \in CC(\mu_o) \text{ avem } \mu_o_i \text{ compatibilă cu } \mu_o_j\}$
- ▶ **Def. 4**
  - ▶ O **clasă de compatibilitate maximă  $MCC(\mu_o)$**  este acea clasa de compatibilitate la care nu mai poate fi adăugată nici o microinstructiune fără a se pierde compatibilitatea.
  - ▶  $MCC(\mu_o) = \{\mu_o_j \mid \text{pt orice } \mu_o_i \notin MCC(\mu_o), \text{ există } \mu_o_j \in MCC(\mu_o) \text{ astfel încât } \mu_o_i \in \mu_o C_k \text{ și } \mu_o_j \in \mu_o C_k\}.$

În mod analog se definește clasa de incompatibilitate maximă  $MIC(\mu_o)$ .

# Costul de implementare a unei clase de compatibilitate

- **Def. 5**
- **Costul de implementare a unei clase de compatibilitate** (măsurat în numărul de biți necesari pentru codificare) este dat de implementarea codificării verticale a microoperațiilor ce compun clasa.

- $\text{Cost } CC_i = \lceil \log_2(|CC_i| + 1) \rceil$

iar costul total de implementare al cuvântului de control

- $\text{Cost } CC = \sum_{i=1}^k \lceil \log_2(|CC_i| + 1) \rceil$

unde k este numărul de clase de compatibilitate.

- **Obs**
- **O clasă de compatibilitate corespunde unui câmp din cadrul microinstrucțiunii.**

# Estimarea costului minim

- ▶ Problema codificării minime presupune două faze distincte:
  - ▶ enumerarea tuturor claselor de compatibilitate maximă MCC;
  - ▶ determinarea unui subset de MCC care să implementeze costul minim pentru codificare.

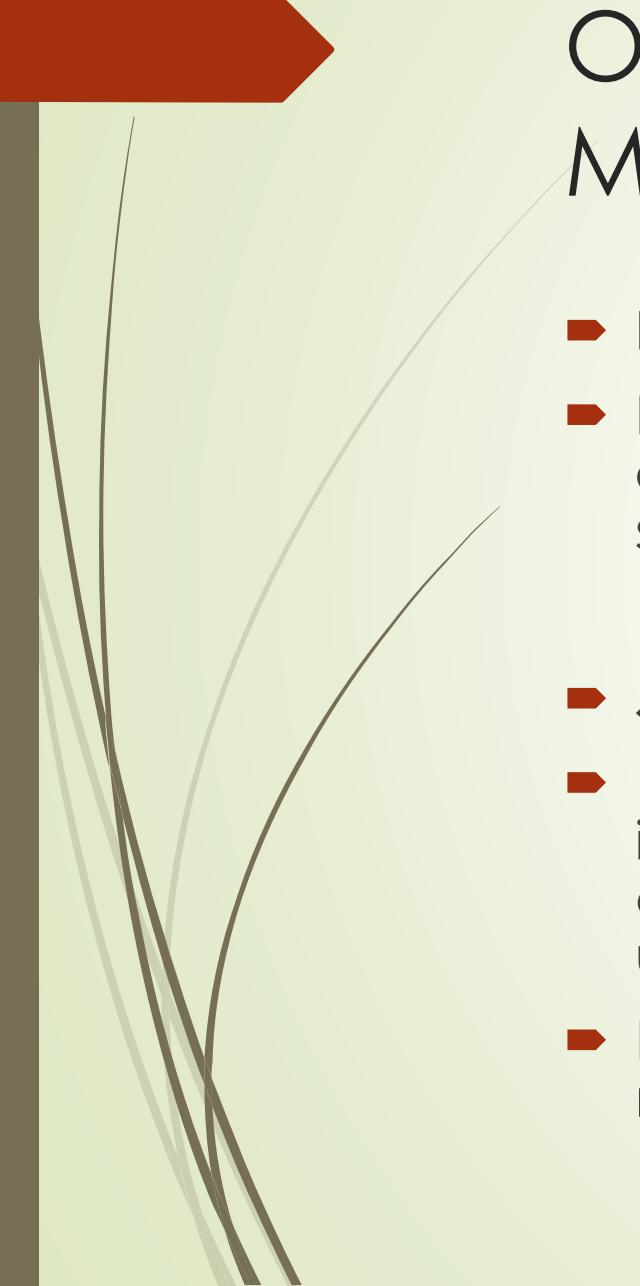
Costul implementării depinde de

- ▶ numărul de MCC necesar pentru acoperirea întregului set de microoperații și
- ▶ numărul de microoperații din fiecare clasă de compatibilitate maximă.
- ▶ De notat faptul că un cuvânt din memoria de control, adică o microinstructiune completă, este un set de clase incompatibile între microoperațiile componente.
- ▶ O clasă de compatibilitate specifică cel mult o microoperație din cadrul unui cuvânt al memoriei de control.

