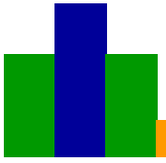
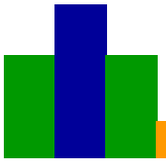


Lecţia 12:
Orgnizarea memoriei - II

G Stefănescu — Universitatea Bucureşti

Arhitectura sistemelor de calcul, Sem.1
Octombrie 2016—Februarie 2017

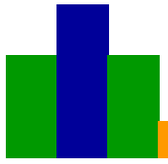
După: D. Patterson and J. Hennessy, Computer Organisation and Design



Orgnizarea memoriei

Cuprins:

- *Generalitati*
- Memoria cache
- Performanta memoriei cache
- Memoria virtuala
- Concluzii, diverse, etc.



Generalitati

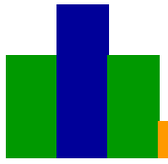
Organizarea memoriei pe nivele: Următorul *exemplu* arată utilitatea organizării *memoriei pe nivele*:

- Presupunem că elaborăm o lucrare accesând diverse cărţi organizate în două moduri:

  Organizare plată: *Toate* cărţile sunt *în bibliotecă* şi aducem *câte una* pe masă;

  Organizare pe nivele: Tinem *unele* cărţi pe *masă*, *altele* în *bibliotecă*; din când în când ducem cărţi de pe masă în bibliotacă şi aducem altele.

- Varianta a doua este clar mai bună (dacă cărţile de pe masă sunt relevante pentru subiectul cercetat).



..Generalitati

Organizarea memoriei pe nivele (cont.)

- In decizia relativ la *ce cărţi ţinem pe masă* este util un *principiu de localizare*:

  Localizare temporală: Tinem pe masă cărţi folosite în ultima vreme;

  Localizare spaţială: Tinem pe masă cărţi care sunt în bibliotecă în locuri apropiate. [Presupunând că biblioteca este aranjată pe domenii, cărţile apropiate sunt din acelaşi domeniu ori domenii înrudite.]

- Similar, memoria calculatorului se poate organiza *pe nivele (ierarhic)*. Memoriile apropiate de procesor sunt *memorii cache.*

..Generalitati

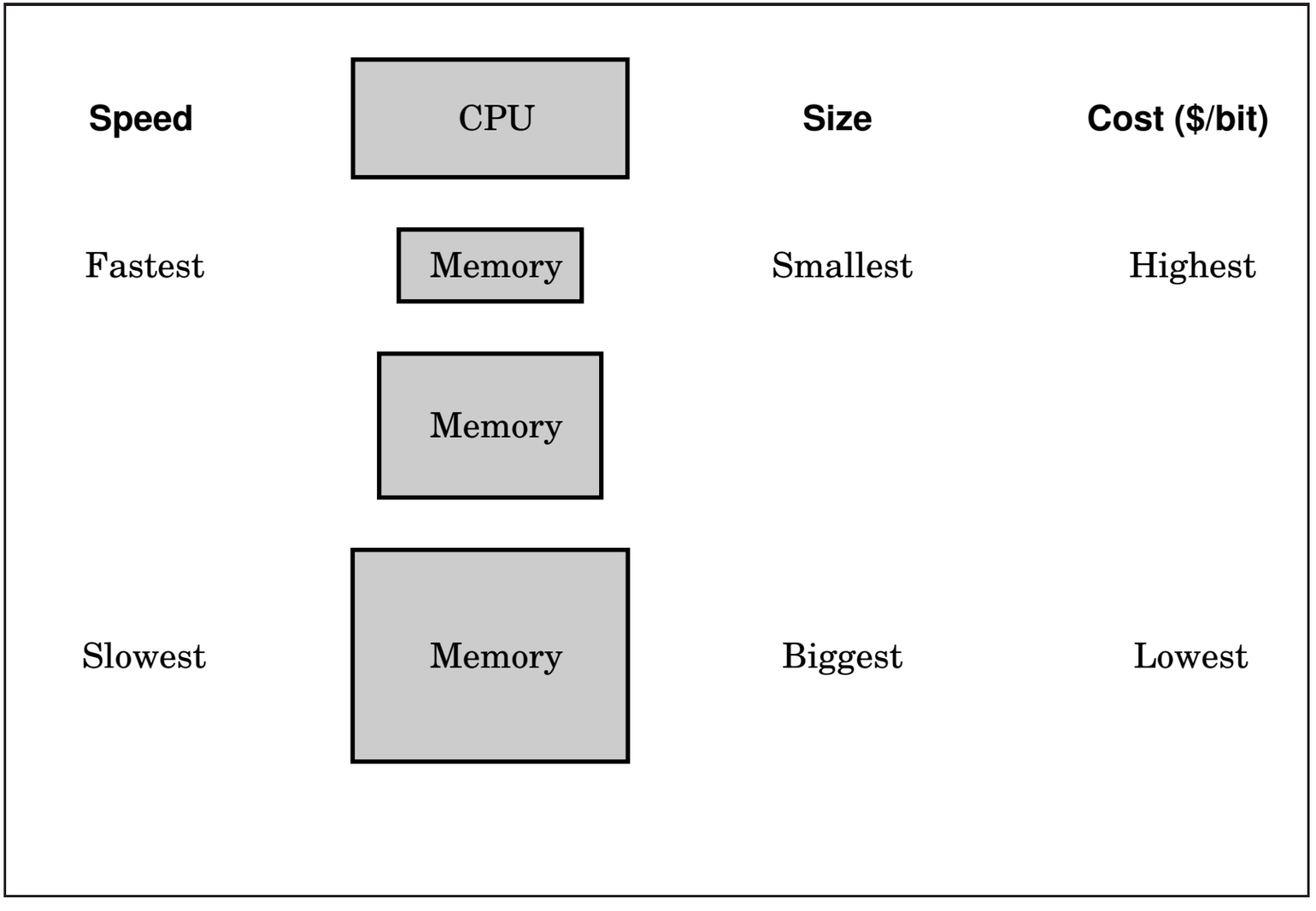
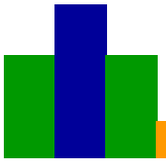


Figura ilustrează *ierarhia de memorii*, producând *iluzia* unei memorii *mari şi rapide*.



..Generalitati

Organizarea memoriei pe nivele (cont.)

- Memoriile mai *apropiate de procesor* sunt *rapide, dar mici şi scumpe*; cele mai *depărtate*, sunt *mari, ieftine, dar lente*.

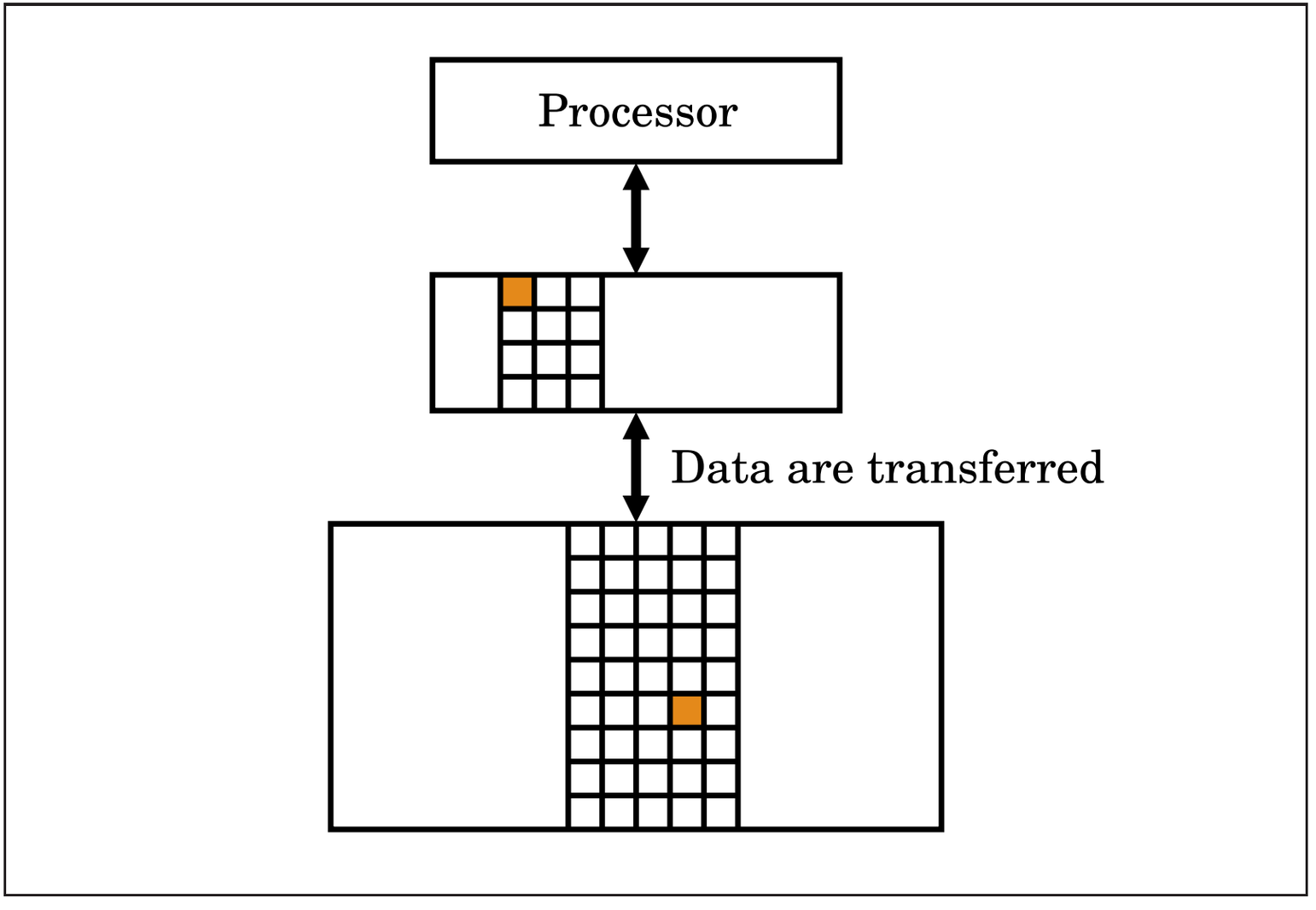
Tehnologie	Timp de acces	Pret pe MB (1997)
SRAM	5-25ns	100-250 USD
DRAM	60-120ns	5-10 USD
disc magnetic	10-20$\times 10^6$ns	0.10-0.20 USD

- Un pic diferit de exemplul cu cărţi, memoriile formează o ierarhie în care *nivelele se conţin unele pe altele*

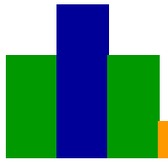
$$nivel\ 1 \subset nivel\ 2 \subset nivel\ 3 \ldots$$

- Transferul de informaţie de pe un nivel pe altul se face pentru un grup mai mare de date, care formează un *bloc*.

..Generalitati



Transferul de date de la un nivel la altul se face folosind *blocuri* de memorie.



..Generalitati

Organizarea memoriei pe nivele (cont.)

