

ARHITECTURA SISTEMELOR DE CALCUL - CURS 0x09

SISTEME MULTI-PROCESOR, IERARHIA MEMORIEI

Cristian Rusu

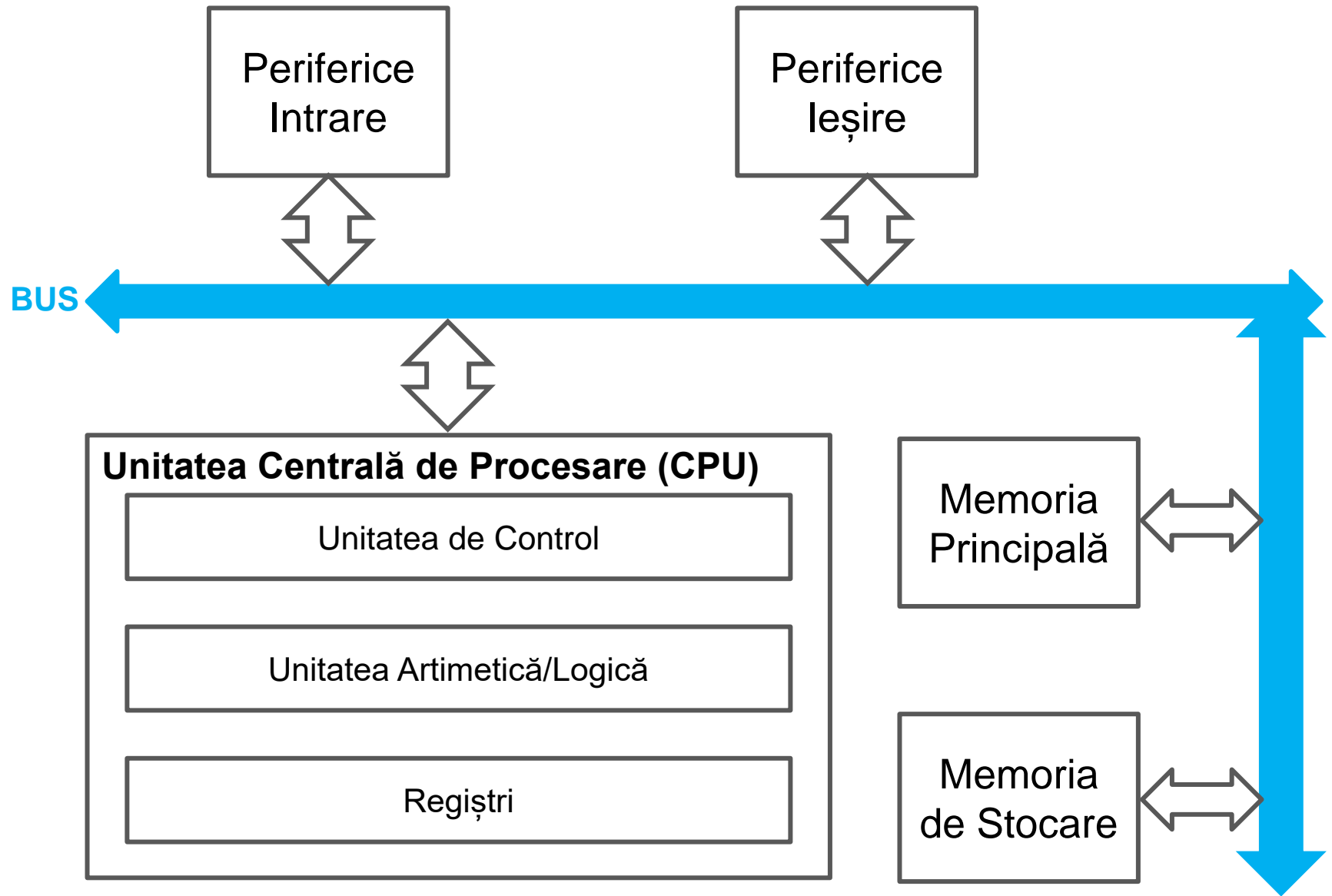
DATA TRECUȚĂ

- **pipelining**
- **branch prediction**
- **out of order execution**

CUPRINS

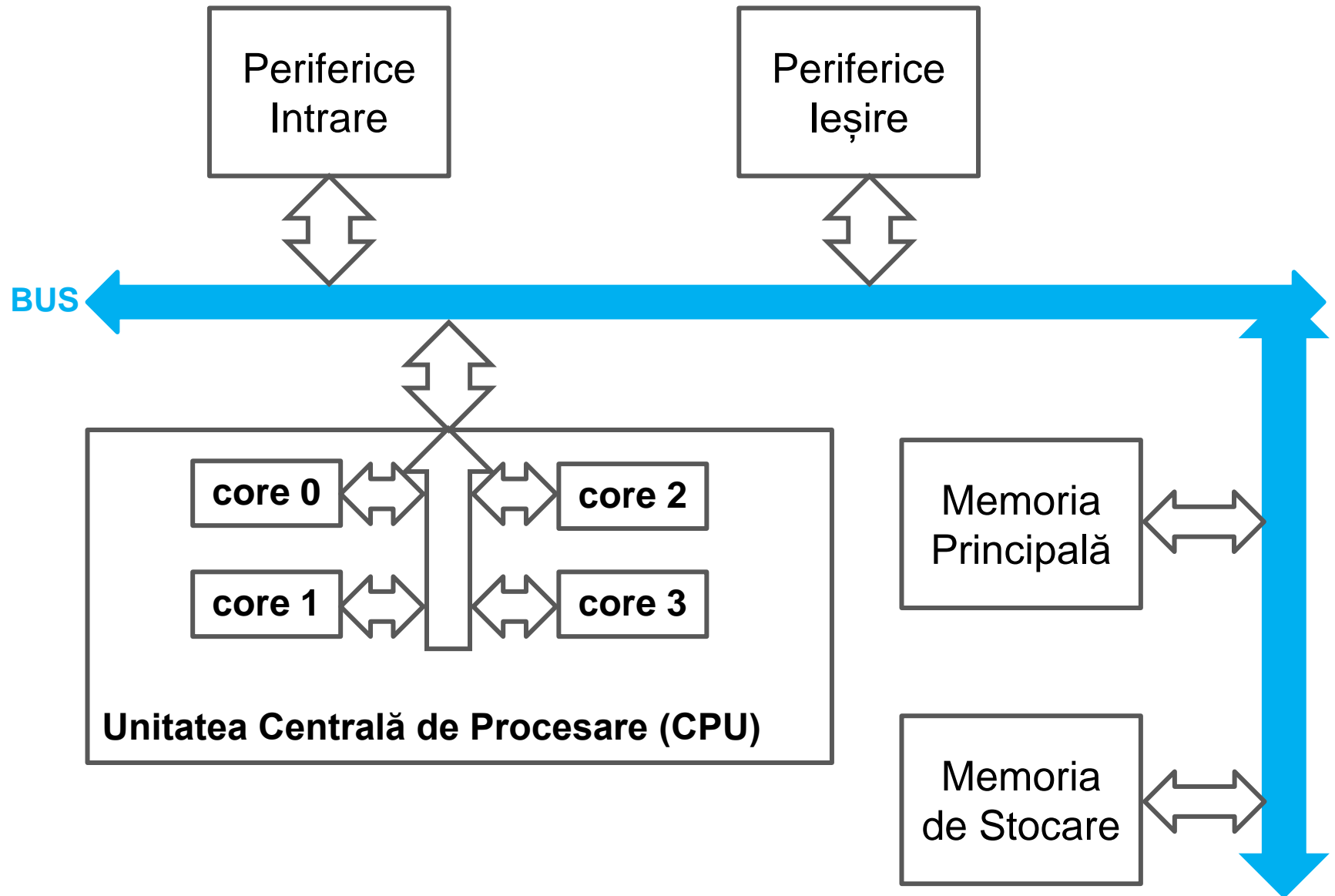
- **sisteme multi-procesor**
- **ierarhia memoriei**
- **caching**