# Clase de compatibilitate maximă asociate (AMCC)

- ▶ **Def. 6**
- ▶ Pentru orice clasă de incompatibilitate IC, clasele de compatibilitate maximă ce conțin un element din IC se numesc **clase de compatibilitate maximă asociate (AMCC) asociate** clasei de incompatibilitate.
- ▶ **Prop. 1**
- ▶ **Pentru orice clasă de incompatibilitate maximă MIC reuniunea claselor de compatibilitate maximă asociate acoperă întreg setul de microoperații.**
- ▶ **Justificare**
- ▶ Fie  $\text{MIC}(\mu_o) = \{\mu_o_1, \dots, \mu_o_{|\text{MIC}|}\}$  formată din  $|\text{MIC}|$  microoperații, unde  $|\text{MIC}| \leq |\mu_B|$ . Presupunem contrariul și anume că reuniunea claselor de compatibilitate maximă asociate nu acoperă întreg setul de microoperații. Fie  $\mu_o_j$  una din aceste microoperații presupuse neacoperite cu proprietatea că:

$\mu_o_j \in \mu_B(\mu_o)$  dar  $\mu_o_j \notin \text{MIC}(\mu_o)$  și  $\mu_o_j \notin \bigcup_{\text{MIC}}$



# O clasa de incompatibilitate maximă MIC nu poate acoperi $\mu B(\mu o)$ .

- ▶ **Prop. 2**
- ▶ Pentru o clasă de incompatibilitate maximă MIC, **reuniunea oricăror k clase de compatibilitate**,  $k < |MIC|$ , nu poate acoperi setul de microoperații  $\mu B(\mu o)$ .
  
- ▶ **Justificare**
- ▶ Considerăm  $MIC(\mu o) = \{\mu o_1, \mu o_2, \dots, \mu o_{|MIC|}\}$ .  $|MIC|$  este limita inferioară pentru numărul de clase de compatibilitate ce satisfac acoperirea, deoarece o clasă de compatibilitate poate acoperi un singur element din MIC.
- ▶ Rezultă că orice reuniune de k clase de compatibilitate  $k < |MIC|$  nu poate acoperi setul de microoperații.

# Clasele MCI si MCC

- ▶ **Clasele de incompatibilitate maximale MCI** sunt determinate pe baza grafului de dependență și a conflictului de resurse și reprezintă microoperațiile specificate de microinstructiunile complete ce descriu microblocul.
- ▶ **Clasele MCC sunt determinate din matricea de microoperații** ce indică incompatibilitatea. Aplicarea operatorului SAU între liniile matricii va specifica microoperațiile ce fac parte din clasa de compatibilitate maximă.
- ▶ **Calcularea MCC este o binecunoscută problemă în teoria automatelor finite**, existând numeroase metode pentru aceasta soluție. AHO arată că problema calculării MCC este o problema "NP complete".

## Prop. 3 partităionare în q câmpuri

- Dacă un set de microoperații  $\mu B(\mu o)$  este partitionat în q câmpuri ale unei microinstrucțiuni, costul minim se va realiza atunci când  $(q-1)$  câmpuri specifică căte o microoperație (au un singur bit), iar cel de al q-lea câmp codifică restul de  $| \mu B(\mu o) | -q+1$  microoperații.

