

Prof. Habil. Dr. Ing. Decebal Popescu -  
[decebal.popescu@upb.ro](mailto:decebal.popescu@upb.ro)

# CALCULATOARE NUMERICE II

# Bibliografie

01

*Computer Architecture  
A Quantitative  
Approach - 5<sup>th</sup> edition*  
- Patterson and  
Hennessey

02

*Modern Processor  
Design Fundamentals  
of Super-scalar  
Processors*

•John Paul Shennon and  
Mikko H. Leposty

03

*Computer  
Organization -  
Hennessey and  
Patterson 5<sup>th</sup> edition -  
ELE 375*

04

**Cursul de PL și CN1 de  
la UPB**

05

ELE 475, ELE 375 și  
COS 475 - PRINCETON  
University

06

6.823 - MIT; CS252 și  
CS152 - UCB; ECE 4750  
- Cornell

# Computer Architecture

- încearcă să vină cu straturile de abstractizare și implementare
- **Definiție:** Arhitectura calculatoarelor este o proiectare a straturilor de abstractizare / implementare care ne permit executarea de informații, prelucrarea de aplicații folosind eficient tehnologiile de fabricație
- Nivelele de abstractizare:
  - Instruction Set Architecture;
  - Microarchitecture;
  - Register-Transfer Level

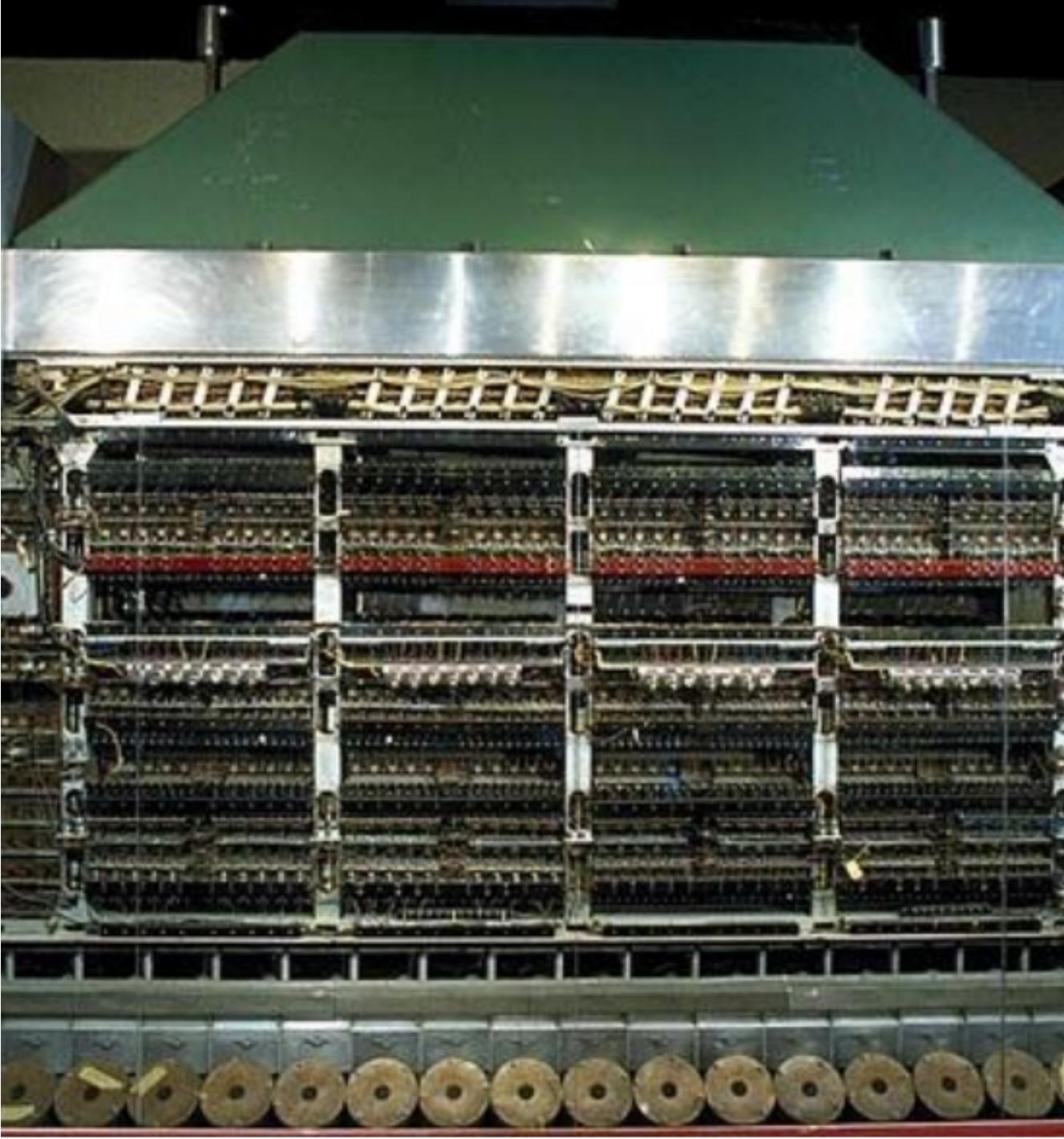
sunt formează Arhitectura Calculatoarelor



# 1940 - 1950: IAS Machine

---

- Proiectată de John von Neumann
- 1952 - prima funcționare
- Această mașină este de fapt construită din tuburi cu vid
- În interiorul acestor tuburi erau comutatoare care puteau fi setate (o poartă care putea fi deschisă sau închisă)
- Înainte de această mașină erau doar mașini electromecanice (rotiță de telefon trimitea pulsuri care acționau un mecanism)

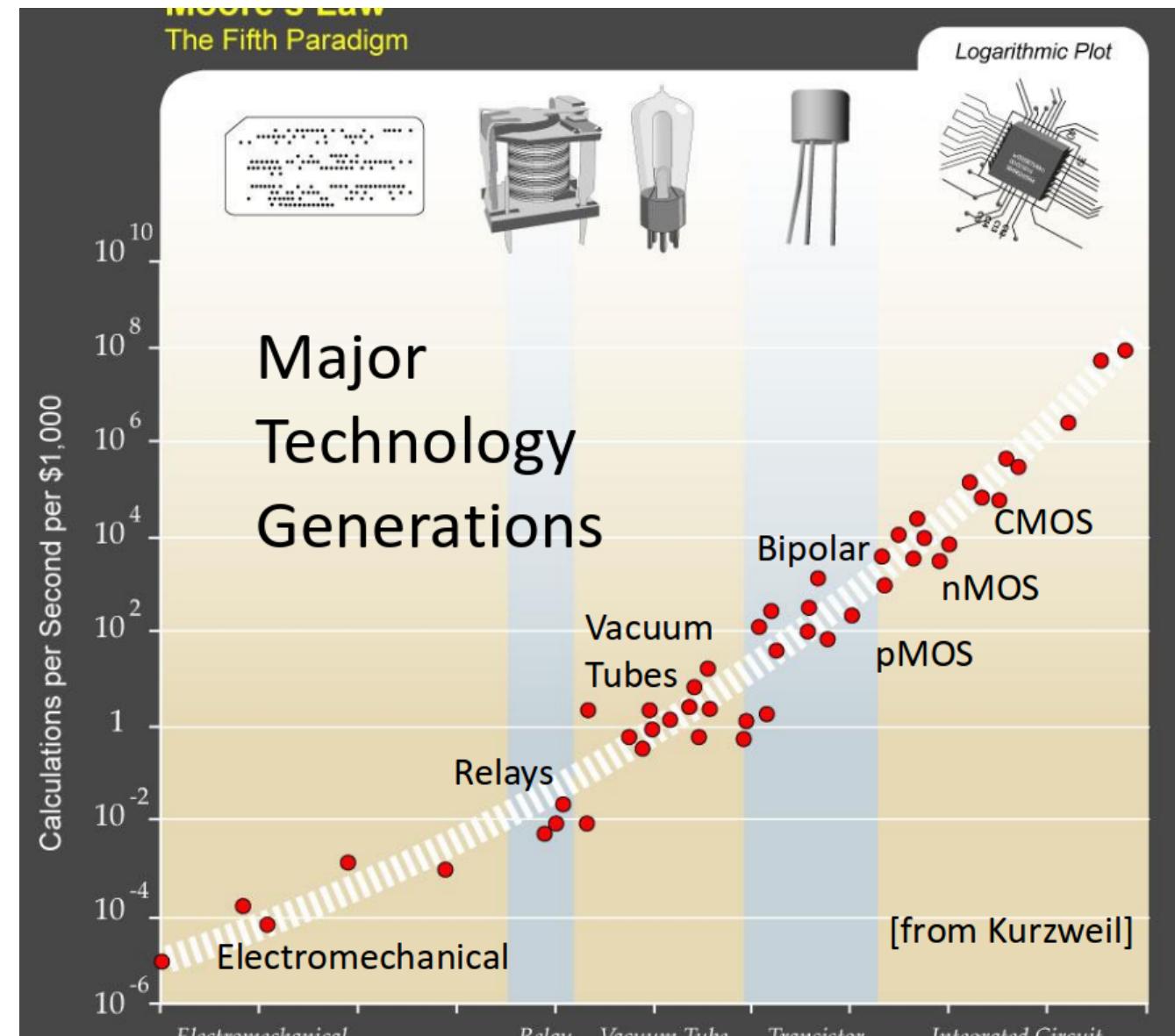


Actualmente



# Legea lui Moore

Gordon Moore a spus că la fiecare optsprezece luni până la doi ani, vor fi de două ori mai mulți tranzistori pe care îi putem cumpără pentru aceeași sumă de dolari.



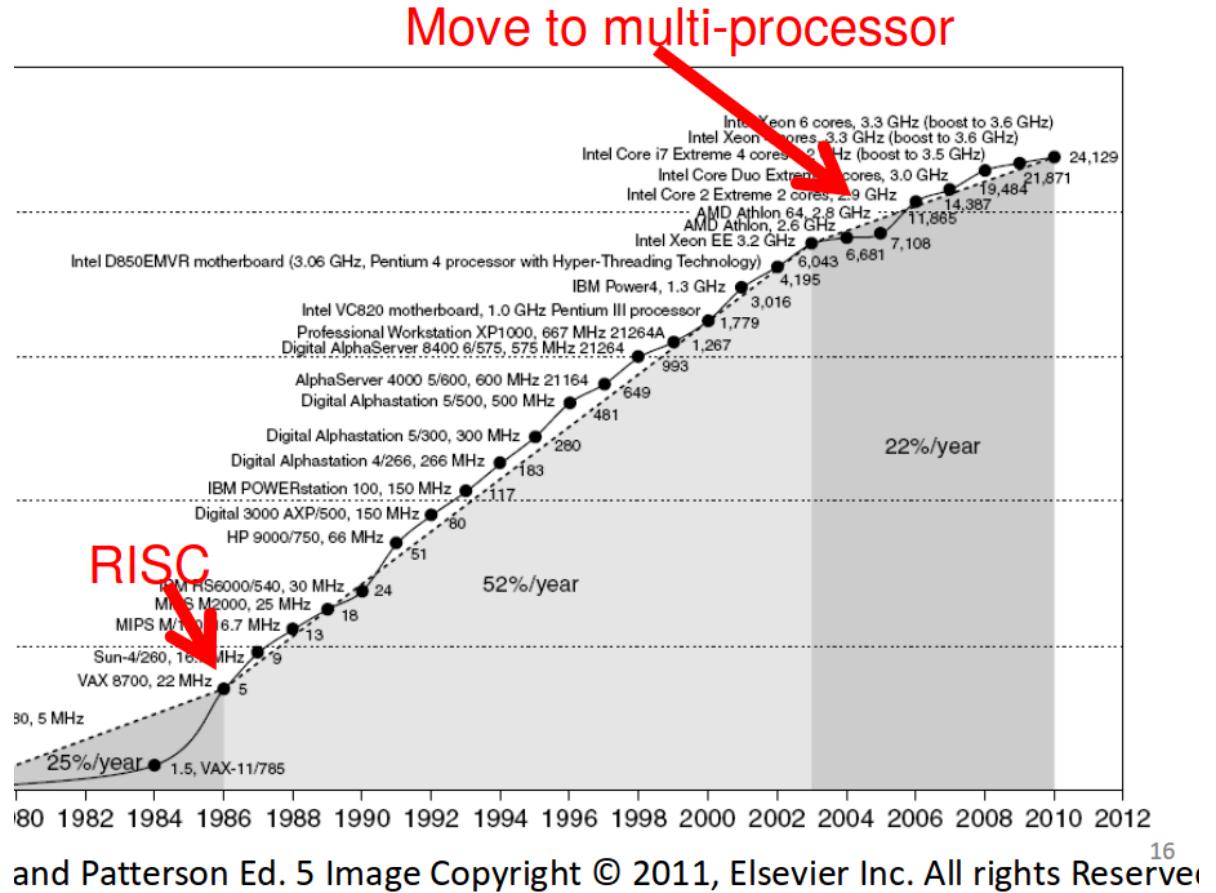
# Performanța procesoarelor secvențiale

performanța crește exponențial

acest lucru subliniează importanța deosebită a arhitecturii calculatoarelor

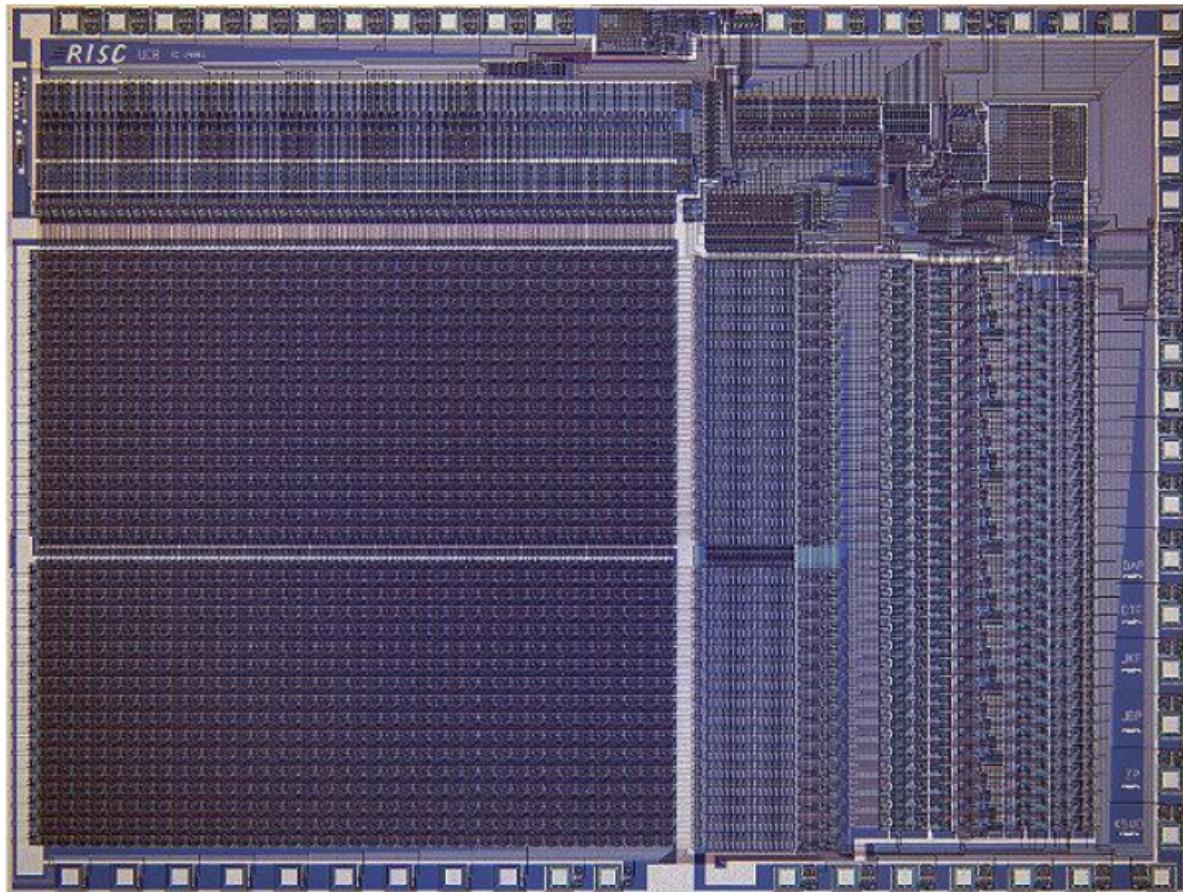
chiar dacă avem un număr din ce în ce mai mare de tranzistori, nu înseamnă nepărat că ei sunt și mai rapizi

arhitectura calculatoarelor înglobează cât mai mulți tranzistori pentru a oferi o performanță cât mai ridicată



and Patterson Ed. 5 Image Copyright © 2011, Elsevier Inc. All rights Reserved  
16

# Evoluția procesoarelor



## RISC 1

- aproximativ 50.000 tranzistoare
- două stagii de pipe
- foarte bun pentru a explica conceptele de bază pentru banda de asamblare și memoria cache

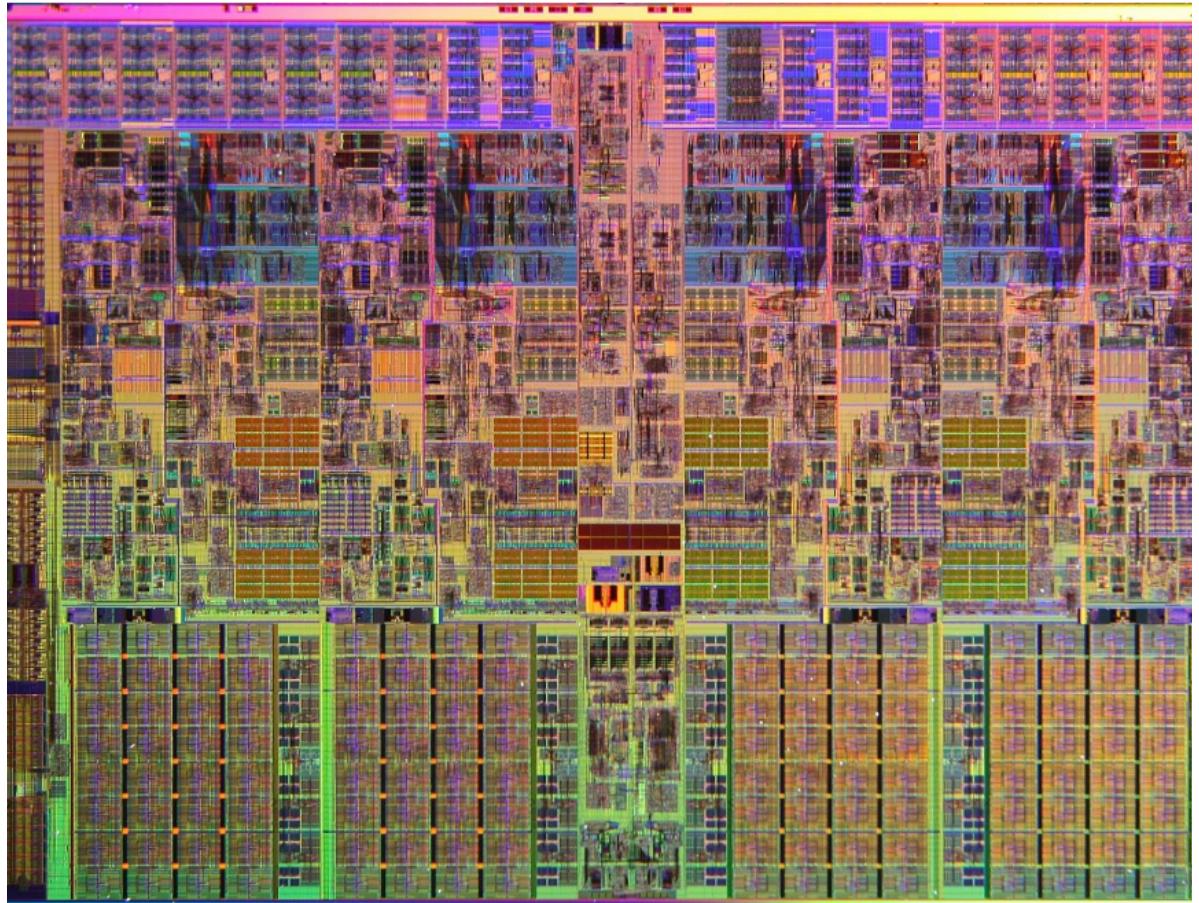
Photo of Berkeley RISC I, © University of California (Berkeley)

# Evoluția procesoarelor

Intel Nehalem Processor -  
Image Credit Intel

Original Core i7

Aproximativ 700.000.000  
tranzistoare



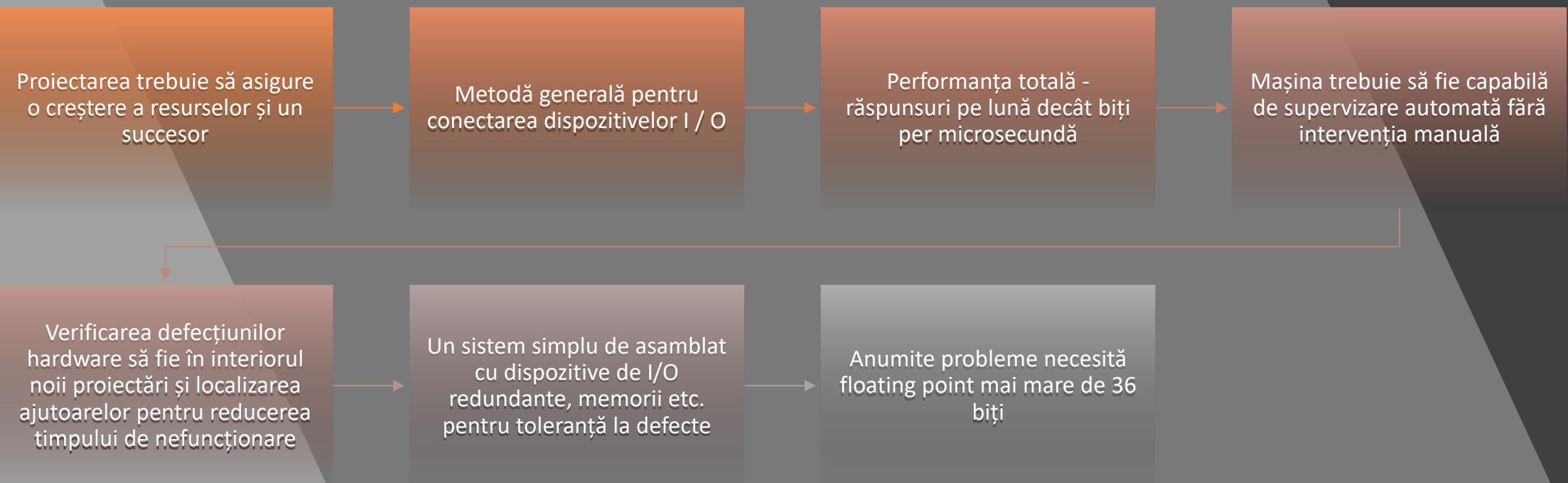
## Conținutul cursului

- Paralelism la nivel de instrucțiune
  - Superscalar
  - VLIW - Very Long Instruction Word
- Paralelismul benzii de asamblare
- Memorii și memorii cache avansate
- Paralelism la nivelul datelor
  - Calculatoare vectoriale
  - GPU
- Paralelism la nivel de thread
  - Multithreading
  - Multiprocessor
  - Multicore
  - Manycore

# Arhitectură versus Microarhitectură

- Arhitectură / ISA (Instruction Set Architecture)
  - Memorii și registre
  - Instrucțiunile și cum lucrează ele
  - Întreruperi
  - I/O
  - Tipuri de date
- Microarhitectură (Organizare)
  - Compromisul necesar pentru implementarea ISA având în vedere anumite constrângeri precum:
    - viteza
    - energia consumată
    - costul final
- *Exemplu:* lungimea BA, numărul de BA, dimensiunea memoriei cache, aria de silicon utilizată, ordinea execuției instrucțiunilor (execuție secvențială sau nu), lățimea bus-ului, dimensionarea ALU
- Această separare a apărut în anii 1960 când IBM a dorit să unifice procesoarele produse de el care până atunci erau dedicate și incompatibile între ele (701 - 7094; 650 - 7074; 702 - 7080; 1401 - 7010)
- procesorul 701 era dedicat operațiunilor bancare (nu neapărat o bună performanță pentru floating point) cât timp 1401 era dedicat calculelor științifice. Soluția: IBM 360

# IBM 360 - principiile de proiectare



# IBM 360 - primul calculator GPR

## Procesorul

- 16 registre generale pe câte 32 de biți
- pot fi utilizate ca registre indexate și registre de bază
- registrul 0 are anumite proprietăți speciale
- 4 registre pentru floating point de 64 biți
- Un PSW (Program Status Word)
- PC, coduri pentru condiții, flag-uri de control

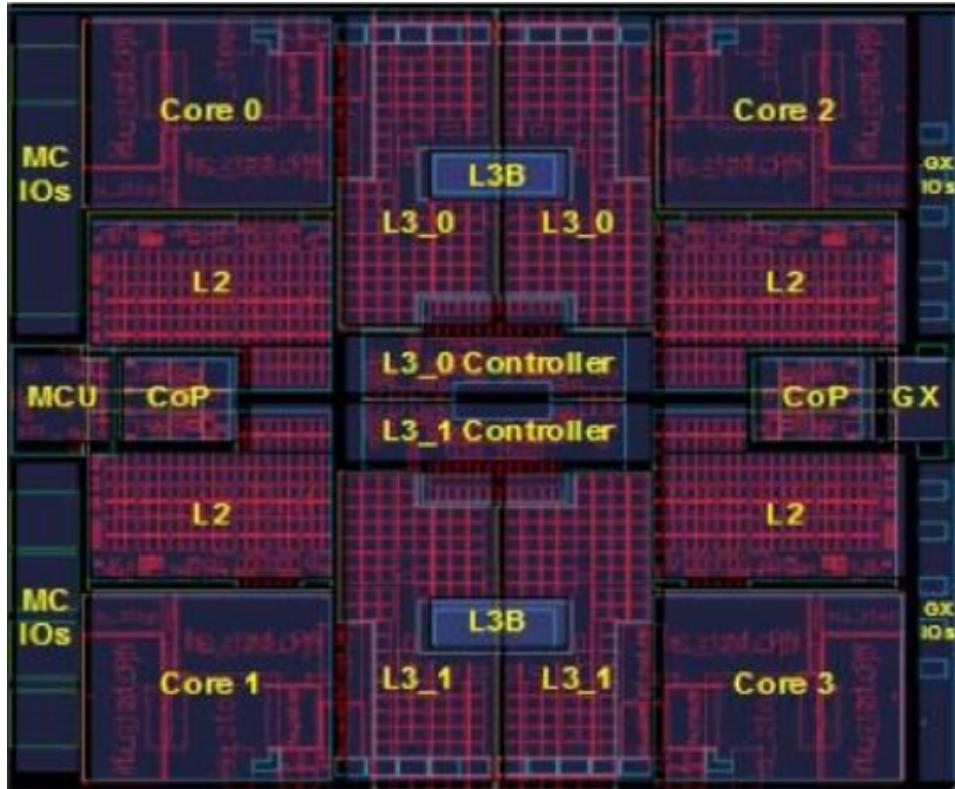
Un calculator pe 32 de biți cu adrese pe 24 de biți

- Dar nici o instrucțiune nu conține o adresă de 24 biți!

Formatele datelor: 8 biți (bytes), 16 biți (semicuvânt), 32 biți (cuvânt), 64 biți (cuvinte duble)

47 de ani mai târziu - microprocesorul z11 (zSeries)

# Z11



[ IBM, Kevin Shum, HotChips, 2010]

Image Credit: IBM

Courtesy of International Business  
Machines Corporation, © International  
Business Machines Corporation.