ARHITECTURA DE BAZĂ



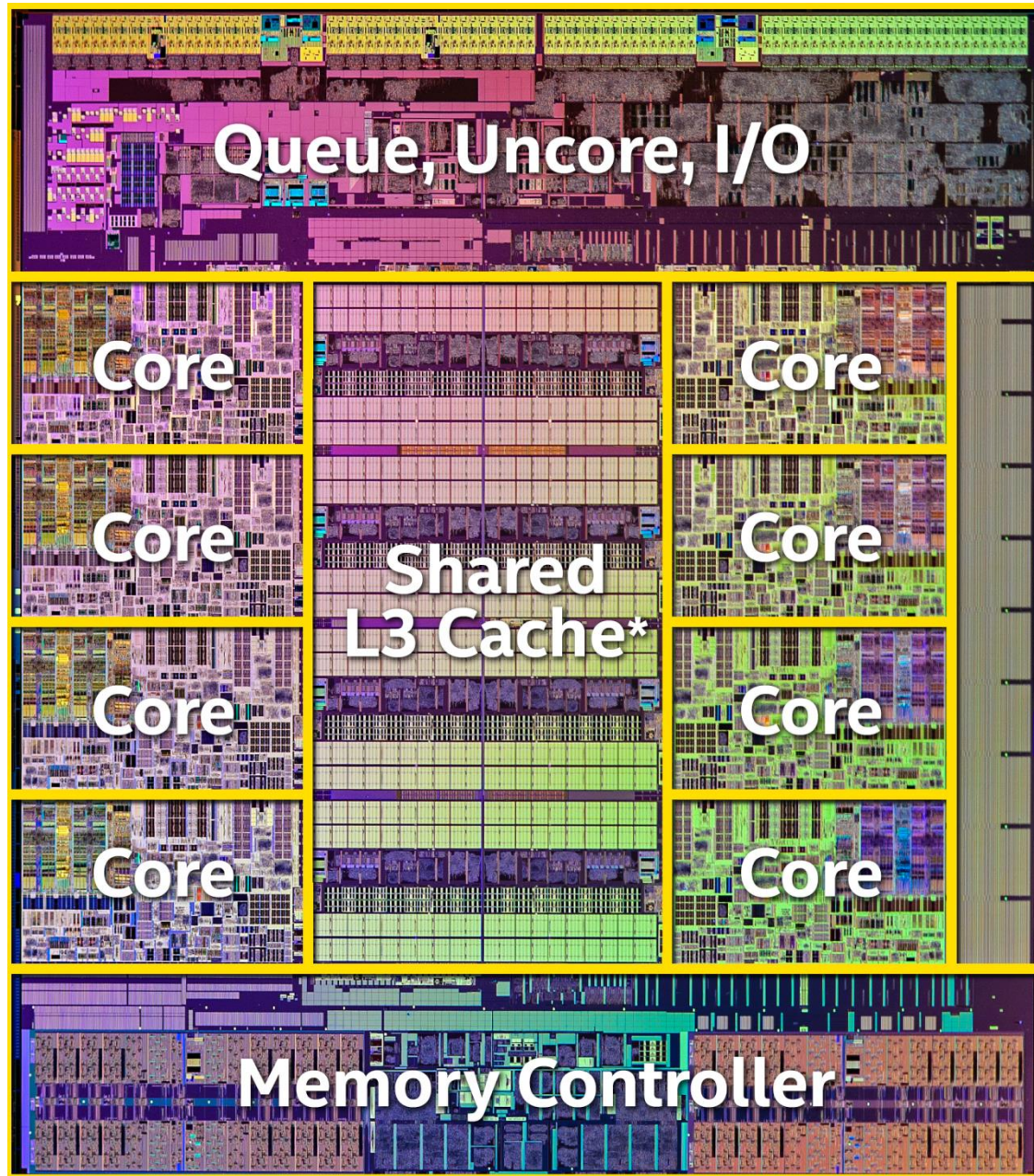
rularea simultană a programelor este “simulată”

ARHITECTURA MULTI-CORE

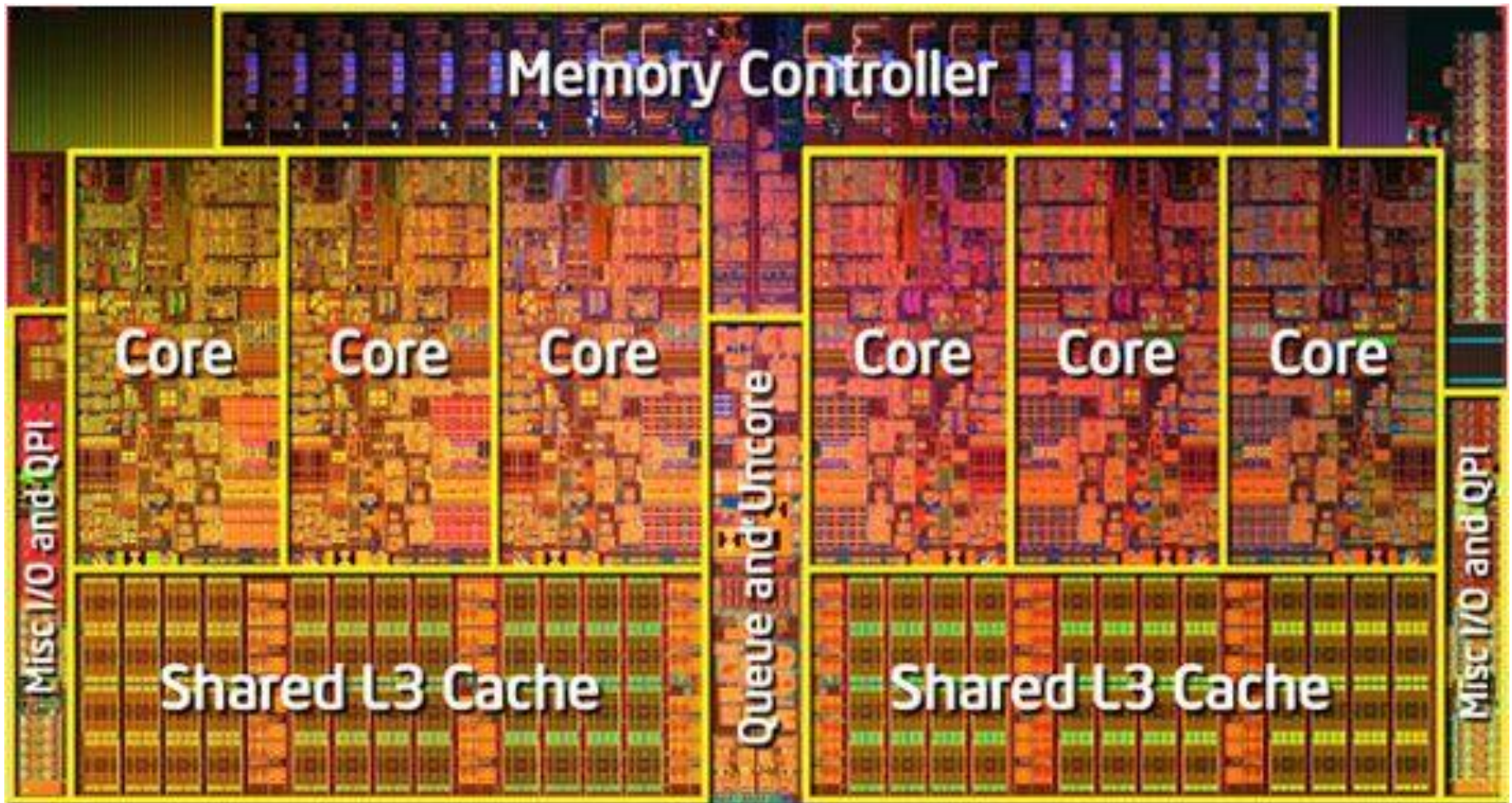


rularea simultană a programelor este "reală"

ARHITECTURA MULTI-CORE



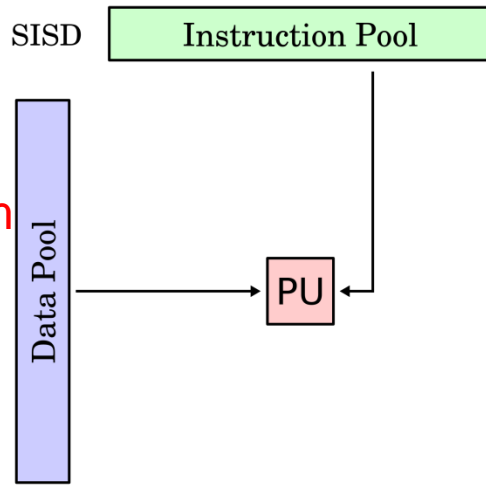
ARHITECTURA MULTI-CORE



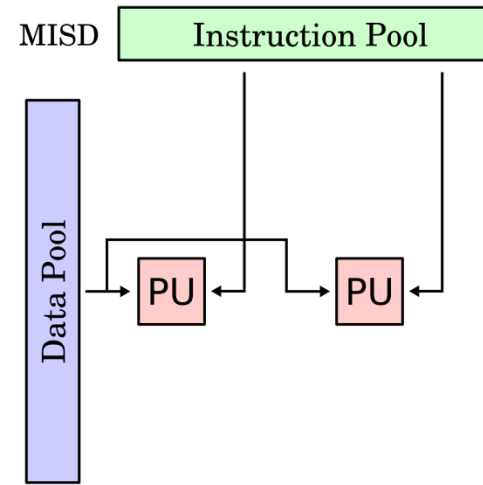
e bine că avem multe core-uri, noua problemă: să aducem datele la core-uri