### Justificare

- Considerăm o partitie arbitrară a LMI biți în q câmpuri. Fie câmpul cu  $b_{max}$  biți, câmpul cu lungimea cea mai mare și fie un oricare alt câmp care are  $b_i$  biți.
- Numărul de microoperații ce poate fi codificat de cele două câmpuri este:
  - **$NMO = (2^{b_{max}} - 1) + (2^{b_i} - 1)$**
  - Facem o modificare în organizarea logică a microinstrucțiunii și mutăm un bit din câmpul cu  $b_i$  biți în câmpul cu  $b_{max}$  biți. În acest caz numărul de microoperații ce se pot codifica este:
    - **$NMO' = (2^{(b_{max}+1)} - 1) + (2^{(b_i-1)} - 1)$**
    - **$NMO' - NMO = 2^{b_{max}} - 2^{b_i} >= 0$  deoarece  $b_{max} > b_i$ .**
  - Deci se pot codifica mai multe operații (microoperații) după modificarea structurii microinstrucțiunii.
  - Repetând procesul de mutare a unui bit dintr-un câmp oarecare în câmpul de lungime maximă, numărul de microoperații ce poate fi codificat crește. Numărul maxim de microoperații ce poate fi codificat rezultă a fi egal cu  $(q+2^{(LMI-q+1)}-2)$ . Transformând totul în cost se obține costul minim:
    - **$Cost = q-1 + \log_2(| \mu B(\mu o) | -q+2) / \text{când } (q-1) \text{ câmpuri specifică fiecare căte o microoperație, iar ultimul câmp } (| \mu B(\mu o) | -q+1) \text{ microoperații.}$**

# Partitionare a setului $\mu_B(\mu_O)$ microoperații în $(q+h)$ câmpuri

## Prop. 4

- Nu este posibilă o soluție de partitionare a setului  $\mu_B(\mu_O)$  microoperații în  $(q+h)$  câmpuri astfel încât costul să fie mai mic decât partitia în  $q$  câmpuri.

## Justificare

Fie

- $C = q-1 + [\log_2(|\mu_B(\mu_O)| - q+2)]$  costul minim obținut prin partitia în  $q$  câmpuri și
- $C = q+h-1 + [\log_2(|\mu_B(\mu_O)| - q-h+2)]$  costul minim obținut prin partitia în  $q+h$  câmpuri.
- Deosebim 2 cazuri și anume:
  - 1)  $h=1$  și  $|\mu_B(\mu_O)| - q-1 = 2^k$
  - 2)  $h \neq 1$  sau  $|\mu_B(\mu_O)| - q+1 \neq 2^k$
- în primul caz:
  - $C_q = q-1+k+1 = q+k$
  - $C_{(q+h)} = q+k$
  - deci  $C_q = C_{(q+h)}$
- în al doilea caz, deoarece :
  - $[\log_2(|\mu_B(\mu_O)| - q+2)] - [\log_2(|\mu_B(\mu_O)| - q-h+2)] < h$
  - rezultă  $C_q < C_{(q+h)}$

## Obsservatie

Costul minim pentru codificarea setului de microoperații  $\mu\mathcal{B}(mO)$  poate să fie mai mare decât cel dat de propoziția 3 adică:

$$C \geq q-1 + [\log_2(|\mu\mathcal{B}(mO)| - q + 2)] \text{ când } |\text{MCC}|_{\max} \leq |\mu\mathcal{B}(mO)| - q.$$

Într-adevăr costul minim corespunde când:

(q-1) câmpuri specifice fiecare câte o microoperație;  
cel de-al q-lea câmp codifică  $|\mu\mathcal{B}(mO)| - q + 1$  microoperații.

Dacă  $|\text{MCC}|_{\max} \leq |\mu\mathcal{B}(mO)| - q$ , rezultă că  $|\text{MCC}|_{\max} < |\mu\mathcal{B}(mO)| - q + 1$ , ceea ce ar face ca în ultimul câmp să nu fie toate microoperațiile compatibile.

În acest caz trebuie renunțat la organizarea microinstrucțiunii în (q-1) câmpuri de 1 bit, dar prin aceasta se mărește și costul de implementare.

Metodă de minimizare a numărului de biți necesari pentru codificarea microinstrucțiunilor complete

- **1. Se alege clasa de incompatibilitate maximă care are cardinalitatea maximă MIC.**

Fie  $\text{MIC}_m \in \text{MIC}$  astfel încât  $|\text{MIC}_m| \geq |\text{MIC}_j| <$  pentru  $j \neq m$ ,  $1 \leq j \leq |\text{MIC}|$

Se generează clasele de compatibilitate maximă asociate AMCC clasei MIC.