- 5.2 GHz în tehnologie IBM 45nm PD-SOI CMOS
- 1.4 miliarde tranzistoare în  $512 \text{ mm}^2$
- adresare virtuală pe 64-biți
  - S/360 a avut 24 biți de extensie și S/370 a avut 31 biți
- Proiectare Quad core
- out-of-order superscalar pipeline
- Out-of-order memory accesses
- Căi de date redundante
  - fiecare instrucțiune realizată în două căi de date paralele și rezultatele sunt comparate
- 64KB L1 I-cache, 128KB L1 D-cache pe chip
- 1.5MB L2 cache privat unificat pe core, pe chip
- 24MB eDRAM L3 cache pe chip
- Scalează până la 96 - core cu 768MB share-uit L4 eDRAM

Aceeași arhitectură  
- microarhitecturi  
diferite

## AMD Phenom X4

- X86 Instruction Set
- Quad Core
- 125W
- Decode 3 Instructions/Cycle/Core
- 64KB L1 I Cache, 64KB L1 D Cache
- 512KB L2 Cache
- Out-of-order
- 2.6GHz

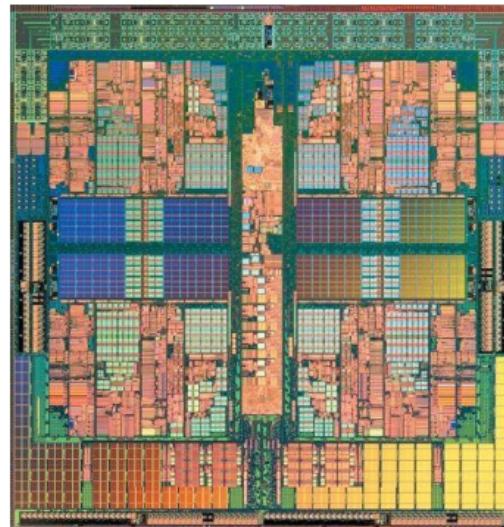


Image Credit: AMD

## Intel Atom

- X86 Instruction Set
- Single Core
- 2W
- Decode 2 Instructions/Cycle/Core
- 32KB L1 I Cache, 24KB L1 D Cache
- 512KB L2 Cache
- In-order
- 1.6GHz

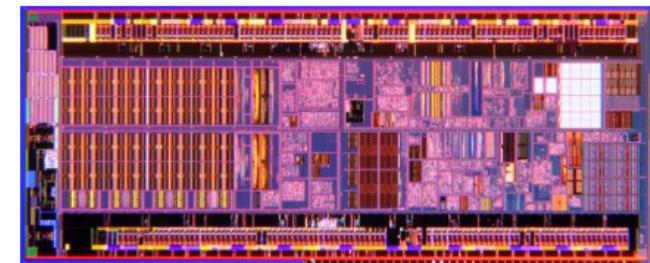


Image Credit: Intel

## Arhitectură și microarhitectură diferită

### AMD Phenom X4

- X86 Instruction Set
- Quad Core
- 125W
- Decode 3 Instructions/Cycle/Core
- 64KB L1 I Cache, 64KB L1 D Cache
- 512KB L2 Cache
- Out-of-order
- 2.6GHz

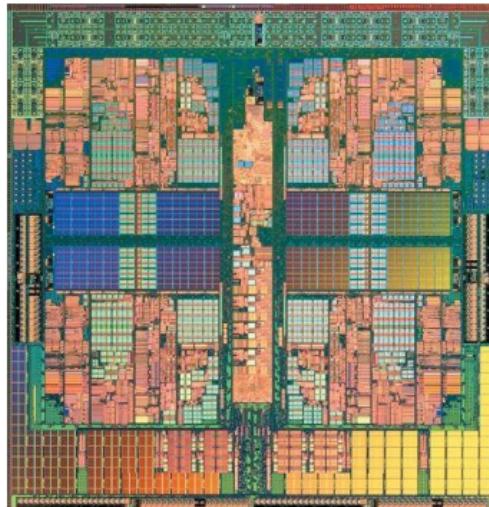


Image Credit: AMD

### IBM POWER7

- Power Instruction Set
- Eight Core
- 200W
- Decode 6 Instructions/Cycle/Core
- 32KB L1 I Cache, 32KB L1 D Cache
- 256KB L2 Cache
- Out-of-order
- 4.25GHz

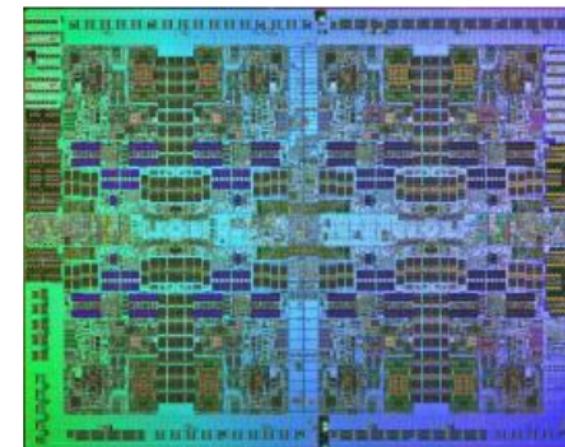


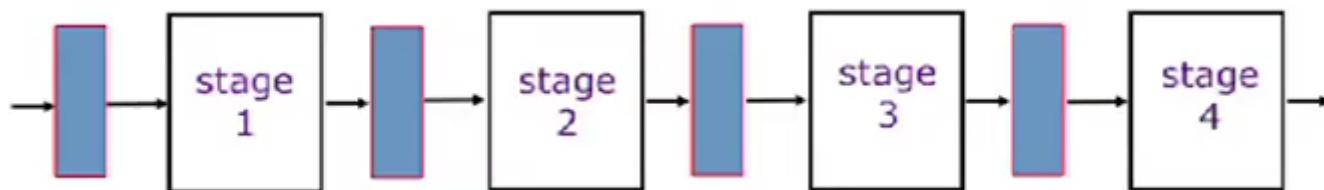
Image Credit: IBM  
Courtesy of International Business Machines  
Corporation, © International Business Machines Corporation.

# **CALCULATOARE NUMERICE 2**

## cursul 2

# Banda de asamblare (pipeline)

- Pentru o bandă de asamblare ideală trebuie:
  - Ca toate obiectele să treacă prin aceleași stagii
  - Între oricare 2 stagii nu este permisă partajarea resurselor
  - Întârzierea de propagare prin toate stagiile este aceeași
  - Tranziția de intrare în banda de asamblare nu este afectată de tranziția între alte stagii

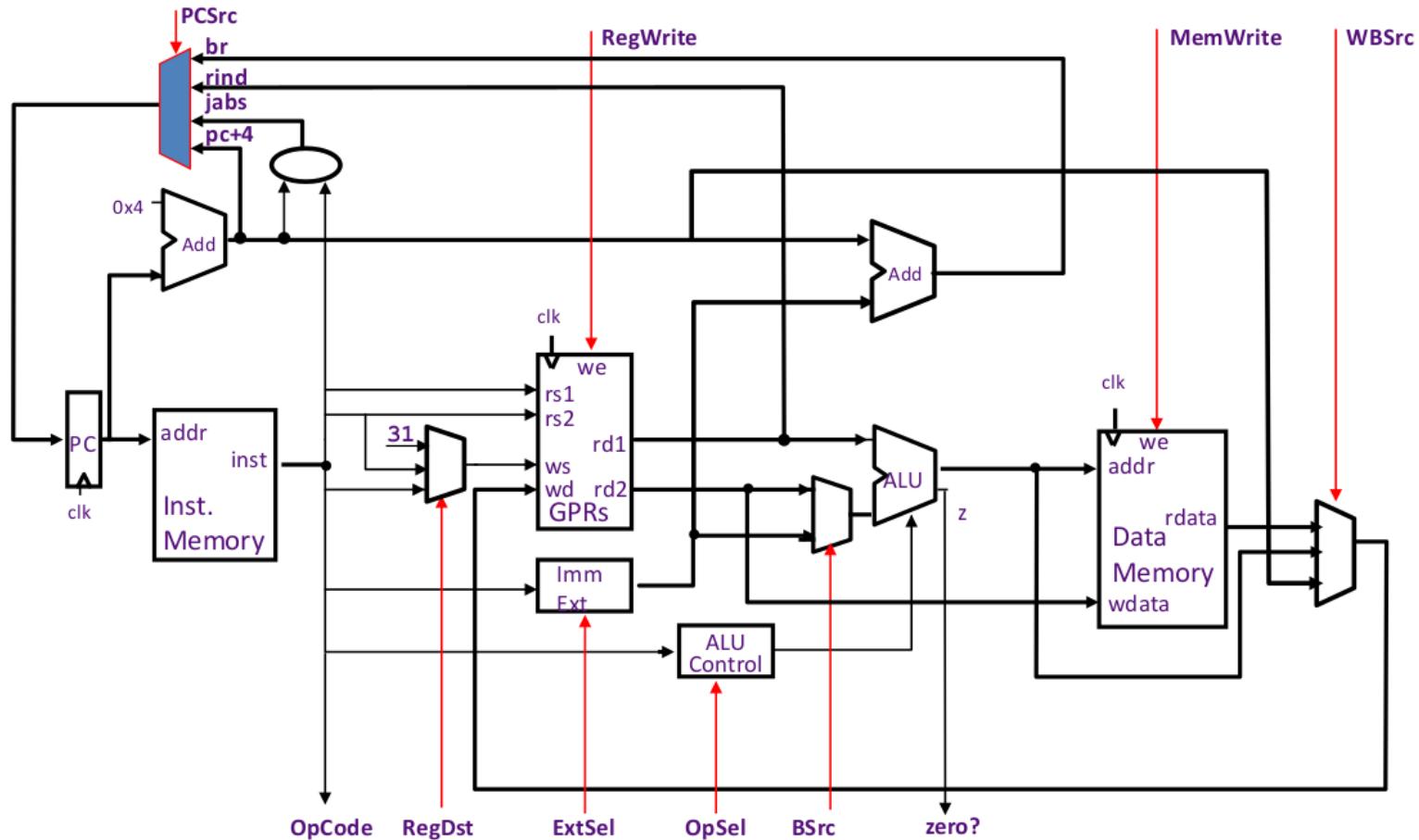


# Banda de asamblare (pipeline)

Toate aceste principii sunt valabile pentru o bandă de asamblare folosită pentru producerea de bunuri (exp: banda de asamblare pentru producerea de mașini)

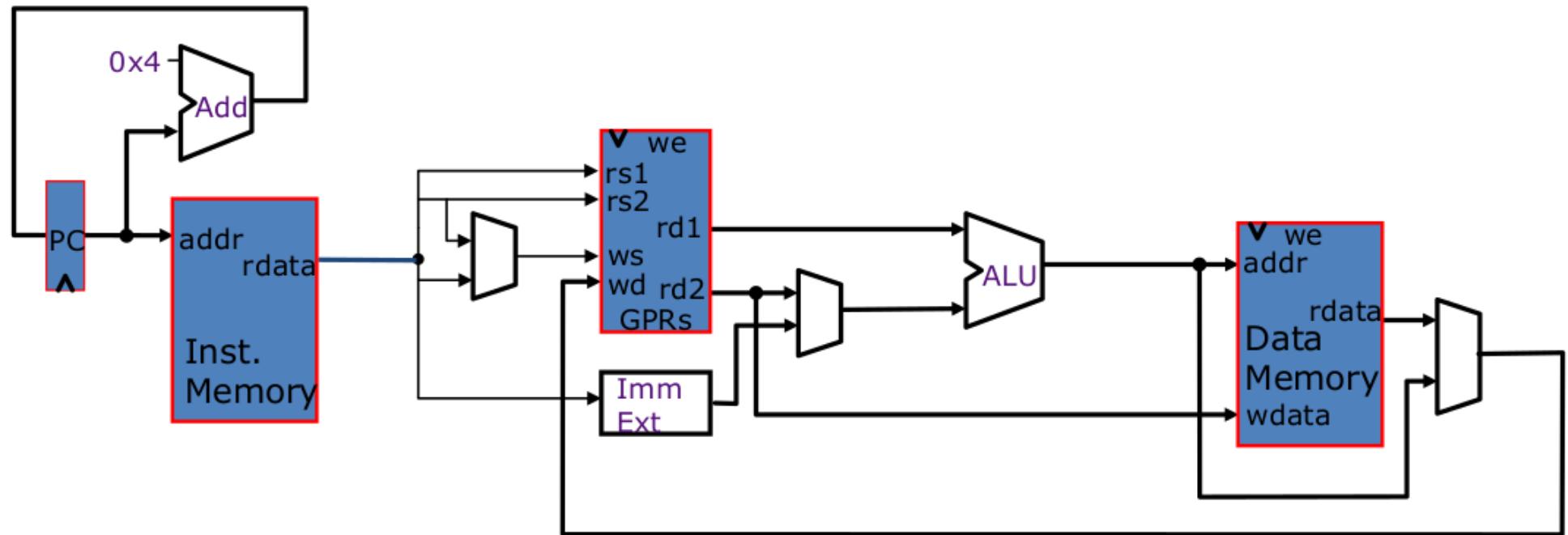
Problema la banda de asamblare folosită în calculatoare este **dependența instrucțiunilor una de alta** (ceea ce conduce la apariția hazardurilor)

# Calea de date MIPS fără BA

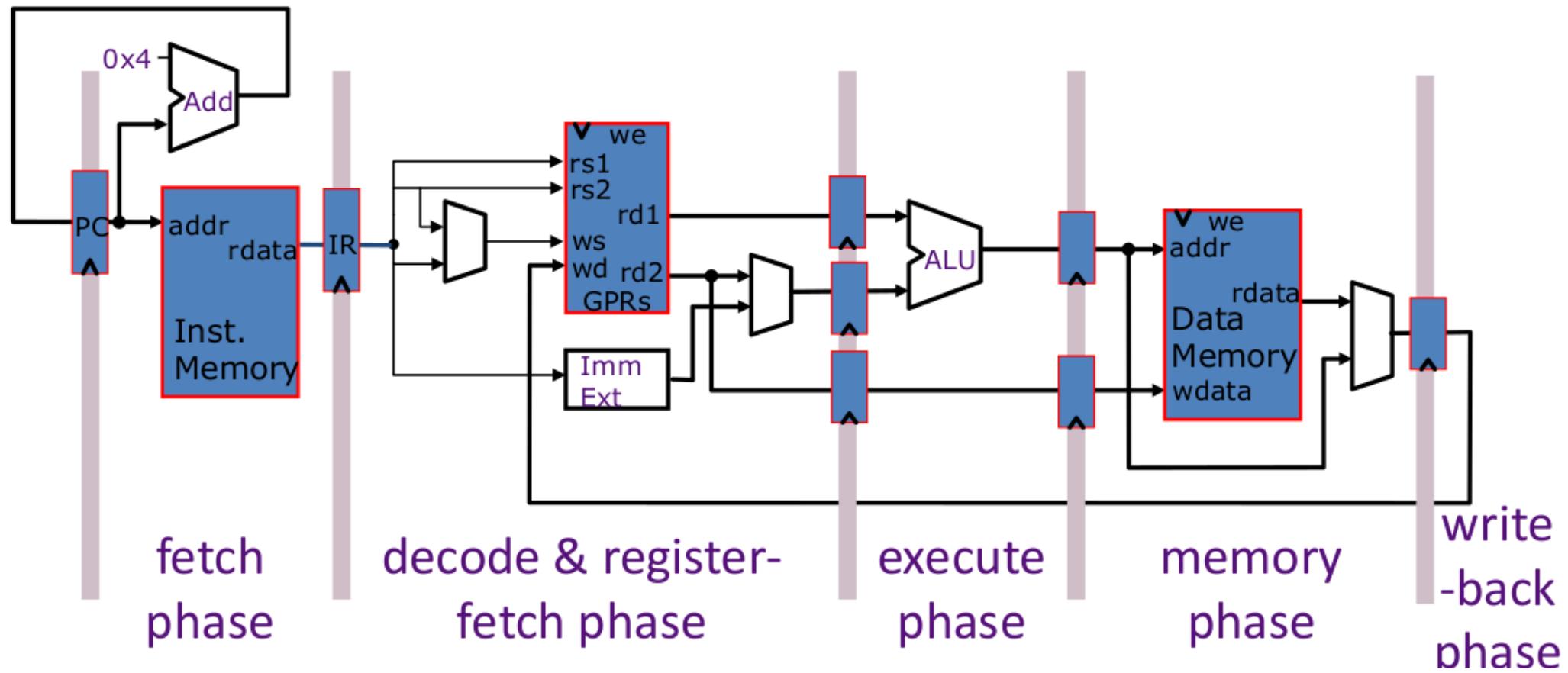


Fiecare instrucție se execută într-un singur ciclu de ceas

# Calea de date MIPS simplificată fără BA

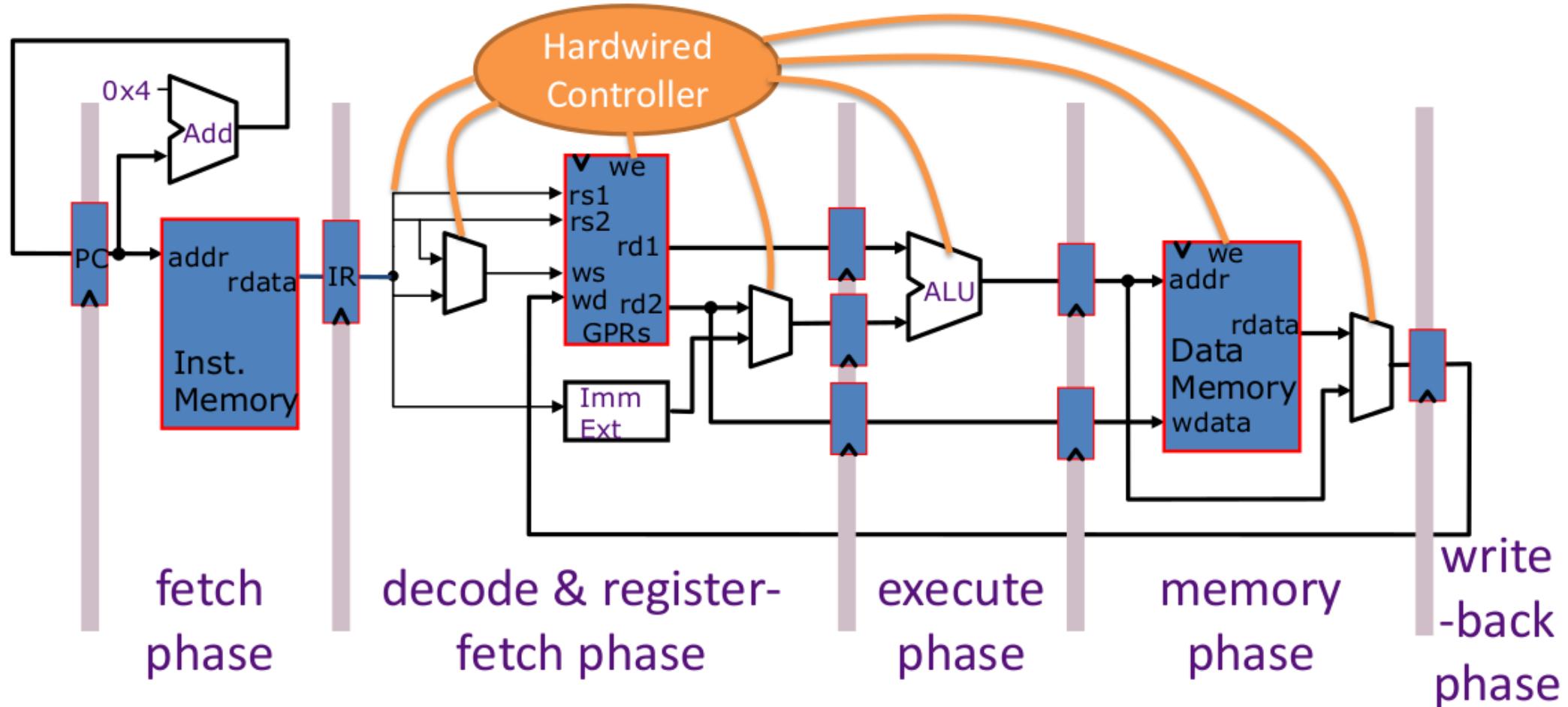


# Calea de date MIPS cu BA

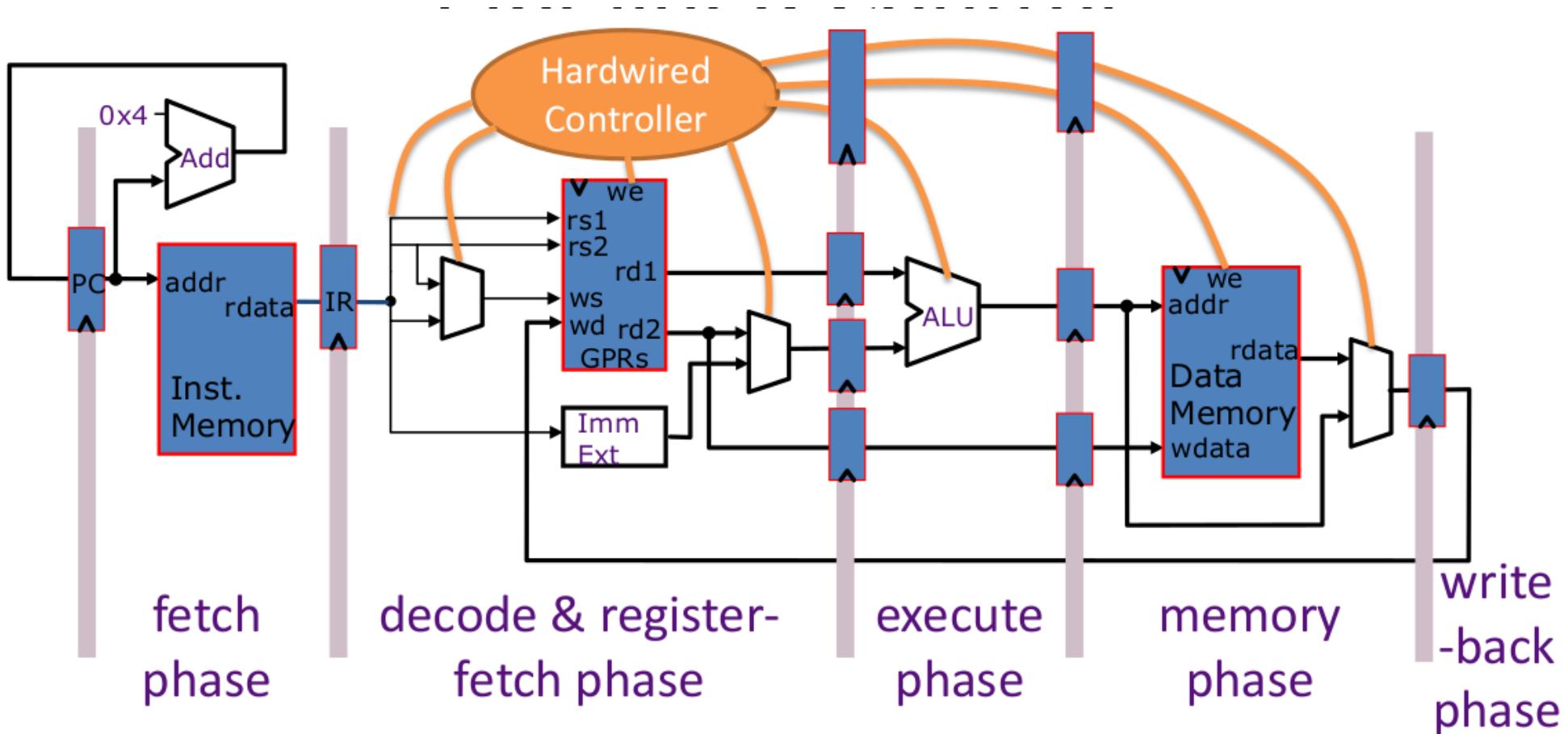


Perioada ceasului se reduce prin executarea instrucțiunii în mai mulți ciclii de ceas

# Controlul



# Controlul



Prin introducerea controlului putem executa instrucțiuni în paralel iar la diferite momente de timp vom executa anumiți pași dintr-o instrucțiune

# Determinarea performanței unui procesor

Performanța unui procesor este dată de legea "Iron"

$$\frac{\text{Timp}}{\text{Program}} = \frac{\text{Instrucțiuni}}{\text{Program}} * \frac{\text{Ciclii}}{\text{Instrucțiune}} * \frac{\text{Timp}}{\text{Ciclu}}$$

Depinde de codul sursă, compilator, ISA

CPI depinde de ISA și microarhitectură

Depinde de microarhitectură și de tehnologia de bază

- Scopul nostru este cel de minimizare al celor 3 factori pentru a putea crește performanța.
- ***CPI = cycles per instruction***
- CPI = 1 și ciclu scurt în cazul BA
- CPI = 1 dar ciclu lung în cazul fără BA

# Determinarea performanței unui procesor

Care micro-arhitectură este mai rapidă ?