- Definim următoarele noţiuni:

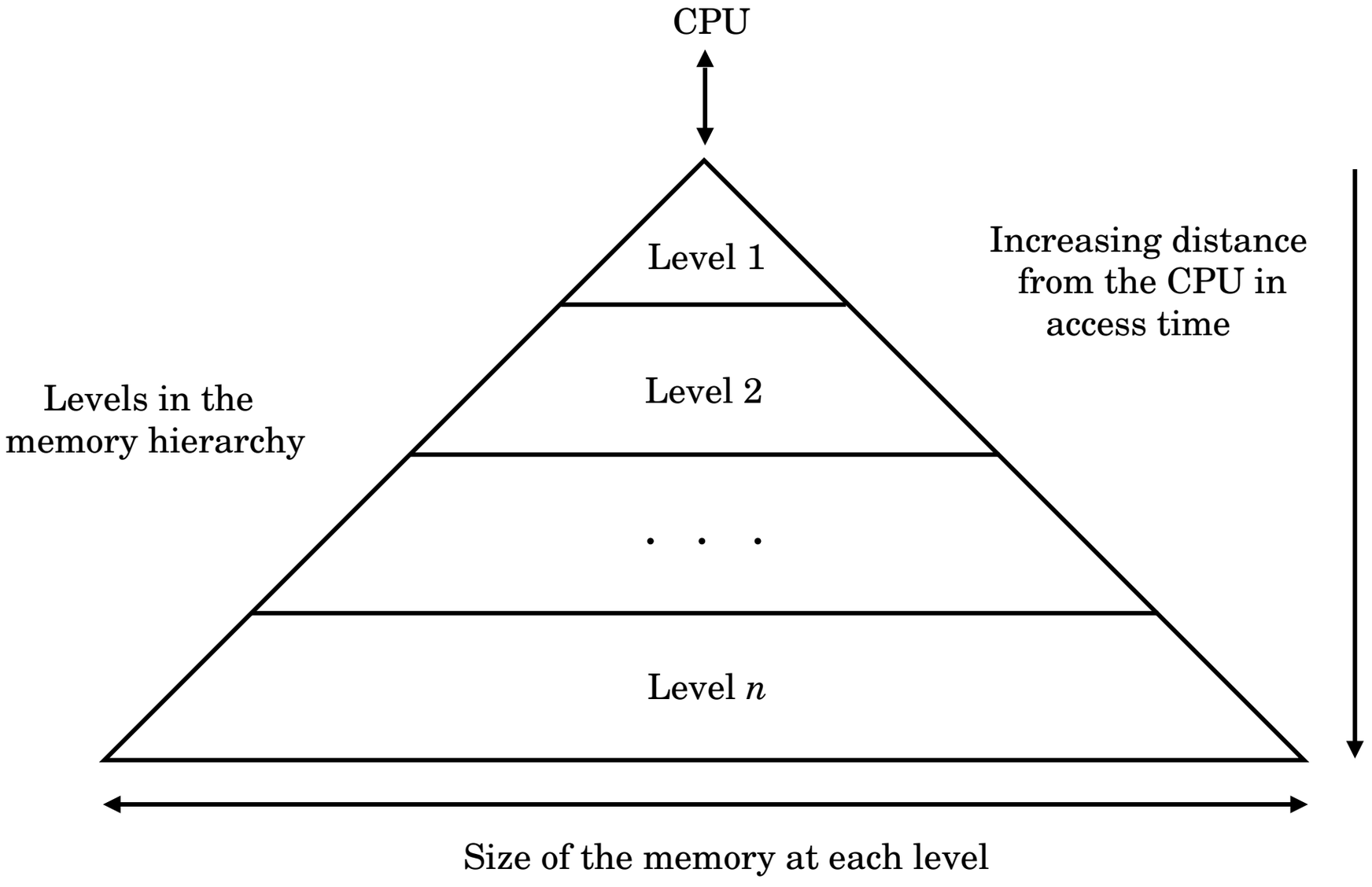
  rata de succes (hit rate) = câte din accesările de memorie sunt rezolvate de nivelul curent;

  rata de eşec (miss rate) = câte din accesările de memorie necesită access la nivelul următor (mai lent);

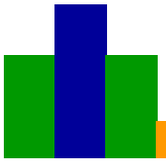
  timp de succes (hit time) = timpul unei accesări cu succes (pe nivelul curent);

  penalizare la eşec (miss penalty) = la eşec, timpul necesar transferării unui *bloc* de la nivelul următor, conţinând data cerută;

..Generalitati



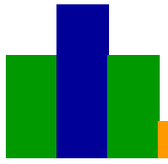
Ierarhia de memorii: *Crescând distanţa* de precesor, *creşte timpul* de acces.



Orgnizarea memoriei

Cuprins:

- Generalitati
- *Memoria cache*
- Performanta memoriei cache
- Memoria virtuala
- Concluzii, diverse, etc.



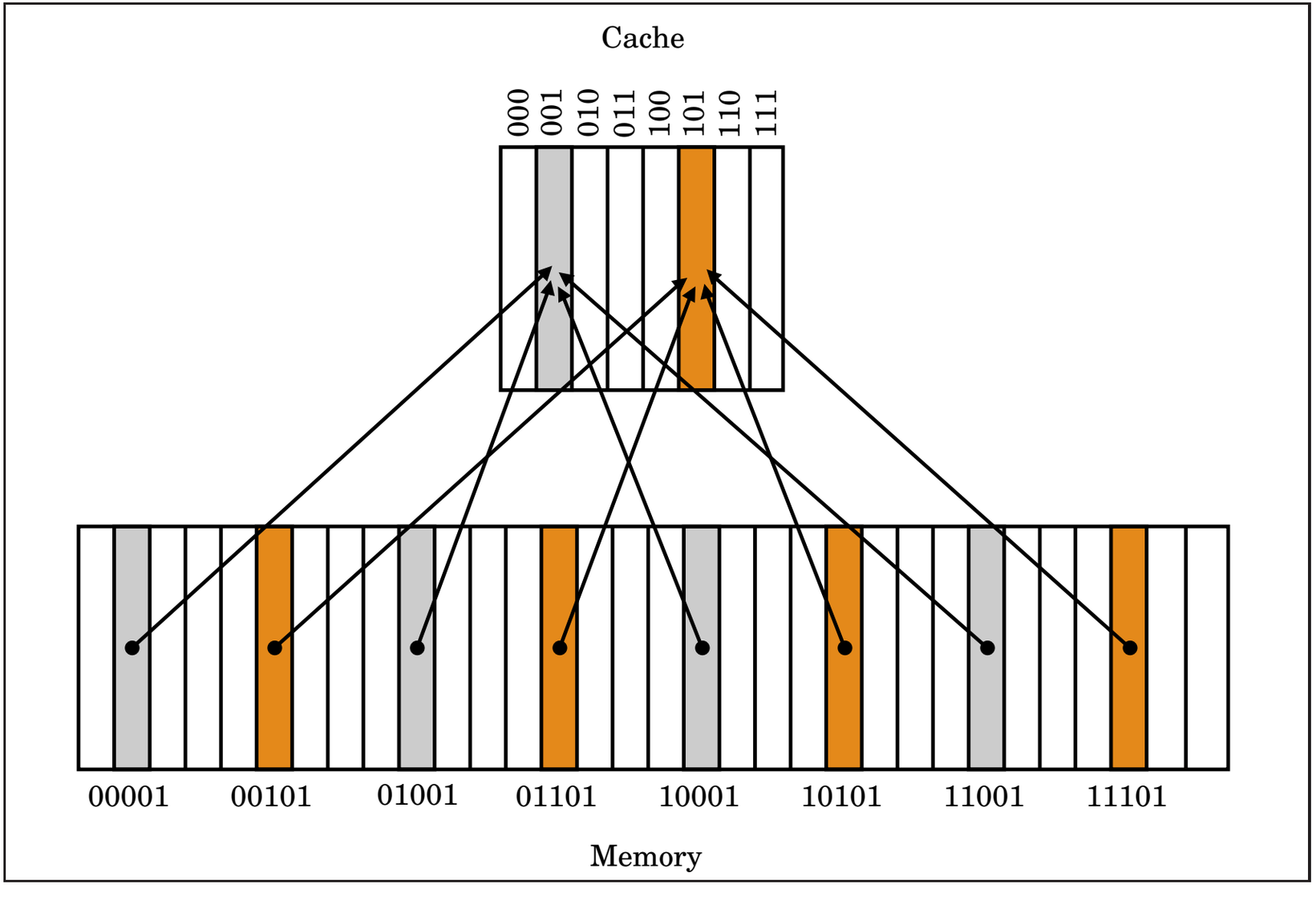
Memoria cache

Memorii cache:

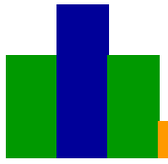
- In sens un pic extins, *memoria cache* este:
  - un *nivel de memorie* între CPU şi memoria principală; ori
  - un *tip de organizare* bazat pe principiul de localizare.
- Incepem cu un caz simplificat, anume procesorul accesează cuvinte (4B), iar blocurile din cache sunt tot de un cuvânt (4B).
- O schemă simplă de organizare a unui cache C de dimensiune N pentru o memorie M este folosind *adresarea directă*, anume:

blocul k de memorie din M
$\mapsto$ *blocul (k modulo N) din C*

..Memoria cache



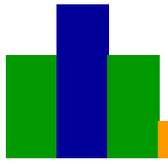
Un cache cu *acces direct* folosind *resturile modulo dimensiunea cache-ului*.



..Memoria cache

Memorii cache (cont.)

- Funcţia de mai sus este surjectivă, dar nu injectivă. Cum ştim ce element din M este memorat curent în C?
- Adăugăm la datele din cache *informaţii auxiliare* (tag-uri), anume *câtul q împărţirii lui k la N*; atunci, $k = q * N + (k \bmod N)$.
- Când dimensiunea cache-ului este $N = 2^n$,
  - adresa lui k în C este dată de ultimii n biţi din k;
  - tag-ul q este dat de biţii din faţă care au rămas.
- Uneori trebie ştiut dacă informaţia de la o adresă din cache este validă ori nu - pentru asta folosim, în plus, un *bit de validitate*.



..Memoria cache

Accesarea unui cache:

- Un exemplu de accesare folosind un cache de 8 blocuri pentru o memorie de 32 blocuri, deci *adresă de accesare are 3 biţi, iar tag-ul 2 biţi*.
- Să presupunem că se accesează datele din M de la adresele (zecimale):

  `22, 26, 22, 26, 16, 3, 16, 18.`

- Plecând cu un cache vid, avem:

  `miss, miss, hit, hit, miss, miss, hit, miss.`

- Evoluţia cache-ului este descrisă în următoarele tabele, care ilustrează doar cazurile de eşec (miss):

..Memoria cache

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10	M[10110]
111	N		

Index	V	Tag	Data
000	N		
001	N		
010	Y	11	M[11010]
011	N		
100	N		
101	N		
110	Y	10	M[10110]
111	N		

Index	V	Tag	Data
000	Y	10	M[10000]
001	N		
010	Y	11	M[11010]
011	N		
100	N		
101	N		
110	Y	10	M[10110]
111	N		

Index	V	Tag	Data
000	Y	10	M[10000]
001	N		
010	Y	11	M[11010]
011	Y	00	M[00011]
100	N		
101	N		
110	Y	10	M[10110]
111	N		