$\text{AMCC}_m = \{\text{MCC} \text{ pt orice } \mu_o \in \text{MIC}_m \exists \text{ MCC astfel încât } \mu_o \in \text{MCC}\}$

- **2. Se formează tabela de acoperire modificată prin considerarea numai a claselor de compatibilitate maximă ce aparțin AMCC.**

**TAM :  $\text{AMCC}_m \times \mu\mathcal{B}(\mu_o) \rightarrow \mathcal{B}$**

Se caută multimea de clase de compatibilitate maximă esențiale  $\{\text{MCC}_e\}$  inclus în  $\text{AMCC}_m$  astfel încât există  $\text{MCC}_e$  unic pentru care :  $\mu_{o_i} \in \text{MCC}_e$  avem  $\text{TAM}(\text{MCC}_e, \mu_{o_i}) = 1$ .

Se elimină coloanele corespunzătoare microoperațiilor ce sunt acoperite de clasele esențiale și cele corespunzătoare microoperațiilor componente ale clasei  $\text{MIC}_m$ , obținându-se o **tabelă de acoperire redusă** :

**TAR :  $\text{AMCC}_m \times (\mu\mathcal{B}(\mu_o) \setminus \text{MIC}_m) \setminus \{\text{MCC}_e\} \rightarrow \mathcal{B}$**

# Cont

- **3. Se generează setul soluțiilor de acoperire a microoperațiilor  $\{\text{MCC}_{\text{ap}}\}$ :  $\{\mu\mathcal{B}(\mu\mathcal{O}) \setminus \text{MIC}_m\} \setminus \{\text{MCC}_e\}$**

Fie  $\text{SOL} = \{\text{SOL}_1, \text{SOL}_2, \dots, \text{SOL}_p\}$  soluțiile de acoperire în care

$$\text{SOL}_j = \wedge / (\{\text{MCC}_e\} \cup \{\text{MCC}_{\text{ap}}\})$$

Se consideră soluția parțială cu cardinalitatea minimă  $\text{SOL}_j$ .

Se generează soluția de acoperire a microoperațiilor ce aparțin clasei  $\text{MIC}_m$ , încă neacoperite:  $\{\text{MCC}_{\text{am}}\}$ .

- **4. Se generează setul soluțiilor de acoperire completă:**

$$\text{SOLC}_j = \text{SOL}_j \wedge \{\text{MCC}_{\text{am}}\}.$$

Se calculează costul soluției de acoperire completă cu cardinalitatea minimă.

# Exemplu

- ▶ Considerăm setul de microinstructiuni din cadrul microsubblocului :  
 $\mu B = \{\mu o_1, \mu o_2, \mu o_3, \mu o_4, \mu o_5, \mu o_6, \mu o_7, \mu o_8, \mu o_9, \mu o_{10}\}$
- ▶ Presupunem că pe baza dependenței de date și a conflictului de resurse între microoperații a rezultat următoarea partiție a microsubblocului :
  - ▶  $\mu IC_1 = \{\mu o_1, \mu o_2, \mu o_4\}$
  - ▶  $\mu IC_2 = \{\mu o_1, \mu o_3, \mu o_5\}$
  - ▶  $\mu IC_3 = \{\mu o_2, \mu o_6, \mu o_8, \mu o_9\}$
  - ▶  $\mu IC_4 = \{\mu o_4, \mu o_5, \mu o_7, \mu o_8\}$
  - ▶  $\mu IC_5 = \{\mu o_3, \mu o_4, \mu o_5, \mu o_6, \mu o_7\}$
  - ▶  $\mu IC_6 = \{\mu o_6, \mu o_9, \mu o_{10}\}$
  - ▶  $\mu IC_7 = \{\mu o_7, \mu o_{10}\}$

- $\mu C_1 = \{\mu o_1, \mu o_2, \mu o_4\}$
- $\mu C_2 = \{\mu o_1, \mu o_3, \mu o_5\}$
- $\mu C_3 = \{\mu o_2, \mu o_6, \mu o_8, \mu o_9\}$
- $\mu C_4 = \{\mu o_4, \mu o_5, \mu o_7, \mu o_8\}$
- $\mu C_5 = \{\mu o_3, \mu o_4, \mu o_5, \mu o_6, \mu o_7\}$
- $\mu C_6 = \{\mu o_6, \mu o_9, \mu o_{10}\}$
- $\mu C_7 = \{\mu o_7, \mu o_{10}\}$