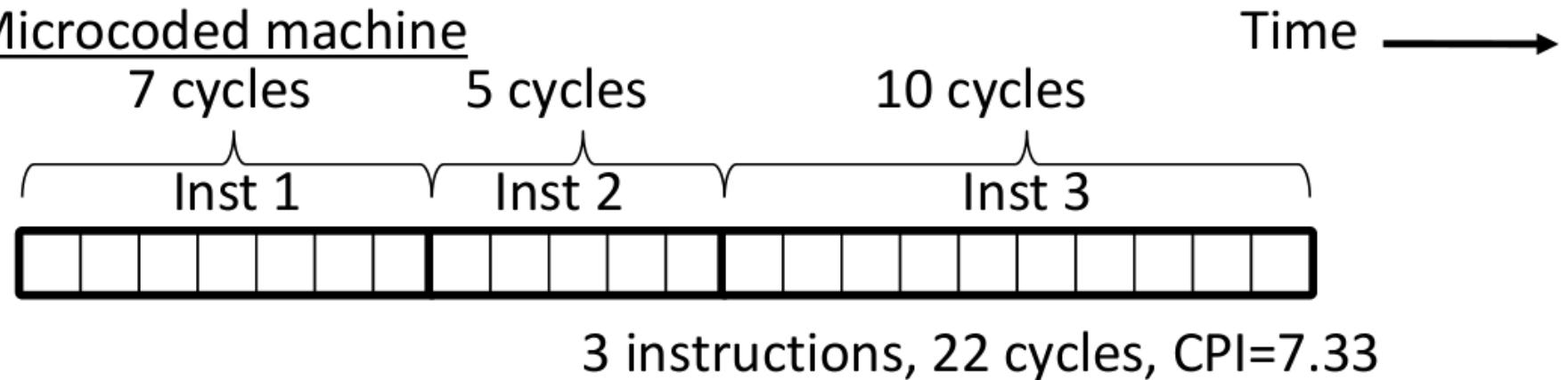
Micro-arhitectură multi-ciclu control fără BA, CPI > 1 și  
timpul de ciclu mic

sau

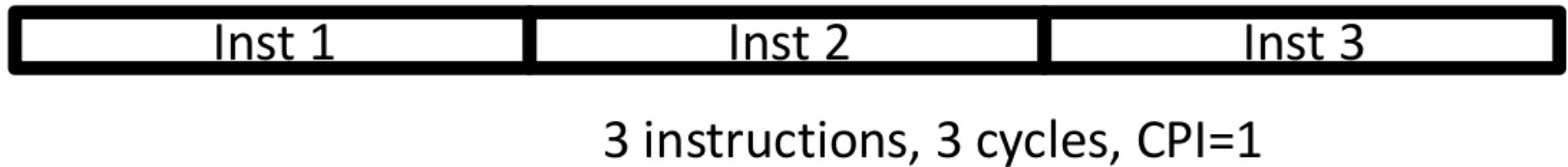
Micro-arhitectură cu BA, CPI = 1 și timpul de ciclu mic

# Exemple

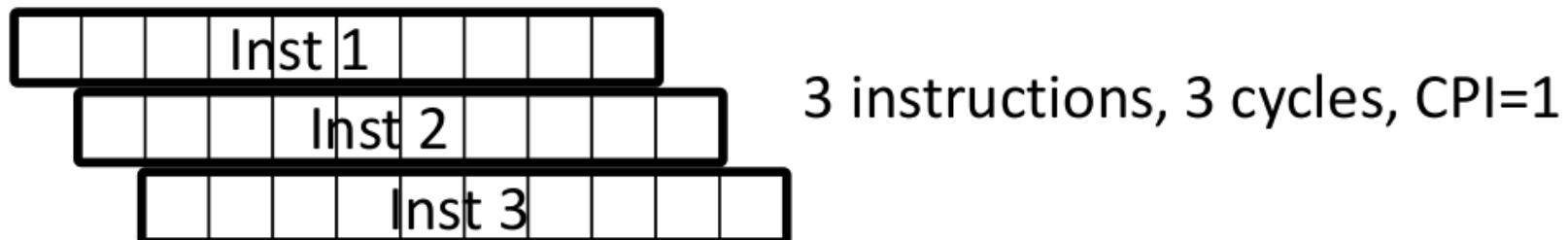
## Microcoded machine



## Unpipelined machine

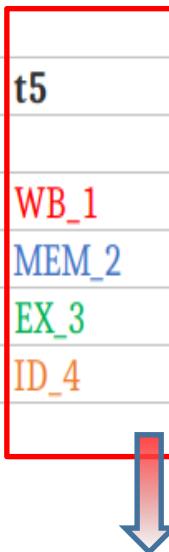


## Pipelined machine



# Diagrame de reprezentare în BA

Timp	t1	t2	t3	t4	t5	t6	t7	t8	.....
Instr_1	IF_1	ID_1	EX_1	MEM_1	WB_1				
Instr_2		IF_2	ID_2	EX_2	MEM_2	WB_2			
Instr_3			IF_3	ID_3	EX_3	MEM_3	WB_3		
Instr_4				IF_4	ID_4	EX_4	MEM_4	WB_4	



Se poate determina în fiecare ciclu de ceas ce se execută în BA

Foarte utilă această posibilitate în determinarea hazardurilor care pot apărea în funcționarea non-ideală a unei BA

# Tipuri de hazarduri

Hazard structural – două instrucțiuni necesită aceeași resursă hardware la același moment de timp

Hazard de date – o instrucțiune depinde de o valoare / dată produsă de o instrucțiune anterioară

Hazard de control – o instrucțiune ce trebuie executată depinde de decizia ce trebuie luată în cadrul unei instrucțiuni anterioare

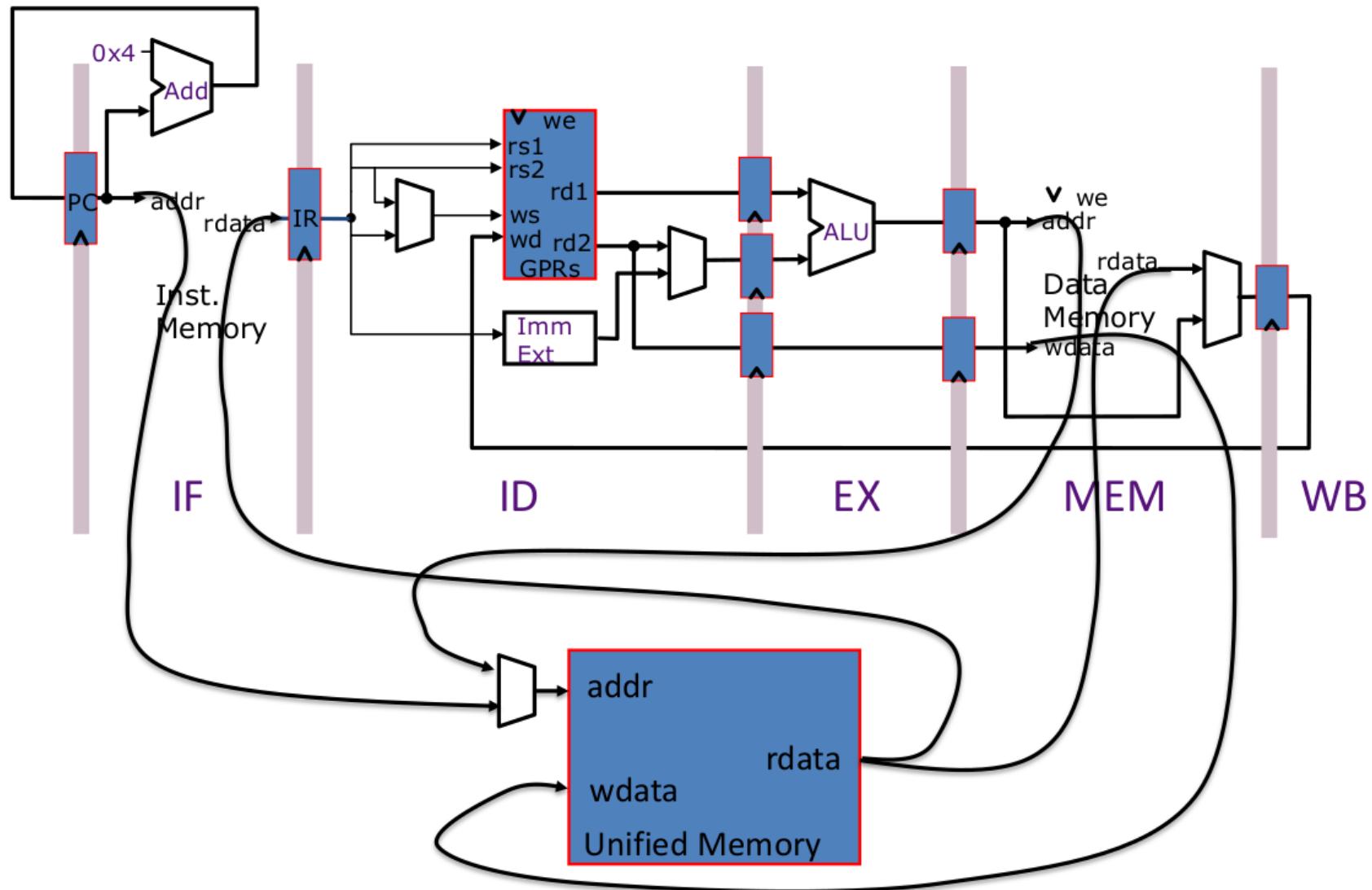
# Hazard structural - soluționare

Programatic – compilatorul nu programează spre execuție instrucțiuni care pot crea hazard structural

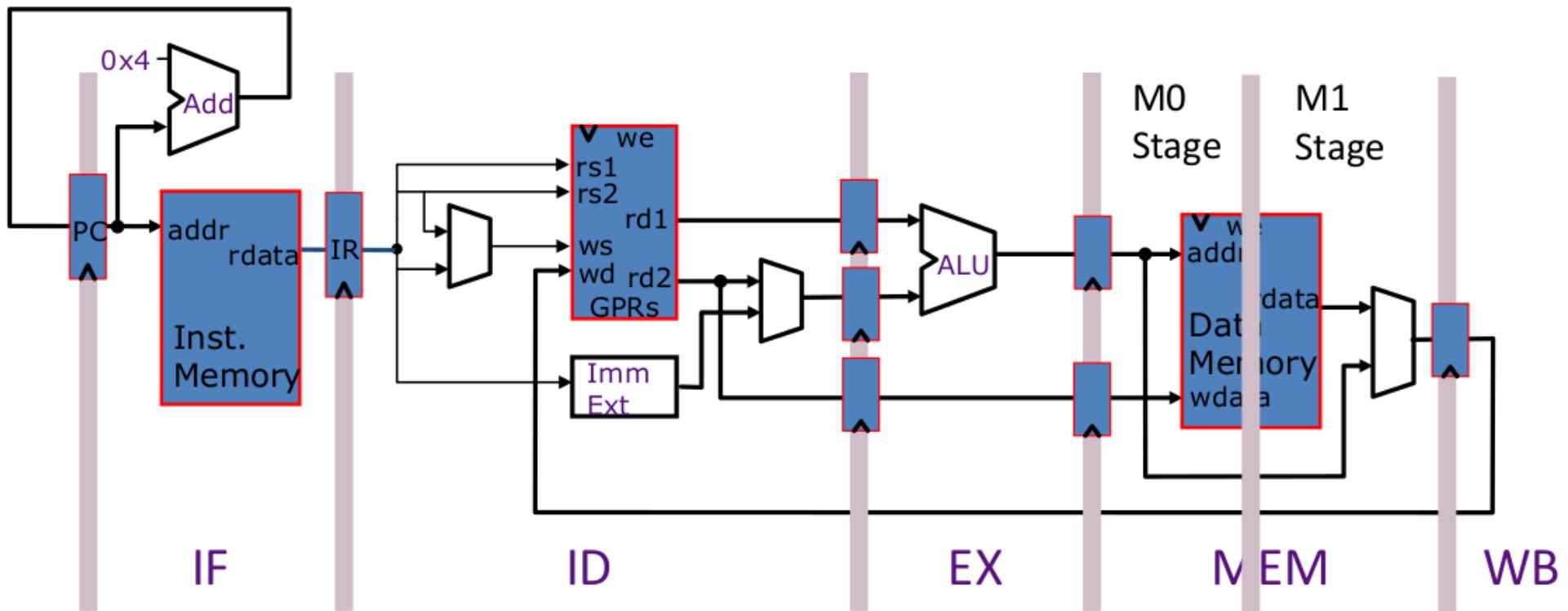
Stall – hardware-ul include logica de control care oprește execuția curentă până când instrucțiunea anterioară nu mai utilizează resursa generatoare de hazard

Duplicare – duplicarea hardware-ului astfel încât fiecare instrucțiune utilizează propriile resurse hardware la același moment de timp

# Hazard structural - exemplu



# Hazard structural - exemplu



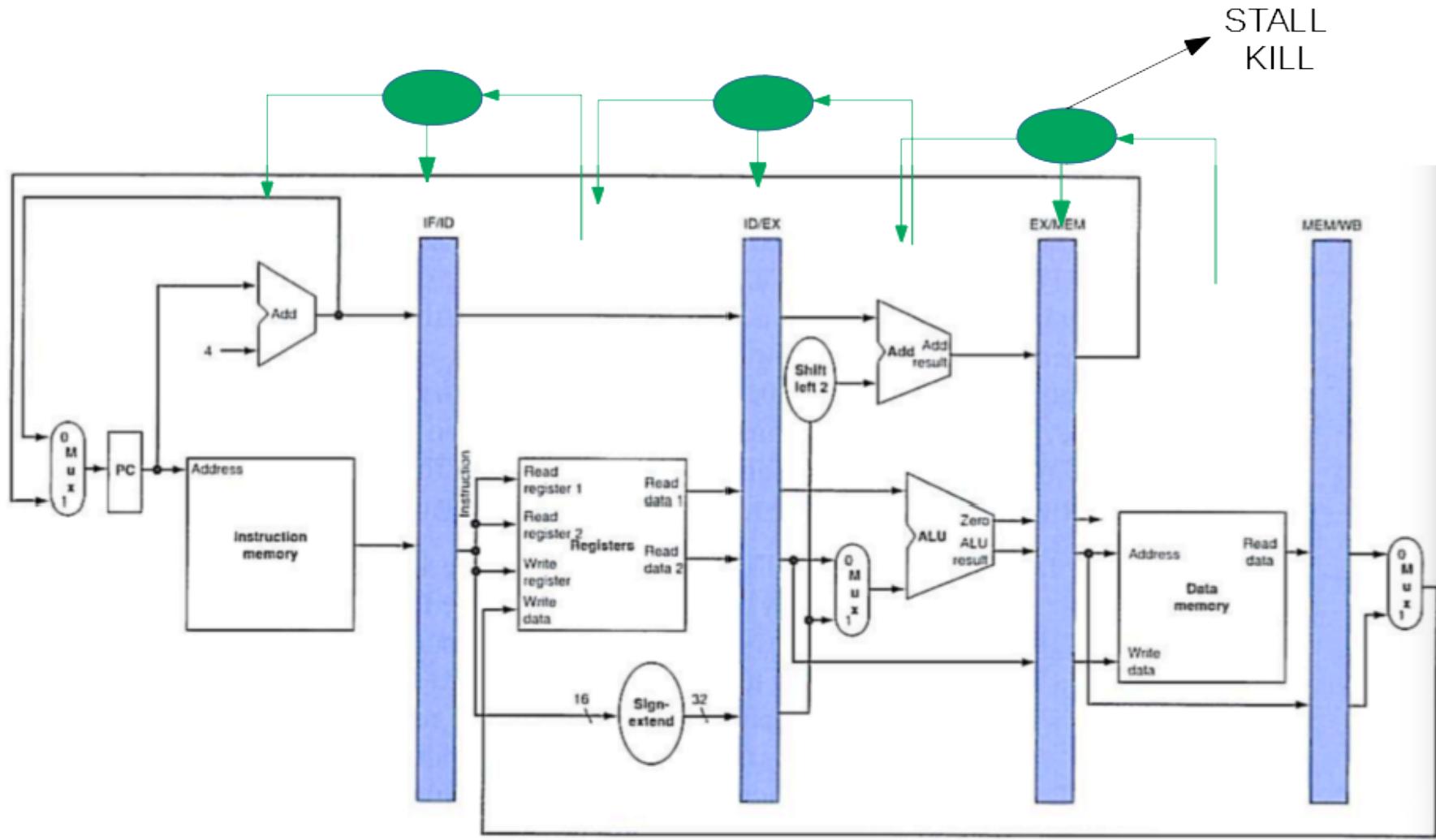
# Hazard de date

- Instrucțiunea planificată spre execuție nu poate fi executată într-unul din ciclurile de ceas ale BA deoarece datele necesare execuției sale nu sunt încă disponibile.
- Soluționarea acestui tip de hazard se poate face prin:
  - Planificare
  - STALL
  - Bypass – calea de date permite valorilor de a fi trimise către un stagiu anterior înainte ca instrucțiunea precedentă să părăsească BA
  - Specularea – se presupune că nu există probleme. În caz contrar, se șterge instrucțiunea speculativă și restart.

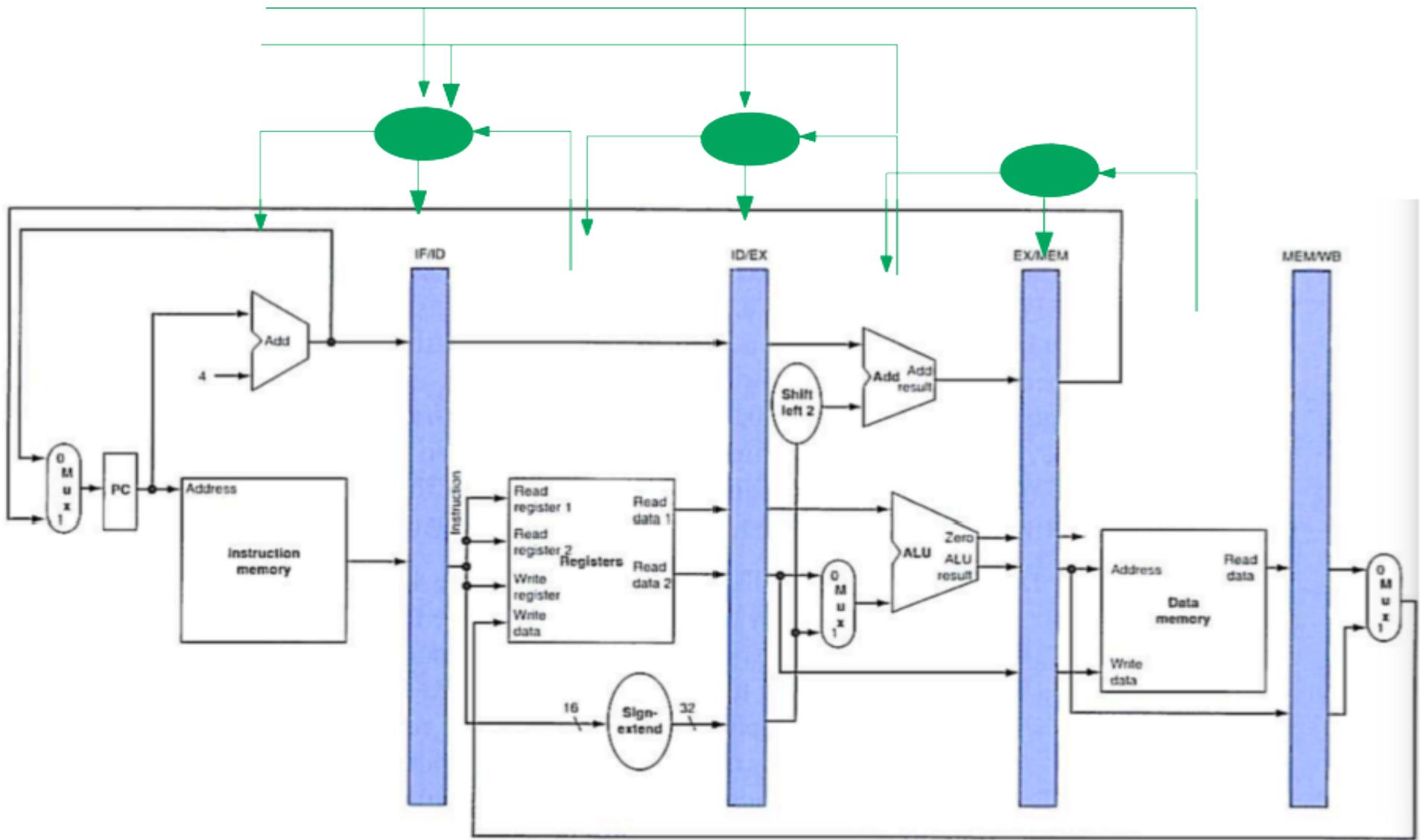
# Hazard de date - exemplu

- ADDI R1, R0, #10
- ADDI R4, R1, #17
- Avem 2 instrucțiuni de adunare cu o valoare imediată – adunăm la registrul R0 valoarea 10 și rezultatul se va salva în R1
- Avem un hazard de tipul RAW
- Problema este că rezultatul primului ADD nu este în R1 când al doilea ADD citește valoarea lui R1.

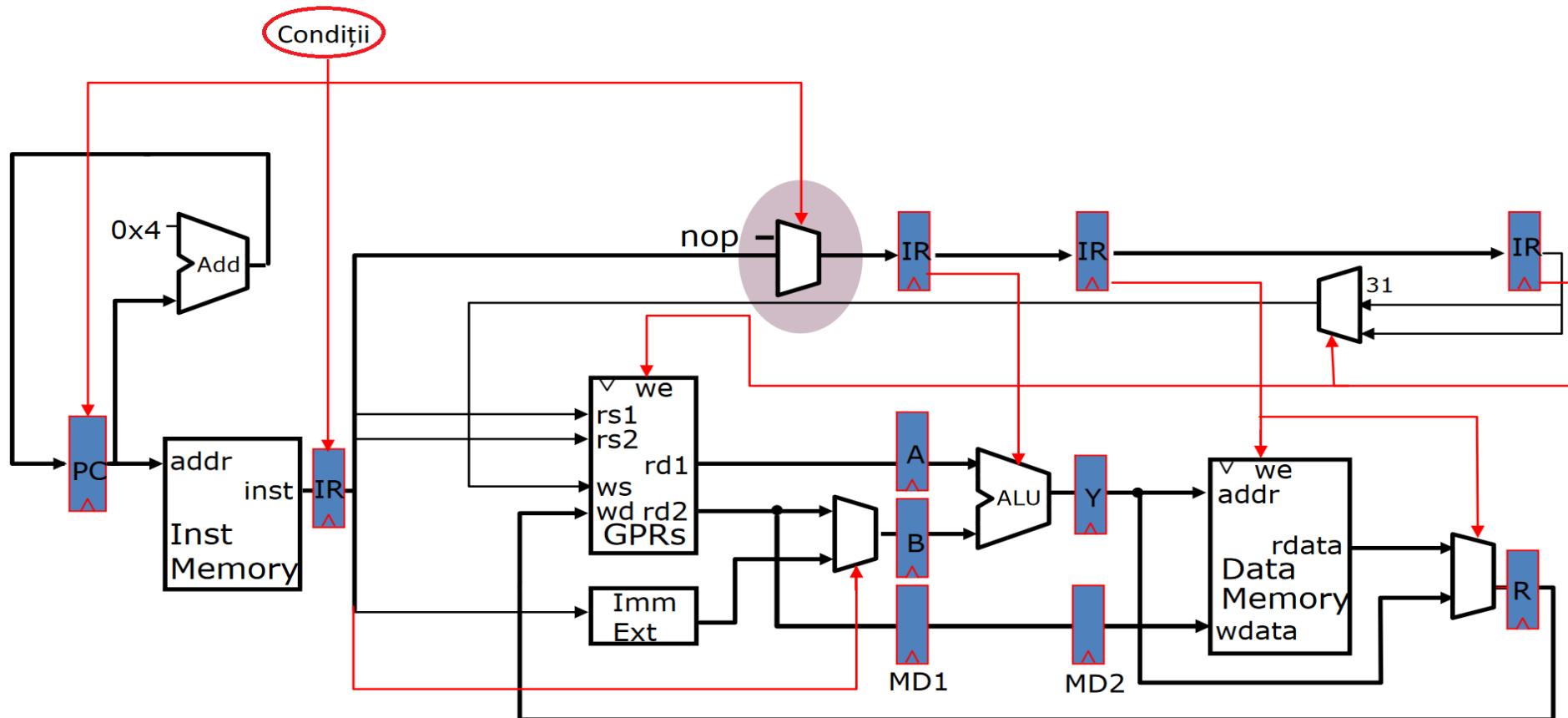
# Hazard de date – o primă soluție



# Hazard de date – soluția optimă



# Hazard de date – introducerea de STALL-uri



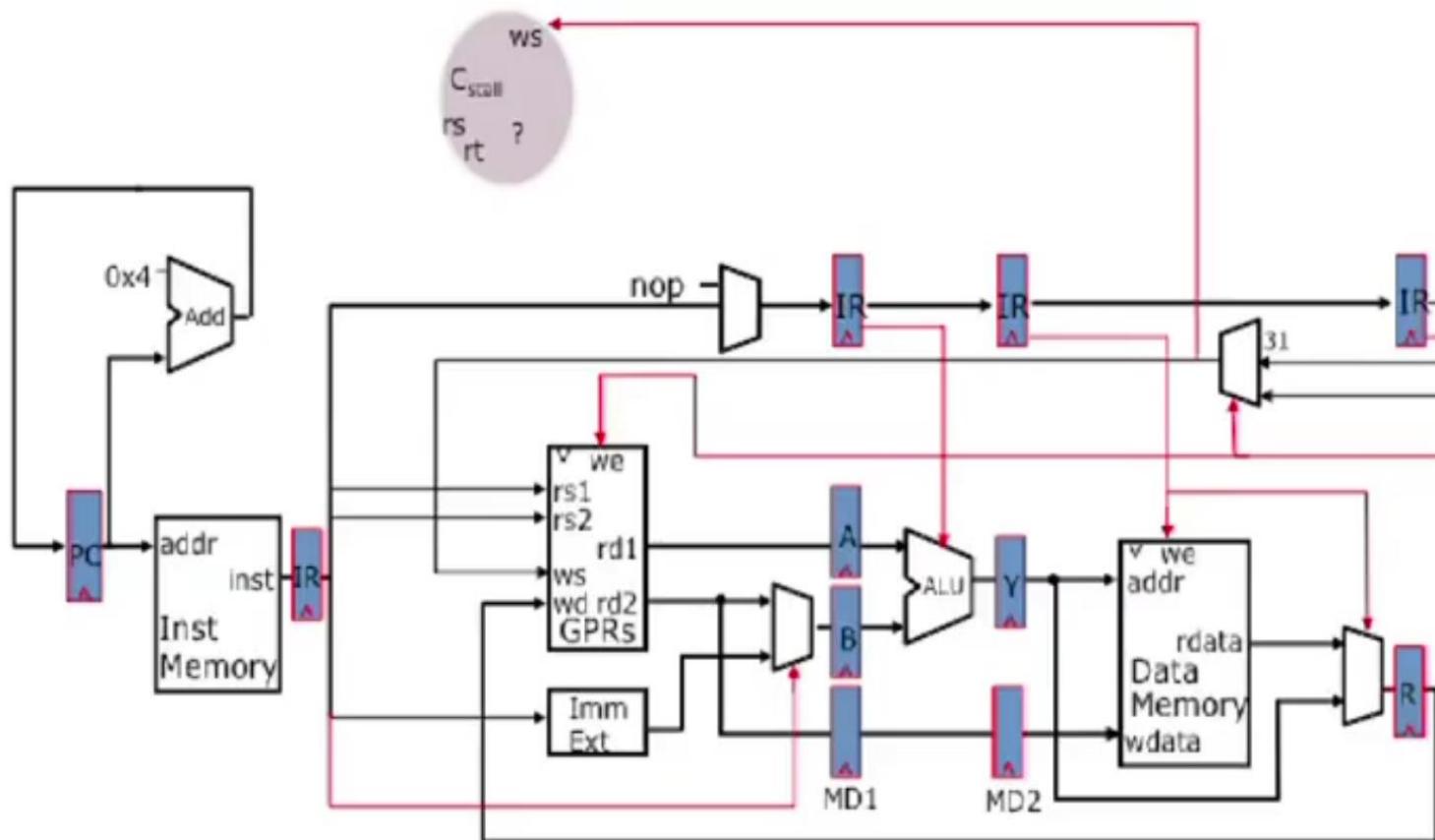
Câte NOP-uri trebuie introduse pentru a putea elimina hazard-ul ?

Cum va arăta execuția programului în BA ?

## O primă concluzie

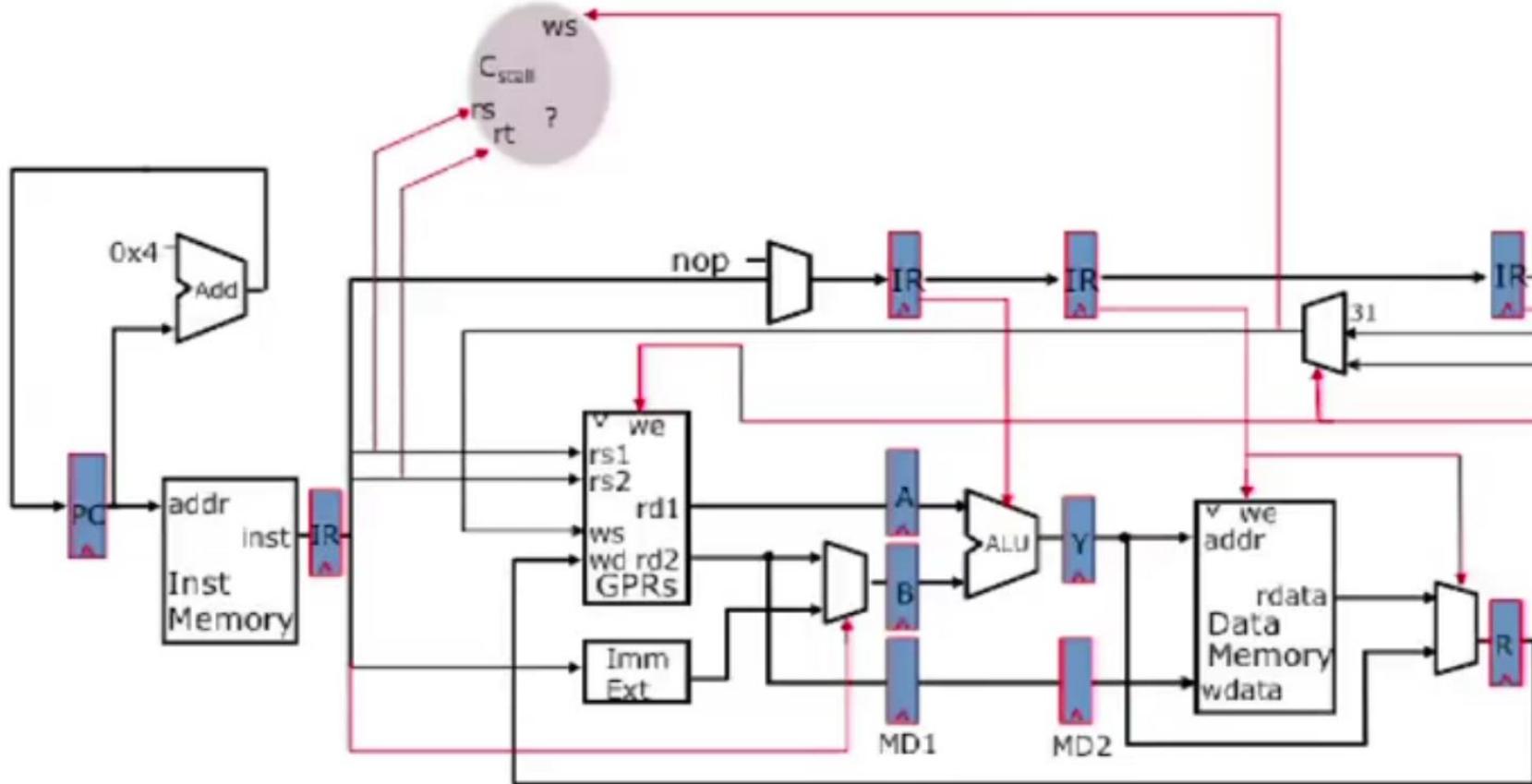
Va trebui să comparăm registrele sursă ale instrucțiunii din stagiu ID cu registrele destinație ale instrucțiunilor neangajate încă.