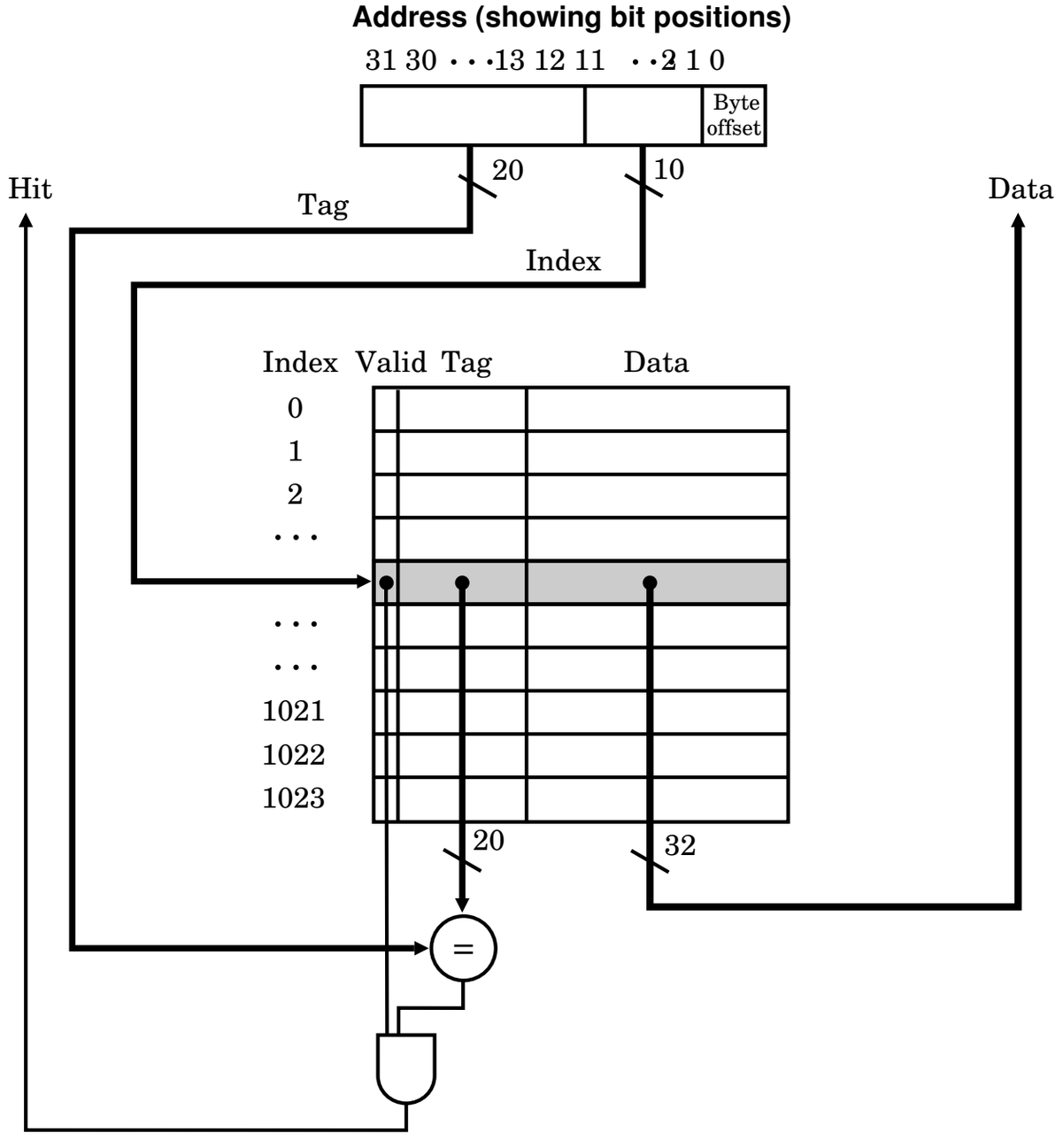
Index	V	Tag	Data
000	Y	10	M[10000]
001	N		
010	Y	10	M[10010]
011	Y	00	M[00011]
100	N		
101	N		
110	Y	10	M[10110]
111	N		

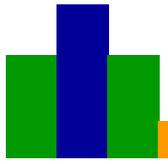
..Memoria cache

Accesarea unui cache:

Un exemplu de control:

- Cache-ul este de 2^{10} cuvinte (2 biţi sunt pentru offsetul de la octeţi la cuvinte);
- Tag-ul foloseşte restul de 20 biţi.
- Cache-ul are încă un bit, pentru validitatea datei în cache.





..Memoria cache

Marimea unui cache:

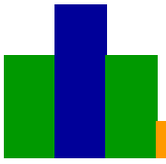
Problemă: Câţi biţi sunt necesari pentru un cache cu acces direct cu 64KB de date şi blocuri de 1 cuvânt, presupunând că folosim adrese de 32b.

Răspuns:

- $64\texttt{KB} = 16\texttt{Kcuvinte} = 2^{14}\texttt{cuvinte}$, deci sunt 2^{14} blocuri; tag-urile au $32-14-2 = 16\texttt{b}$; în plus, avem 1b de validitate; deci, spaţiul suplimentar pentru tag + bit este 17b.
- Memoria cache totală este

$$2^{14} * (32+17)\texttt{b} = 2^{14} * 49\texttt{b} = 784 * 2^{10}\texttt{b} = 784\texttt{Kb}$$

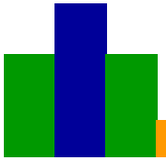
- Numărul total de biţi este cu circa 50% peste cel pentru date.



..Memoria cache

Tratarea esecurilor:

- Dacă în procesor la încărcarea unei date ori instrucţiuni apare un *eşec* trebuie căutată informaţia în *nivelul următor* de memorie, semnificativ mai *lent*.
- Soluţia este de a *îngheţa* activitatea CPU până soseşte data ori instrucţiunea necesară, apoi continuăm execuţia.
- In rezolvarea problemei, pe lângă controlul din procesor se foloseşte un *control separat* pentru actualizarea memoriei cache.
- Ingheţarea activităţii procesorului se face ca la versiunea pipeline, dar mult mai simplu: *conservăm conţinutul regiştrilor* (uzuali ori pipeline).

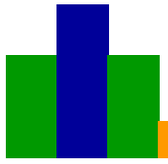


..Memoria cache

Tratarea esecurilor (cont.)

Detalii pentru tratarea eşecul încărcării unei instrucţiuni:

- Se trimite PC-ul original (deci PC–4) la memoria principală;
- Se trimite un semnal de citire din memorie şi se aşteaptă rezultatul;
- Se scrie în memoria cache, punând rezultatul în zona de date şi completând tag-ul cu biţii din faţa ai adresei PC (din ALU) şi setând bitul de validitate;
- Se restartează execuţia, reâncărcând instrucţiunea din memoria cache.

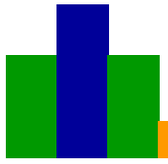


..Memoria cache

Tratarea esecurilor (cont.) Cum reducem penalizarea datorată eşecului? O soluţie este *stall on use*:

- Se aşteaptă *activ*, procesorul încercând să execute alte intrucţiuni.
- Strategia este *bună* pentru *eşecul la date*, dar *nu la instrucţiuni*, căci acestea depind unele de altele.
- *Eficienţa* (numărul de cicluri salvate) depinde de interdependenţa datelor, de numărul de *instrucţiuni independente* cu care se poate alimenta procesorul aşteptând sosirea datei cerute.

Nota: Există si alte strategii, unele discutate ulterior: mai multe nivele de memorii cache, o aranjare mai flexibilă a blocurilor in memorie, etc.



..Memoria cache

Studiu de caz: DECStation 3100:

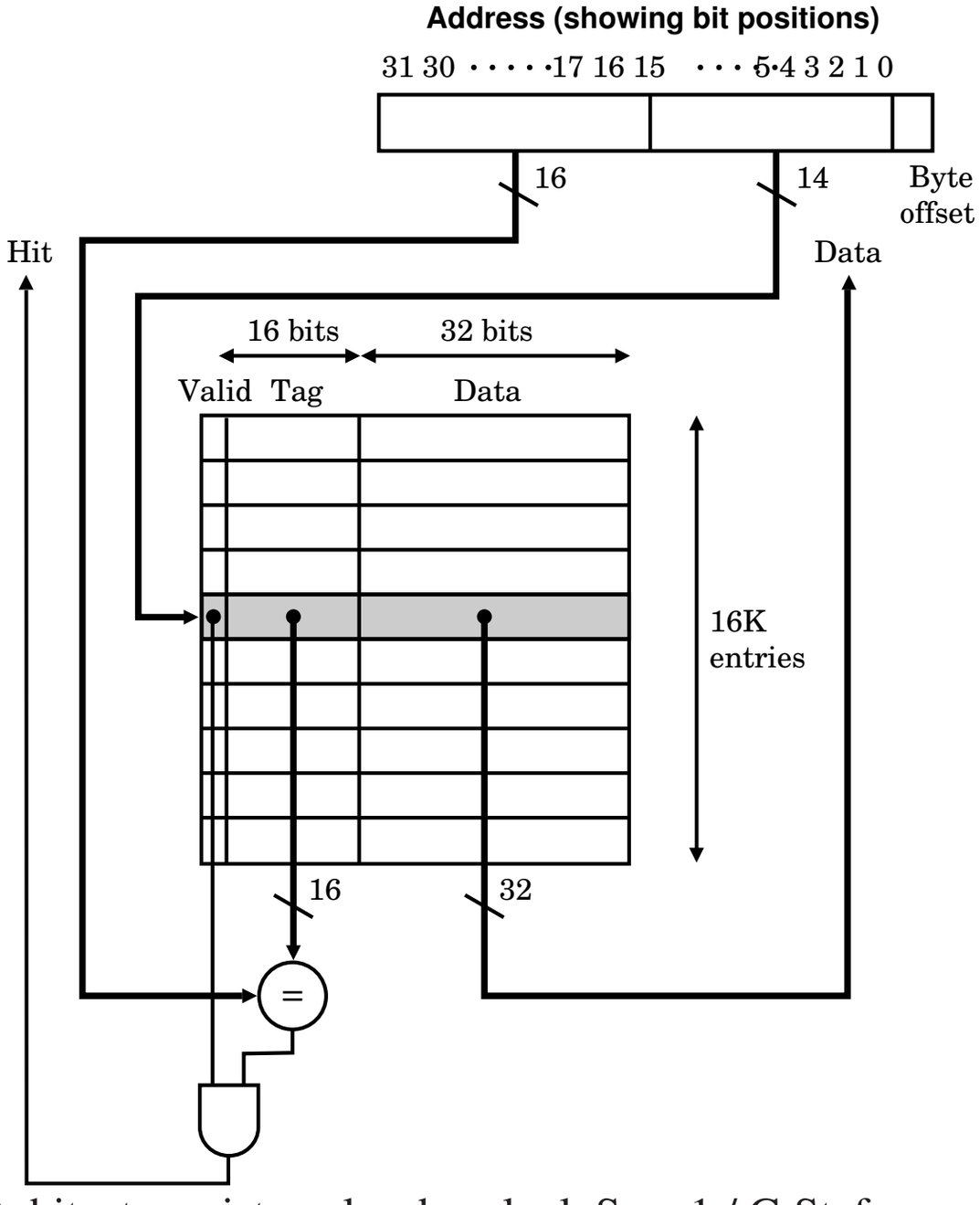
- Este procesor de tip pipeline; în activitate intensă accesează *o înstrucţiune* şi *o dată pe ciclu* pipeline.
- Are *2 memorii cache* de 64KB (una pentru instrucţiuni, una pentru date), cu blocuri de 1 cuvânt.
- Accesarea memoriei pentru *citire* se produce astfel:
  - Trimitem adresa la cache (din PC ori ALU);
  - Dacă cache-ul returnează succes (hit), informaţia este pe liniile de intrare;
  - Dacă cache-ul returnează eşec (miss), se trimite adresa la memoria principală, se scrie în cache, şi se continuă ca mai sus.