# Tabela de incompatibilitate

- ▶  $\mu C_1 = \{\mu o_1, \mu o_2, \mu o_4\}$
- ▶  $\mu C_2 = \{\mu o_1, \mu o_3, \mu o_5\}$
- ▶  $\mu C_3 = \{\mu o_2, \mu o_6, \mu o_8, \mu o_9\}$
- ▶  $\mu C_4 = \{\mu o_4, \mu o_5, \mu o_7, \mu o_8\}$
- ▶  $\mu C_5 = \{\mu o_3, \mu o_4, \mu o_5, \mu o_6, \mu o_7\}$
- ▶  $\mu C_6 = \{\mu o_6, \mu o_9, \mu o_{10}\}$
- ▶  $\mu C_7 = \{\mu o_7, \mu o_{10}\}$

| $\mu o \setminus \mu o$ | $\mu o_1$ | $\mu o_2$ | $\mu o_3$ | $\mu o_4$ | $\mu o_5$ | $\mu o_6$ | $\mu o_7$ | $\mu o_8$ | $\mu o_9$ | $\mu o_{10}$ |
|-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|--------------|
| $\mu o_1$               | X         | X         | X         | X         | X         |           |           |           |           |              |
| $\mu o_2$               | X         | X         |           | X         |           | X         |           | X         | X         |              |
| $\mu o_3$               | X         |           | X         | X         | X         | X         | X         |           |           |              |
| $\mu o_4$               | X         | X         | X         | X         | X         | X         | X         | X         |           |              |
| $\mu o_5$               | X         |           | X         | X         | X         | X         | X         | X         |           |              |
| $\mu o_6$               |           | X         | X         | X         | X         | X         | X         | X         | X         | X            |
| $\mu o_7$               |           |           | X         | X         | X         | X         | X         | X         |           | X            |
| $\mu o_8$               |           | X         |           | X         | X         | X         | X         | X         | X         |              |
| $\mu o_9$               |           | X         |           |           |           | X         |           | X         | X         | X            |
| $\mu o_{10}$            |           |           |           |           |           | X         | X         |           | X         | X            |

# Clasele maximale de incompatibilitate

- ▶  $MIC_1 = \{\mu o_1, \mu o_2, \mu o_4\}$
- ▶  $MIC_2 = \{\mu o_1, \mu o_3, \mu o_4, \mu o_5\}$
- ▶  $MIC_3 = \{\mu o_2, \mu o_4, \mu o_6, \mu o_8\}$
- ▶  $MIC_4 = \{\mu o_2, \mu o_6, \mu o_8, \mu o_9\}$
- ▶  $MIC_5 = \{\mu o_3, \mu o_4, \mu o_5, \mu o_6, \mu o_7\}$
- ▶  $MIC_6 = \{\mu o_4, \mu o_5, \mu o_6, \mu o_7, \mu o_8\}$
- ▶  $MIC_7 = \{\mu o_6, \mu o_7, \mu o_{10}\}$
- ▶  $MIC_8 = \{\mu o_6, \mu o_9, \mu o_{10}\}$
- ▶  $MIC_{max} = MIC_5 \text{ sau } MIC_6$

| $\mu o \backslash \mu o$ | $\mu o_1$ | $\mu o_2$ | $\mu o_3$ | $\mu o_4$ | $\mu o_5$ | $\mu o_6$ | $\mu o_7$ | $\mu o_8$ | $\mu o_9$ | $\mu o_{10}$ |
|--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|--------------|
| $\mu o_1$                | X         | X         | X         | X         | X         |           |           |           |           |              |
| $\mu o_2$                | X         | X         |           | X         |           | X         |           | X         | X         |              |
| $\mu o_3$                | X         |           | X         | X         | X         | X         | X         |           |           |              |
| $\mu o_4$                | X         | X         | X         | X         | X         | X         | X         | X         |           |              |
| $\mu o_5$                | X         |           | X         | X         | X         | X         | X         | X         |           |              |
| $\mu o_6$                |           | X         | X         | X         | X         | X         | X         | X         | X         | X            |
| $\mu o_7$                |           |           | X         | X         | X         | X         | X         |           |           | X            |
| $\mu o_8$                |           | X         |           | X         | X         | X         | X         | X         |           |              |
| $\mu o_9$                |           | X         |           |           |           | X         |           | X         | X         | X            |
| $\mu o_{10}$             |           |           |           |           |           | X         | X         | X         | X         | X            |