# Determinarea semnalului de control $C_{STALL}$



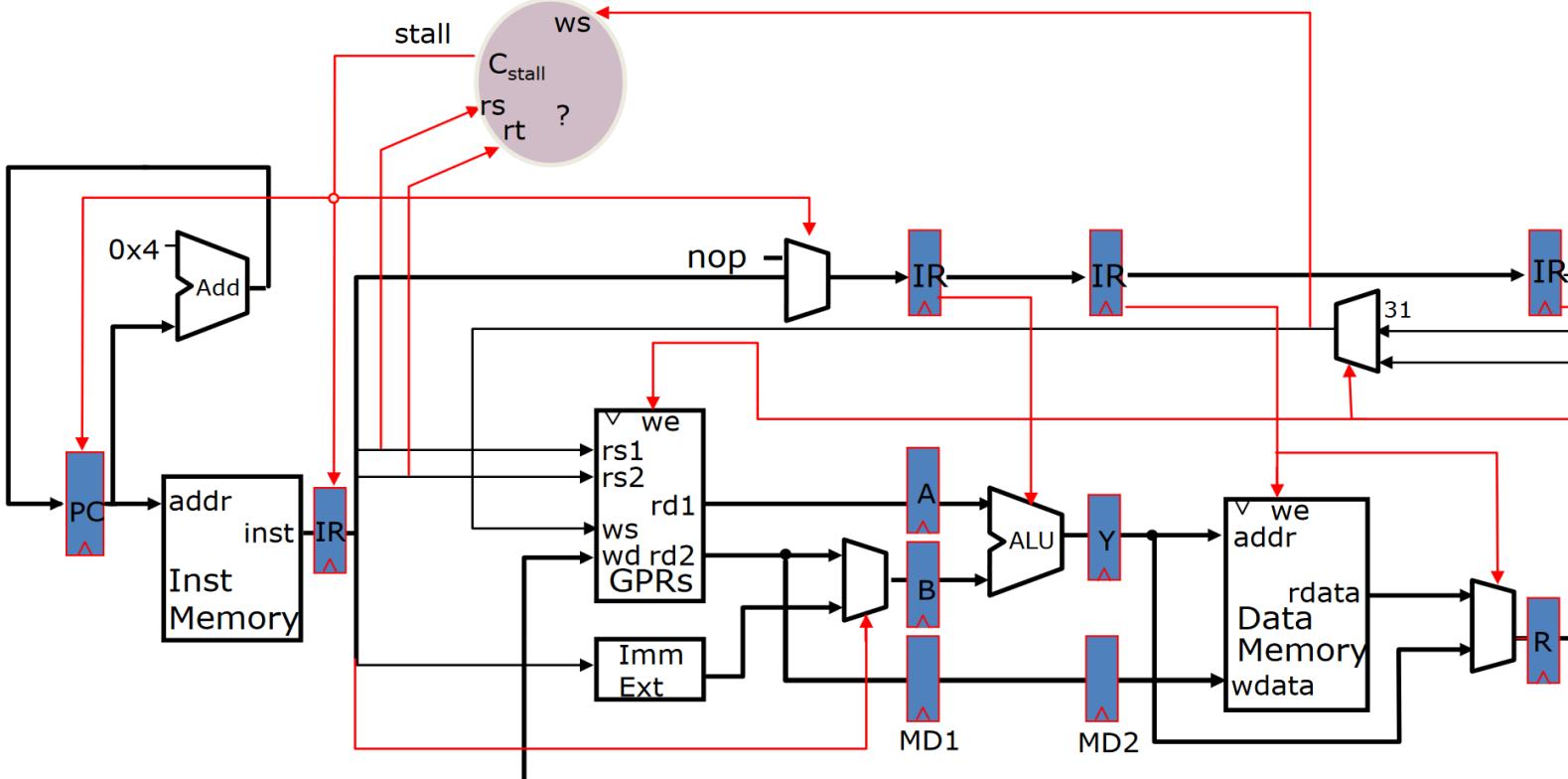
Primul lucru care trebuie verificat – destinația operandului

# Determinarea semnalului de control $C_{STALL}$



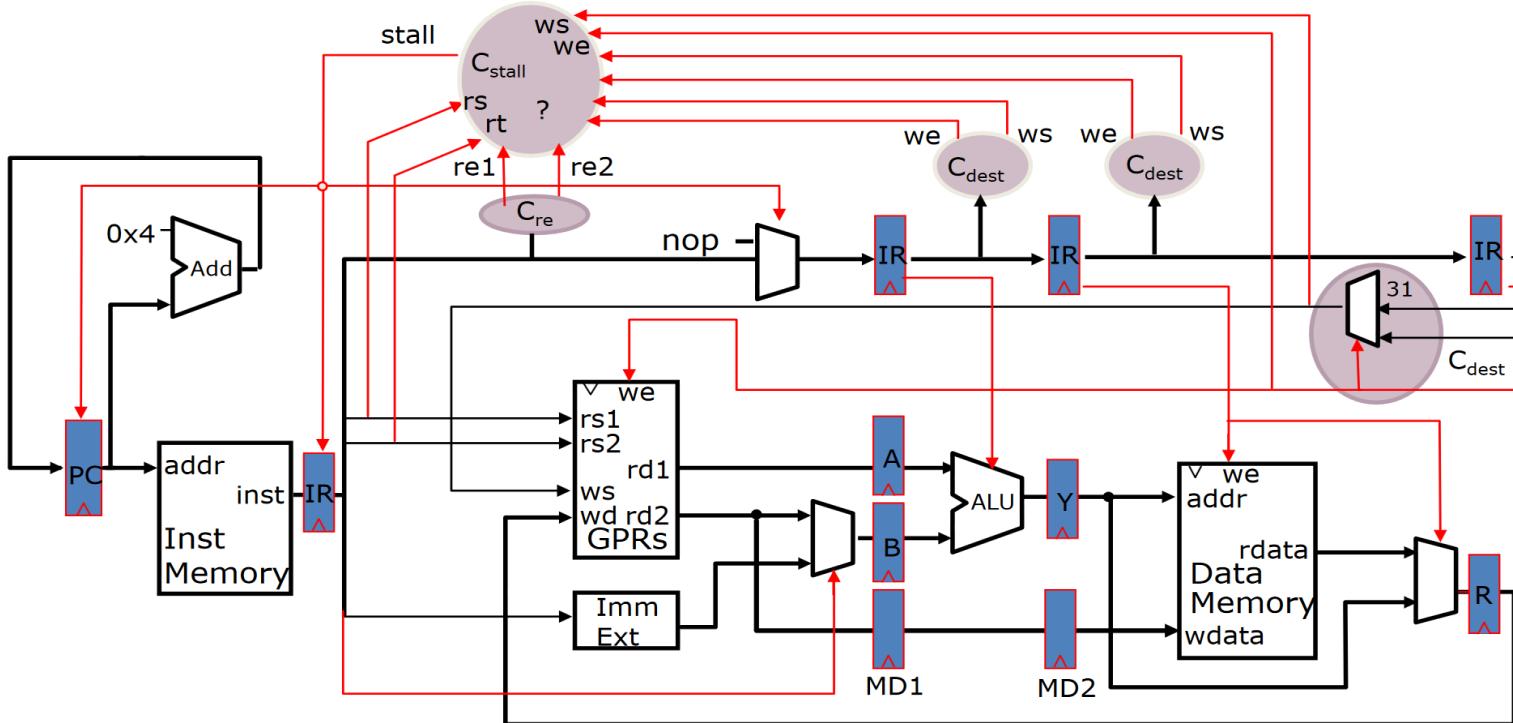
Se compară destinația operandului cu cele două surse de intrare.

# Determinarea semnalului de control $C_{STALL}$



- Ce am realizat până acum ?
- Putem spune STALL oricărei instrucțiuni ce este faza IF a BA-ului și inseră instrucțiuni no op (NOP-uri) în BA în stagiile următoare.

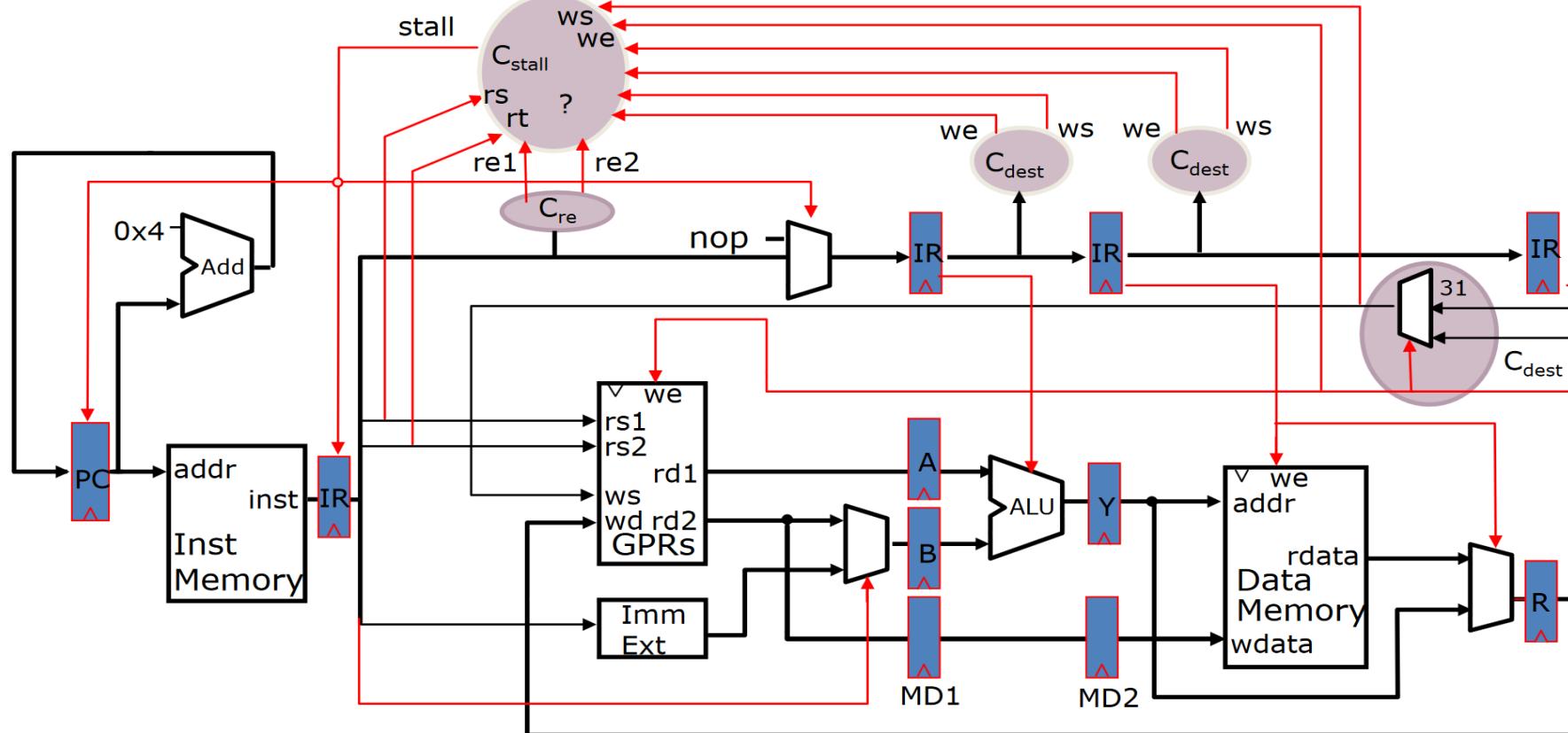
# Determinarea semnalului de control $C_{STALL}$



- Calcularea acestor biți se face în stagiul 2 (ID) al benzii de asamblare și vor fi transferați apoi prin stagiile următoare mai exact stagiile 3 și 4.
- Abia acum putem spune cu certitudine că atunci când registrele din ID se potrivesc cu cele din WB vom avea un STALL.

# Determinarea semnalului de control $C_{STALL}$

- De ce este MUX-ul din stagiul WB încircuit ?



- Solutia finală ignorând instrucțiunile de jump și branch.

# Final pentru semnalul de control $C_{STALL}$

		<i>source(s)</i>	<i>destination</i>	
ALU	$rd \leftarrow (rs) \text{ func } (rt)$	rs, rt	rd	
ALUI	$rt \leftarrow (rs) \text{ op immediate}$	rs	rt	
LW	$rt \leftarrow M [(rs) + \text{immediate}]$	rs	rt	
SW	$M [(rs) + \text{immediate}] \leftarrow (rt)$	rs, rt		
BZ	<i>cond</i> ( $rs$ ) <i>true</i> : $PC \leftarrow (PC) + \text{immediate}$ <i>false</i> : $PC \leftarrow (PC) + 4$	rs		
J	$PC \leftarrow (PC) + \text{immediate}$	rs		
JAL	$r31 \leftarrow (PC), PC \leftarrow (PC) + \text{immediate}$		31	
JR	$PC \leftarrow (rs)$	rs		
JALR	$r31 \leftarrow (PC), PC \leftarrow (rs)$	rs	31	

$C_{dest}$   
 $ws = \text{Case opcode}$   
 ALU  $\Rightarrow rd$   
 ALUi, LW  $\Rightarrow rt$   
 JAL, JALR  $\Rightarrow R31$

$we = \text{Case opcode}$   
 ALU, ALUi, LW  $\Rightarrow (ws \neq 0)$   
 JAL, JALR  $\Rightarrow on$   
 ...  $\Rightarrow off$

$C_{re}$   
 $re1 = \text{Case opcode}$

ALU, ALUi,  
 LW, SW, BZ,  
 JR, JALR  
 J, JAL

$\Rightarrow on$   
 $\Rightarrow off$

$re2 = \text{Case opcode}$

ALU, SW  
 ...

2

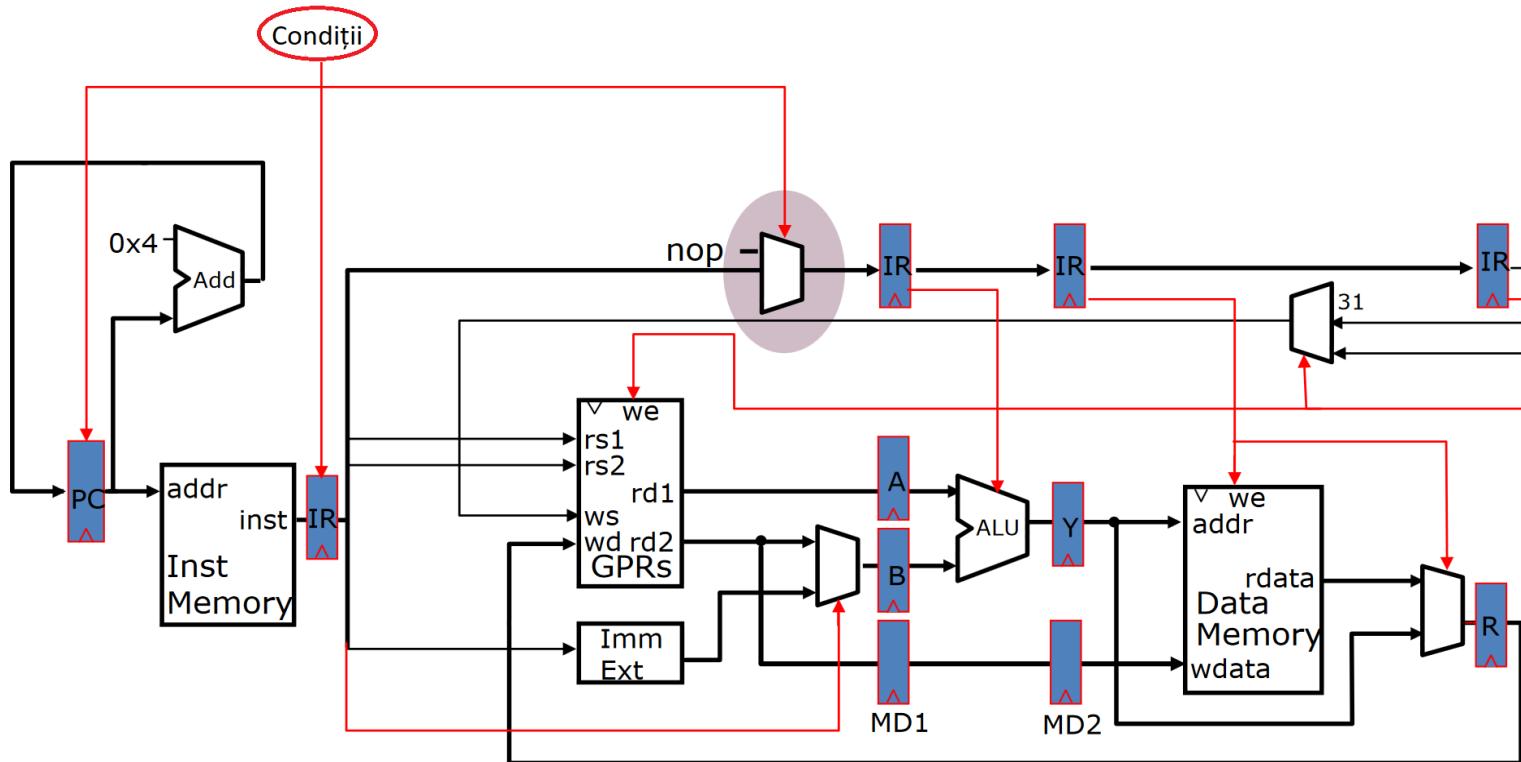
$C_{stall}$

$$\text{stall} = ((rs_D = ws_E).we_E + (rs_D = ws_M).we_M + (rs_D = ws_W).we_W) . re1_D + ((rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W) . re2_D$$

1

3

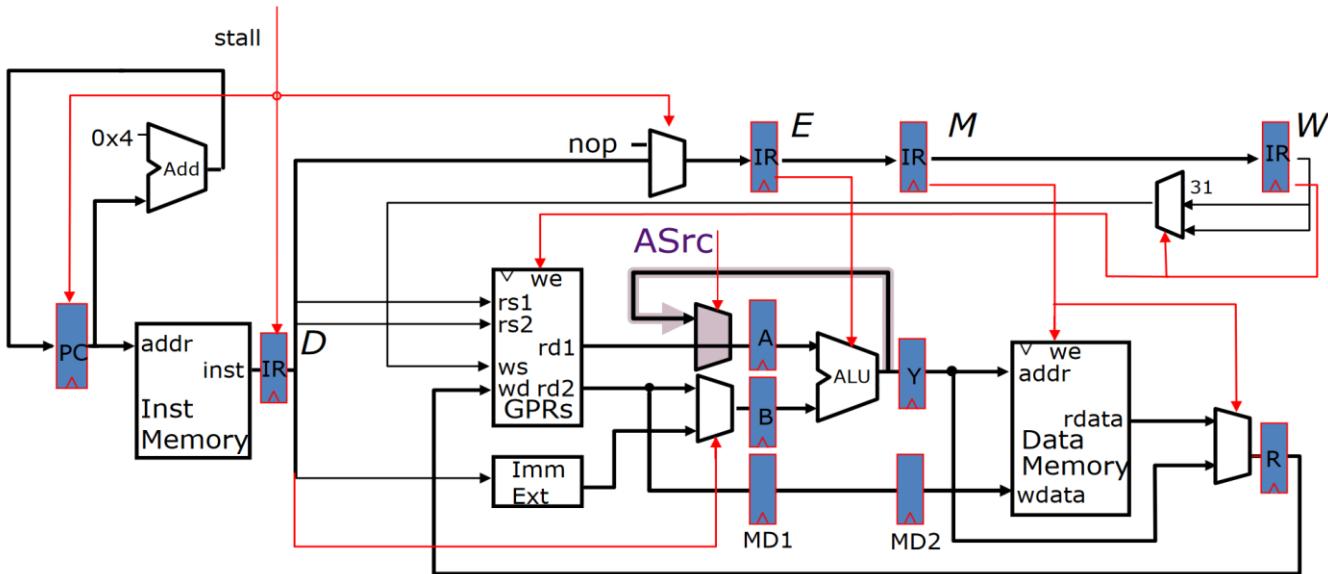
# Hazarduri pentru LOAD și STORE



Mem [Regs[r1] + 7 ] <- Regs [r2]  
Regs[r4] <- Mem [Regs[r3] + 5 ]

Ce se întâmplă dacă Regs[r1] + 7 == Regs[r3] + 5 ?

# BYPASS



Când ne ajută această metodă ?

$$\begin{aligned} r1 &<- r0 + 10 \\ r4 &<- r1 + 17 \end{aligned}$$

$$\begin{aligned} r1 &<- \text{Mem}[r0 + 10] \\ r4 &<- r1 + 17 \end{aligned}$$

$$\begin{aligned} \text{JAL } 500 \\ r4 &<- r31 + 17 \end{aligned}$$

# Noile ecuații de control

$$\text{stall} = (\cancel{((rs_D = ws_E).we_E + (rs_D = ws_M).we_M + (rs_D = ws_W).we_W).re1_D} + ((rt_D = ws_E).we_E + (rt_D = ws_M).we_M + (rt_D = ws_W).we_W).re2_D)$$

$ws = \text{Case opcode}$

ALU	$\Rightarrow rd$
ALUi, LW	$\Rightarrow rt$
JAL, JALR	$\Rightarrow R31$

$we = \text{Case opcode}$

ALU, ALUi, LW	$\Rightarrow (ws \neq 0)$
JAL, JALR	$\Rightarrow \text{on}$
...	$\Rightarrow \text{off}$

Controlul pentru multiplexorul nou introdus     $ASrc = (rs_D = ws_E).we_E.re1_D$

# Noile ecuații de control rescrise

$\text{we-bypass}_E = \begin{cases} \text{Case opcode}_E \\ \text{ALU, ALUi} \Rightarrow (ws \neq 0) \\ \dots \Rightarrow \text{off} \end{cases}$

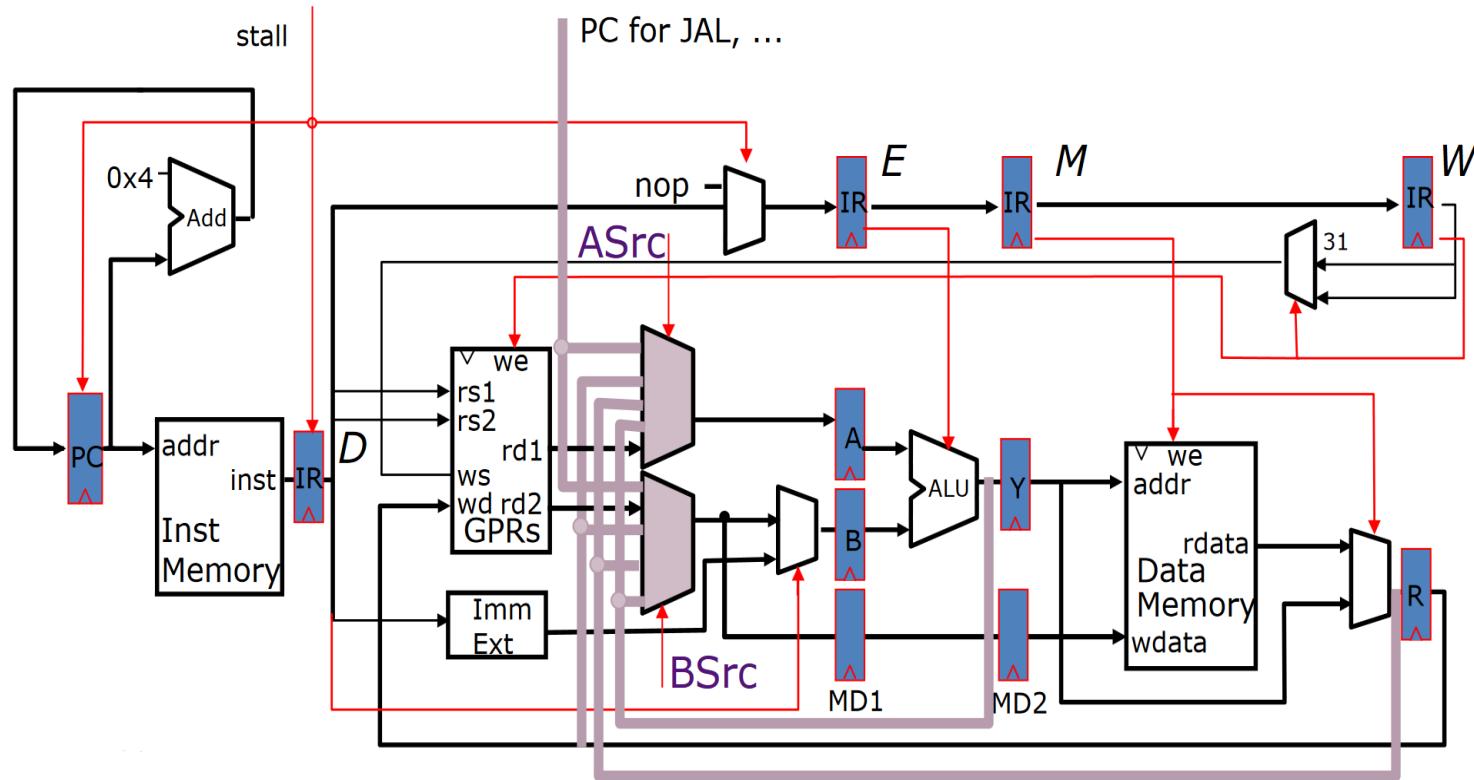
$\text{we-stall}_E = \begin{cases} \text{Case opcode}_E \\ \text{LW} \Rightarrow (ws \neq 0) \\ \text{JAL, JALR} \Rightarrow \text{on} \\ \dots \Rightarrow \text{off} \end{cases}$

$$\text{ASrc} = (rs_D = ws_E). \text{we-bypass}_E . re1_D$$

$$\begin{aligned} \text{stall} = & ((rs_D = ws_E). \text{we-stall}_E + \\ & (rs_D = ws_M). we_M + (rs_D = ws_W). we_W). re1_D \\ & + ((rt_D = ws_E) . we_E + (rt_D = ws_M) . we_M + (rt_D = ws_W) . we_W) . re2_D \end{aligned}$$

# BYPASS – varianta finală

## Datapath complet cu bypass



$$\begin{aligned}
 \text{stall} = & (rs_D = ws_E) \cdot (\text{opcode}_E = LW_E) \cdot (ws_E \neq 0) \cdot re1_D \\
 & + (rt_D = ws_E) \cdot (\text{opcode}_E = LW_E) \cdot (ws_E \neq 0) \cdot re2_D
 \end{aligned}$$

# Hazarduri de control - PC-ul

Cum calculăm următorul PC ?