ARHITECTURA MULTI-CORE

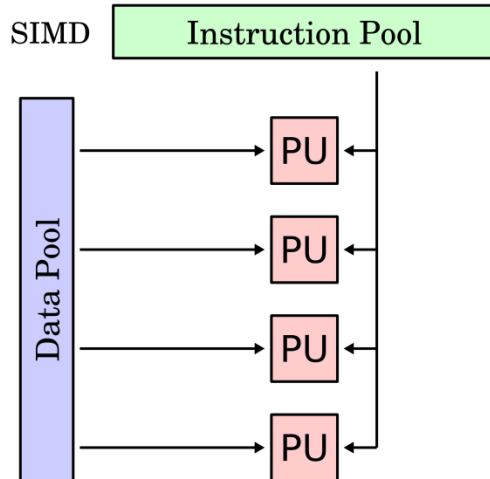
- **taxonomia Flynn**



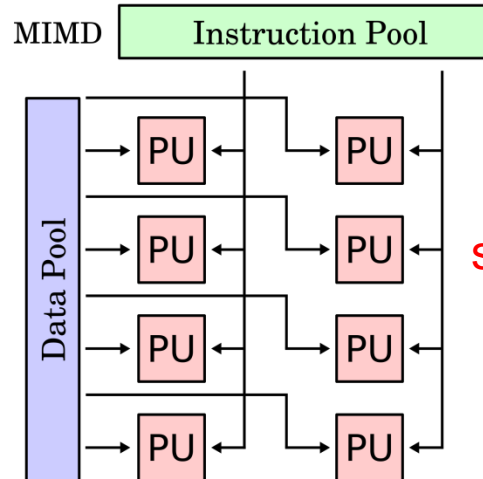
orice sistem
uni-core



verificare
calcole



xmm



sisteme
superscalare
multi-core,
sisteme
distribuite

ARHITECTURA MULTI-CORE

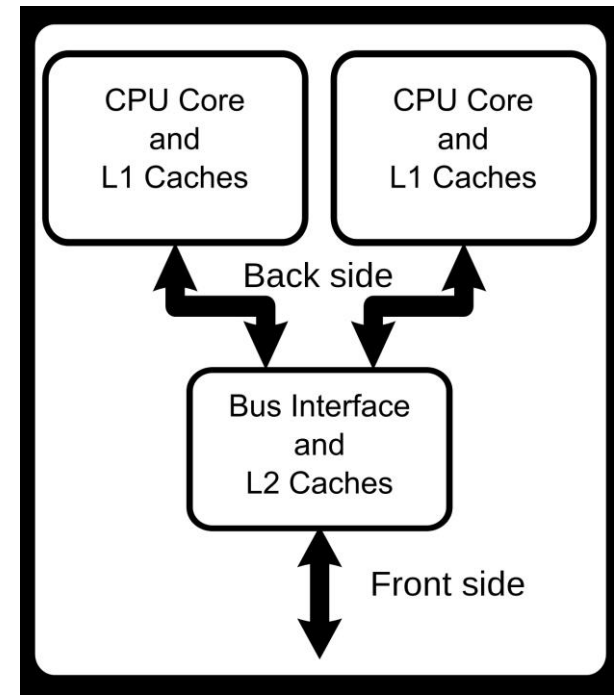
- **problema:**
 - sigur toate core-urile vor instrucțiuni
 - probabil toate core-urile vor să acceseze memoria
 - din când în când core-uri vor să facă operații I/O
 - toate comunică pe același bus
 - conflicte de acces
 - coadă de priorități pentru acces
- **soluția:**
 - fiecare core are “o memorie” locală cu care să poată comunica rapid
 - ierarhizarea memoriei

TIPURI DE PARALELISM

- **la nivel de biți**
 - modificarea dimensiunii “cuvintelor” procesorului (procesoarea pe 16, 32 și 64 de biți)
 - cu câți biți poate sistemul de calcul să opereze
- **la nivel de instrucțiune**
 - pipelines
- **la nivel de task-uri**
 - multi-thread
 - multi-process
- **la nivel de blocuri**
 - vectorizarea operațiilor
 - operații pe blocuri

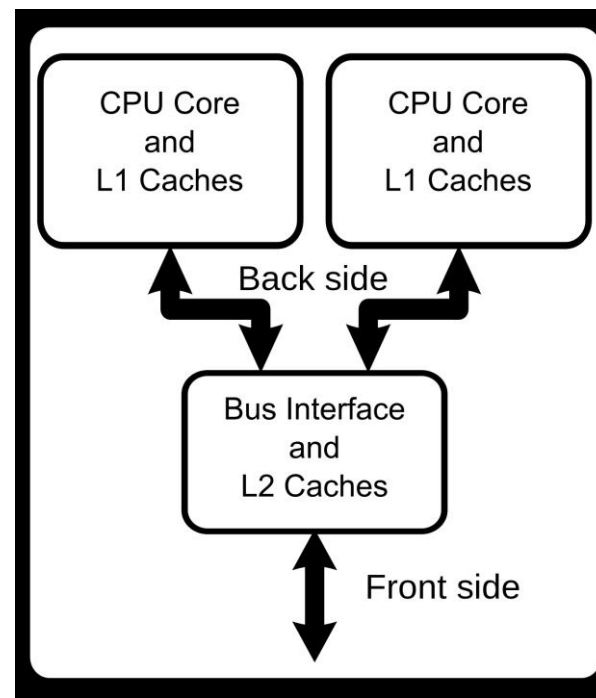
TIPURI DE PARALELISM

- în trecut 1 procesor = 1 unitate de calcul
- de câteva decenii 1 procesor = mai multe (2,4,...) unități de calcul
 - sisteme multi-core
 - unitățile de calcul au resurse proprii: regiștrii, ALU, FPU, cache...
 - dar unele resurse sunt împărtășite între toate unitățile de calcul (cache L3, controller-ul de memorie)
- avantaje: coerența cache-ului
- dezavantaj: software special



TIPURI DE PARALELISM

- în trecut 1 procesor = 1 unitate de calcul
- de câteva decenii 1 procesor = mai multe (2,4,...) unități de calcul
 - sisteme multi-thread: Hyper-Threading, Chip Multi-threading
 - unitățile de calcul au resurse proprii: regiștrii (și cam atât)
 - restul resurselor de calcul sunt împărtășite de thread-uri
 - thread-urile sunt blocate dacă resursele de calcul sunt ocupate
- avantaj: dacă resursa este disponibilă
- dezavantaj: competiție pentru resurse
- hyper-threading = logical cores

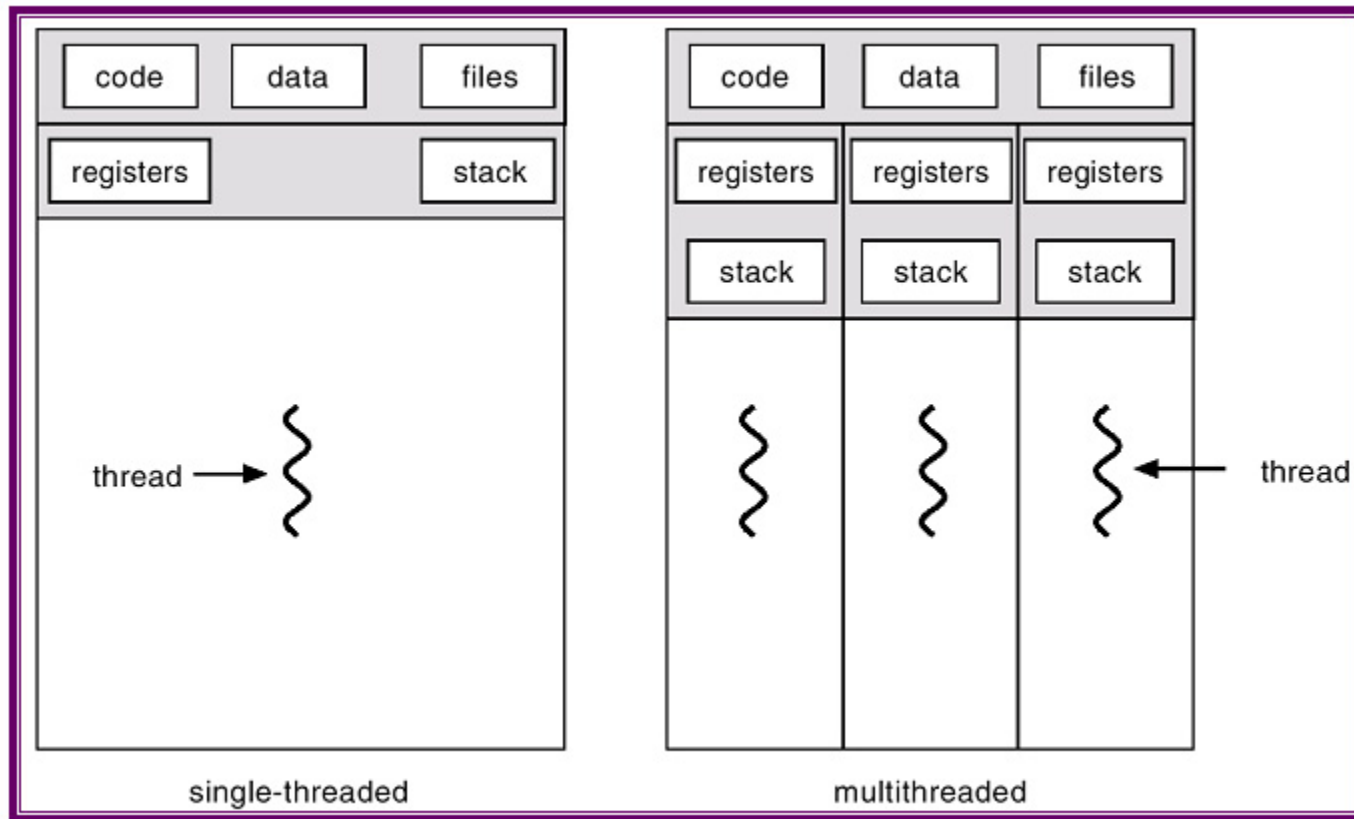


TIPURI DE PARALELISM

- **Instruction-level parallelism (ILP)**
 - Instruction Pipelining
 - Register Renaming
 - Speculative Execution
 - Branch Prediction
 - Value Prediction
 - Memory Dependence Prediction
 - Cache Latency Prediction
 - Out-of-order Execution
 - Dataflow Analysis/Execution