# Clasele maximale de compatibilitate

- $MCC_1 = \{\mu o_1, \mu o_6\}$
- $MCC_2 = \{\mu o_1, \mu o_7, \mu o_9\}$
- $MCC_3 = \{\mu o_1, \mu o_8, \mu o_{10}\}$
- $MCC_4 = \{\mu o_2, \mu o_3, \mu o_{10}\}$
- $MCC_5 = \{\mu o_2, \mu o_5, \mu o_{10}\}$
- $MCC_6 = \{\mu o_2, \mu o_7\}$
- $MCC_7 = \{\mu o_3, \mu o_8, \mu o_{10}\}$
- $MCC_8 = \{\mu o_3, \mu o_9\}$
- $MCC_9 = \{\mu o_4, \mu o_9\}$
- $MCC_{10} = \{\mu o_4, \mu o_{10}\}$
- $MCC_{11} = \{\mu o_5, \mu o_9\}$

| $\mu o \backslash \mu o$ | $\mu o_1$ | $\mu o_2$ | $\mu o_3$ | $\mu o_4$ | $\mu o_5$ | $\mu o_6$ | $\mu o_7$ | $\mu o_8$ | $\mu o_9$ | $\mu o_{10}$ |
|--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|--------------|
| $\mu o_1$                | X         | X         | X         | X         | X         |           |           |           |           |              |
| $\mu o_2$                | X         | X         |           | X         |           | X         |           | X         | X         |              |
| $\mu o_3$                | X         |           | X         | X         | X         | X         | X         |           |           |              |
| $\mu o_4$                | X         | X         | X         | X         | X         | X         | X         | X         |           |              |
| $\mu o_5$                | X         |           | X         | X         | X         | X         | X         | X         |           |              |
| $\mu o_6$                |           | X         | X         | X         | X         | X         |           | X         | X         | X            |
| $\mu o_7$                |           |           | X         | X         | X         | X         | X         |           |           | X            |
| $\mu o_8$                |           | X         |           | X         | X         | X         | X         | X         | X         |              |
| $\mu o_9$                |           | X         |           |           |           | X         |           | X         | X         | X            |
| $\mu o_{10}$             |           |           |           |           |           | X         | X         |           | X         | X            |

# Tabela de acoperire a microoperațiilor de către clasele maximale de compatibilitate

- $MCC_1 = \{\mu o_1, \mu o_6\}$
- $MCC_2 = \{\mu o_1, \mu o_7, \mu o_9\}$
- $MCC_3 = \{\mu o_1, \mu o_8, \mu o_{10}\}$
- $MCC_4 = \{\mu o_2, \mu o_3, \mu o_{10}\}$
- $MCC_5 = \{\mu o_2, \mu o_5, \mu o_{10}\}$
- $MCC_6 = \{\mu o_2, \mu o_7\}$
- $MCC_7 = \{\mu o_3, \mu o_8, \mu o_{10}\}$
- $MCC_8 = \{\mu o_3, \mu o_9\}$
- $MCC_9 = \{\mu o_4, \mu o_9\}$
- $MCC_{10} = \{\mu o_4, \mu o_{10}\}$
- $MCC_{11} = \{\mu o_5, \mu o_9\}$

| $MCC \setminus \mu o$ | $\mu o_1$ | $\mu o_2$ | $\mu o_3$ | $\mu o_4$ | $\mu o_5$ | $\mu o_6$ | $\mu o_7$ | $\mu o_8$ | $\mu o_9$ | $\mu o_{10}$ |
|-----------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|--------------|
| $MCC_1$               | x         |           |           |           |           | x         |           |           |           |              |
| $MCC_2$               | x         |           |           |           |           |           | x         |           | x         |              |
| $MCC_3$               | x         |           |           |           |           |           |           | x         |           | x            |
| $MCC_4$               |           | x         | x         |           |           |           |           |           |           | x            |
| $MCC_5$               |           | x         |           |           | x         |           |           |           |           | x            |
| $MCC_6$               |           | x         |           |           |           |           |           | x         |           |              |
| $MCC_7$               |           |           | x         |           |           |           |           | x         |           | x            |
| $MCC_8$               |           |           |           | x         |           |           |           |           | x         |              |
| $MCC_9$               |           |           |           |           | x         |           |           |           |           | x            |
| $MCC_{10}$            |           |           |           |           | x         |           |           |           |           | x            |
| $MCC_{11}$            |           |           |           |           |           | x         |           |           | x         |              |