- Jump
  - Opcode
  - Offset
  - PC
- Jump Register
  - Opcode
  - Valoarea registrului
- Salturi condiționate
  - Opcode
  - PC
  - Registrul – pentru condiție
  - Offset

Restul instrucțiunilor au nevoie doar de opcode și PC

# Hazarduri de control

	t0	t1	t2	t3	t4	t5	t6	t7	....
(I <sub>1</sub> ) $r1 \leftarrow (r0) + 10$	IF <sub>1</sub>	ID <sub>1</sub>	EX <sub>1</sub>	MA <sub>1</sub>	WB <sub>1</sub>				
(I <sub>2</sub> ) $r3 \leftarrow (r2) + 17$		IF <sub>2</sub>	IF <sub>2</sub>	ID <sub>2</sub>	EX <sub>2</sub>	MA <sub>2</sub>	WB <sub>2</sub>		
(I <sub>3</sub> )			IF <sub>3</sub>	IF <sub>3</sub>	ID <sub>3</sub>	EX <sub>3</sub>	MA <sub>3</sub>	WB <sub>3</sub>	
(I <sub>4</sub> )				IF <sub>4</sub>	IF <sub>4</sub>	ID <sub>4</sub>	EX <sub>4</sub>	MA <sub>4</sub>	WB <sub>4</sub>

Introducem NOP cu toate că instrucțiunile ce trebuie executate nu prezintă hazarduri.

**DE CE ?**

# Hazarduri de control - PC-ul

## JUMP

1. Este necesar să ne uităm la opcode pentru a determina dacă este JUMP sau nu
2. Trebuie să ne uităm la offset-ul care vine din instrucțiune precum și la valoarea curentă a PC-ului
3. Decodificatorul va stabili că avem o instrucțiune JUMP, stagiul DECODE din banda de asamblare va indica o instrucțiune JUMP  $\rightarrow$  PC + offset (cel mai probabil operația se va face în ALU, altfel trebuie introdus un sumator)
4. Executăm instrucțiunea următoarea de la adresa calculată

# Hazarduri de control - PC-ul

## JUMP Register

1. Opcode ne va spune dacă avem o instrucțiune de acest tip sau nu
2. Nu este necesară utilizarea offset-ului deoarece sărim direct la valoarea registrului – doar în cazul MIPS (altfel folosim registru indirect sau o instrucțiune de tipul jump register sau chiar o instrucțiune de tipul jump indirect la memorie)

# Hazarduri de control – speculăm PC+4

I1 – 096 ADD

I2 – 100J 304

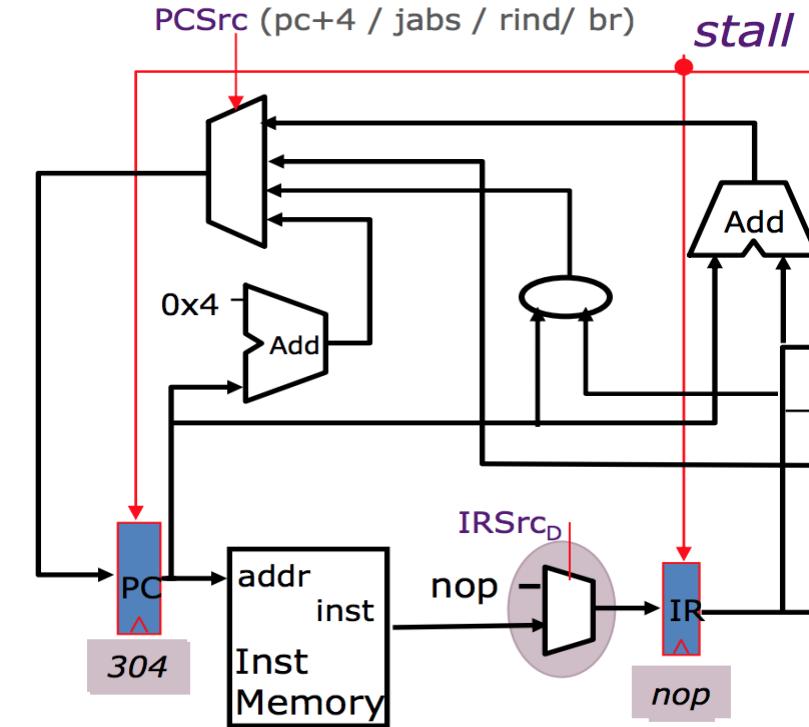
I3 – 104 ADD

I4 – 304 ADD

Trebuie ca instrucțiunea 3 să nu se execute

=»

trebuie introdus un multiplexor care introduce NOP-uri înainte de registrul IR.



$$\begin{aligned} \text{IRSrc}_D &= \begin{cases} \text{Case } \text{opcode}_D \\ \text{J, JAL} \\ \dots \end{cases} \\ &\Rightarrow \text{nop} \\ &\Rightarrow \text{IM} \end{aligned}$$

# Hazarduri de control – speculăm PC+4

I1 – 096ADD

I2 – 100J 304

I3 – 104ADD

I4 – 304ADD

(I<sub>1</sub>) 096: ADD  
(I<sub>2</sub>) 100: J 304  
(I<sub>3</sub>) 104: ADD  
(I<sub>4</sub>) 304: ADD

	t0	t1	t2	t3	t4	t5	t6	t7	...
	IF <sub>1</sub>	ID <sub>1</sub>	EX <sub>1</sub>	MA <sub>1</sub>	WB <sub>1</sub>				
		IF <sub>2</sub>	ID <sub>2</sub>	EX <sub>2</sub>	MA <sub>2</sub>	WB <sub>2</sub>			
		IF <sub>3</sub>	nop	nop	nop	nop			
			IF <sub>4</sub>	ID <sub>4</sub>	EX <sub>4</sub>	MA <sub>4</sub>	WB <sub>4</sub>		

# Hazarduri de control – salturi condiționale

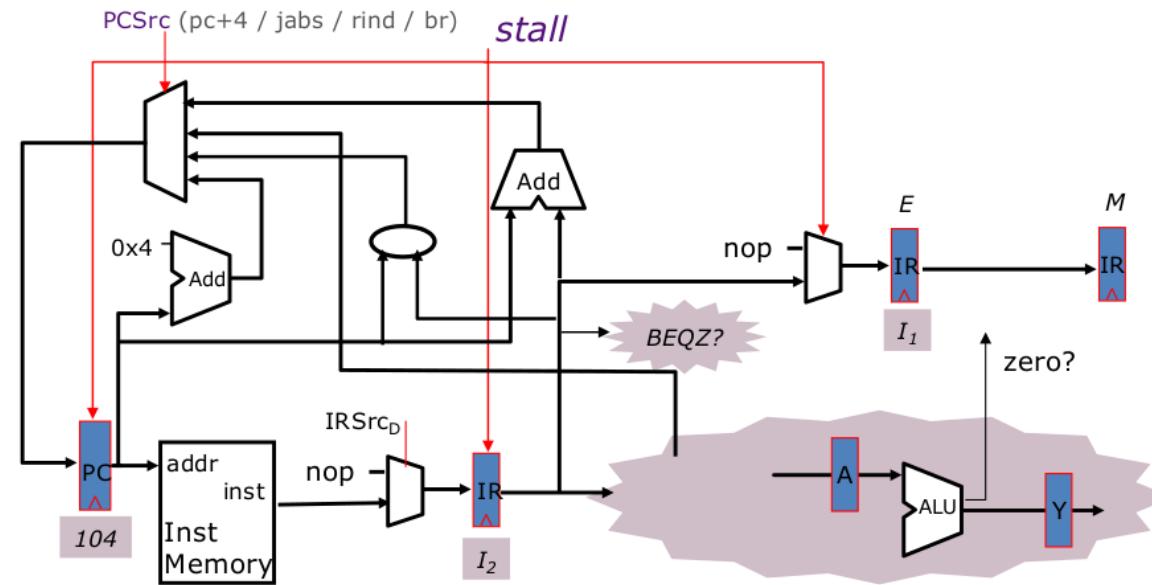
I1 – 096 ADD

I2 – 100 **BEQZ** r1 + 200

I3 – 104 ADD

108.....

I4 – 304 ADD



Condiția de salt nu este cunoscută până în stagiul EX

# Hazarduri de control – salturi condiționale

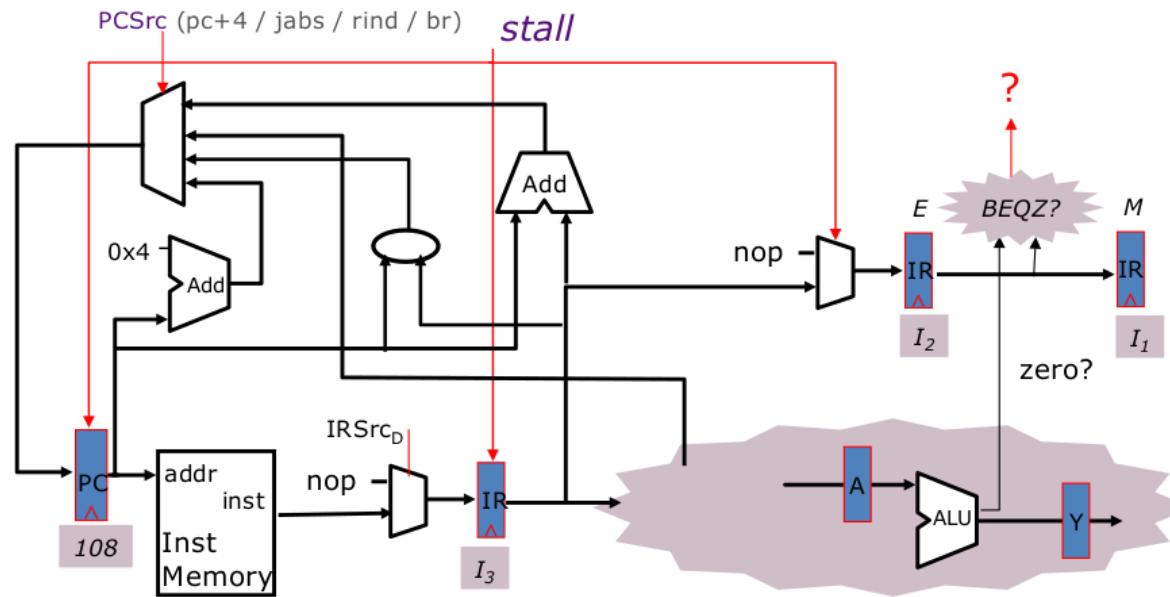
I1 - 096ADD

|2 – 100 BEQZ r1 + 200

I3 – 104 ADD

108.....

I4 – 304ADD



Dacă instrucțiunea BEQZ este luată atunci:

1. se elimină cele 2 instrucțiuni
  2. instrucțiunea din stagiu DECODE nu este validă

# Hazarduri de control – calcul adresă

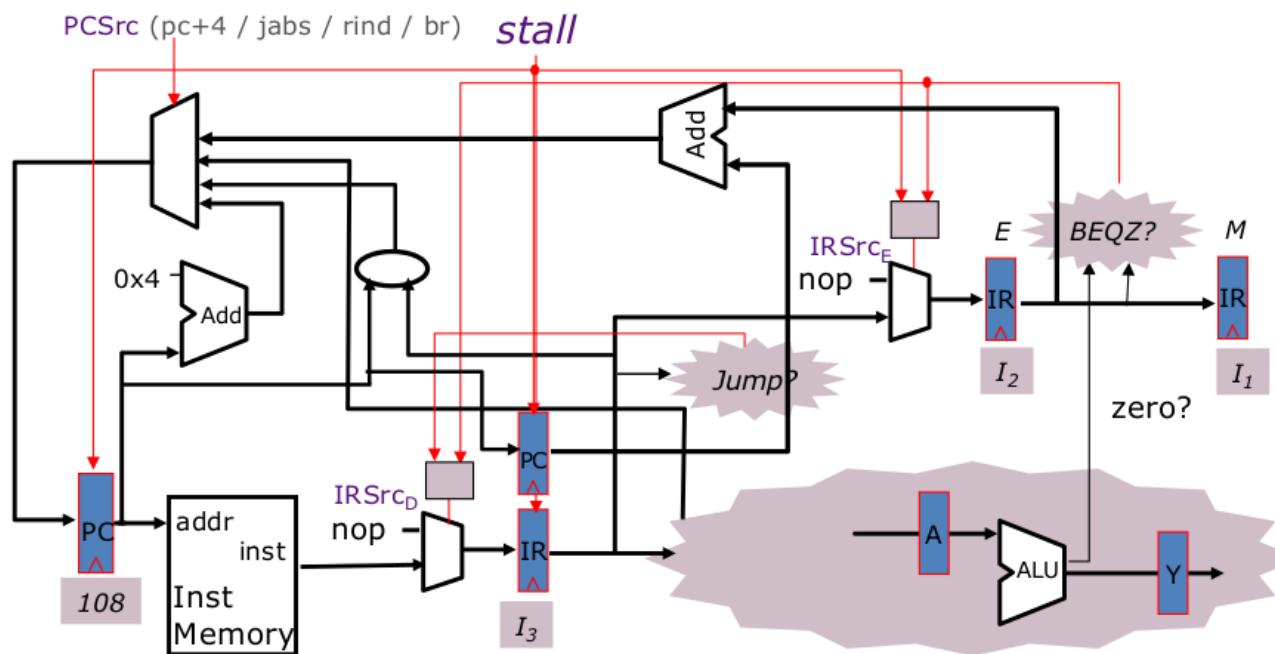
I1 – 096 ADD

I2 – 100 **BEQZ** r1 + 200

I3 – 104 ADD

108.....

I4 – 304 ADD



## Hazarduri de control – noua ecuație pentru STALL

$$\begin{aligned} \text{stall} = & ( ((\text{rs}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rs}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rs}_D = \text{ws}_W) \cdot \text{we}_W) \cdot \text{re1}_D \\ & + ((\text{rt}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rt}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rt}_D = \text{ws}_W) \cdot \text{we}_W) \cdot \text{re2}_D ) \\ & \cdot !((\text{opcode}_E = \text{BEQZ}) \cdot z + (\text{opcode}_E = \text{BNEZ}) \cdot !z) \end{aligned}$$

De ce nu stăm dacă branch-ul este luat în considerare ?

# Hazarduri de control – ecuațiile de control

$PCSrc = \text{Case opcode}_E$

BEQZ.z, BNEZ.!z  $\Rightarrow$  br  
...  $\Rightarrow$   
 $\text{Case opcode}_D$   
J, JAL  $\Rightarrow$  jabs  
JR, JALR  $\Rightarrow$  rind  
...  $\Rightarrow$  pc+4

$IRSrc_D = \text{Case opcode}_E$

BEQZ.z, BNEZ.!z  $\Rightarrow$  nop  
...  $\Rightarrow$   
 $\text{Case opcode}_D$   
J, JAL, JR, JALR  $\Rightarrow$  nop  
...  $\Rightarrow$  IM

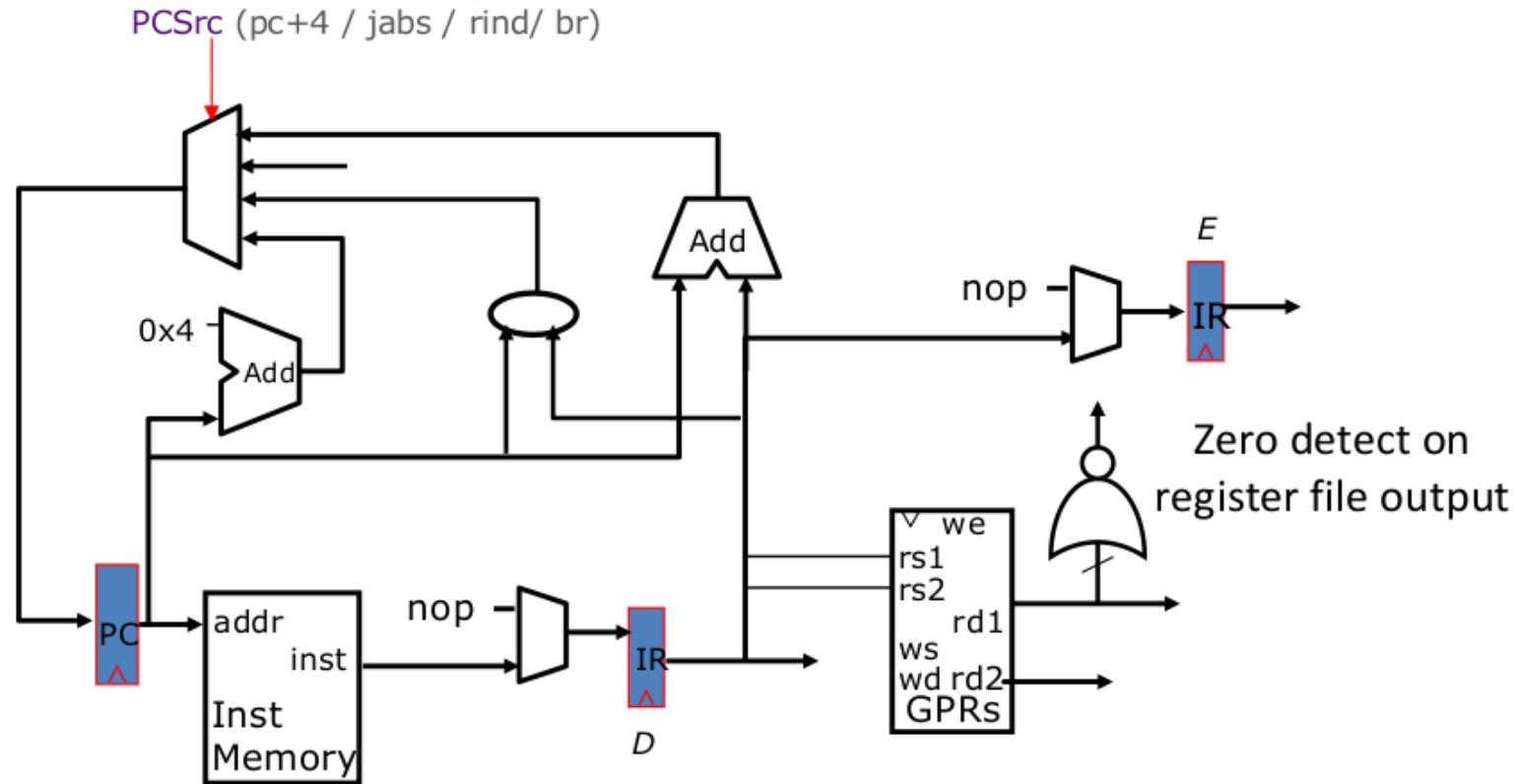
$IRSrc_E = \text{Case opcode}_E$

BEQZ.z, BNEZ.!z  $\Rightarrow$  nop  
...  $\Rightarrow$  stall.nop + !stall.IR<sub>D</sub>

# Hazarduri de control – branch

		t0	t1	t2	t3	t4	t5	t6	t7	...
(I <sub>1</sub> )	096: ADD	IF <sub>1</sub>	ID <sub>1</sub>	EX <sub>1</sub>	MA <sub>1</sub>	WB <sub>1</sub>				
(I <sub>2</sub> )	100: BEQZ +200		IF <sub>2</sub>	ID <sub>2</sub>	EX <sub>2</sub>	MA <sub>2</sub>	WB <sub>2</sub>			
(I <sub>3</sub> )	104: ADD			IF <sub>3</sub>	ID <sub>3</sub>	nop	nop	nop		
(I <sub>4</sub> )	108:				IF <sub>4</sub>	nop	nop	nop	nop	
(I <sub>5</sub> )	304: ADD					IF <sub>5</sub>	ID <sub>5</sub>	EX <sub>5</sub>	MA <sub>5</sub>	WB <sub>5</sub>

# Hazarduri de control – branch – reducerea penalității



Un NOP din banda de asamblare poate fi eliminat dacă folosim un comparator în stagiul DECODE

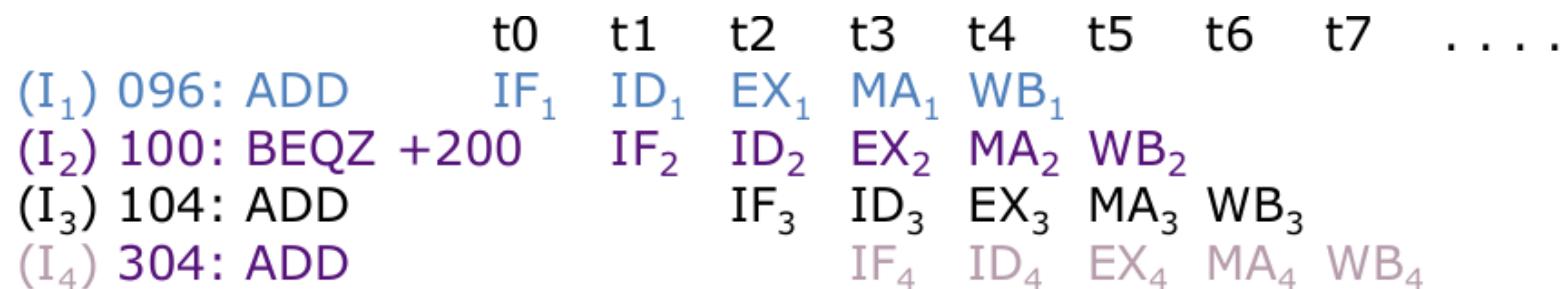
# Hazarduri de control – branch – reducerea penalității

*Prin intermediul compilatorului*

Folosim instrucțiuni de sloturi pentru întârzierea branch-urilor

Unele arhitecturi au prevăzute 2 – 3 sloturi

I <sub>1</sub>	096	ADD
I <sub>2</sub>	100	BEQZ r1 +200
I <sub>3</sub>	104	ADD ←—————
I <sub>4</sub>	304	ADD



# Hazarduri – CONCLUZII

***BYPASS complet este prea scump de implementat*** - unele căi de by-pass utilizate frecvent pot crește timpul ciclului de ceas și altfel se contracarează beneficiului reducerii CPI-ului

***Instrucțiunile LOAD au o latență de 2 ciclii***

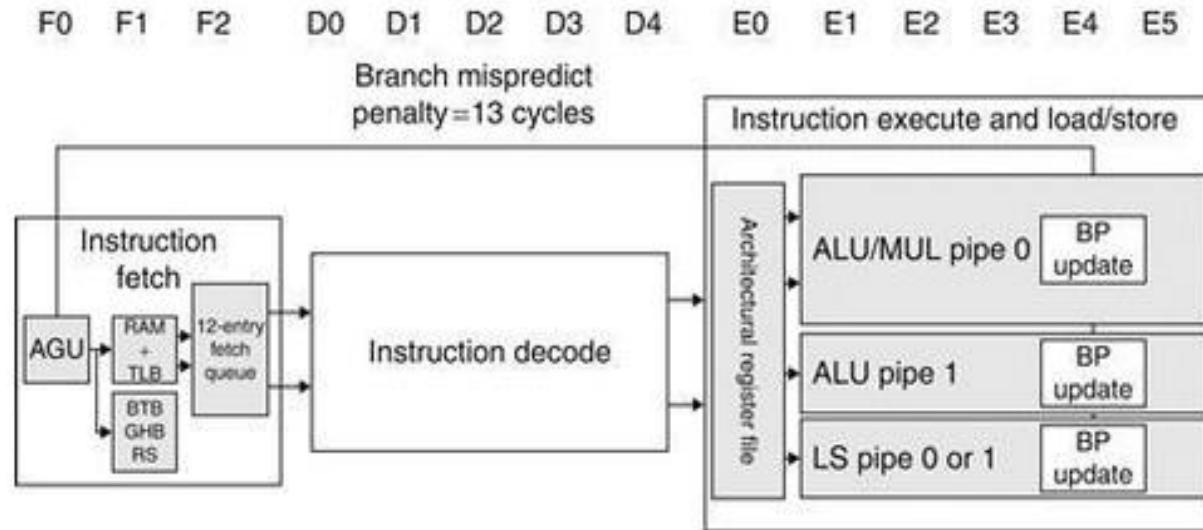
Instrucțiunile după LOAD nu pot utiliza rezultatul instrucțiunii LOAD

MIPS 1 definea încărcarea slot-urilor de întârziere. MIPS 2 a adăugat banda de asamblare care folosește interlock în hardware

MIPS - “Microprocessor without Interlocked Pipeline Stages”

***Branchurile condiționale pot cauza NOP-uri*** - Se elimină instrucțiunile următoare dacă nu avem slot-uri de întârziere

# Banda de asamblare – ARM Cortex 8



AGU – Address Generation Unit

BTB – Branch Target Buffer

GHB – Global History Buffer

RS – Return stack

# Banda de asamblare – ARM Cortex 8

Primele 3 stagii citesc două instrucțiuni la același moment de timp pentru a menține buffer-ul de prefetch plin. Acest buffer are capacitatea de 12 instrucțiuni

Se utilizează un branch predictor pe 2 nivele, ambele nivele având un buffer de 512 intrări și un buffer cu istorie globală de 4096 intrări. Stiva de returnare are 8 intrări.

Dacă o predicție branch este greșită, banda de asamblare se golește rezultând 13 ciclii de ceas ca și penalitate

Cele 5 stagii din pipe de decode determină dacă există dependințe între perechile de instrucțiuni, ceea ce va forța o execuție secvențială a instrucțiunilor.

Cele 6 stagii de execuție a unei instrucțiuni oferă un pipe pentru instrucțiunile de load și store precum și 2 pipe-uri pentru operațiile aritmetice.

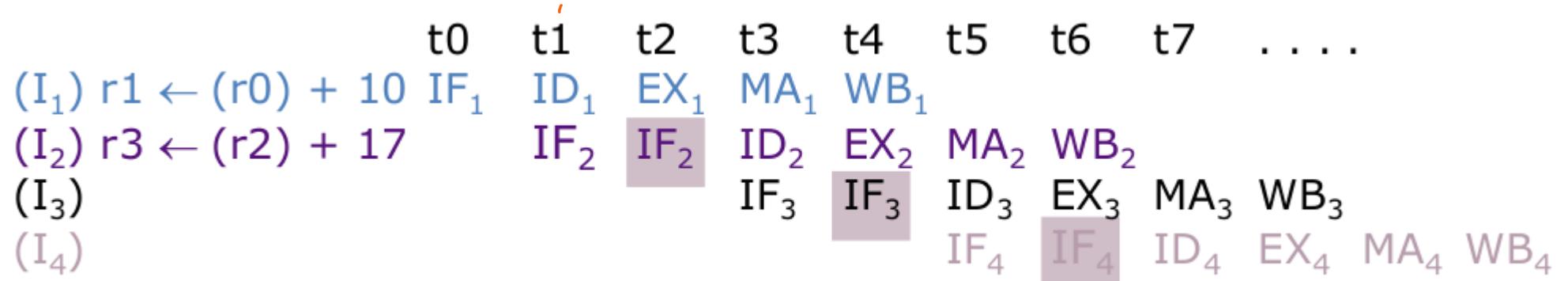
# Hazarduri de control - PC-ul

Cum calculăm următorul PC ?

- Jump
  - Opcode
  - Offset
  - PC
- Jump Register
  - Opcode
  - Valoarea registrului
- Salturi condiționate
  - Opcode
  - PC
  - Registru – pentru condiție
  - Offset

Restul instrucțiunilor au nevoie doar de opcode și PC

# Hazarduri de control



Introducem NOP cu toate că instrucțiunile ce trebuie executate nu prezintă hazarduri.