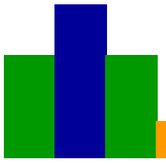
..Memoria cache

Studiu de caz: DECStation 3100:

Cache-ul la DECStation 3100:

- Conţine 16K blocuri de 1 cuvânt; deci, indexul are 14b;
- Tag-ul foloseşte restul de 32–(14+2)=16 biţi.



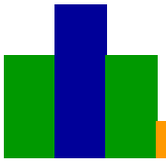


..Memoria cache

Studiu de caz: DECStation 3100:

- Accesarea memoriei pentru *scriere* se produce astfel:
  - Scriem data în cache.
  - Cum memoria principală poate avea o dată diferită, cache-ul şi memoria pot fi *inconsistente*.
  - Consistenţa la DECStation 3100 se menţine prin *scriere şi în cache şi în memorie*.
- Tehnica de consistenţă de mai sus se numeşte *scriere simultană*.
- Exemple de rate de eşec la DECStation 3100 (la citire)

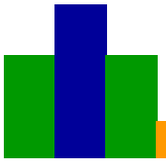
Program	Esec la instructiuni	Esec la date	Esec combinat
`gcc`	6.1%	2.1%	5.4%
`spice`	1.2%	1.3%	1.2%



..Memoria cache

Memorii tampon, consistenta:

- O altă tehnică de consistenţă este de *scriere la loc (write-back)*:
  - datele scrise se scriu curent doar în cache;
  - ele se scriu în memoria principală *numai când* blocul din cache trebuie înlocuit cu altul din memorie.
- Eşecurile *de scriere* pot produce penalizări de multe cicluri (procesorul aşteaptă până se scrie în memoria principală). Intârzierea se poate diminua folosind *memorii buffer (tampon)*.
- La DECStation 3100 mărimea buffer-ul de scriere este de 4 cuvinte.



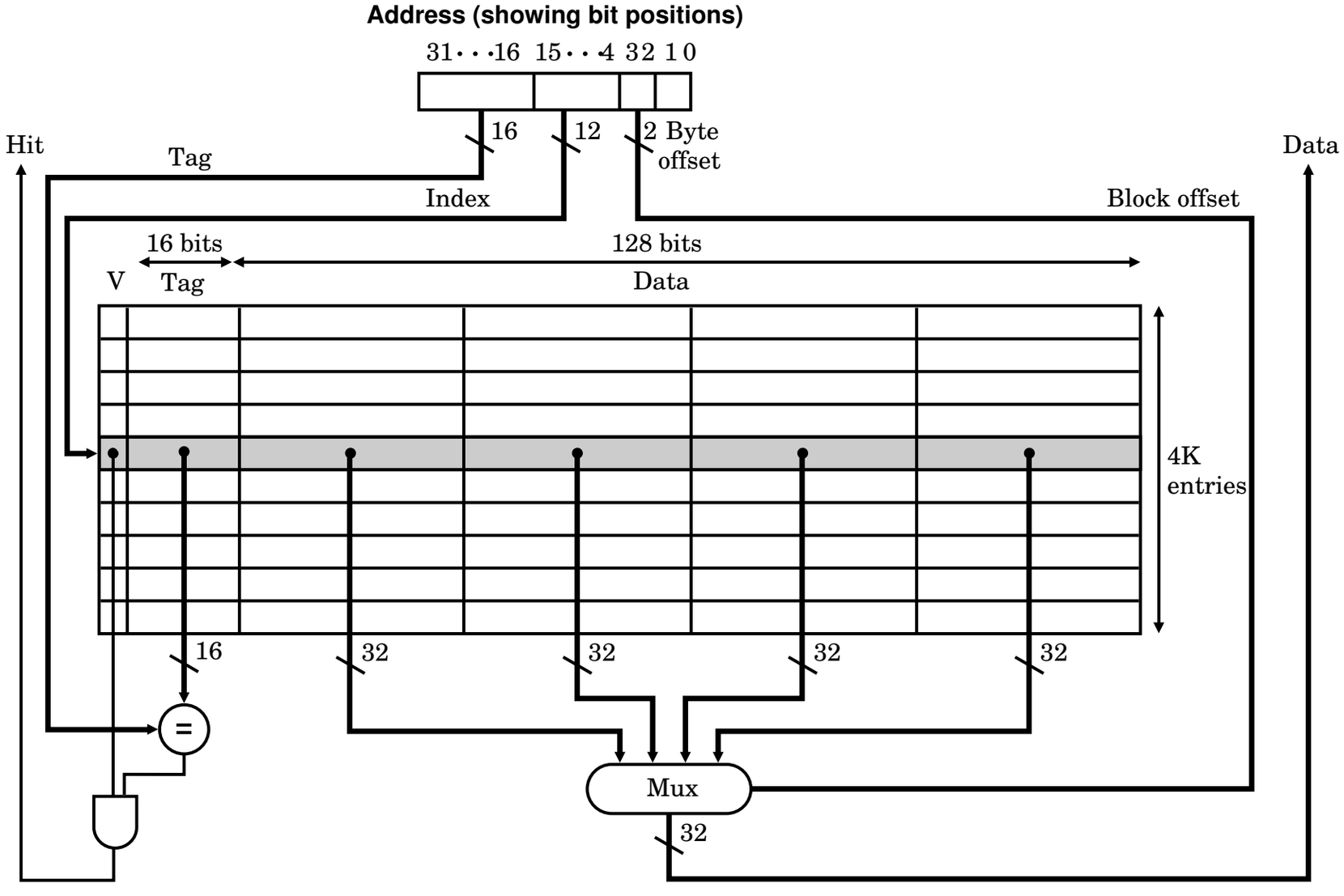
..Memoria cache

Folosirea localizarii spatiale:

- Transferul de informaţii între cache şi memorie se face în grupuri mai mari, numite *blocuri*.
- Putem folosi acelaşi sitem de adresare, dar acum relativ la blocuri mai mari de cuvânt; după localizarea blocului, se identifică poziţia datei.
- *Intrebare:* Unde este data de la adresa 1208B într-un cache de 64 blocuri cu 16B fiecare?

  Răspuns: $1208 = 75 \times 16 + 8$; blocul din cache este $75 \bmod 64 = 11$; elementul din bloc este cel cu index $8/4 = 2$ (al 3-lea cuvînt).

..Generalitati



Un cache de 64KB care utilizează *blocuri de 4 cuvinte* (16B).

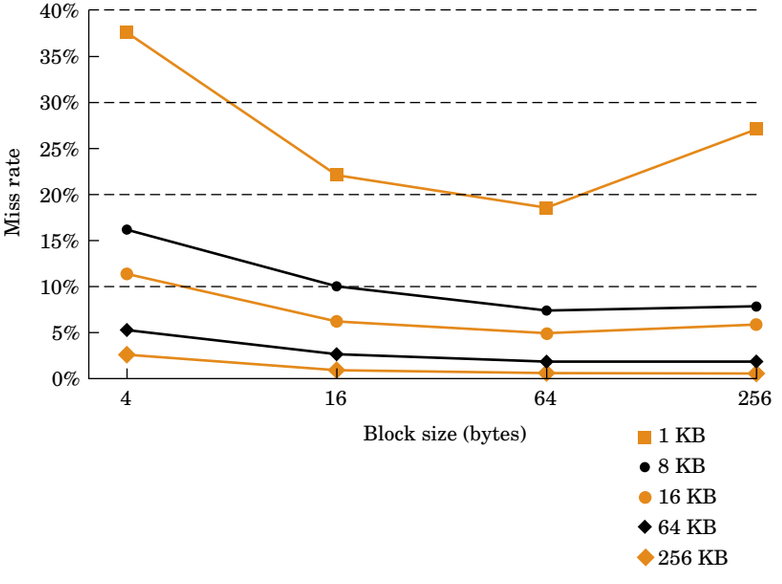
..Memoria cache

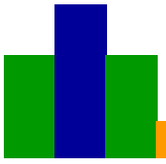
Folosirea localizarii spatiale (cont.)

- Exemple de rate de eşec la DECStation 3100 cu blocuri de lungimi diferite

Program	Marime bloc	Esec la instructiuni	Esec la date	Esec combinat
`gcc`	1	6.1%	2.1%	5.4%
	4	2.0%	1.7%	1.9%
`spice`	1	1.2%	1.3%	1.2%
	4	0.3%	0.6%	0.4%

- După cum se observă, în general, mărirea dimensiunii blocului *scade rata de eşec*. (In contrapondere, creşterea dimensiunii blocului creşte penalizarea la eşec.)





..Memoria cache

Organizare a memoriei cu cache-uri: Să analizăm un caz ipotetic, cu următorii timpi de acces pentru un eşec:

- 1 ciclu de ceas - trimis adresa
- 15 cicluri - iniţializarea memoriei DRAM
- 1 ciclu - trimis data

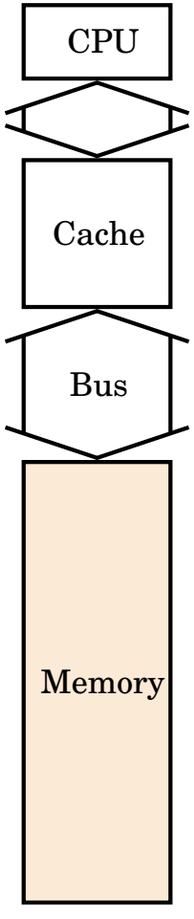
Lărgime de bandă de 1 cuvânt: Pentru un cache cu blocuri de 4 cuvinte şi interacţie cu memoria pe 1 cuvânt, penalizarea la eşec este de:

1 (trimis adresă) + 4*15 (4 initializari de acces) + 4*1 (transfer 4 date din bloc) = 65 cicluri

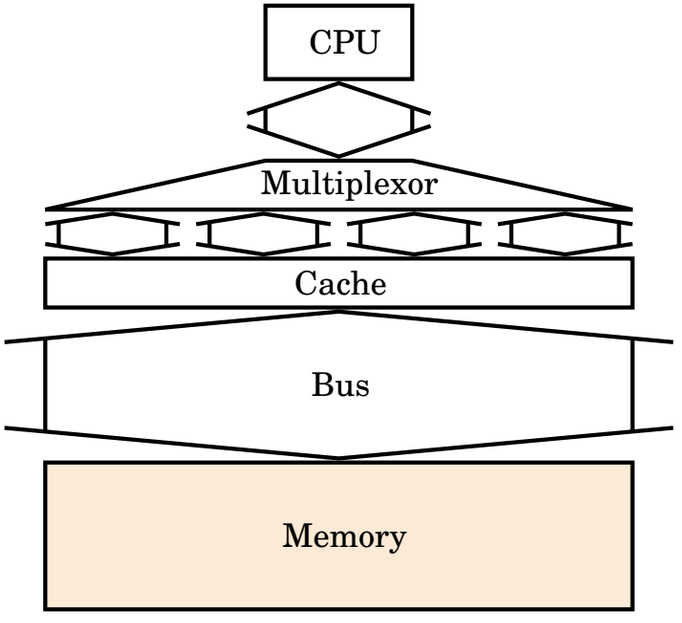
Lărgime de bandă de 2 cuvinte: Dacă interacţia cu memoria este pe 2 cuvinte, penalizarea devine:

1 (trimis adresă) + 2*15 (2 initializari de acces) + 2*1 (transfer 4 date din bloc) = 33 cicluri

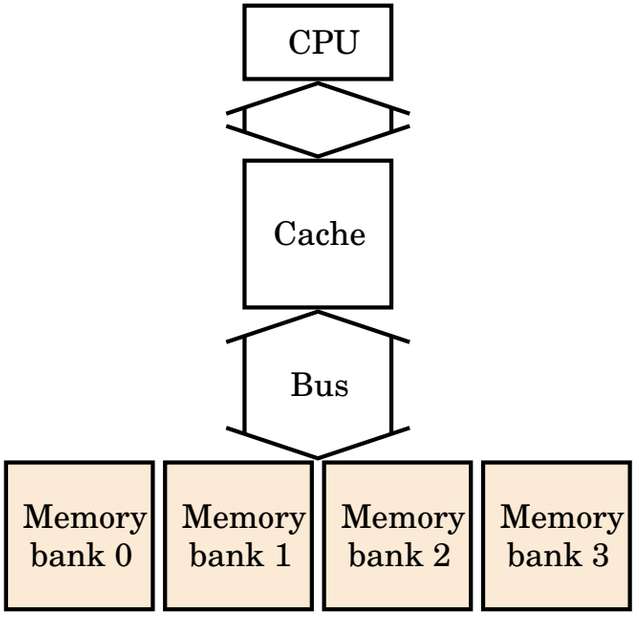
..Generalitati



a. One-word-wide memory organization

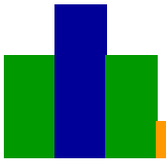


b. Wide memory organization



c. Interleaved memory organization

Exemple de *organizări de memorie* (relativ la *largimea de bandă* a interacţii între memorii).