TIPURI DE PARALELISM

- Task parallelism (ILP)
 - multi-thread
 - multi-process



TIPURI DE PARALELISM

Processes Performance App history Startup Users Details Services



CPU
44% 2.18 GHz



Memory
22/256 GB (9%)



Disk 0 (C:)
SSD
0%



Disk 1
HDD
0%



Ethernet
Ethernet
S: 864 R: 64.0 Kbps



Ethernet
VirtualBox Host-Only Network
S: 0 R: 0 Kbps



Ethernet
VMware Network Adapter VMnet1
S: 0 R: 0 Kbps



Ethernet
VMware Network Adapter VMnet8
S: 0 R: 0 Kbps



GPU 0
NVIDIA Quadro RTX 4000
7% (34 °C)



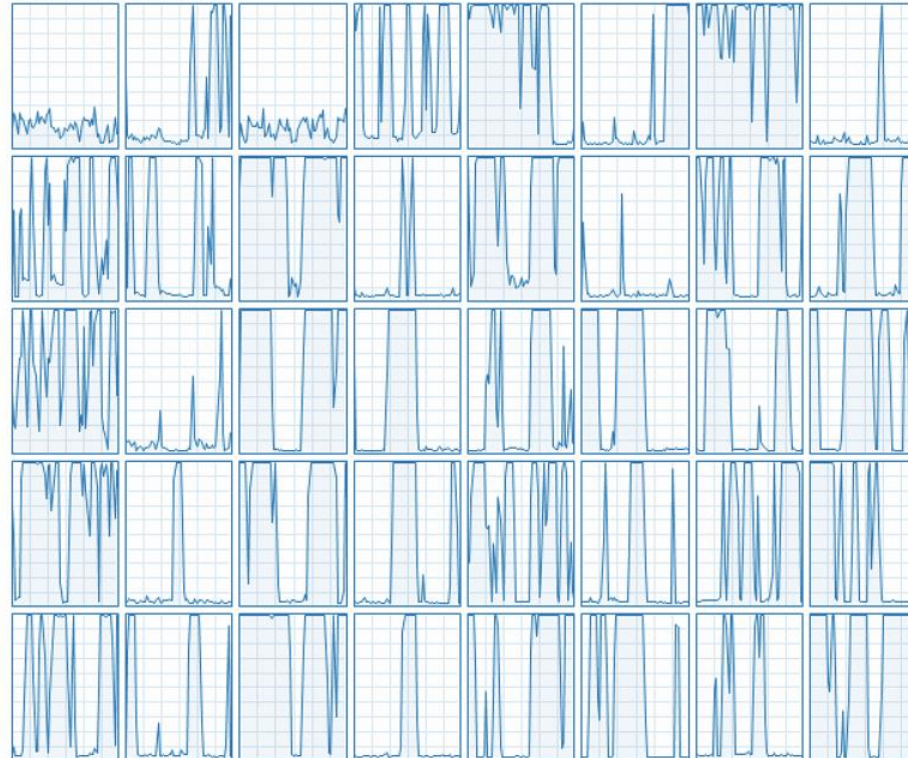
GPU 1
NVIDIA Quadro RTX 4000
0% (32 °C)

CPU

Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz

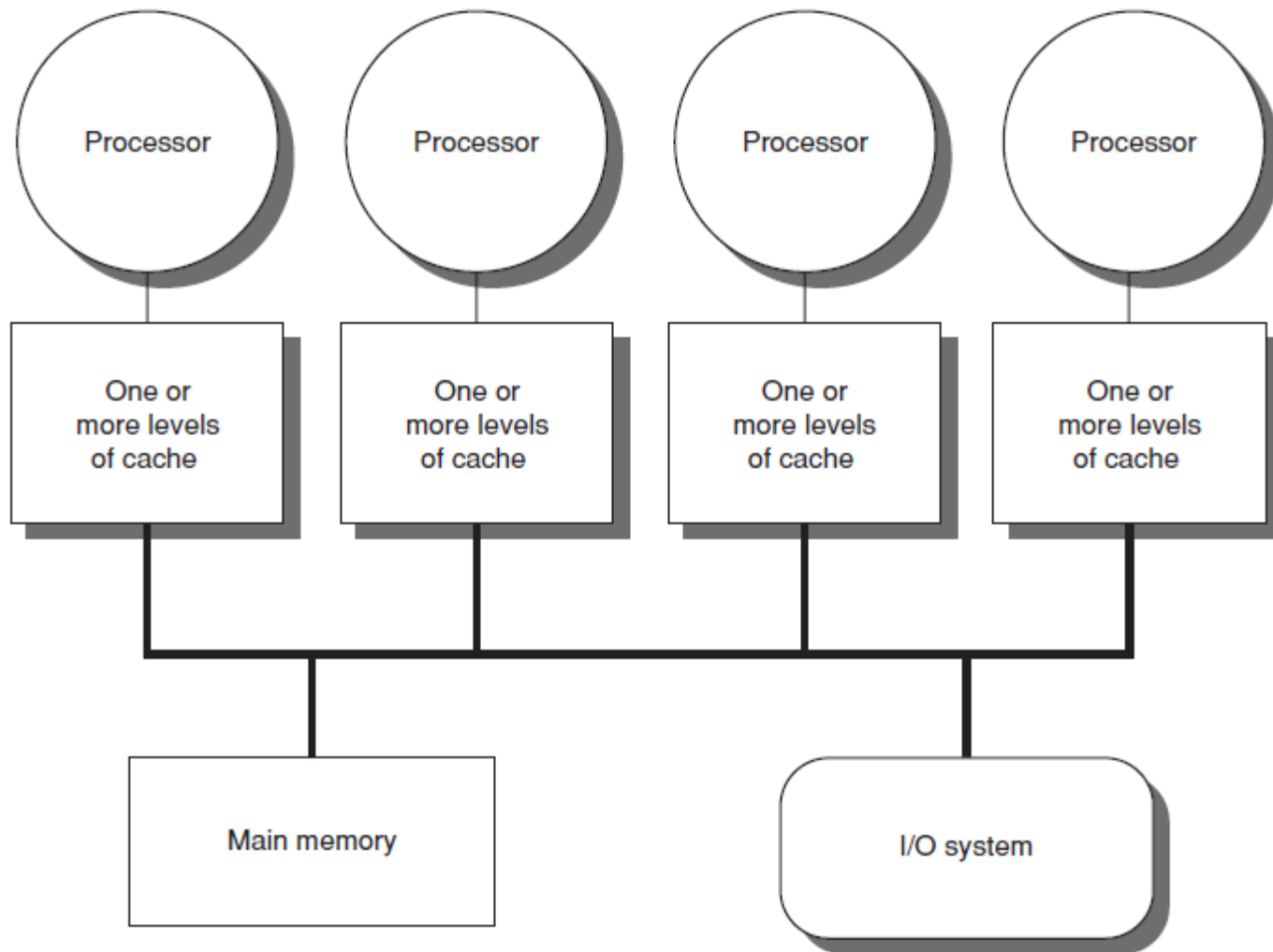
% Utilization over 60 seconds

100%



Utilization	Speed	Base speed:	2.19 GHz
44%	2.18 GHz	Sockets:	2
		Cores:	20
Processes	Threads	Handles	Logical processors: 40
304	4332	200610	Virtualization: Enabled
Up time		L1 cache:	1.2 MB
51:19:44:41		L2 cache:	20.0 MB
		L3 cache:	27.5 MB