*DE CE ?*

# Hazarduri de control - PC-ul

## JUMP

1. Este necesar să ne uităm la opcode pentru a determina dacă este JUMP sau nu
2. Trebuie să ne uităm la offset-ul care vine din instrucțiune precum și la valoarea curentă a PC-ului
3. Decodificatorul va stabili că avem o instrucțiune JUMP, stagiul DECODE din banda de asamblare va indica o instrucțiune JUMP -» PC + offset (cel mai probabil operația se va face în ALU, altfel trebuie introdus un sumator)
4. Executăm instrucțiunea următoarea de la adresa calculată

# Hazarduri de control - PC-ul

## JUMP Register

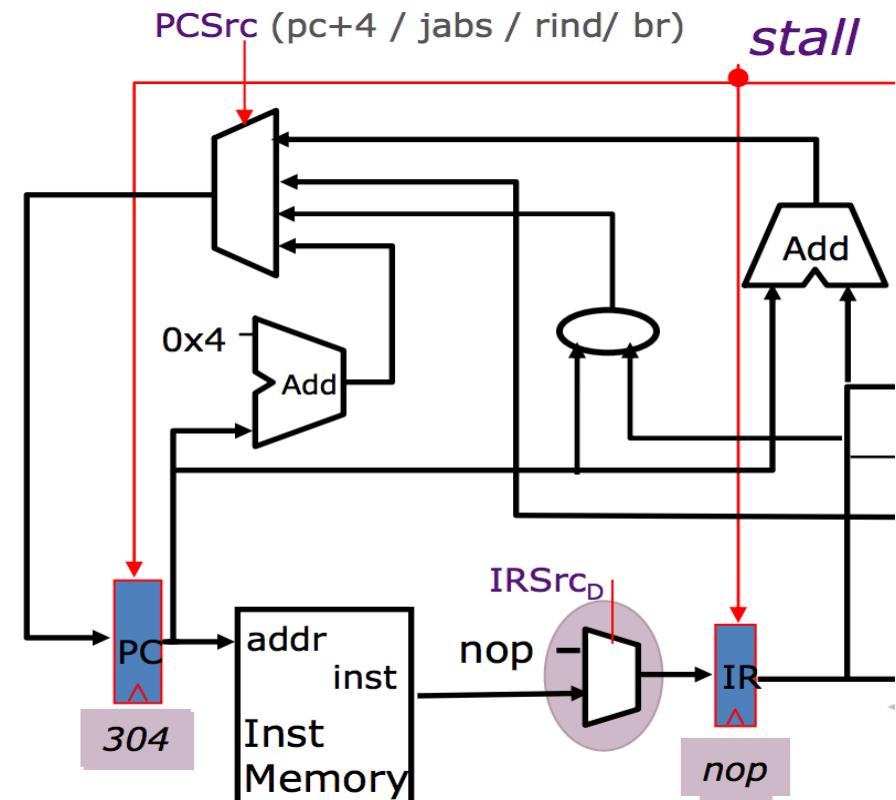
1. Opcode ne va spune dacă avem o instrucțiune de acest tip sau nu
2. Nu este necesară utilizarea offset-ului deoarece sărim direct la valoarea registrului – doar în cazul MIPS (altfel folosim registru indirect sau o instrucțiune de tipul jump register sau chiar o instrucțiune de tipul jump indirect la memorie)

# Hazarduri de control – speculăm PC+4

I1 – 096	ADD
I2 – 100	J 304
I3 – 104	ADD
I4 – 304	ADD

Trebuie ca instrucția 3 să nu se execute

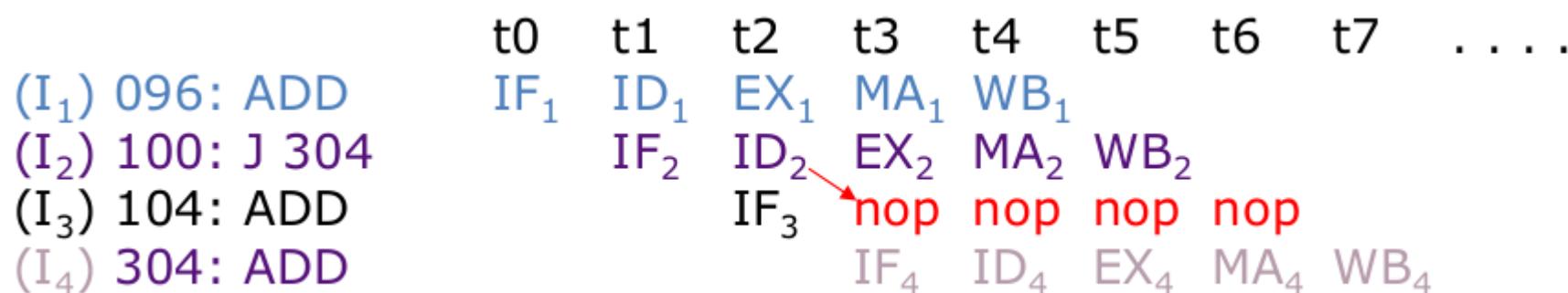
=»  
trebuie introdus un multiplexor care  
introduce NOP-uri înainte de registrul IR.



$$\begin{aligned} \text{IRSrc}_D &= \text{Case opcode}_D \\ &\quad J, \text{JAL} \\ &\quad \dots \\ &\Rightarrow \text{nop} \\ &\Rightarrow \text{IM} \end{aligned}$$

# Hazarduri de control – speculăm PC+4

I1 – 096	ADD
I2 – 100	J 304
I3 – 104	ADD
I4 – 304	ADD



# Hazarduri de control – salturi condiționale

I1 – 096

I2 – 100

I3 – 104

108

I4 – 304

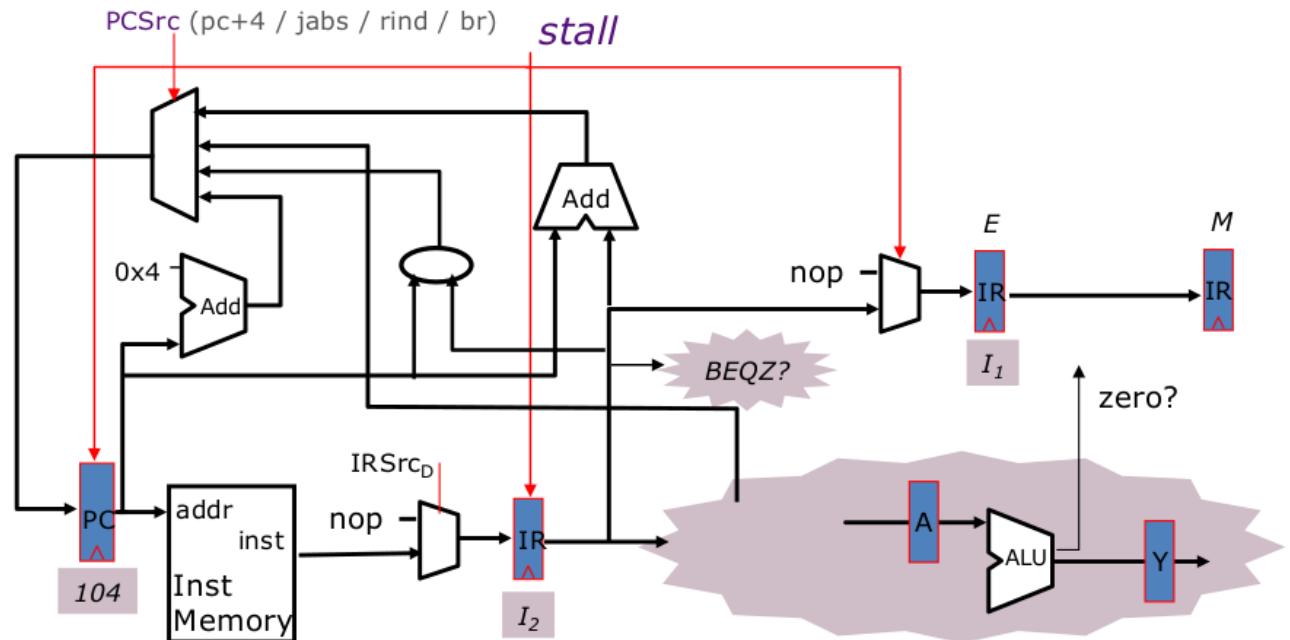
ADD

**BEQZ** r1 + 200

ADD

.....

ADD



Condiția de salt nu este cunoscută până în stagiul EX

# Hazarduri de control – salturi condiționale

I1 – 096

I2 – 100

I3 – 104

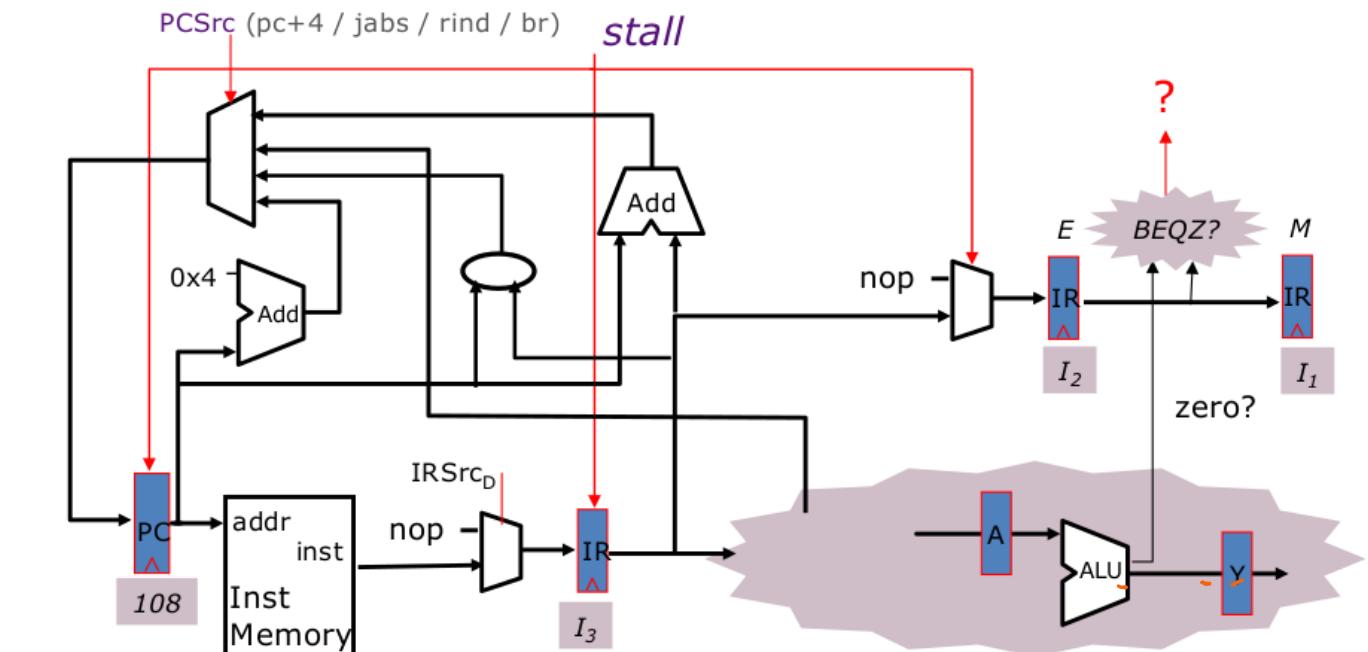
.....

I4 – 304

ADD

**BEQZ** r1 + 200

ADD

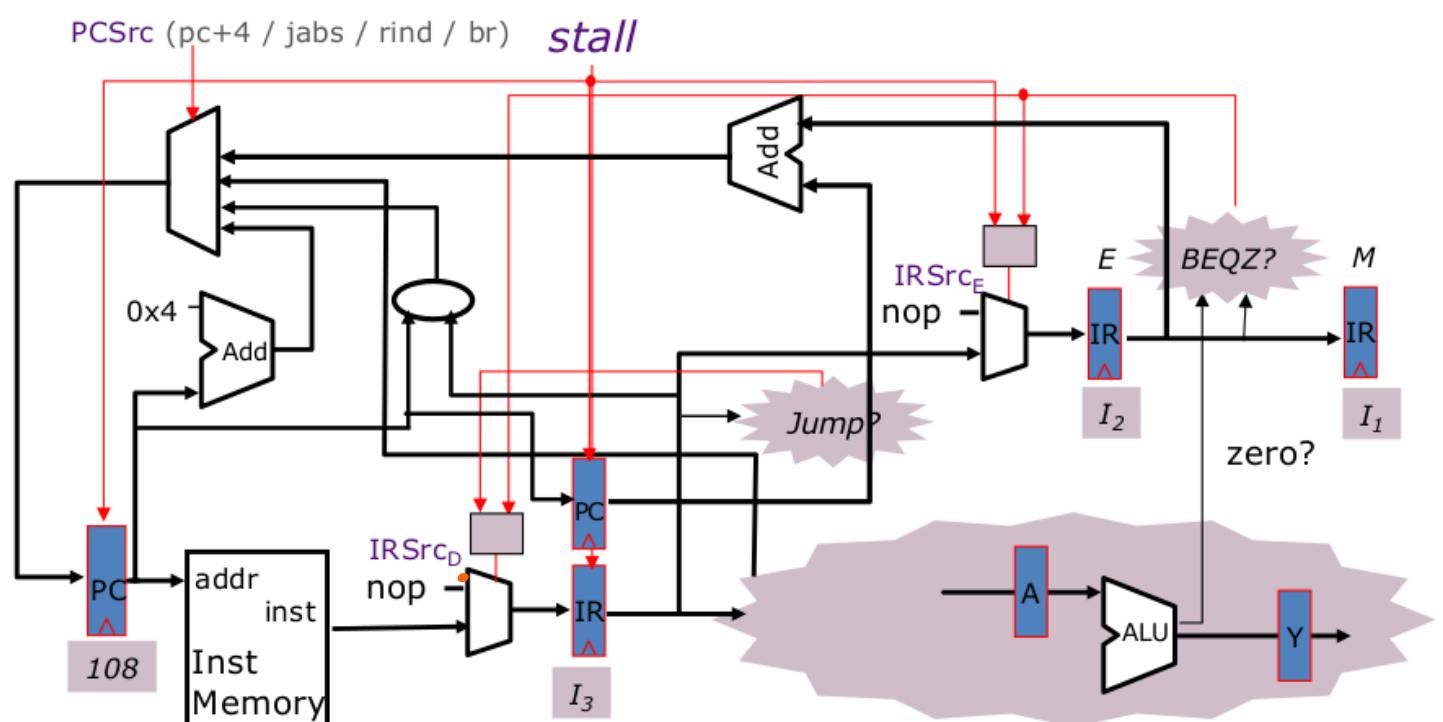


Dacă instrucțiunea BEQZ este luată atunci:

1. se elimină cele 2 instrucțiuni
2. instrucțiunea din stagiul DECODE nu este validă

# Hazarduri de control – calcul adresă

I1 – 096            ADD  
 I2 – 100          ***BEQZ r1 + 200***  
 I3 – 104          ADD  
 108                .....  
 I4 – 304          ADD



# Hazarduri de control – noua ecuație pentru STALL

$$\begin{aligned} \text{stall} = & ((\text{rs}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rs}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rs}_D = \text{ws}_W) \cdot \text{we}_W) \cdot \text{re1}_D \\ & + ((\text{rt}_D = \text{ws}_E) \cdot \text{we}_E + (\text{rt}_D = \text{ws}_M) \cdot \text{we}_M + (\text{rt}_D = \text{ws}_W) \cdot \text{we}_W) \cdot \text{re2}_D \\ & \cdot !((\text{opcode}_E = \text{BEQZ}) \cdot z + (\text{opcode}_E = \text{BNEZ}) \cdot !z) \end{aligned}$$

De ce nu stăm dacă branch-ul este luat în considerare ?

# Hazarduri de control – ecuațiile de control

$\text{PCSrc} = \text{Case opcode}_E$

$\text{BEQZ.z, BNEZ.!z} \Rightarrow \text{br}$   
 $\dots \Rightarrow$

$\text{Case opcode}_D$

$J, \text{JAL} \Rightarrow \text{jabs}$   
 $\text{JR, JALR} \Rightarrow \text{rind}$   
 $\dots \Rightarrow \text{pc+4}$

$\text{IRSrc}_D = \text{Case opcode}_E$

$\text{BEQZ.z, BNEZ.!z} \Rightarrow \text{nop}$   
 $\dots \Rightarrow$

$\text{Case opcode}_D$

$J, \text{JAL}, \text{JR, JALR} \Rightarrow \text{nop}$   
 $\dots \Rightarrow \text{IM}$

$\text{IRSrc}_E = \text{Case opcode}_E$

$\text{BEQZ.z, BNEZ.!z} \Rightarrow \text{nop}$

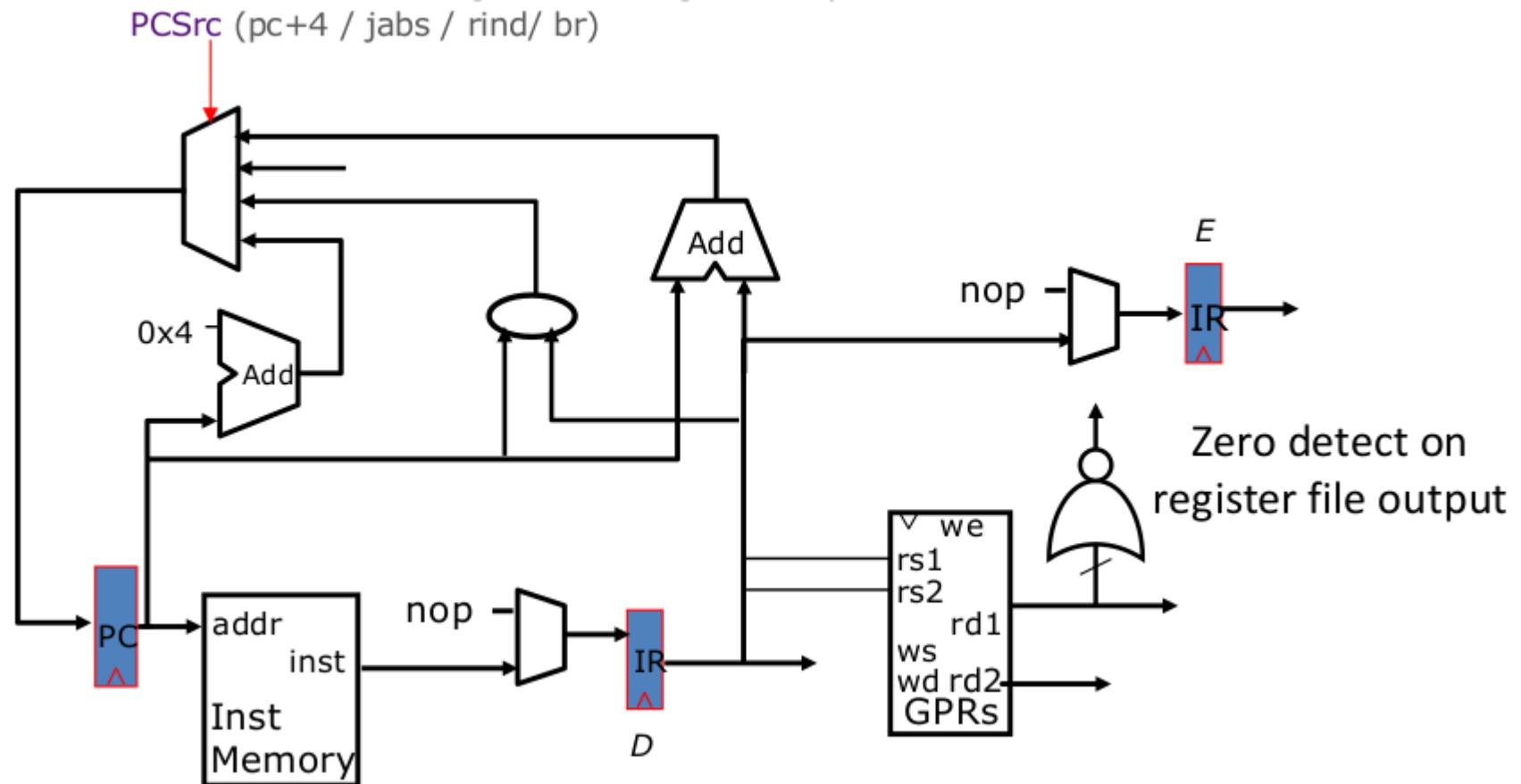
$\dots$

$\Rightarrow \text{stall.nop} + !\text{stall.IR}_D$

# Hazarduri de control – branch

	t0	t1	t2	t3	t4	t5	t6	t7	...
(I <sub>1</sub> ) 096: ADD	IF <sub>1</sub>	ID <sub>1</sub>	EX <sub>1</sub>	MA <sub>1</sub>	WB <sub>1</sub>				
(I <sub>2</sub> ) 100: BEQZ +200		IF <sub>2</sub>	ID <sub>2</sub>	EX <sub>2</sub>	MA <sub>2</sub>	WB <sub>2</sub>			
(I <sub>3</sub> ) 104: ADD			IF <sub>3</sub>	ID <sub>3</sub>	hop	nop	nop		
(I <sub>4</sub> ) 108:				IF <sub>4</sub>	hop	nop	nop	nop	
(I <sub>5</sub> ) 304: ADD					IF <sub>5</sub>	ID <sub>5</sub>	EX <sub>5</sub>	MA <sub>5</sub>	WB <sub>5</sub>

# Hazarduri de control – branch – reducerea penalității



Un NOP din banda de asamblare poate fi eliminat dacă folosim un comparator în stagiul DECODE

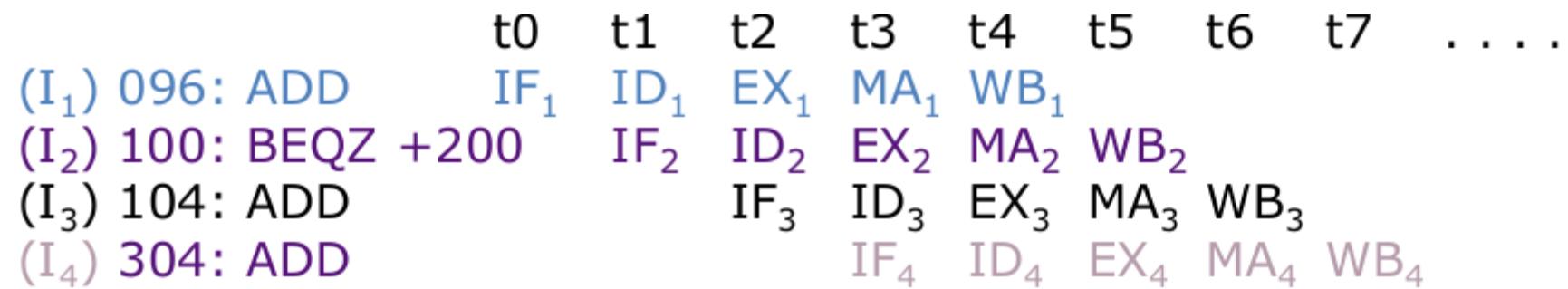
# Hazarduri de control – branch – reducerea penalității

*Prin intermediul compilatorului*

Folosim instrucțiuni de sloturi pentru întârzierea branch-urilor

Unele arhitecturi au prevăzute 2 – 3 sloturi

I <sub>1</sub>	096	ADD
I <sub>2</sub>	100	BEQZ r1 +200
I <sub>3</sub>	104	ADD
I <sub>4</sub>	304	ADD



# Hazarduri – CONCLUZII

**BYPASS complet este prea scump de implementat** - unele căi de by-pass utilizate frecvent pot crește timpul ciclului de ceas și altfel se contracarează beneficiului reducerii CPI-ului

**Instrucțiunile LOAD au o latență de 2 ciclii**

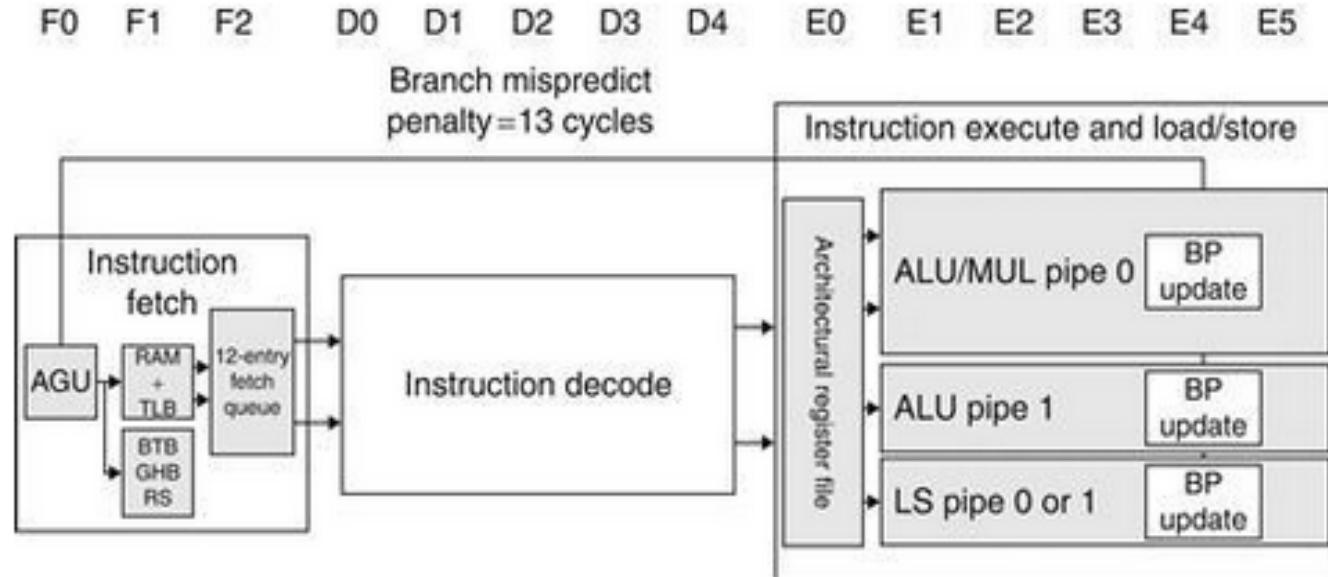
Instrucțiunile după LOAD nu pot utiliza rezultatul instrucțiunii LOAD

MIPS 1 definea încărcarea slot-urilor de întârziere. MIPS 2 a adăugat banda de asamblare care folosește interlock în hardware

MIPS - “Microprocessor without Interlocked Pipeline Stages”

**Branch-urile condiționale pot cauza NOP-uri** - Se elimină instrucțiunile următoare dacă nu avem slot-uri de întârziere

# Banda de asamblare – ARM Cortex 8



AGU – Address Generation Unit

BTB – Branch Target Buffer

GHB – Global History Buffer

RS – Return stack

# Banda de asamblare – ARM Cortex 8

Primele 3 stagii citesc două instrucțiuni la același moment de timp pentru a menține buffer-ul de prefetch plin. Acest buffer are capacitatea de 12 instrucțiuni

Se utilizează un branch predictor pe 2 nivele, ambele nivele având un buffer de 512 intrări și un buffer cu istorie globală de 4096 intrări. Stiva de returnare are 8 intrări.

Dacă o predicție branch este greșită, banda de asamblare se golește rezultând 13 ciclii de ceas ca și penalitate

Cele 5 stagii din pipe de decode determină dacă există dependințe între perechile de instrucțiuni, ceea ce va forța o execuție secvențială a instrucțiunilor.