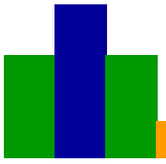
..Memoria cache

Organizare a memoriei cu cache-uri (cont.)

- Varianta a 2-a este mai rapidă, dar mai costisitoare şi poate introduce întârzieri din cauza multiplexoarelor şi controlului dintre CPU şi cache-uri.
- Alternativ, putem păstra interacţia pe 1 cuvânt, dar citi mai mult la un acces de memorie.

Lărgime de bandă de 1 cuvânt şi interleaving: Dacă interacţia cu memoria este pe 1 cuvânt dar putem citi 4 date la un acces de memorie, penalizarea devine:

1 (trimis adresă) + 1*15 (1 initializare de acces) + 4*1 (transfer 4 date din bloc) = 20 cicluri

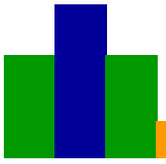


..Memoria cache

Tipuri de memorii:

- Organizarea este descrisă sub forma $d \times w$, unde
  d - numărul de locaţii adresabile;
  w - lărgimea fiecărei locaţii (valori populare: 4 şi 8 biţi);

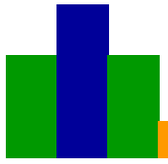
  Exemple: DRAM de 16Mb de tip 4M $\times$ 4; Memorie de 64MB din 4 cipuri DRAM de de tip 4M $\times$ 16; etc.
- Cu timpul, s-au creat combinaţii DRAM + SRAM, anume se accesează o linie întregă (*pagină*) dintr-o configuraţie matricială de DRAM, rezultatul punându-se într-un SRAM; Exemplu: *EDO RAM* (EDO = Extended Data Out).
- Versiuni mai recente sunt *SDRAM*-uri (DRAM-uri sincrone) care folosesc ceas pentru a accesa o secvenţa de locaţii succesive (engl. “burst”) - evită sincronizarea şi adrese multiple.



Orgnizarea memoriei

Cuprins:

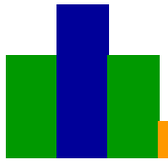
- Generalitati
- Memoria cache
- *Performanta memoriei cache*
- Memoria virtuala
- Concluzii, diverse, etc.



Performanta memoriei cache

Generalitati:

- *Timpul de execuţie* CPU poate fi separat în: (1) timp de *execuţie propriu-zisă* t_e şi (2) timp de *aşteptare pentru accesul la memorie* t_a (aici, doar eşecurile contează).
- La rândul lui, t_a se împarte în: (1) timp de *aşteptare pentru citire* t_r şi (2) *pentru scriere* t_w.
- La citire, contează *rata de eşecuri* şi *penalizarea la un eşec*;
- La scriere, ca la citire + întârzierile produse *când buffer-ul este plin*.
- In fine, *ratele* de eşec variază la *instrucţiuni* şi la *date*.



..Performanta memoriei cache

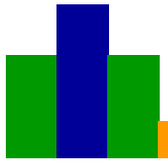
Exemplu: Pentru `gcc` [43% ALU, 23% load, 13% store, 19% branch, 2% jump], presupunem că avem:

- o rată de eşec de 2% la instrucţiuni şi 4% la date;
- CPI-ul este 2 fără blocări de memorie;
- penalizarea la toate eşecurile este 40 de cicluri

Cu cât este mai rapidă o maşină cu cache perfect (fără eşecuri)?

Răspuns:

- Eşec pentru instrucţiuni: Eşec (instr.) $= I \times 0.02 \times 40 = 0.80 \times I$ cicluri ($I =$ numărul de instrucţiuni);
- Pentru date (load + store = 36%): Eşec (date) $= I \times 0.36 \times 0.04 \times 40 = 0.58 \times I$;
- Total blocări pentru acces memorie: $0.80 \times I + 0.58 \times I = 1.38 \times I$;
- Performanta: $CPU_{\text{cu\_blocari}}/CPU_{\text{cache\_perfect}} = (2 + 1.38)/2 = 1.69$.



Performanta memoriei cache

Procesor mai rapid:

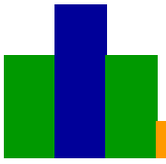
- Pe exemplul anterior, dacă *reducem CPI de la 2 la 1* (păstrând frecvenţa de ceas), cache-ul perfect aduce un spor de

$$(1 + 1.38)/1 = 2.38$$

  Dar, în versiunea cu blocări, ponderea blocărilor creşte semnificativ de la

$$1.38/3.38 = 41\% \quad \text{la} \quad 1.38/2.38 = 58\%$$

- Similar, dacă *frecvenţa de ceas creşte*, performanţa nu creşte proporţional în prezenţa eşecurilor de access la memorie (vezi exemplul următor).
- Concluzie: CPI-ul mic, ori frecvenţa sporită *creşte impactul eşecurilor* de acces la memorie.



..Performanta memoriei cache

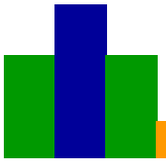
Exemplu: Repetăm exemplul, cu aceleaşi date (în particular, păstrăm ratele de eşec anterioare), dar

- dublăm frecvenţa de ceas.

Cu cât este mai rapidă noua maşină, dacă nu avem cache perfect?

Răspuns:

- Măsurate in frecvenţa nouă, penalizările la eşec sunt de 80 de cicluri;
- Eşec (total) $= I \times (0.02 \times 80 + 0.36 \times 0.04 \times 80) = 2.75 \times I$ cicluri;
  Deci avem un CPI de $2 + 2.75 = 4.75$.
- Performanta: $CPU_{\text{cu\_ceas\_lent}} / CPU_{\text{cu\_ceas\_rapid}} = 3.38 / [(2 + 2.75) \times 0.5] = 1.42$.
- Deci, la cache imperfect, crescând frecvenţa cu 100%, performanta creşte cu numai 42%!



Reducerea numarului de esecuri

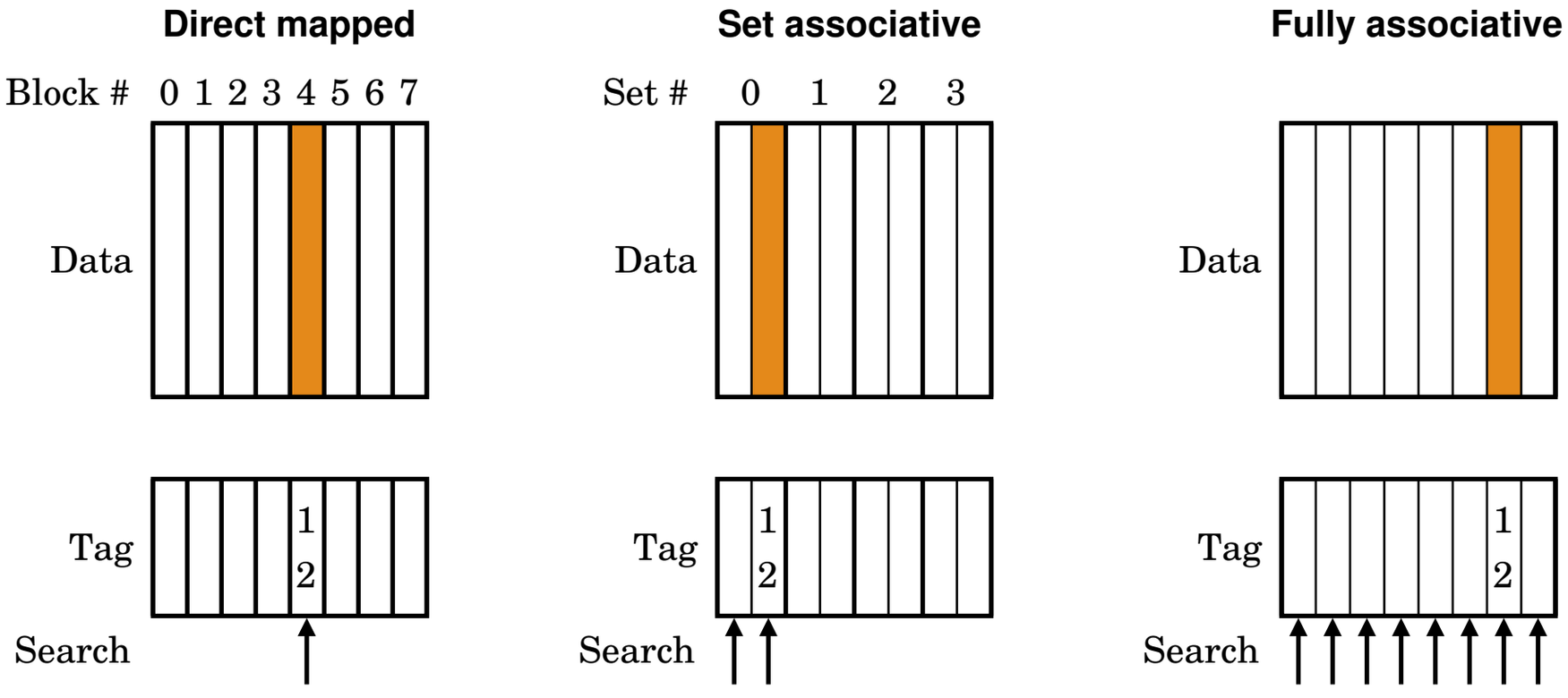
Alte organizari de cache:

- Schema de adresare directă (modulo N) este simplă, dar poate produce un număr mare de eşecuri.
- Se poate folosi o schemă de adresare *complet asociativă* în care orice bloc de memorie poate fi pus în orice bloc din cache;
- Intre ele, putem folosi scheme de adresare *n-asociative* (ori *set-asociative*), unde pentru un bloc de memorie există o mulţime de n locaţii în cache unde poate fi plasat.
- Dacă sunt N mulţimi, atunci

  blocul k din *memorie* $\mapsto$ mulţimea $k \bmod N$ din *cache*

  Apoi blocul *se caută* în interiorul mulţimii asociate ca mai sus.

..Reducerea numarului de esecuri



Exemple de *organizări de cache*: adresare *directă*, *partial asociativă*, şi *complet asociativă*.