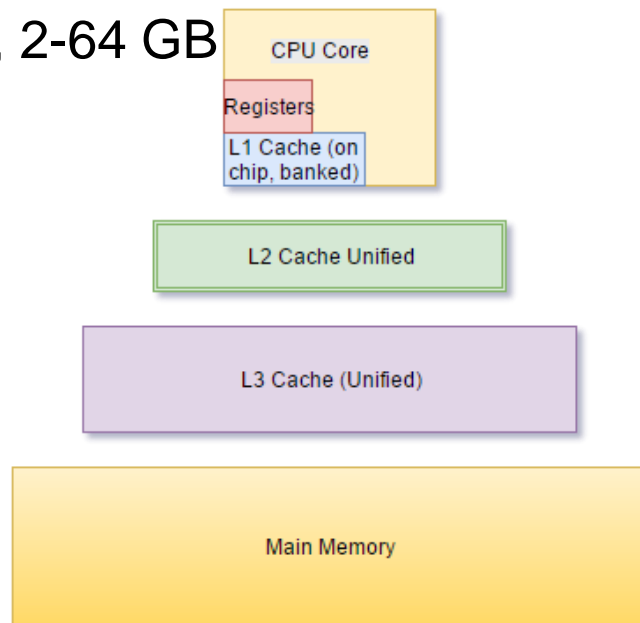
IERARHIZAREA MEMORIEI



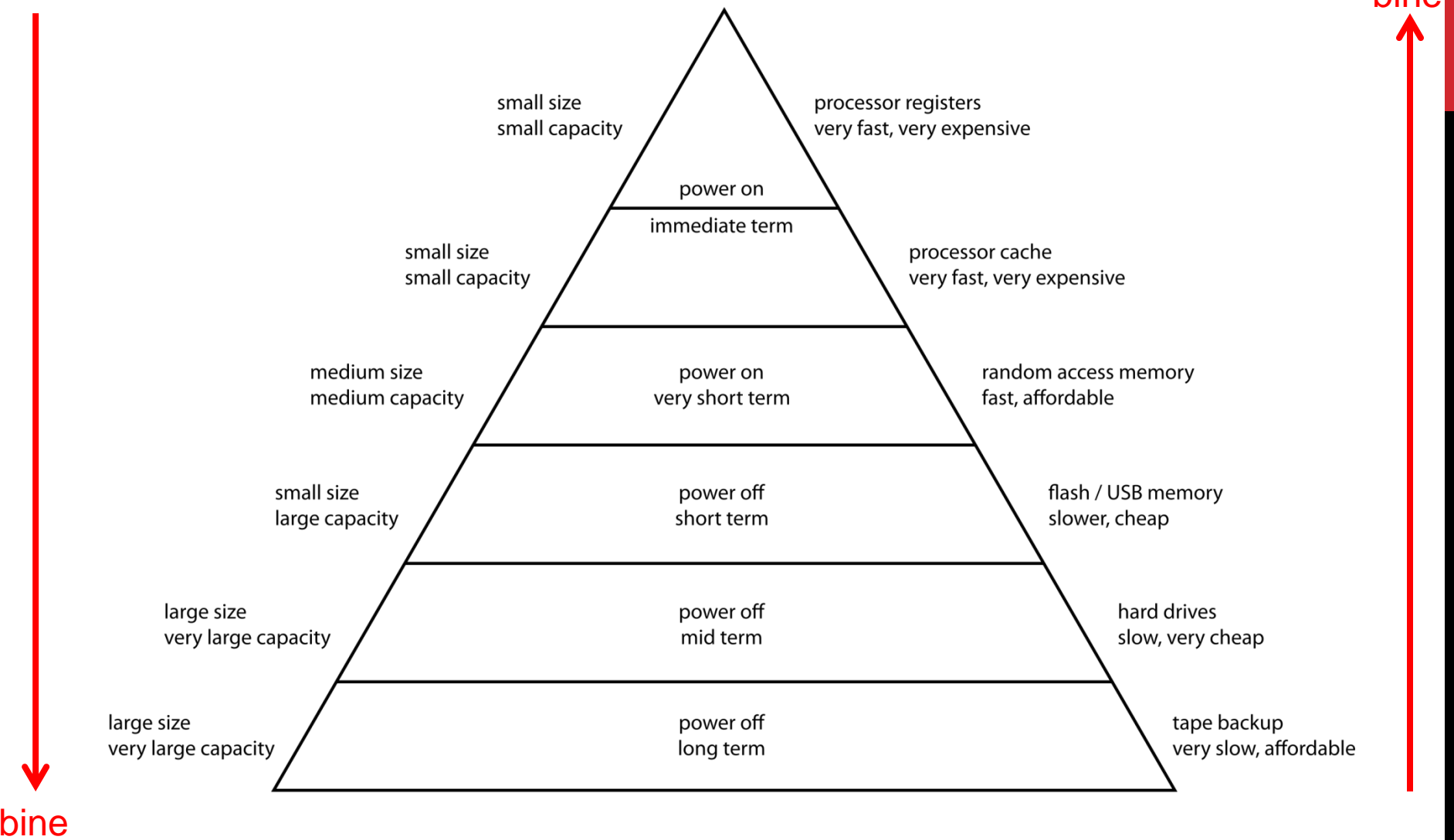
IERARHIZAREA MEMORIEI

- **tipuri de memorie**

- regiștrii procesorului: acces imediat, 100-1000 bytes
- cache L0: acces foarte rapid, 5-20 kbytes
- cache L1 (cache instrucțiuni și date): 700GB/s, 100-500 kbytes
- cache L2: 200GB/s, 500-1000 kbytes
- cache L3 (de obicei partajat): 100GB/s, 1-5 MB
- memoria principală RAM: 100-500 MB/s, 2-64 GB
- disc HD/SSD: 10-100 MB/s, 1TB

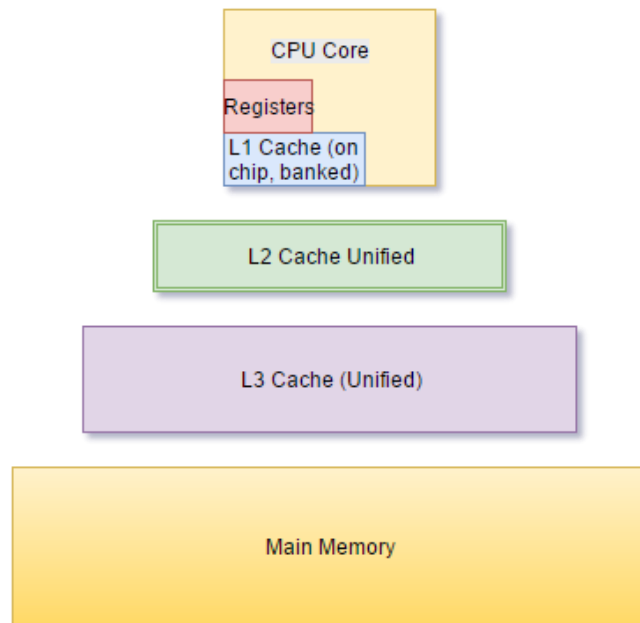


IERARHIZAREA MEMORIEI



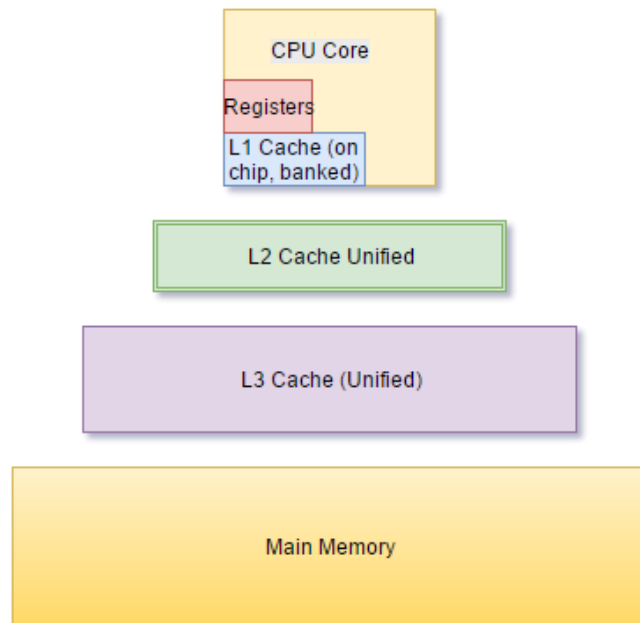
IERARHIZAREA MEMORIEI

- de ce e bine să avem cache?
- presupunem că timpii de acces sunt:
 - în memoria principală: 50 ns
 - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
 - în L2: 5 ns cu 1% miss rate
 - în L3: 10 ns cu 0.2% miss rate
- să presupunem că vrem să accesăm o bucată de memorie, cât ne costă ca timp dacă:
 - verificăm în RAM:



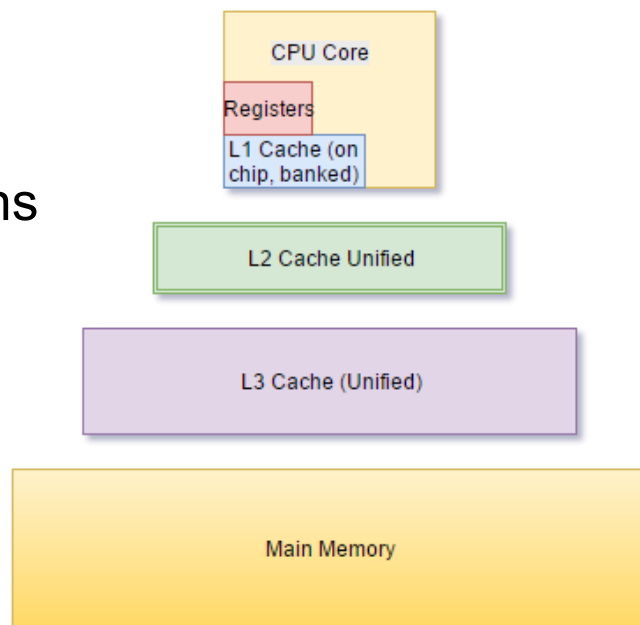
IERARHIZAREA MEMORIEI

- de ce e bine să avem cache?
- presupunem că timpii de acces sunt:
 - în memoria principală: 50 ns
 - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
 - în L2: 5 ns cu 1% miss rate
 - în L3: 10 ns cu 0.2% miss rate
- să presupunem că vrem să accesăm o bucată de memorie, cât ne costă ca timp dacă:
 - verificăm în RAM: 50 ns
 - verificăm în L1:



IERARHIZAREA MEMORIEI

- de ce e bine să avem cache?
- presupunem că timpii de acces sunt:
 - în memoria principală: 50 ns
 - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
 - în L2: 5 ns cu 1% miss rate
 - în L3: 10 ns cu 0.2% miss rate
- să presupunem că vrem să accesăm o bucată de memorie, cât ne costă ca timp dacă:
 - verificăm în RAM: 50 ns
 - verificăm în L1: $1 \text{ ns} + (0.1 \times 50 \text{ ns}) = 6 \text{ ns}$
 - verificăm în L2:



IERARHIZAREA MEMORIEI

- de ce e bine să avem cache?
- presupunem că timpii de acces sunt:
 - în memoria principală: 50 ns
 - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
 - în L2: 5 ns cu 1% miss rate
 - în L3: 10 ns cu 0.2% miss rate
- să presupunem că vrem să accesăm o bucată de memorie, cât ne costă ca timp dacă:
 - verificăm în RAM: 50 ns
 - verificăm în L1: $1 \text{ ns} + (0.1 \times 50 \text{ ns}) = 6 \text{ ns}$
 - verificăm în L2: $1 \text{ ns} + (0.1 \times (5 \text{ ns} + (0.01 \times 50 \text{ ns}))) = 1.55 \text{ ns}$
 - verificăm în L3:

IERARHIZAREA MEMORIEI

- de ce e bine să avem cache?
- presupunem că timpii de acces sunt:
 - în memoria principală: 50 ns
 - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
 - în L2: 5 ns cu 1% miss rate
 - în L3: 10 ns cu 0.2% miss rate
- să presupunem că vrem să accesăm o bucată de memorie, cât ne costă ca timp dacă:
 - verificăm în RAM: 50 ns
 - verificăm în L1: $1 \text{ ns} + (0.1 \times 50 \text{ ns}) = 6 \text{ ns}$
 - verificăm în L2: $1 \text{ ns} + (0.1 \times (5 \text{ ns} + (0.01 \times 50 \text{ ns}))) = 1.55 \text{ ns}$
 - verificăm în L3: $1 \text{ ns} + (0.1 \times (5 \text{ ns} + (0.01 \times (10 \text{ ns} + (0.002 \times 50 \text{ ns}))))) = 1.5101 \text{ ns}$

observați trade-off-ul între viteza de acces și probabilitatea de miss rate

cât este miss rate în RAM?

IERARHIZAREA MEMORIEI

- de ce e bine să avem cache?
- presupunem că timpii de acces sunt:
 - în memoria principală: 50 ns
 - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
 - în L2: 5 ns cu 1% miss rate
 - în L3: 10 ns cu 0.2% miss rate
- să presupunem că vrem să accesăm o bucată de memorie, cât ne costă ca timp dacă:
 - verificăm în RAM: 50 ns
 - verificăm în L1: $1 \text{ ns} + (0.1 \times 50 \text{ ns}) = 6 \text{ ns}$
 - verificăm în L2: $1 \text{ ns} + (0.1 \times (5 \text{ ns} + (0.01 \times 50 \text{ ns}))) = 1.55 \text{ ns}$
 - verificăm în L3: $1 \text{ ns} + (0.1 \times (5 \text{ ns} + (0.01 \times (10 \text{ ns} + (0.002 \times 50 \text{ ns}))))) = 1.5101 \text{ ns}$

observați trade-off-ul între viteza de acces și probabilitatea de miss rate

cât este miss rate în RAM? 0% (în RAM sigur avem informația)
cu ce dimensiune are legătură miss rate?

IERARHIZAREA MEMORIEI

- de ce e bine să avem cache?
- presupunem că timpii de acces sunt:
 - în memoria principală: 50 ns
 - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
 - în L2: 5 ns cu 1% miss rate
 - în L3: 10 ns cu 0.2% miss rate
- să presupunem că vrem să accesăm o bucată de memorie, cât ne costă ca timp dacă:
 - verificăm în RAM: 50 ns
 - verificăm în L1: $1 \text{ ns} + (0.1 \times 50 \text{ ns}) = 6 \text{ ns}$
 - verificăm în L2: $1 \text{ ns} + (0.1 \times (5 \text{ ns} + (0.01 \times 50 \text{ ns}))) = 1.55 \text{ ns}$
 - verificăm în L3: $1 \text{ ns} + (0.1 \times (5 \text{ ns} + (0.01 \times (10 \text{ ns} + (0.002 \times 50 \text{ ns}))))) = 1.5101 \text{ ns}$

observați trade-off-ul între viteza de acces și probabilitatea de miss rate

cât este miss rate în RAM? 0% (în RAM sigur avem informația)
cu ce dimensiune are legătură miss rate? cu dimensiunea memoriei

IERARHIZAREA MEMORIEI

- de ce e bine să avem cache?
- presupunem că timpii de acces sunt:
 - în memoria principală: 50 ns
 - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
 - în L2: 5 ns cu 1% miss rate
 - în L3: 10 ns cu 0.2% miss rate
- să presupunem că vrem să accesăm o bucată de memorie, cât ne costă ca timp dacă:
 - verificăm în RAM: 50 ns
 - verificăm în L1: $1 \text{ ns} + (0.1 \times 50 \text{ ns}) = 6 \text{ ns}$
- cu un miss rate de 10% merită să avem cache L1
- pentru ce probabilitate miss rate nu mai merită cache L1?

IERARHIZAREA MEMORIEI

- de ce e bine să avem cache?
- presupunem că timpii de acces sunt:
 - în memoria principală: 50 ns
 - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
 - în L2: 5 ns cu 1% miss rate
 - în L3: 10 ns cu 0.2% miss rate
- să presupunem că vrem să accesăm o bucată de memorie, cât ne costă ca timp dacă:
 - verificăm în RAM: 50 ns
 - verificăm în L1: $1 \text{ ns} + (0.1 \times 50 \text{ ns}) = 6 \text{ ns}$
 - cu un miss rate de 10% merită să avem cache L1
 - pentru ce probabilitate miss rate nu mai merită cache L1?
 - $p = 49 / 50 = 98\%$

IERARHIZAREA MEMORIEI

- de ce e bine să avem cache?
- presupunem că timpii de acces sunt:
 - în memoria principală: 50 ns
 - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
 - în L2: 5 ns cu 1% miss rate
 - în L3: 10 ns cu 0.2% miss rate
- **Exemplu:**
 - memoria RAM este 1 GB
 - memoria cache L1 este 128 kbytes
 - memoria RAM este de aproximativ 8000 de ori mai multă decât memoria cache L1, deci cum putem avea miss rate 10%?
 - ne bazăm pe **principiul de localizare**

CACHE

- **principiul de localizare**
 - presupunem că avem în RAM un vector de 1000 elemente



- cache L1 este gol



- cache L2 este gol



citim elementul a_0 (avem nevoie să procesăm vectorul): elementul nu e în L1, nu e în L2, trebuie să mergem în RAM, dar după ce îl citim din RAM copiem un segment din vector în L1 și L2

CACHE

- **principiul de localizare**
 - presupunem că avem în RAM un vector de 1000 elemente

a_0	a_1	a_2	a_3	a_4	a_5	a_6	...	a_{998}	a_{999}
-------	-------	-------	-------	-------	-------	-------	-----	-----------	-----------

- cache L1 conține a_0 și următoarele elemente din vector

a_0	a_1	a_2	a_3
-------	-------	-------	-------

- cache L2 conține a_0 și mai multe elemente din vector

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

citim elementul a_1 : este deja în L1 (cache hit!)

CACHE

- **principiul de localizare**
 - presupunem că avem în RAM un vector de 1000 elemente

a_0	a_1	a_2	a_3	a_4	a_5	a_6	...	a_{998}	a_{999}
-------	-------	-------	-------	-------	-------	-------	-----	-----------	-----------

- cache L1 conține a_0 și următoarele elemente din vector

a_0	a_1	a_2	a_3
-------	-------	-------	-------

- cache L2 conține a_0 și mai multe elemente din vector

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

citim elementul a_2 : este deja în L1 (cache hit!)

CACHE

- **principiul de localizare**
 - presupunem că avem în RAM un vector de 1000 elemente

a_0	a_1	a_2	a_3	a_4	a_5	a_6	...	a_{998}	a_{999}
-------	-------	-------	-------	-------	-------	-------	-----	-----------	-----------

- cache L1 conține a_0 și următoarele elemente din vector

a_0	a_1	a_2	a_3
-------	-------	-------	-------

- cache L2 conține a_0 și mai multe elemente din vector

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

citim elementul a_3 : este deja în L1 (cache hit!)