## Obs

- ▶ Se observă că există două clase maximale de incompatibilitate cu cardinalitate 5 ( $MIC_5$  și  $MIC_6$ ).
- ▶ Costul minim absolut, conform proprietății 4 este **C = 7**.
- ▶ **Să considerăm  $AMCC_6$  - clasele maximale de compatibilitate asociate clasei maximale de incompatibilitate  $MIC_6$  :**  
 $AMCC_6=\{MCC_1, MCC_2, MCC_3, MCC_5, MCC_6, MCC_7, MCC_9, MCC_{10}, MCC_{11}\}$
- ▶ Conform propoziției 1 clasele maximale de compatibilitate din  **$AMCC_6$**  acoperă toate microoperațiile microsubblockului.

# Tabela de acoperire modificată

|            | $\mu o_1$ | $\mu o_2$ | $\mu o_3$ | $\mu o_4$ | $\mu o_5$ | $\mu o_6$ | $\mu o_7$ | $\mu o_8$ | $\mu o_9$ | $\mu o_{10}$ |
|------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|--------------|
| $MCC_1$    | x         |           |           |           |           | x         |           |           |           |              |
| $MCC_2$    | x         |           |           |           |           |           | x         |           | x         |              |
| $MCC_3$    | x         |           |           |           |           |           |           | x         |           | x            |
| $MCC_5$    |           | x         |           |           | x         |           |           |           |           | x            |
| $MCC_6$    |           | x         |           |           |           |           | x         |           |           |              |
| $MCC_7$    |           |           | x         |           |           |           |           | x         |           | x            |
| $MCC_9$    |           |           |           | x         |           |           |           |           | x         |              |
| $MCC_{10}$ |           |           |           | x         |           |           |           |           |           | x            |
| $MCC_{11}$ |           |           |           |           | x         |           |           |           | x         |              |

Se observă că  $MCC_1$  și  $MCC_7$  sunt esențiale și vor face parte din soluția finală.

# Tabela de acoperire redusă

- ▶ Considerăm, conform algoritmului, acoperite microoperațiile:
  - ▶ incluse în clasele de compatibilitate maximă **MCC<sub>1</sub>** și **MCC<sub>7</sub>**, esentiale și
  - ▶ cele componente ale clasei de incompatibilitate maximă **MIC<sub>6</sub>**.
- ▶ Astfel, din tabela de acoperire se elimină microoperațiile
  - ▶ **μo<sub>1</sub>, μo<sub>6</sub>, μo<sub>3</sub>, μo<sub>8</sub>, μo<sub>10</sub>, respectiv μo<sub>4</sub>, μo<sub>5</sub>, μo<sub>7</sub>**.

| MCC \ μo          | μo <sub>2</sub> | μo <sub>9</sub> |
|-------------------|-----------------|-----------------|
| MCC <sub>2</sub>  |                 | x               |
| MCC <sub>5</sub>  | x               |                 |
| MCC <sub>6</sub>  | x               |                 |
| MCC <sub>9</sub>  |                 | x               |
| MCC <sub>11</sub> |                 | x               |

# Acoperirea microoperațiilor $\mu o_2$ și $\mu o_9$

- ➡ Se observă că tabela de acoperire s-a redus substanțial. Acoperirea microoperațiilor  $\mu o_2$  și  $\mu o_9$  se poate face cu ajutorul următoarelor clase maximale de compatibilitate :
  - ➡ MCC<sub>2</sub>, MCC<sub>5</sub>
  - ➡ MCC<sub>2</sub>, MCC<sub>6</sub>
  - ➡ MCC<sub>5</sub>, MCC<sub>9</sub>
  - ➡ MCC<sub>5</sub>, MCC<sub>11</sub>
  - ➡ MCC<sub>6</sub>, MCC<sub>9</sub>
  - ➡ MCC<sub>6</sub>, MCC<sub>11</sub>

| MCC \ $\mu o$     | $\mu o_2$ | $\mu o_9$ |
|-------------------|-----------|-----------|
| MCC <sub>2</sub>  |           | x         |
| MCC <sub>5</sub>  | x         |           |
| MCC <sub>6</sub>  | x         |           |
| MCC <sub>9</sub>  |           | x         |
| MCC <sub>11</sub> |           | x         |