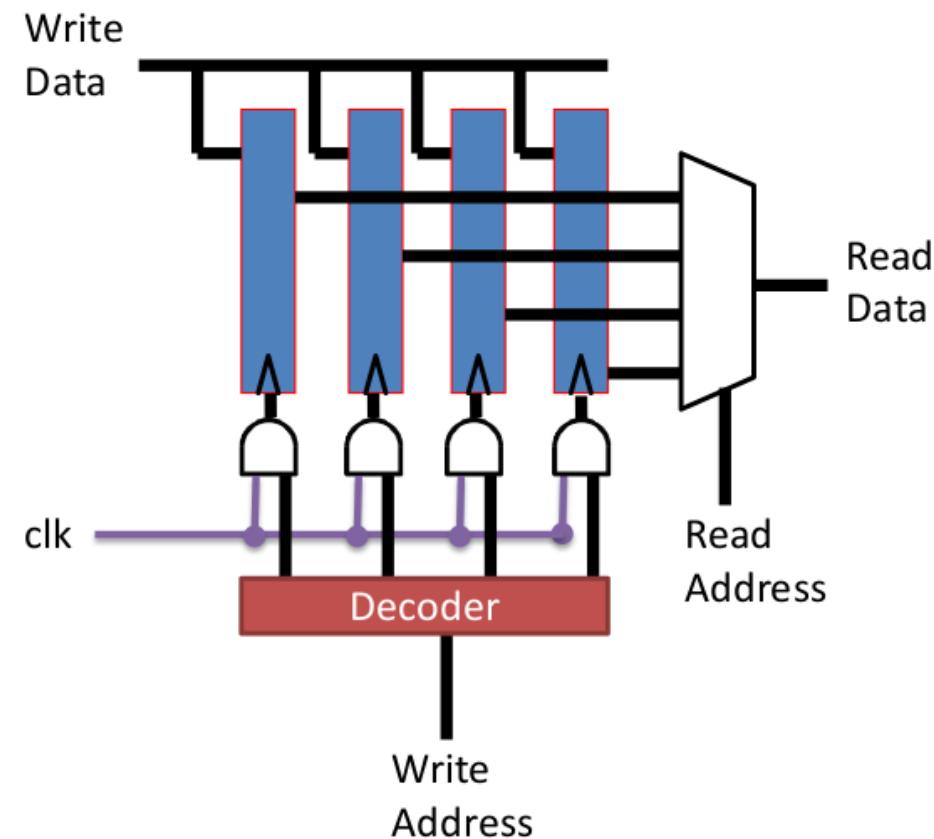
Cele 6 stagii de execuție a unei instrucțiuni oferă un pipe pentru instrucțiunile de load și store precum și 2 pipe-uri pentru operațiile aritmetice.

# **MEMORIA**

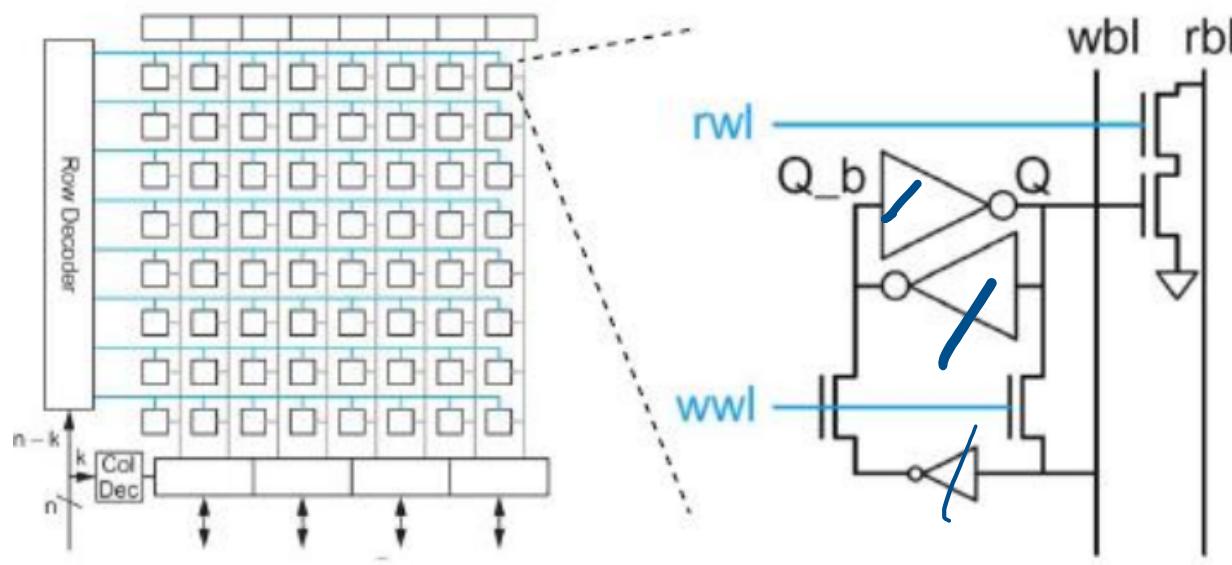
# MEMORIA

- Tehnologii folosite
- Motivarea folosirii memoriei cache
- Clasificarea memoriilor cache
- Performanța memoriei cache

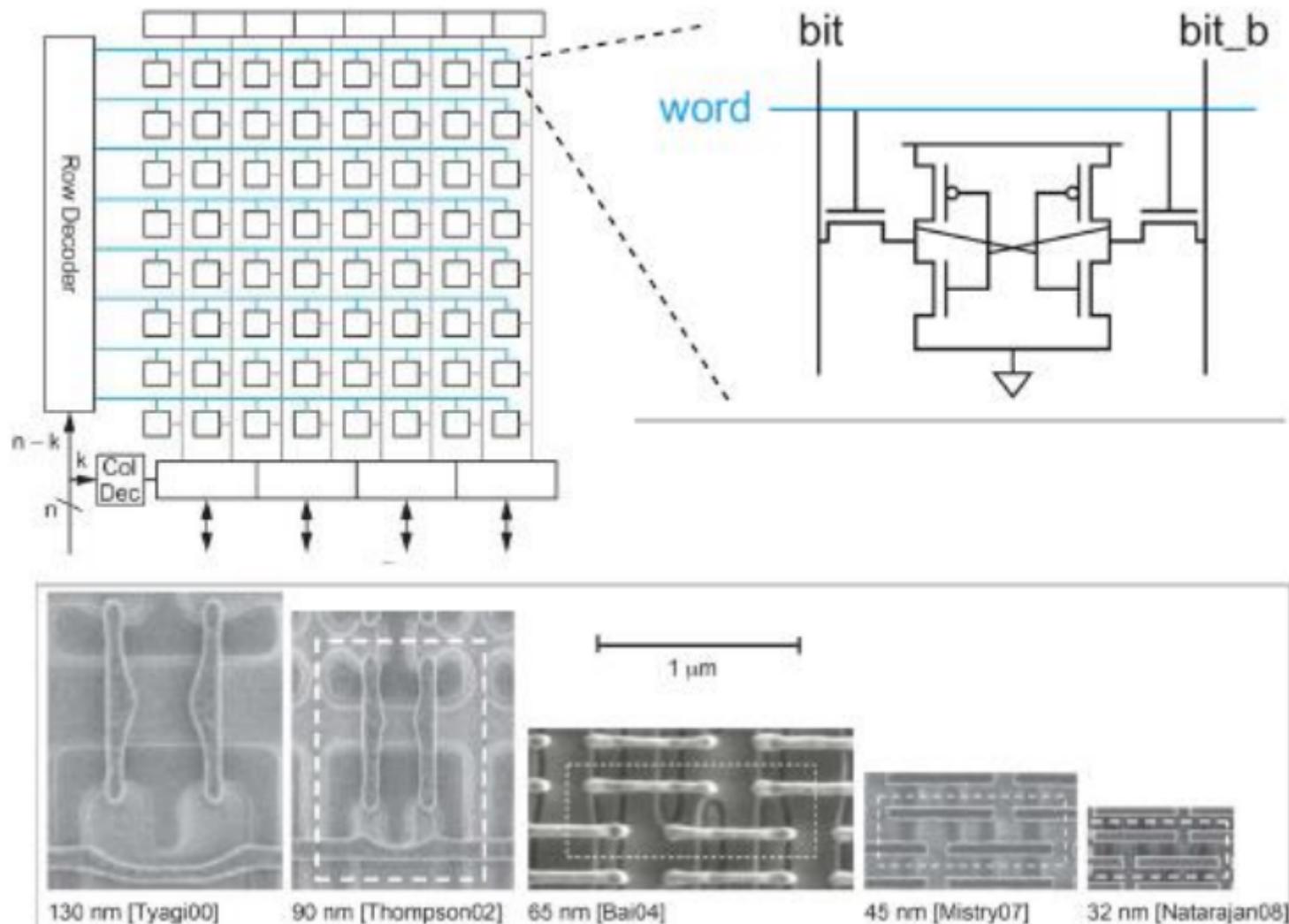
# MEMORIA - registru



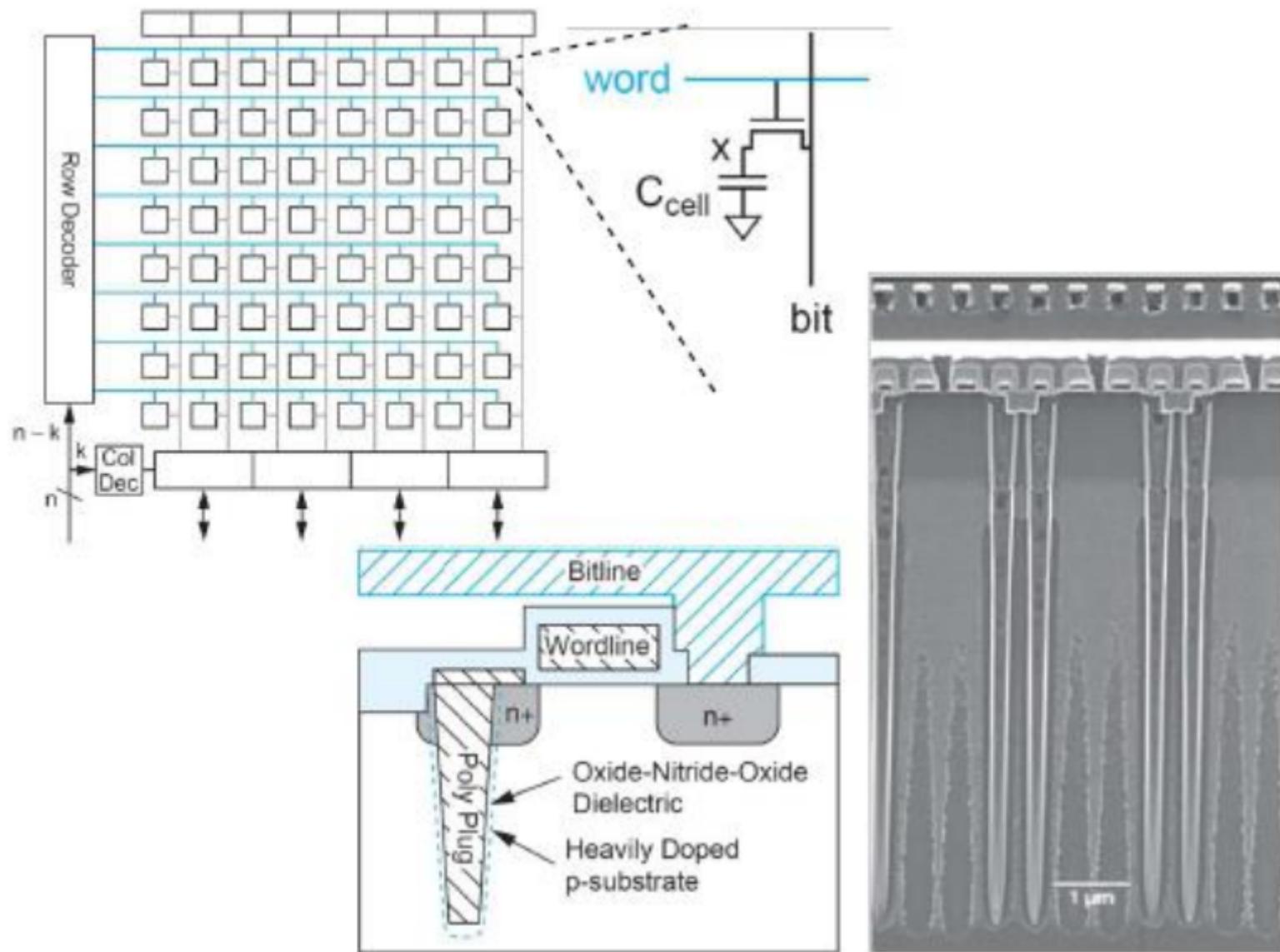
# MEMORIA - array



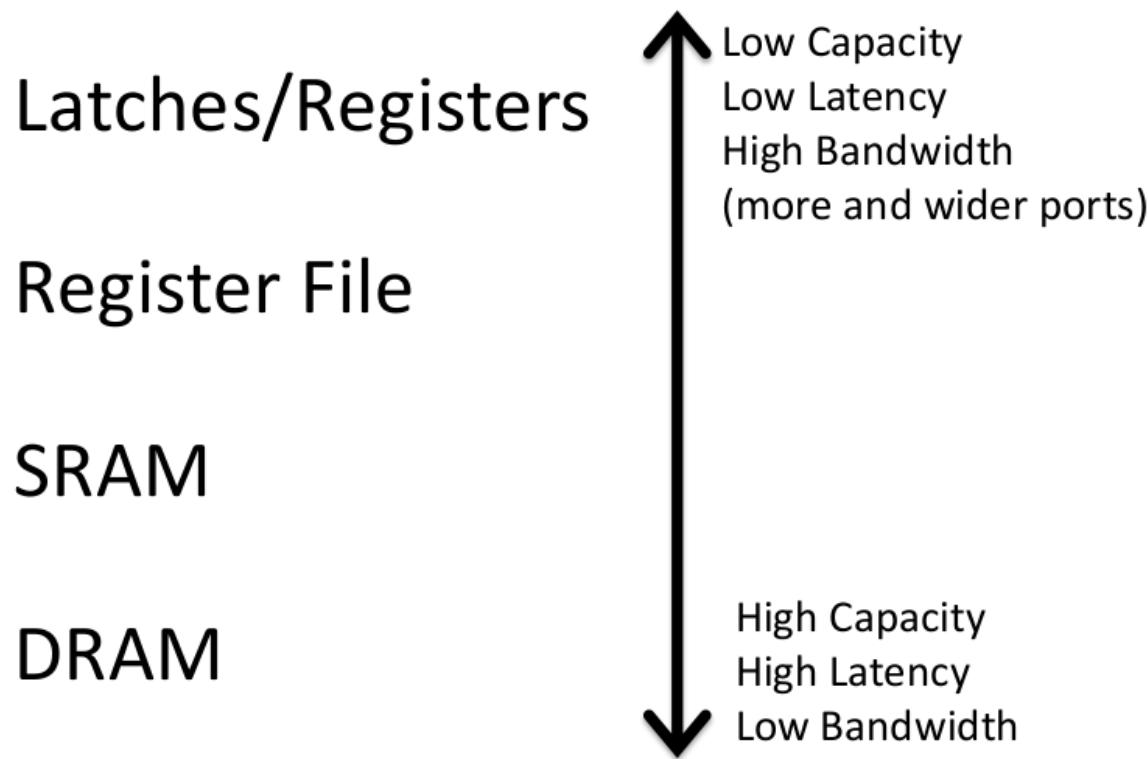
# MEMORIA - SRAM



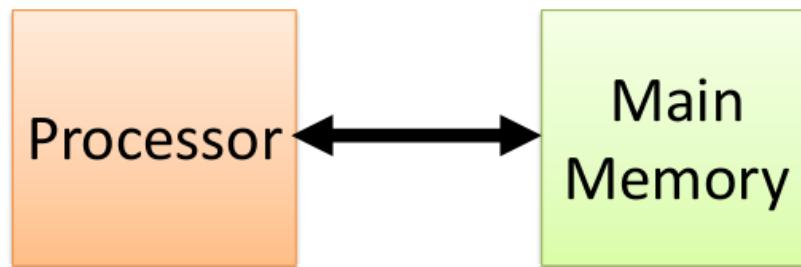
# MEMORIA - DRAM



# MEMORIA

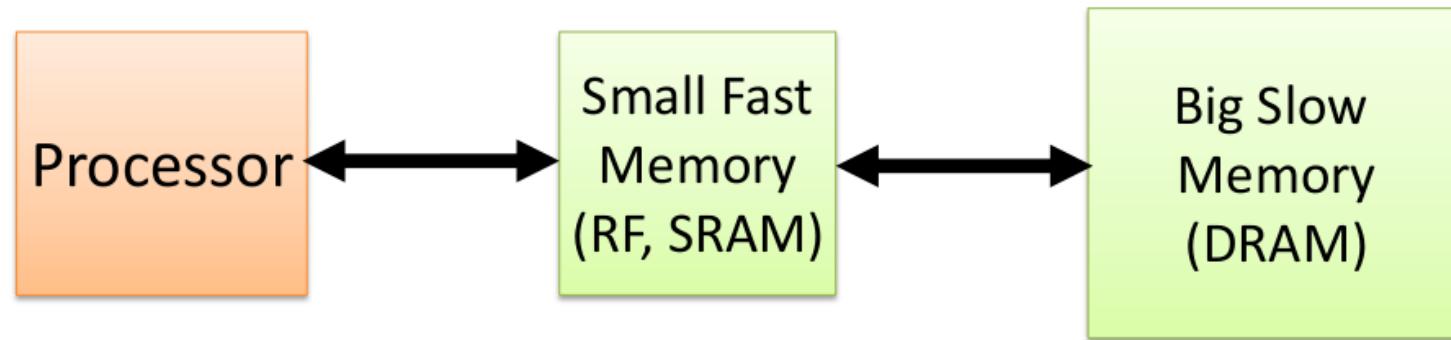


# MEMORIA CACHE



- Performanța procesorului este limitată de lățimea benzii pentru memorie și latența memoriei.
- Lățimea benzii este reprezentată de numărul de accesări / unitatea de timp
- Latența (*timpul pentru un singur acces*) memoriei principale este »» decât ciclul procesorului

# IERARHIA DE MEMORIE



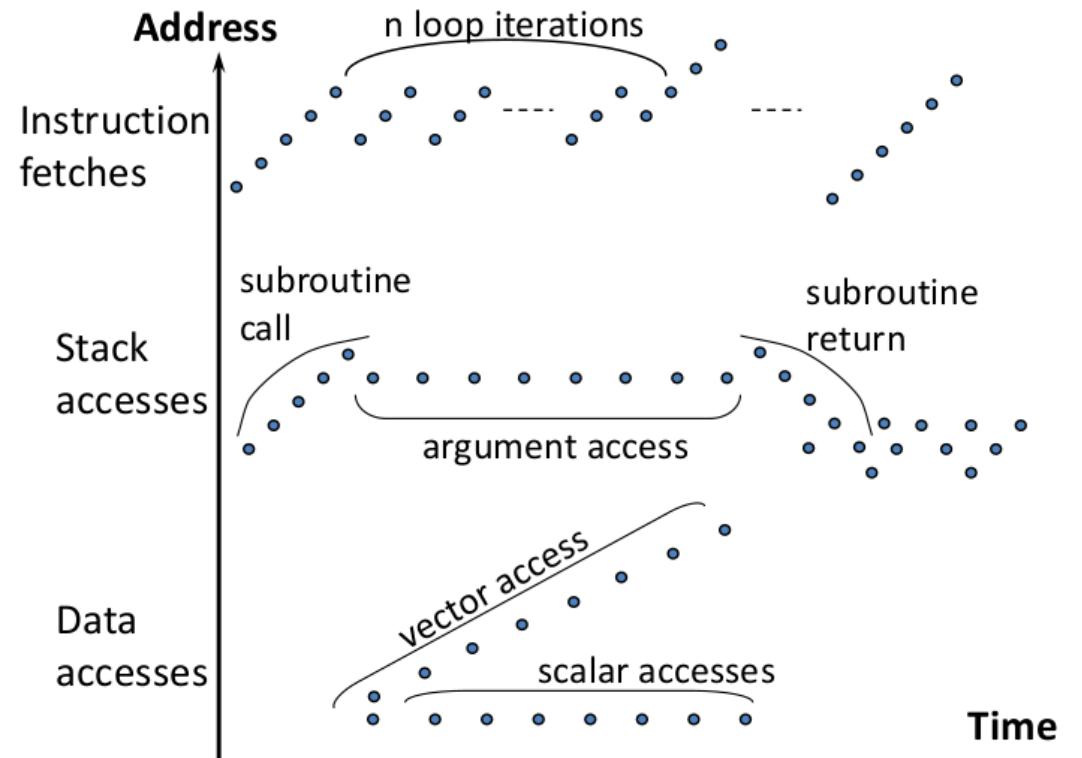
Capacitate: Register << SRAM << DRAM

Latență: Register << SRAM << DRAM

După accesarea datelor:

- dacă data este într-o memorie rapidă -> latență scăzută pentru accesarea SRAM-ului
- dacă data nu este într-o memorie rapidă -> latență mare pentru accesarea DRAM-ului

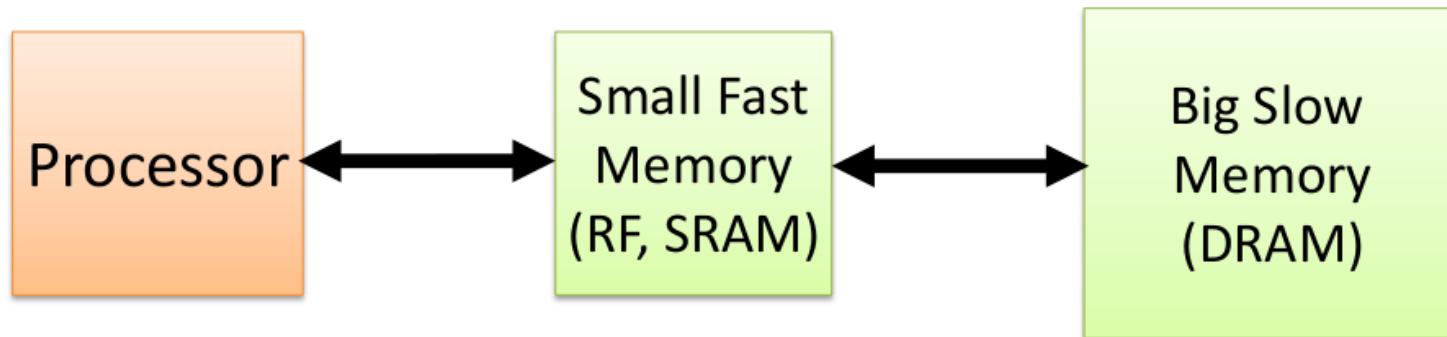
# PRINCIPII



**Localizare temporară** – dacă o locație este referită, atunci ea va fi referită din nou în viitorul apropiat

**Localizare spațială** – dacă o locație este referită, atunci locațiile învecinate ei vor fi referite în viitorul apropiat

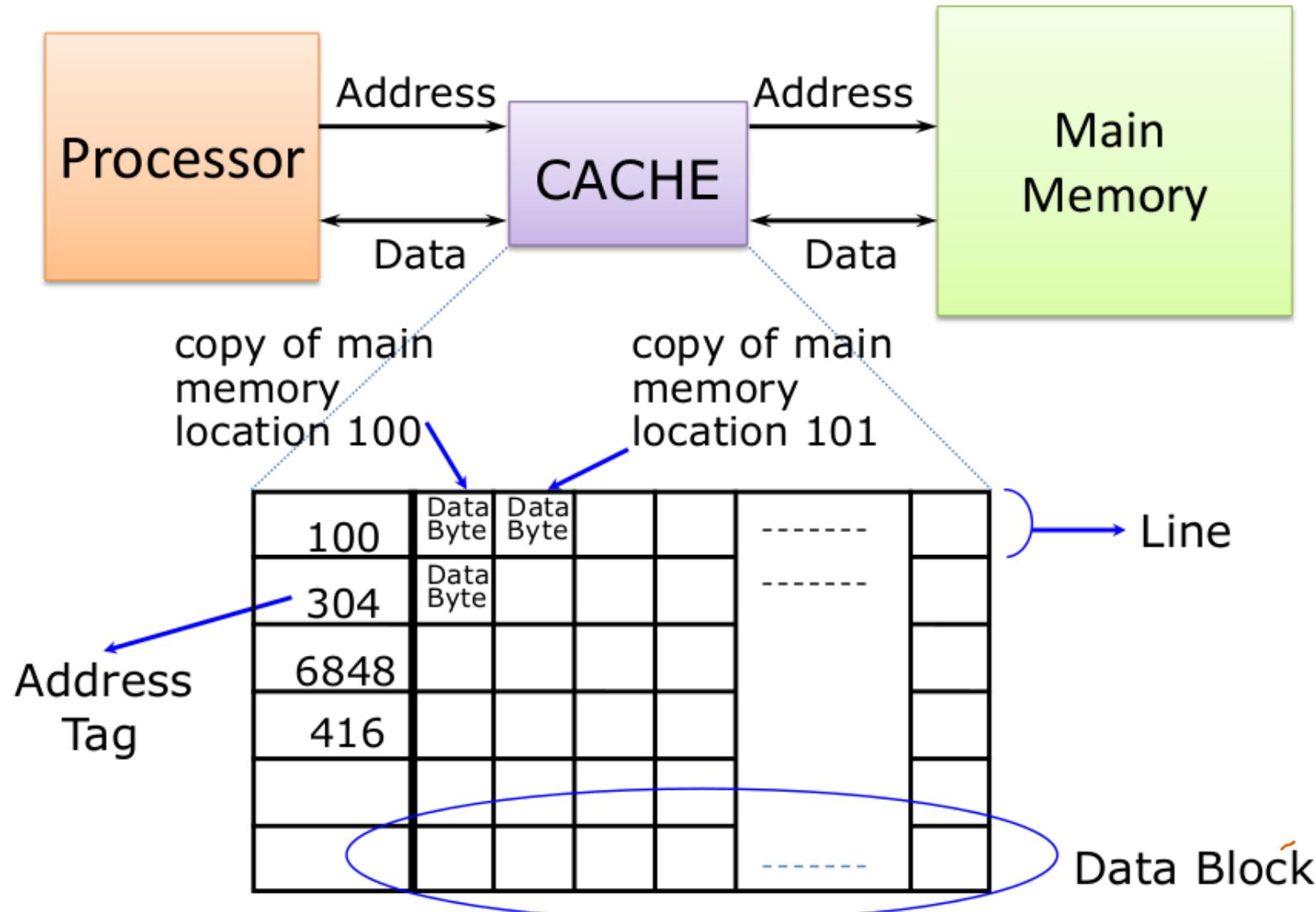
# MEMORIA CACHE



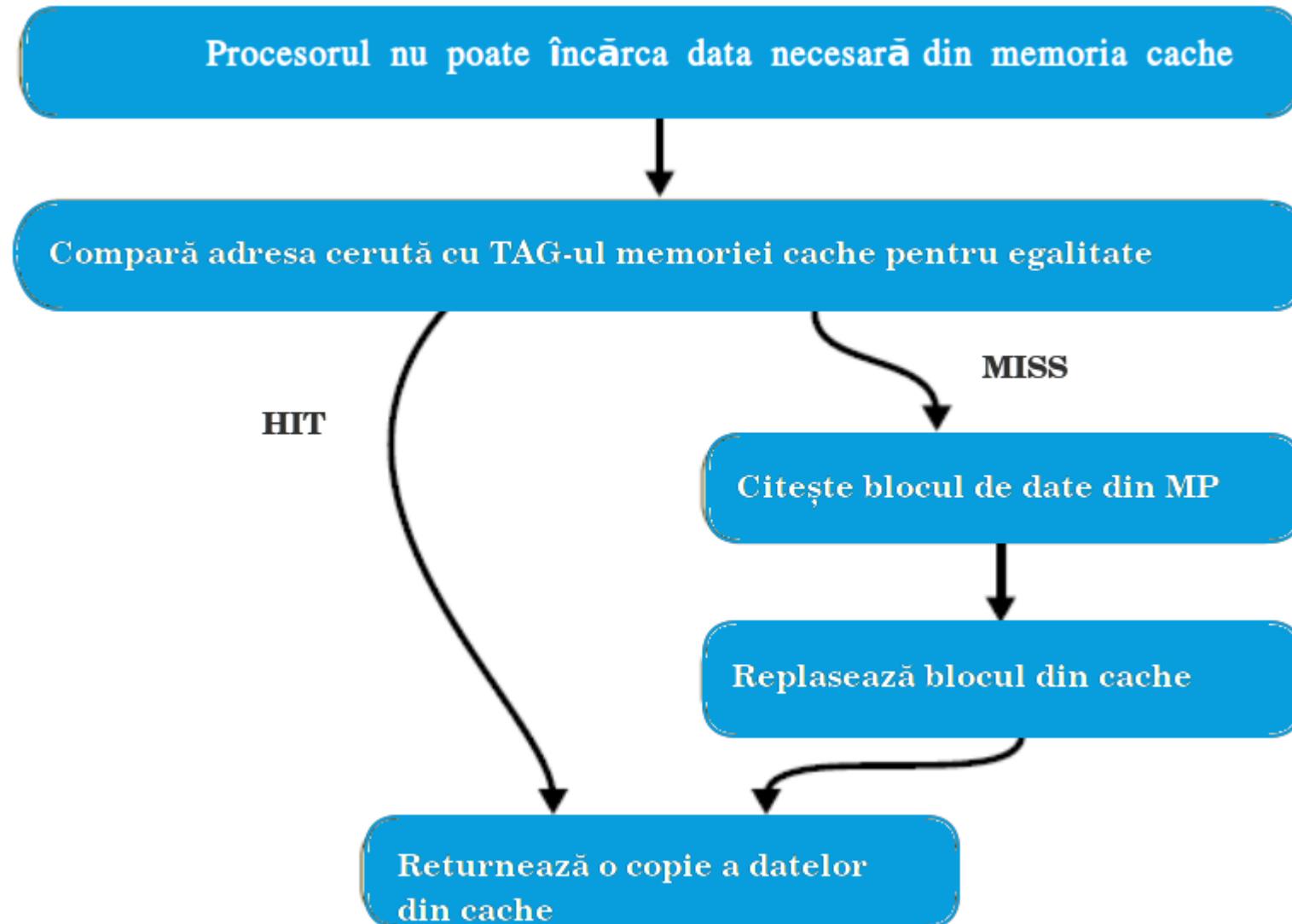
Memoria cache exploatează ambele tipuri de localizări a datelor