..Reducerea numarului de esecuri

Exemple de adresări pentru un cache de capacitate 8:

directă,

2-asociativă,

4-asociativă,

8-asociativă.

One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

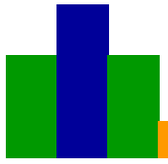
Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data



..Performanta memoriei cache

Exemplu: Avem date 3 cache-uri de 4 cuvinte, care sunt respectiv

- *complet asociativ*, *2-asociativ*, şi cu *adresare directă*.

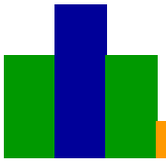
Câte eşecuri există la procesarea secvenţei de adrese: `0,8,0,6,8`?

Răspuns:

Cache cu accesare directă: $0 \mapsto 0, 6 \mapsto 2, 8 \mapsto 0 \Rightarrow$ 0(miss), 8(miss), 0(miss), 6(miss), 8(miss), deci 5 eşecuri;

Cache 2-asociativ: $0 \mapsto 0, 6 \mapsto 0, 8 \mapsto 0 \Rightarrow$ 0(miss), 8(miss), 0(hit), 6(miss), 8(miss), deci 4 eşecuri;

Cache complet asociativ: $0 \mapsto 0, 6 \mapsto 0, 8 \mapsto 0 \Rightarrow$ 0(miss), 8(miss), 0(hit), 6(miss), 8(hit), deci 3 eşecuri;



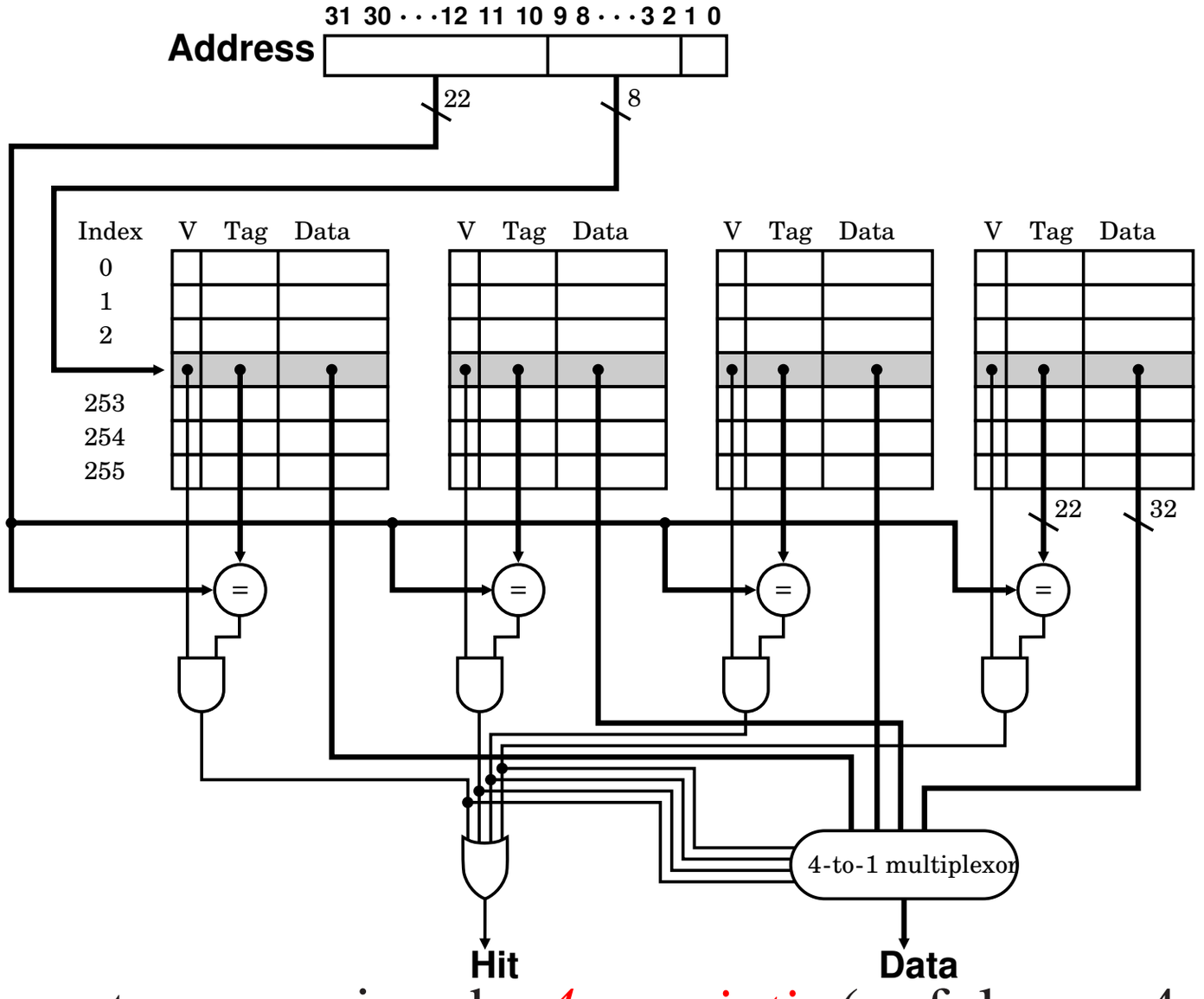
..Reducerea numarului de esecuri

Exemplu (concret, pe DECStation 3100)

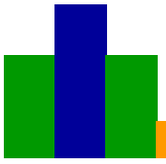
Program	Asociativitate	Esec la instructiuni	Esec la date	Esec combinat
`gcc`	1	2.0%	1.7%	1.9%
	2	1.6%	1.4%	1.5%
	4	1.6%	1.4%	1.5%
`spice`	1	0.3%	0.6%	0.4%
	2	0.3%	0.6%	0.4%
	4	0.3%	0.6%	0.4%

Datele sunt mici; doar la `gcc` *se vede un câstig de 20% in trecerea de la 1- la 2-asociativtate.*

..Reducerea numarului de esecuri



Implementare a unui cache *4-asociativ* (se folosesc 4-comparatori şi un multiplexor 4-to-1).



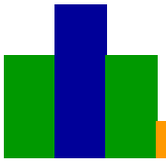
..Reducerea numarului de esecuri

Complemente:

Mărimea tag-urilor: In genere, mărind numărul de elemente într-un set (gradul de asociativitate) creşte mărimea tag-ului.

De exemplu, un cache de 4K blocuri de 4 cuvinte (adrese de 32b) foloseşte pentru tag-uri: 64Kb la adresare directă, 68Kb la cache 2-asociativ, 72Kb la cache 4-asociativ, si 112Kb la cache complet-asociativ.

Tehnica de înlocuire: La memorii asociative, dacă mulţimea de locaţii pentru un bloc este complet ocupată, trebuie înlocuit un bloc. Strategia uzuală este *LRU (least recently used)* - cel mai vechi bloc este înlocuit.

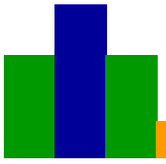


..Reducerea numarului de esecuri

Complemente:

Cache-uri pe mai multe nivele:

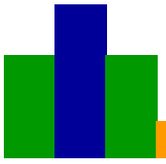
- Este util să se organizeze cache-uri pe mai multe nivele, spre exemplu cu un nivel SRAM suplimentar între procesor şi DRAM-uri.
- Prezenţa unui cache suplimentar poate reduce timpul de penalizare la eşec, dacă putem găsi blocul în cache-ul secundar.
- Cu 2 cache-uri, prima se poate focaliza pe minimizarea timpului de succes, iar a doua pe reducerea timpului de penalizare la eşec.
- Folosirea de mai multe cache-uri multiplică numărul de tipuri de eşec care pot apare.



Orgnizarea memoriei

Cuprins:

- Generalitati
- Memoria cache
- Performanta memoriei cache
- *Memoria virtuala*
- Concluzii, diverse, etc.



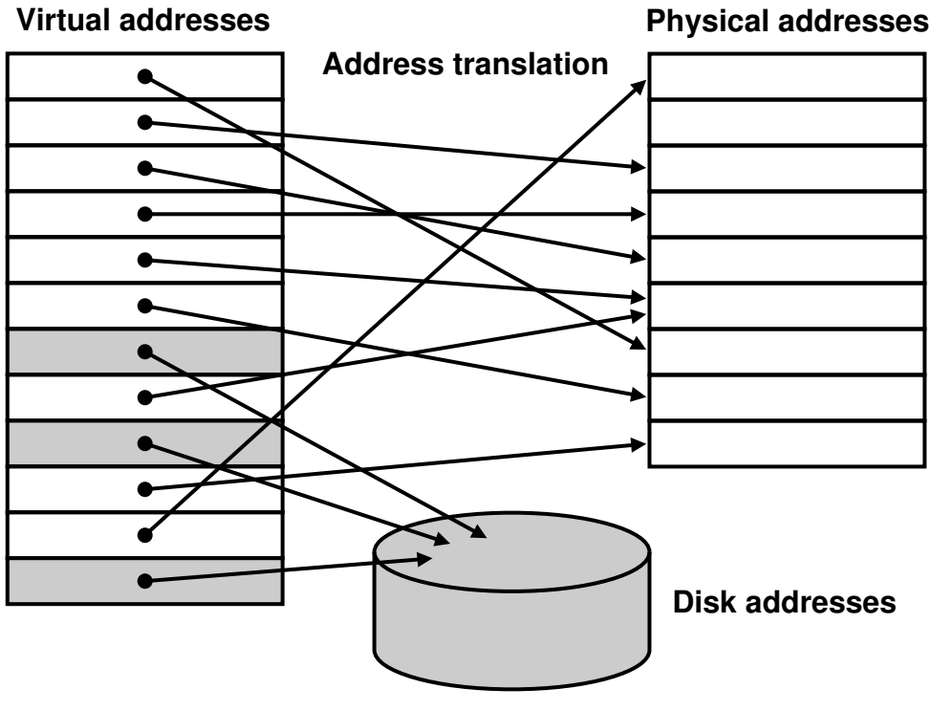
Memoria virtuala

Generalitati:

- Memoria *principală* poate fi considerată un "cache" pentru următoare - cea *secundară* (implementată magnetic pe hard disk). Tehnica rezultată este de *memorie virtuală*.
- Fiecare program care rulează are unicul său spaţiu de adrese în memoria virtuală.
- O adresă din memoria virtuală are asociată o *adresă fizică* în memoria principală şi una pe disk.
- Memoria principală fiind mai mică, blocurile din memoria virtuală se încărcă (şi descărcă) succesiv în memoria principală.
- Blocul de transfer între memorii (disk, virtuală, principală) se numeşte *pagină*. Localizarea unei date în memorie se face cu *numărul de pagină* şi *deplasarea* (offset-ul) în pagină.

..Memoria virtuala

Generalitati (cont.)