# Setul soluțiilor

- ▶  $SOL = \{SOL_1, SOL_2, SOL_3, SOL_4, SOL_5, SOL_6\}$  astfel :
- ▶  $SOL = \{$ 
  - $MCC_1 MCC_7 MCC_2 MCC_5 ;$
  - $MCC_1 MCC_7 MCC_2 MCC_6 ;$
  - $\textcolor{red}{MCC_1 MCC_7 MCC_5 MCC_9} ;$
  - $MCC_1 MCC_7 MCC_5 MCC_{11} ;$
  - $MCC_1 MCC_7 MCC_6 MCC_9 ;$
  - $MCC_1 MCC_7 MCC_6 MCC_{11}$ $\}$
- ▶ Considerând soluția :  $\textcolor{red}{MCC_1 MCC_7 MCC_5 MCC_9} \in SOL$ , se acoperă toate microoperațiile cu excepția  $\mu_7 \in MIC_6$ .
- ▶ Pentru acoperirea lui  $\mu_7$  se poate lua una din clasele maximale de compatibilitate  $MCC_2$  sau  $MCC_6$ .
- ▶  $SOLC_{31} = \{MCC_1, MCC_7, MCC_5, MCC_9, MCC_2\}$
- ▶  $SOLC_{32} = \{MCC_1, MCC_7, MCC_5, MCC_9, MCC_6\}$

# Setul soluțiilor complete

- ▶  $SOLC_3 = \{SOLC_{31}, SOC_{32}\}$

unde

- ▶  $SOLC_{31}=\{MCC_1, MCC_7, MCC_5, MCC_9, MCC_2\}$
- ▶  $SOLC_{32}=\{MCC_1, MCC_7, MCC_5, MCC_9, MCC_6\}$

Ambele soluții complete au aceeași cardinalitate ce specifică numărul minim de câmpuri necesar pentru codificarea microinstrucțiunilor ce descriu microsubblockul.

Alegând  $SOLC_{31}$ , rezultă următoarea grupare a microoperațiilor:

MC2                    MC7                    MC5                    MC9                    MC1  
 $\{ \mu o_1 \mu o_7 \mu o_9 ; \mu o_3 \mu o_8 \mu o_{10} ; \mu o_2 \mu o_5 ; \mu o_4 ; \mu o_6 \}$

Microoperațiile cuprinse într-un camp nu mai apar în alte campuri

Sunt mai multe posibilități

MC2                    MC5                    MC7                    MC9                    MC1  
 $\{ \mu o_1 \mu o_7 \mu o_9 ; \mu o_2 \mu o_5 \mu o_{10} ; \mu o_3 \mu o_8 ; \mu o_4 ; \mu o_6 \}$

- ▶ ce necesită 8 biți pentru codificare.

- ▶  $MCC_1=\{\mu o_1, \mu o_6\}$
- ▶  $MCC_2=\{\mu o_1, \mu o_7, \mu o_9\}$
- ▶  $MCC_3=\{\mu o_1, \mu o_8, \mu o_{10}\}$
- ▶  $MCC_4=\{\mu o_2, \mu o_3, \mu o_{10}\}$
- ▶  $MCC_5=\{\mu o_2, \mu o_5, \mu o_{10}\}$
- ▶  $MCC_6=\{\mu o_2, \mu o_7\}$
- ▶  $MCC_7=\{\mu o_3, \mu o_8, \mu o_{10}\}$
- ▶  $MCC_8=\{\mu o_3, \mu o_9\}$
- ▶  $MCC_9=\{\mu o_4, \mu o_9\}$
- ▶  $MCC_{10}=\{\mu o_4, \mu o_{10}\}$
- ▶  $MCC_{11}=\{\mu o_5, \mu o_9\}$