CACHE

- **principiul de localizare**
 - presupunem că avem în RAM un vector de 1000 elemente

a_0	a_1	a_2	a_3	a_4	a_5	a_6	...	a_{998}	a_{999}
-------	-------	-------	-------	-------	-------	-------	-----	-----------	-----------

- cache L1 conține a_0 și următoarele elemente din vector

a_0	a_1	a_2	a_3
-------	-------	-------	-------

- cache L2 conține a_0 și mai multe elemente din vector

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

citim elementul a_4 : nu este în L1 (cache miss!), dar este în L2, deci îl citim de acolo și actualizăm în L1

CACHE

- **principiul de localizare**

- presupunem că avem în RAM un vector de 1000 elemente

a_0	a_1	a_2	a_3	a_4	a_5	a_6	...	a_{998}	a_{999}
-------	-------	-------	-------	-------	-------	-------	-----	-----------	-----------

- cache L1 conține a_4 și următoarele elemente din vector

a_4	a_5	a_6	a_7
-------	-------	-------	-------

- cache L2 conține a_0 și mai multe elemente din vector

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

citim elementul a_5 : este deja în L1 (cache hit!)

... când nu mai găsim nici în L2, mergem din nou în RAM și citim un nou subset din vector ($a_{10} \dots a_{19}$)

CACHE

- **principiul de localizare**
 - presupunem că avem în RAM un vector de 1000 elemente

a_0	a_1	a_2	a_3	a_4	a_5	a_6	...	a_{998}	a_{999}
-------	-------	-------	-------	-------	-------	-------	-----	-----------	-----------

- cache L1 conține a_4 și următoarele elemente din vector

a_4	a_5	a_6	a_7
-------	-------	-------	-------

- cache L2 conține a_0 și mai multe elemente din vector

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

miss rate pentru cache L1?

miss rate pentru cache L2?

CACHE

- **principiul de localizare**
 - presupunem că avem în RAM un vector de 1000 elemente

a_0	a_1	a_2	a_3	a_4	a_5	a_6	...	a_{998}	a_{999}
-------	-------	-------	-------	-------	-------	-------	-----	-----------	-----------

- cache L1 conține a_4 și următoarele elemente din vector

a_4	a_5	a_6	a_7
-------	-------	-------	-------

- cache L2 conține a_0 și mai multe elemente din vector

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

miss rate pentru cache L1? **25%**

miss rate pentru cache L2? **10%**

CACHE

- **principiul de localizare**
 - presupunem că avem în RAM un vector de 1000 elemente

a_0	a_1	a_2	a_3	a_4	a_5	a_6	...	a_{998}	a_{999}
-------	-------	-------	-------	-------	-------	-------	-----	-----------	-----------

- cache L1 conține a_4 și următoarele elemente din vector

a_4	a_5	a_6	a_7
-------	-------	-------	-------

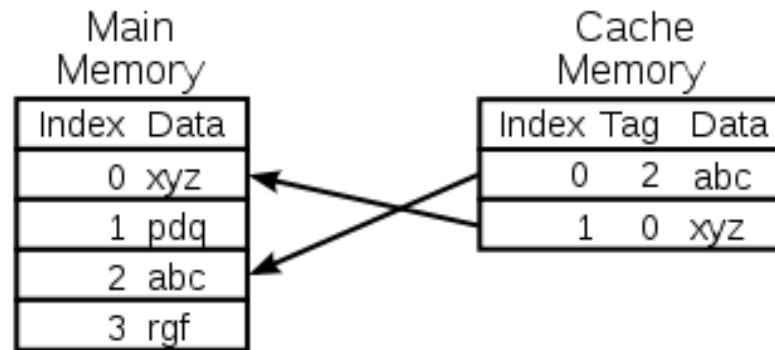
- cache L2 conține a_0 și mai multe elemente din vector

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------

aici am accesat secvențial vectorul, dacă îl accesăm aleator, totul este pierdut (deci chiar dacă memoria se numește RAM, nu e deloc bine să accesăm datele complet Random)

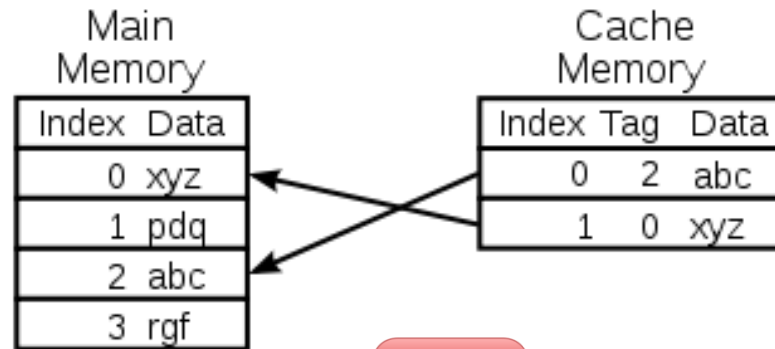
CACHE

- **corespondența dintre locația din cache și locația din RAM**
 - este realizată printr-un tabel

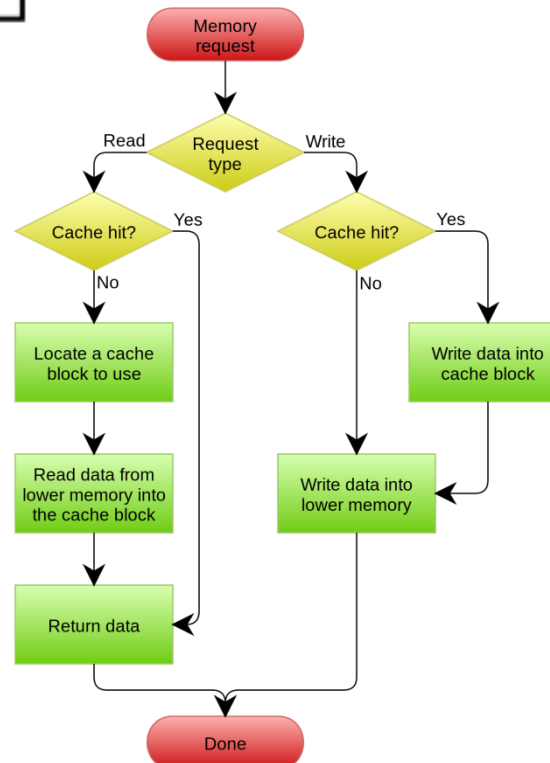


CACHE

- corespondența dintre locația din cache și locația din RAM
 - este realizată printr-un tabel

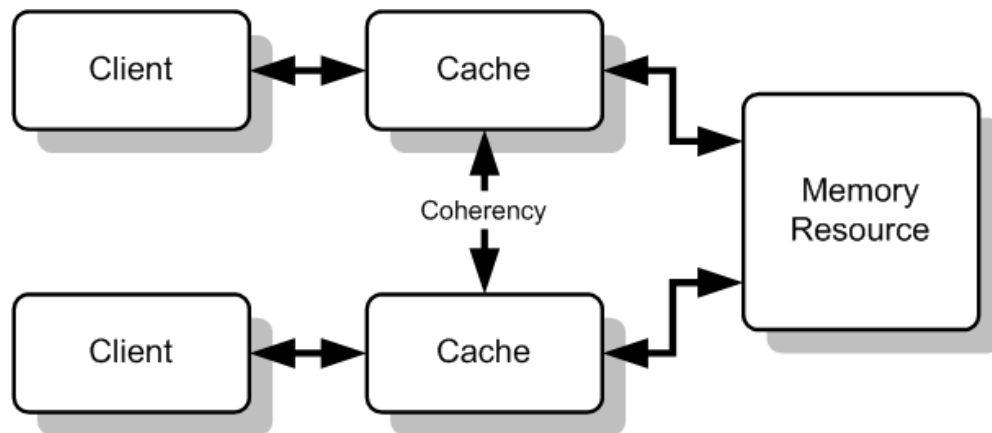


- algoritmul general:



CACHE

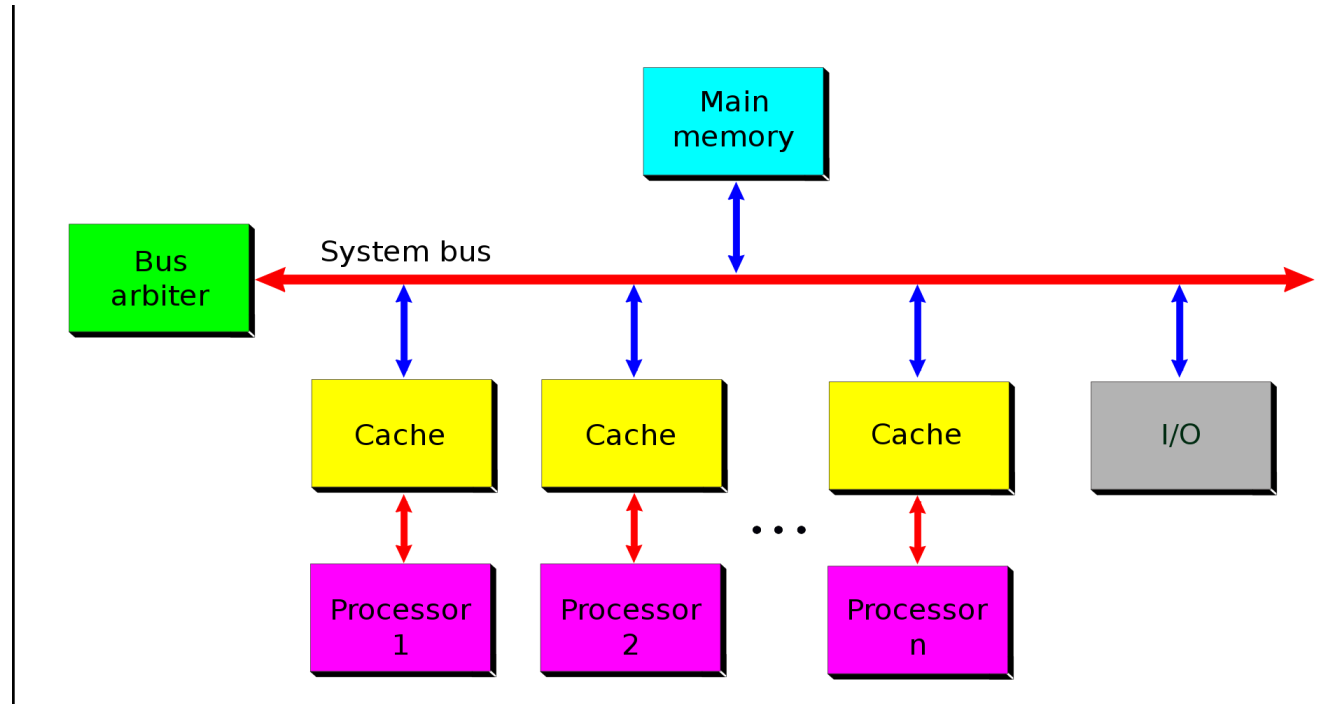
- **corespondența dintre locația din cache și locația din RAM**
 - la citire nu sunt niciodată probleme
 - la scriere lucrurile se complică
 - rezultatul este scris în cache
 - și celelalte memorii trebuie să fie anunțate de noua valoare
 - L1/L2/L3/RAM etc.
 - situația se complică și mai mult dacă sunt cache-uri diferite pentru fiecare core pe care îl avem: cache coherence (protocol pentru consistența tuturor cache-urilor)



IERARHIZAREA MEMORIEI

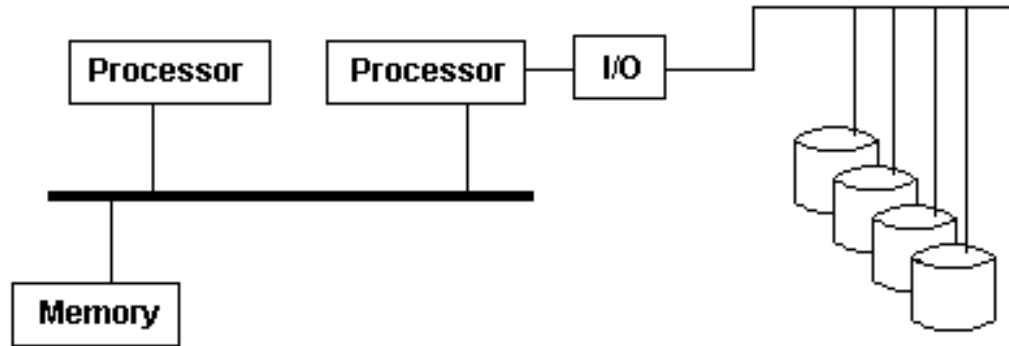
- **când programați, nu vedeți această ierarhizare**
- **cine e responsabil de ce anume?**
 - programatorul: transfer între HD/SSD și RAM (citire de pe disc)
 - logică hardware: din/în RAM în/din memoriile cache
 - compilatorul: generează cod care exploatează cache-ul
- **cât timp performanța este acceptabilă totul e OK, apoi Assembly**

SYMMETRIC MULTIPROCESS SYSTEMS



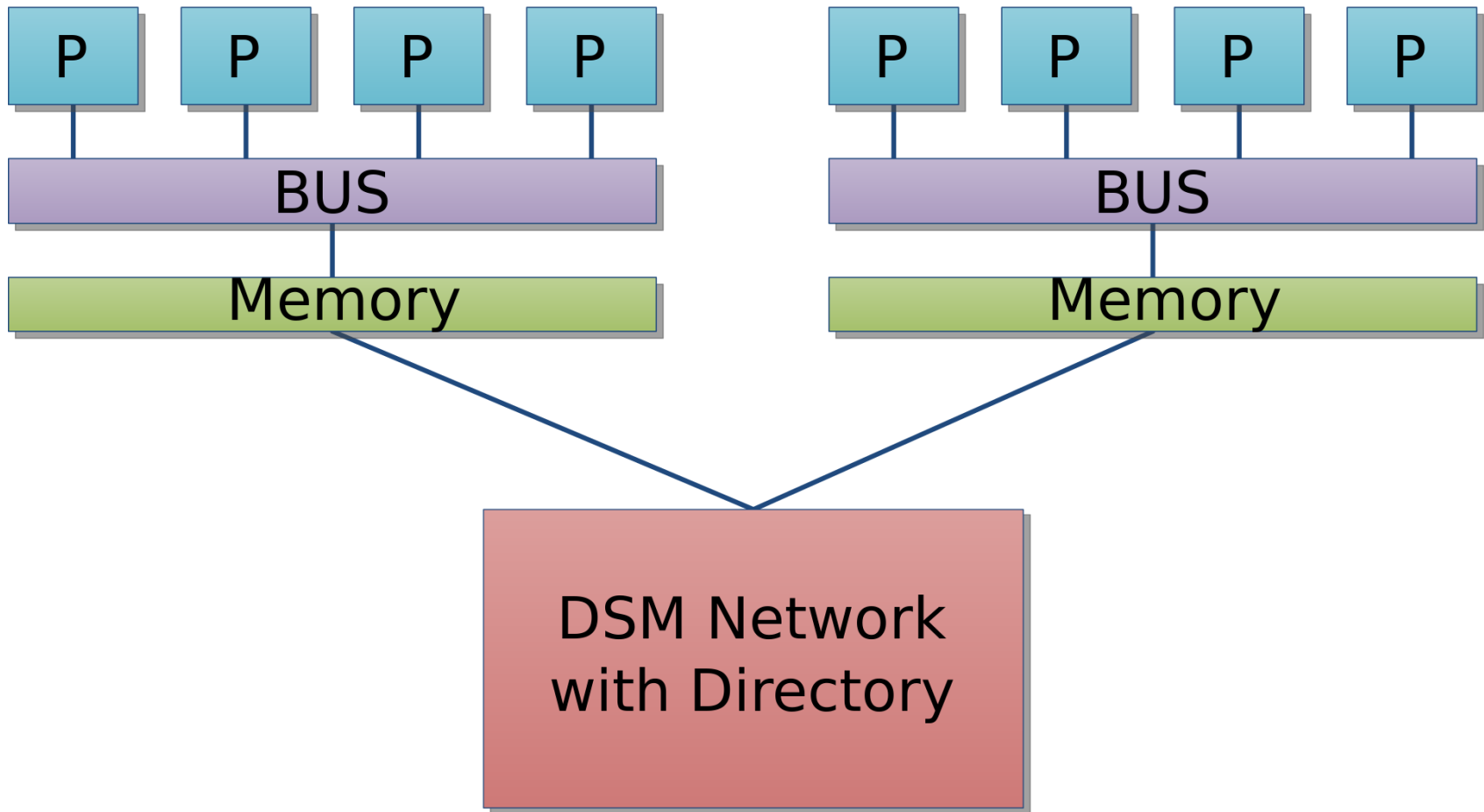
- **UMA (Uniform Memory Access)**
- procese diferite pe procesoare diferite, dar e nevoie de modificarea programelor ca acestea să ruleze paralel
- dezavantaj: cache coherence

ASYMMETRIC MULTIPROCESS SYSTEMS



- **fiecare procesor are se ocupă de ceva diferit**
 - unul execute programe
 - altul se ocupă de I/O

NON-UNIFORM MEMORY ACCESS



- rezolvă problema accesului la memorie de către procesoare multiple (fiecare procesor are memoria/cache-ul său)
- procesoarele așteaptă mai puțin să ajunge datele la ele

CE AM FĂCUT ASTĂZI

- **structura multi-core a calculatoarelor**
- **ierarhizarea memoriei**
- **beneficile cache-ului**

DATA VIITOARE ...

- **ne apropiem de final**
 - Cursul 0x0B va fi în prima săptămână de școală din ianuarie
 - Cursul 0x0C va fi în a doua săptămână de școală din ianuarie
 - nu facem curs, avem verificarea
 - am să vă postez detalii cu privire la testul de verificare
 - online pe moodle, tot anul
 - în ianuarie, nu se mai fac laboratoare/seminare noi
 - la laborator fiecare grupă are verificarea de laborator
 - la seminar facem recuperări, vă răspund la întrebări etc.
- **Cursul 0x0B**
 - performanța calculatoarelor
 - un demo interesant (sper)

LECTURĂ SUPLIMENTARĂ

- **PH book**
 - 5 Large and Fast: Exploiting Memory Hierarchy
 - 6.6 Introduction to Graphics Processing Units
- Erik Demaine, Cache-Oblivious Algorithms: Medians & Matrices,
<https://www.youtube.com/watch?v=CSqbjfCCLrU>