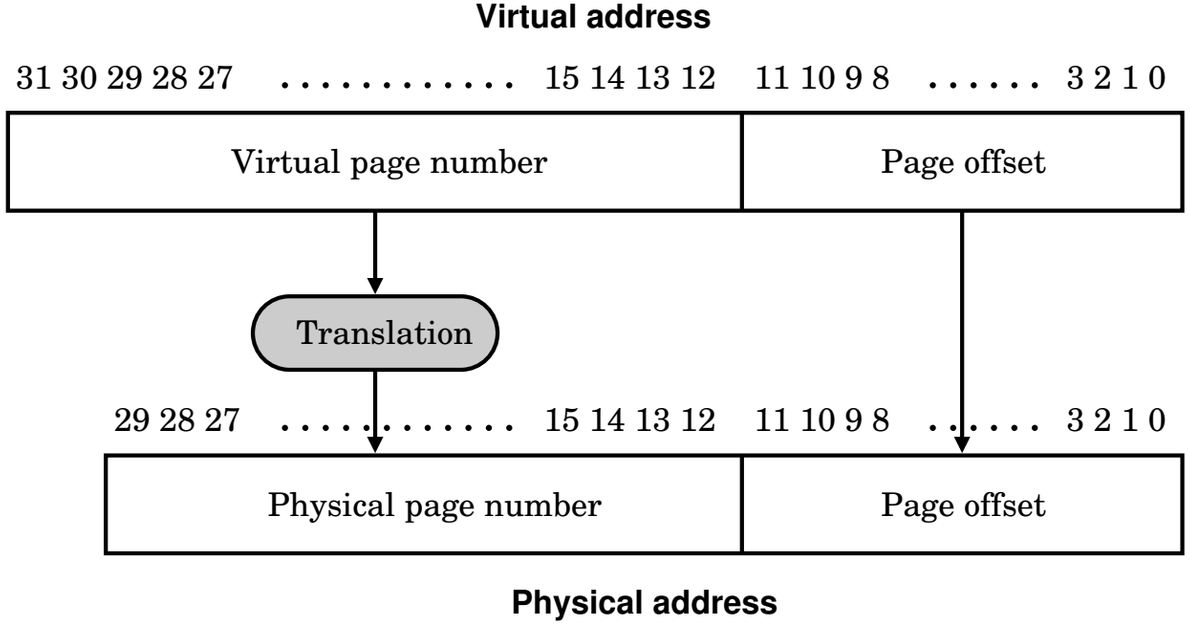


- *Memoria virtuală* şi corespondenţele la cea *magnetică* (pe disk) şi cea *principală*.
- Putem observa un caz de *sharing* (memorie comună): 2 adrese virtuale au, *în acelaşi timp*, aceeaşi adresă fizică, deci 2 programe au cod ori date comune.

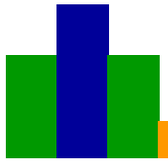
..Memoria virtuala

Generalitati (cont.)

- Exemplu de corespondenţă de la adrese virtuale la adrese fizice



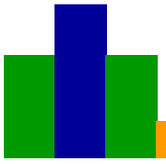
- Paginile au $2^{12} = 4$KB, deci offset-ul necesită 12b de adresă;
- Pentru adrese fizice, numărul de pagină foloseşte 18b, deci memoria principală este de cel mult 1GB;
- Memoria virtuală foloseşte 20b, deci este până la 4GB.



..Memoria virtuala

Generalitati (cont.) Eşecul de a găsi o dată în memorie se numeşte aici *page fault* (eroare de pagină). Dat fiind accesul lent la hard disk, penalizarea pentru page-fault este enormă: *milioane de cicluri* de procesor. Soluţii:

- Paginile trebuie să fie *mari* spre a amortiza timpul de acces; usual de la 4KB la 64KB;
- Organizările trebuie să *evite* page-fault, deci folosim organizări *asociative*.
- Se poate folosi tehnică *software* complicată spre a evita page-fault (penalizarea poate fi mai mică).
- Tehnica de scriere nu este simultană, ci *scris la loc* (scriem pe disk doar la schimbarea paginii din memorie).



Memoria virtuala

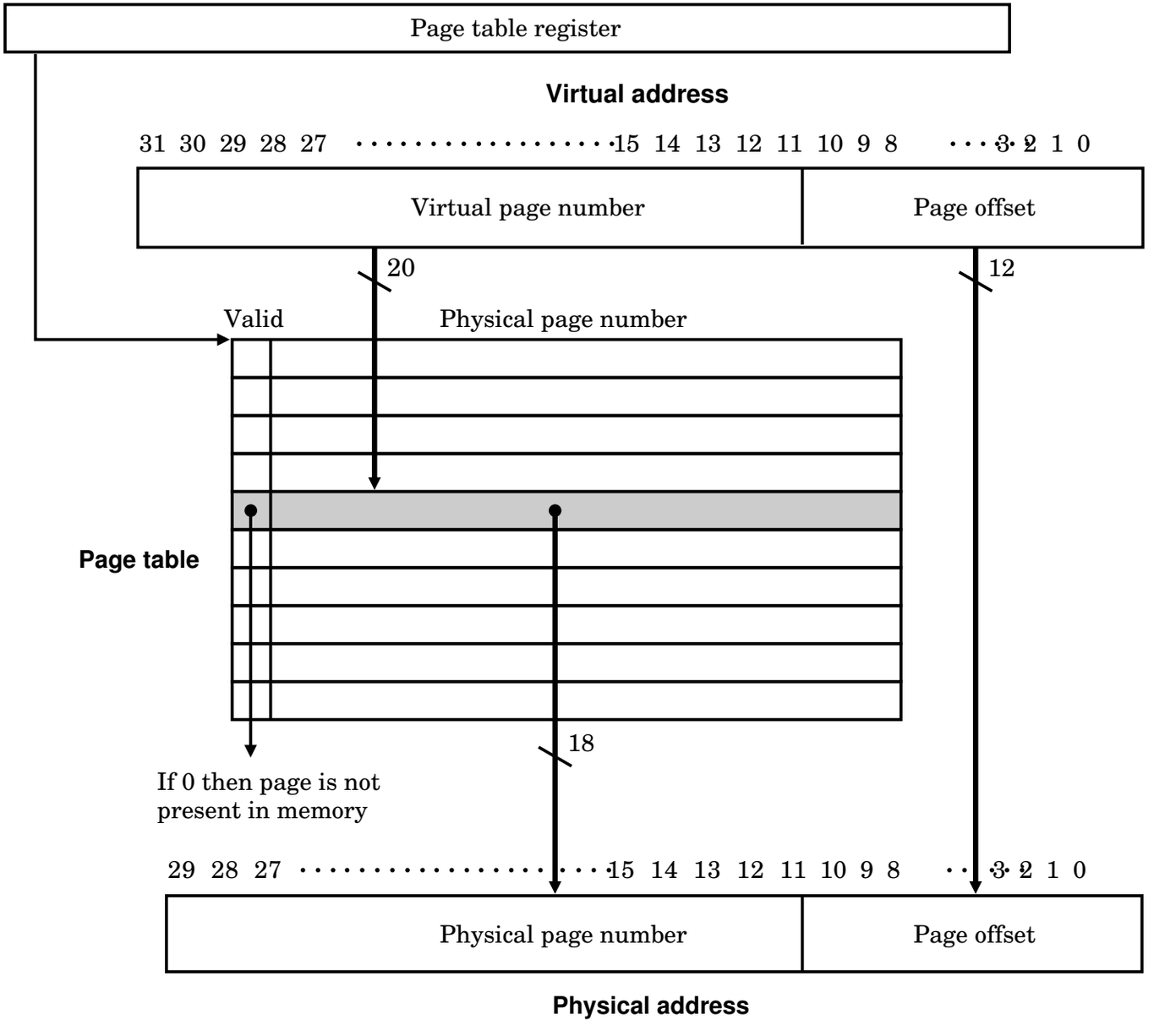
Plasarea paginilor:

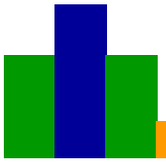
- Pentru a evita page-fault, paginile se pun *oriunde* în memorie, folosind algoritmi şi structuri de date sofisticate (plasare complet-asociativă).
- In memoria virtuală, exită un *tabel de pagini* care conţine indexul paginii în adresare virtuală şi adresa fizică corespunzătoare.
- Fiecare *program are propriul său tabel* de pagini.
- Există în hardware un *registru* special *pentru tabelul de pagini*.
- In fine, există un *bit de validitate* (ca la cache), care spune dacă pagina este prezentă în memorie.

..Memoria virtuala

Plasarea paginilor (cont.)

Figura descrie cum se obţine *adresa fizică* folosind *registrul pentru tabelul de pagini* şi *numărul virtual al paginii* în tabel.



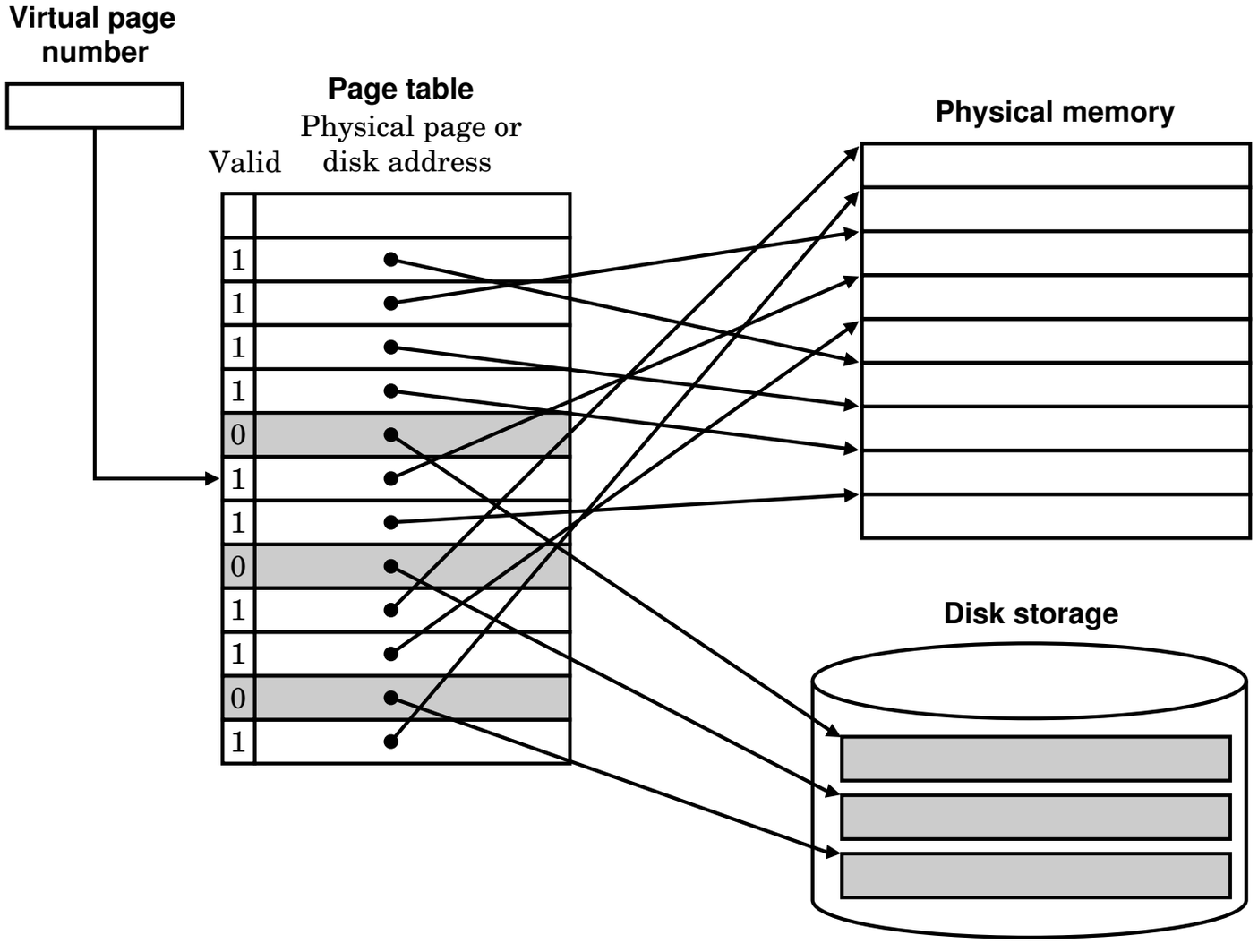


Memoria virtuala

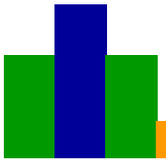
Page-fault:

- Apare *page-fault* când *bitul de validitate este 0* (pagina nu e prezentă în memorie). Controlul este preluat de *sistemul de operare*.
- Pentru a putea şti locaţiile de pe disk ale paginilor din memoria virtuală, sistemul de operare crează o *structură* de date cu *locaţiile pe disk* ale paginilor unui *proces* la crearea lui.
- Dacă toate locaţiile din memorie sunt ocupate, sistemul de operare *alege o pagină* care va fi înlocuită.
- Modelul uzual de selecţie este *LRU (least recently used)*. Exemplu: dacă ultimile pagini accesate au fost 10,12,9,7,11,10 şi se cere 8 (care nu e prezentă), se înlocuieşte 12 cu 8. [Implementarea lui LRU poate fi *costisitoare*; alternativ, putem folosi un *bit de utilizare*.]

..Memoria virtuala



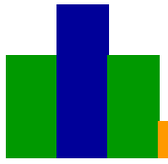
Căutare a unei pagini în memorie: dacă bitul de validitate e 0, pagina este *doar pe disk* (de obicei, se foloseşte o tabelă diferită pentru locaţiile de pe disk).



Memoria virtuala

Scriere:

- Scrierea pe disk este *lentă*, deci se preferă evitarea ei.
- Tehnica uzuală este *scrierea pe loc*, anume scriem pagina pe disk când pagina este înlocuită în memoria principală.
- Suplimentar, se poate folosi un *dirty-bit* care spune dacă pagina a fost modificată în memorie şi trebuie realmente rescrisă pe disk.



Memoria virtuala

TLB: (*Translation-lookaside buffer*)

- Pentru a eficientiza acccesarea tabelului cu pagini se foloseşte o tabelă mică cu accesurile curente, anume o *memorie cache* numită *TLB* (translation-lookaside buffer).
- Organizarea este de tip *cache peste tabelul de pagini*.
- Dacă apare un *eşec în TLB*, există o şansă să găsim adresa fizică a paginii *în tabelul cu pagini*. Dacă nu, se invocă sistemul de operare pentru acces la *disk*.
- Eşecurile TLB pot fi rezolvate cu *hardware ori software* (de regulă tehnici similare).
- La înlocuirea unei intrări din TLB, singurele valori de scris în tabelul de pagini sunt *biţii use + dirty* (restul nu se modifică).

..Memoria virtuala

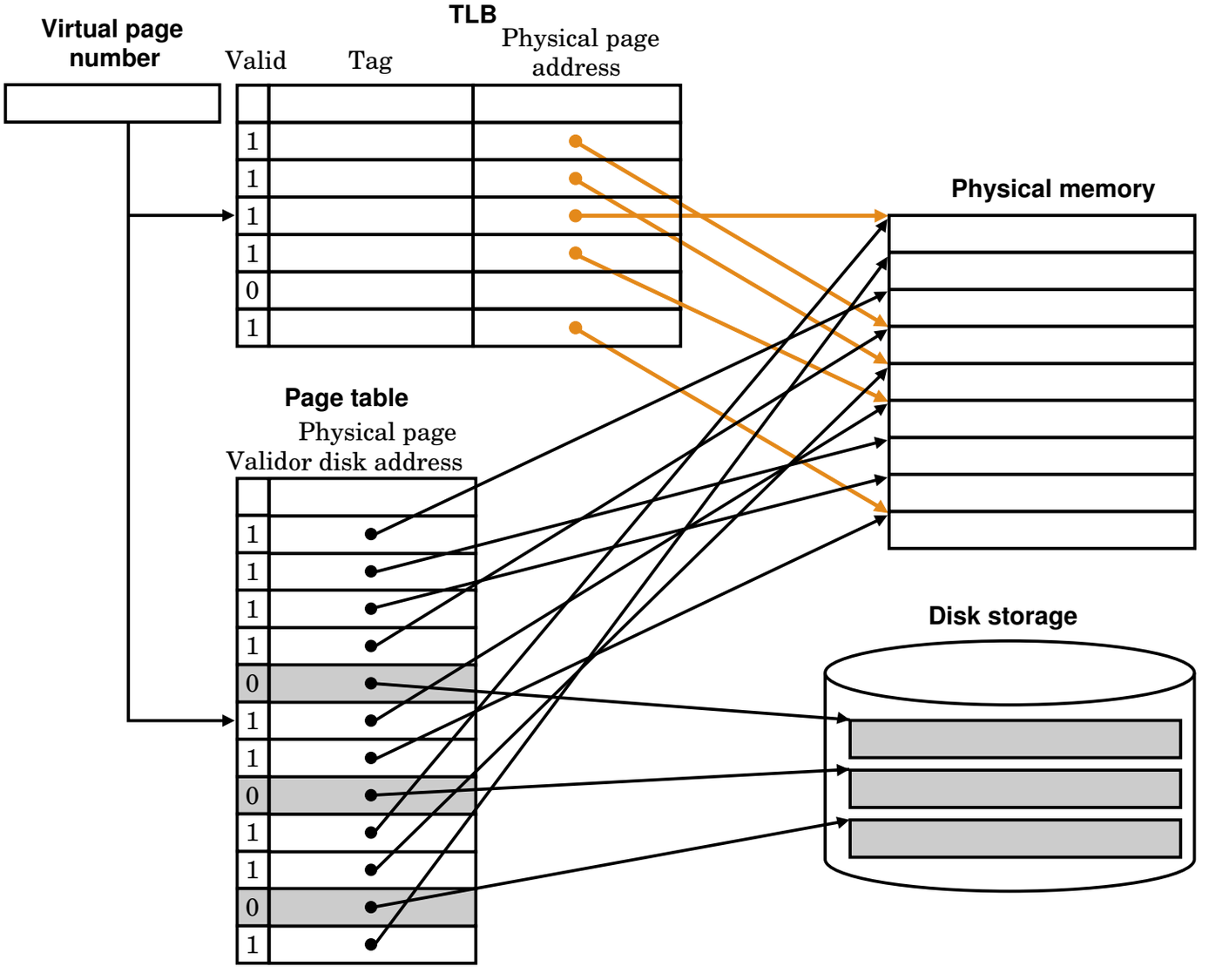
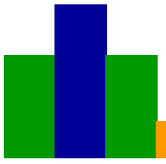


Figura prezintă buffer-ul *TLB* care este un fel de *cache pentru tabela de pagini* relativ la adresele fizice care apar.



Memoria virtuala

TLB (cont.)

Caracteristici tipice pentru TLB:

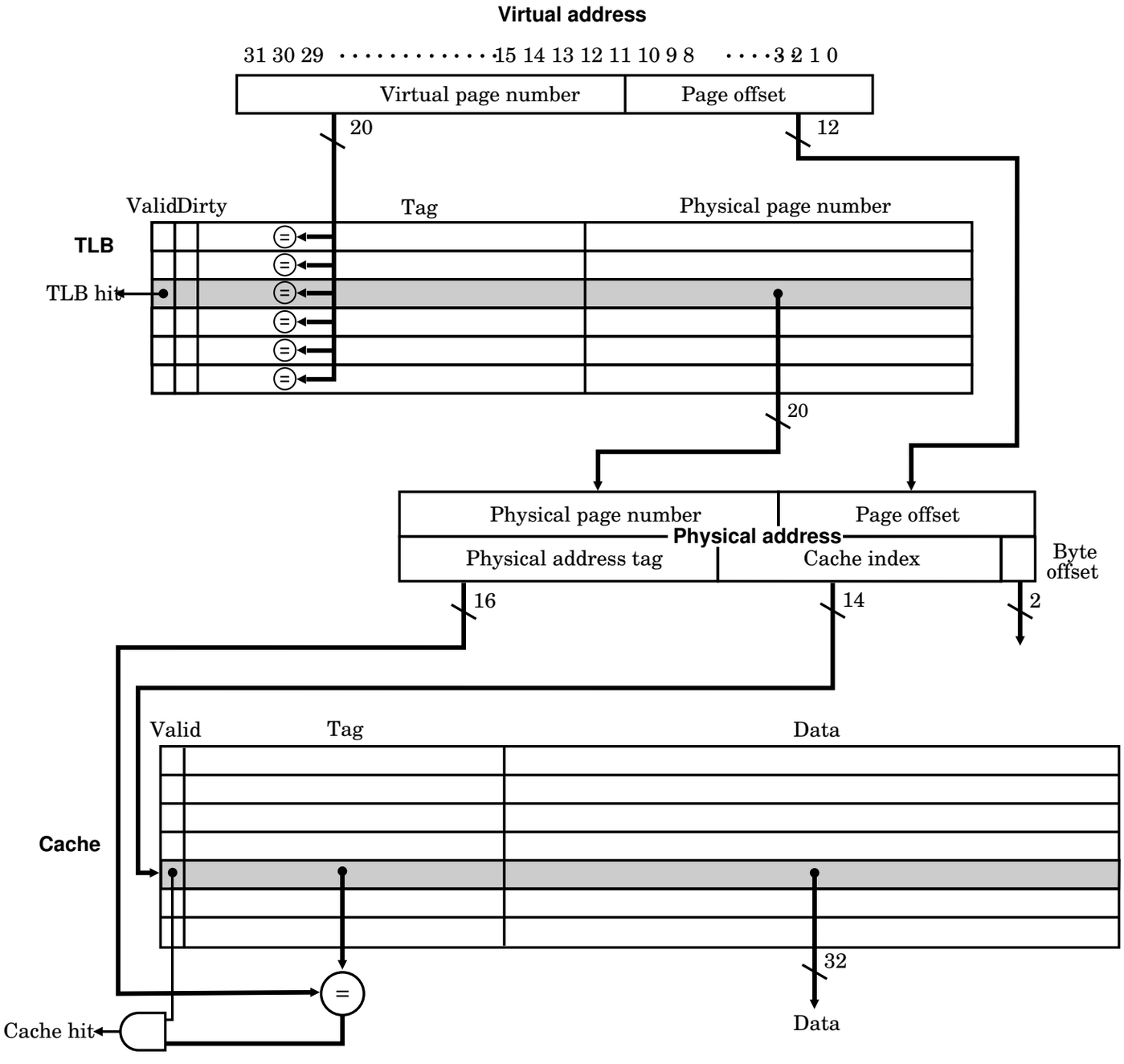
- Mărime: 32-4096 intrări;
- Mărime bloc: 1-2 pagini;
- Timp de succes: 0.5-1 cicluri;
- Penalizare la eşec: 10-30 cicluri;
- Rate de eşec: 0.01-1%.

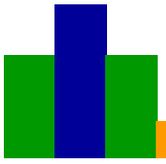
..Memoria virtuala

TLB (cont.)

Implementare integrată a *memoriei virtuale*, *TLB*-urilor şi *cache*-urilor.

Exemplu este de la DECStation 3100.





Orgnizarea memoriei

Cuprins:

- Generalitati
- Memoria cache
- Performanta memoriei cache
- Memoria virtuala
- *Concluzii, diverse, etc.*