- temporară -» prin menținerea conținutului locațiilor accesate cel mai recent
- spațială -» prin citirea blocurilor de date din jurul locației accesate cel mai recent

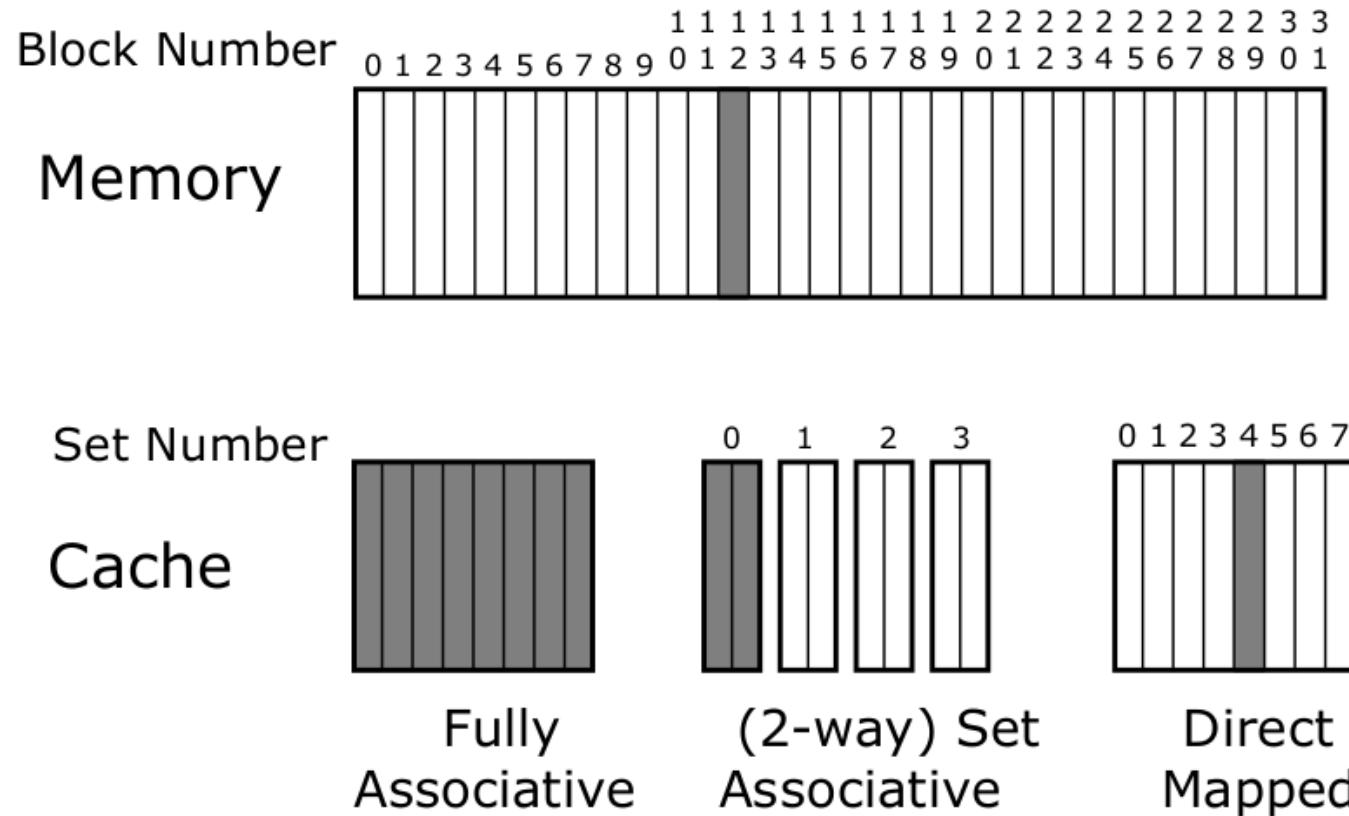
# MEMORIA CACHE



# Algoritm memorie cache LOAD



# Unde plasăm un bloc în cache ?



- Unde poate fi plasat blocul 12 ?

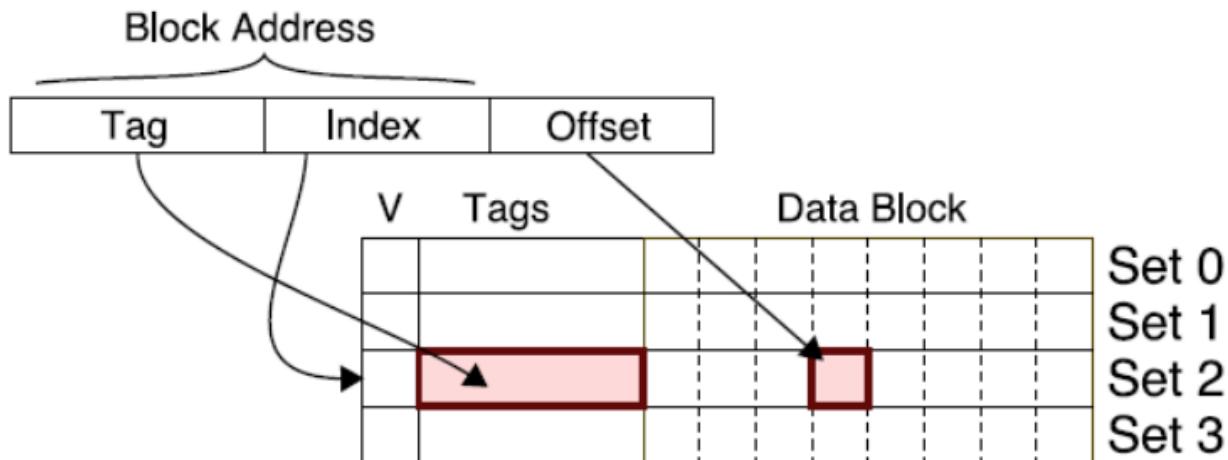
# Parametrii memoriei cache

- Orice memorie cache poate fi definită de 3 parametrii:
- ***Numărul de linii cache – C***
- ***Dimensiunea liniei de cache - L***
- ***Gradul de asociativitate - m***

# Parametrii memoriei cache

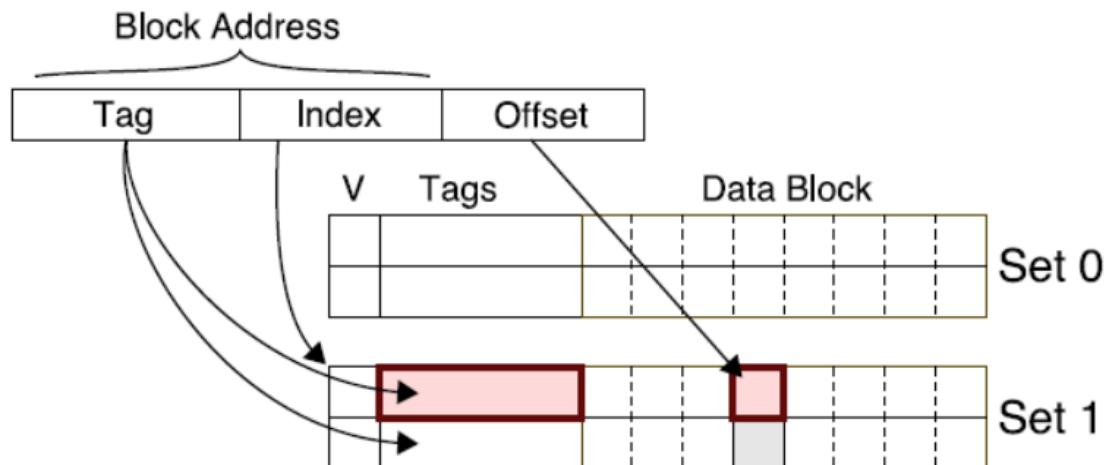
- Orice adresa de memorie generată de ALU se descompune în:
- **TAG** - t
- **INDEX** - i
- DEPLASAMENT sau **OFFSET** – notat d
- Deoarece există L bytes / linie =>  $d = \log(2)L$  – cei mai puțin semnificativi biți din adresa
- Deoarece avem C/m linii / block =>  $i = \log(2) C/m$
- Restul de bitii până la 32 vor reprezenta valoarea lui t

# Regăsirea blocului în cache



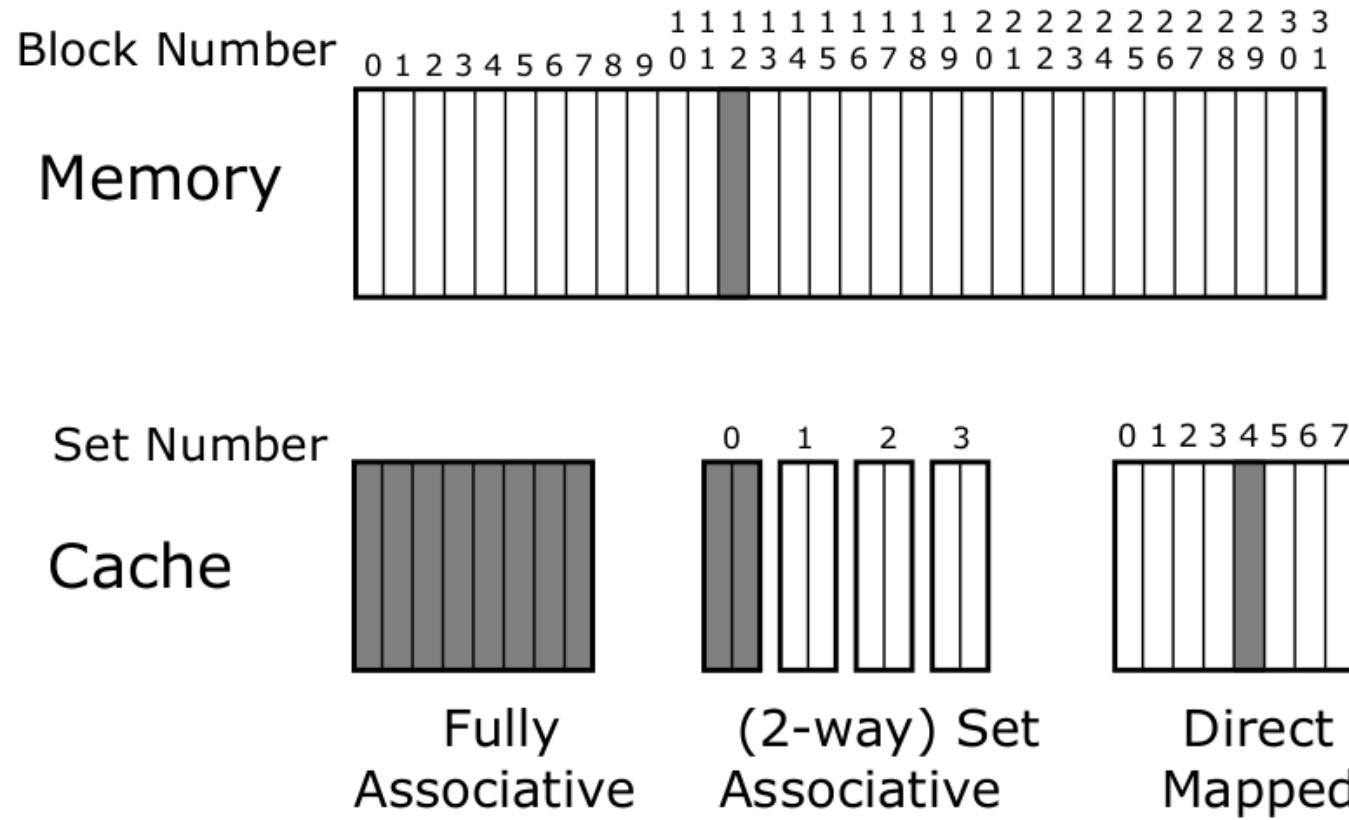
- Memorie cache cu mapare directă – 4 linii cache și bloc de 8B
  - Întâi utilizăm Index-ul și Offset-ul pentru a identifica o potențială potrivire, apoi verificăm TAG-ul

# Regăsirea blocului în cache



- Memorie cache set asociativă cu 2 căi – 4 linii cache și bloc de 8B
- Se caută în paralel toate potențialele blocuri printr-o verificare paralelă a TAG-ului
- Metodă scumpă deoarece necesită folosirea a unui număr mare de comparație

# Unde replasăm un bloc în cache ?



- Unde poate fi replasat blocul 12 ?
    - Într-o memorie cache cu mapare directă -» oriunde
    - Într-o memorie cache cu mapare set-asociativă -» random, LRU, FIFO sau NMRU

# Scrierea în memoria cache - HIT

- Scrierea în memoria cache se poate face prin mai multe metode:
- Doar scriem în memoria cache – această tehnică se numește **WRITEBACK** sau COPYBACK
- Memoria cache trebuie să conțină un indicator care să reflecte modificarea conținutului liniei – dirty bit. Acest bit trebuie setat când avem un HIT pe scriere
- Scriere în memoria cache dar și în nivelul următor al ierarhiei de memorie – **WRITE-THROUGH**.
  - În acest caz nu mai este nevoie de dirty bit. Această metodă generează trafic de memorie mai mare decât precedenta metoda.

# Scrierea în memoria cache - MISS

- Scrierea în memoria cache se poate face prin mai multe metode:
  - **No Write Allocate** - doar scriere în memoria principală
  - **Write Allocate** - citim blocul în memoria cache apoi scriem

# Performanța memoriei cache

- Ce anume încearcă să facă cache-ul?
- atunci când se face un LOAD sau un STORE, încercăm să micșorăm ciclii/instrucțiune pentru a putea procesa instrucțiunile de LOAD sau STORE.
- Dacă, dacă reușim cumva să aducem în cache, datele care reduc probabilitatea unei pierderi de cache, putem să scădem timpul necesar pentru procesare
- Timpul Mediu de Acces pentru Memorie = Timp Hit + (Rata Miss \* Penalitatea Miss)
- Ca să putem crește performanța mai întâi trebuie să determinăm tipurile de MISS ce pot apărea

# Memoria cache – tipuri de MISS – cei trei „C”

- **Compulsory** (obligatorii) - prima referință a unui bloc. O posibilitate de eliminare a acestui MISS este introducerea unei unități de prefetch. Practic va trebui cumva intuit care date vor fi folosite în viitor, aduse și utilizate în caz de MISS.
- **Capacitate** - capacitatea memoriei cache este prea mică. Trebuie făcut un compromis între capacitate și preț. Cu cât memoria cache este mai mare cu atât rata de MISS va scădea
- **Conflict** - pot apărea coliziuni în memoriile cache care nu au asociativitate totală. Practic soluția de proiectare aleasă necesită o asociativitate crescută.

# Memoria cache – îmbunătățirea performanțelor

- construirea de memorii cache mici și simple (se reduce timpul de HIT ceea ce este foarte bine conform ecuației anterioare) - memorii cache pe 1, 2, 4, 8 căi
- reducerea ratei de MISS prin folosirea în cadrul memoriei cache a unor blocuri de date de dimensiune cât mai mare. Ca și avantaje putem menționa: overhead-ul de tag este mai mic, transferuri mai rapide de la DRAM. Ca și dezavantaje: dacă nu utilizăm datele atunci ocupăm de pomană magistrala, crește numărul de conflicte deoarece avem mai puține blocuri.
- reducerea ratei de MISS dacă dimensiunea memoriei cache este mare
  - “Dacă dimensiunea memoriei cache este dublă, rata de MISS în mod uzual descrește cu aproximativ  $\sqrt{2}$ ”
- reducerea ratei de MISS dacă se crește asociativitatea memoriei cache
- Ce dezavantaj am avea în cazul unei memorii associative pe 2 căi vs o memorie cache cu mapare directă ?

# Calculatoare Numerice 2

Cursul 5

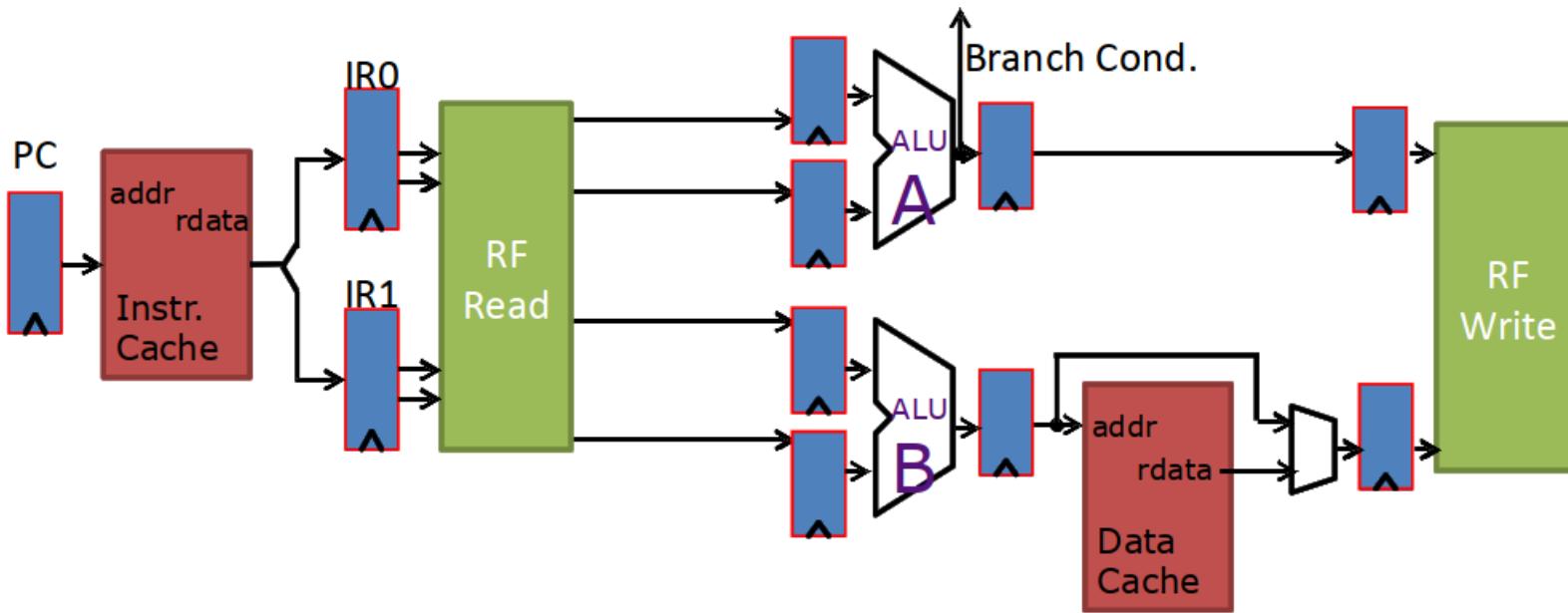
# Tipuri de hazarduri

- Considerăm că se execută secvența:  $r_k \ll - r_i \ op \ r_j$
- Dependență de date (RAW)
  - $r_3 \ll - r_1 \ op \ r_2$
  - $r_5 \ll - r_3 \ op \ r_4$
- Anti-dependență (WAR)
  - $r_3 \ll - r_1 \ op \ r_2$
  - $r_1 \ll - r_4 \ op \ r_5$
- Dependență de ieșire (WAW)
  - $r_3 \ll - r_1 \ op \ r_2$
  - $r_3 \ll - r_6 \ op \ r_7$

# Introducere superscalar

- Procesoarele studiate pînă acum sunt fundamental limitate la CPI »= 1
- Procesoarele superscalare au CPI «= 1 (**Instruction Per Cycle » 1**) deoarece se execută mai multe instrucțiuni în paralel
- Procesoare IO și OOO
  - IO – in order – execuția programului se face așa cum este definită de programator
  - OOO – out of order – execuția programului se poate face executând mai multe instrucțiuni în paralel

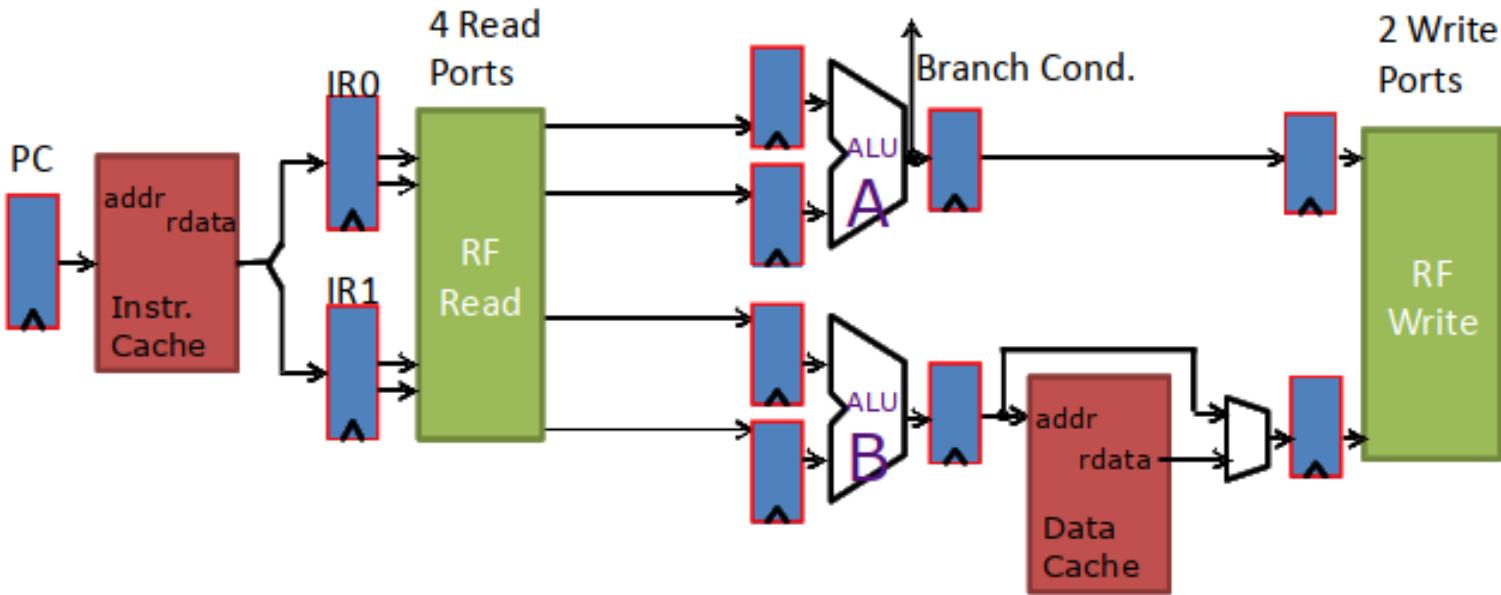
# Superscalar IO 2 căi



Pipe A: Operații întregi și salturi (branch-uri)

Pipe B: Operații întregi și Memoria

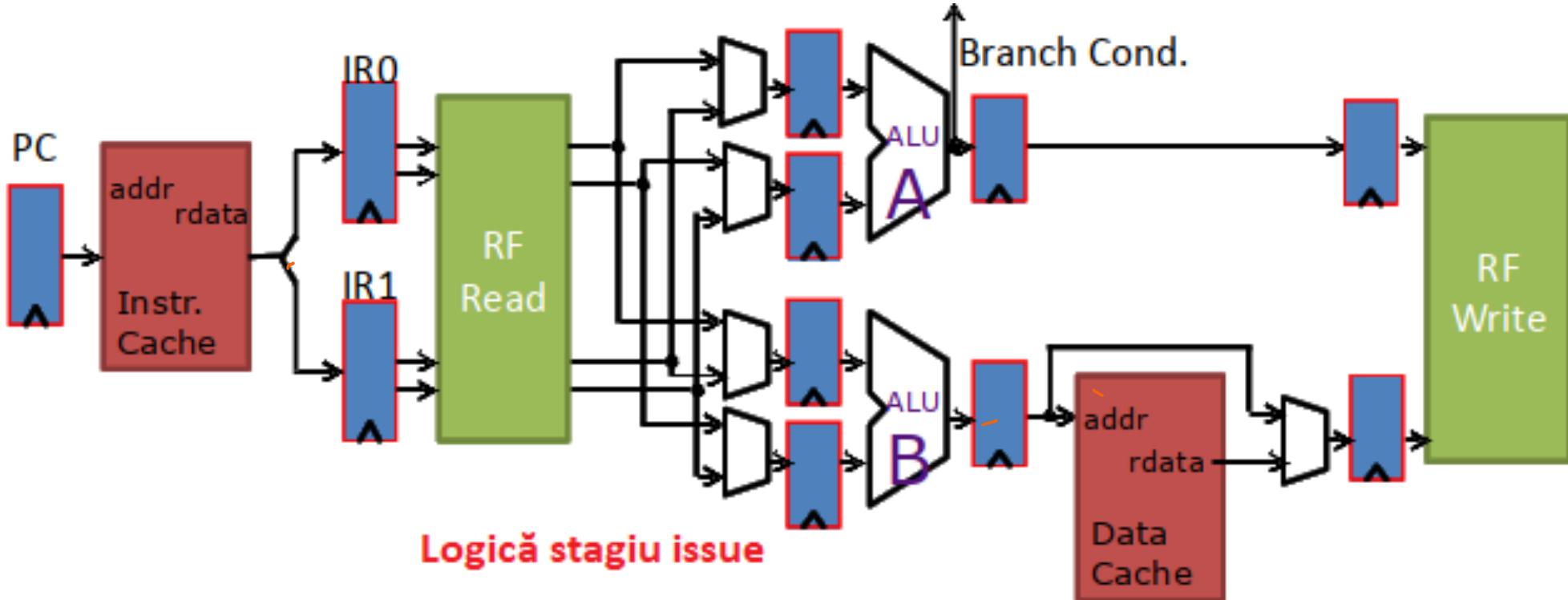
# Superscalar 10 2 căi



Pipe A: Operații întregi și salturi (branch-uri)  
Pipe B: Operații întregi și Memoria

Citește 2 instrucțiuni la același moment de timp

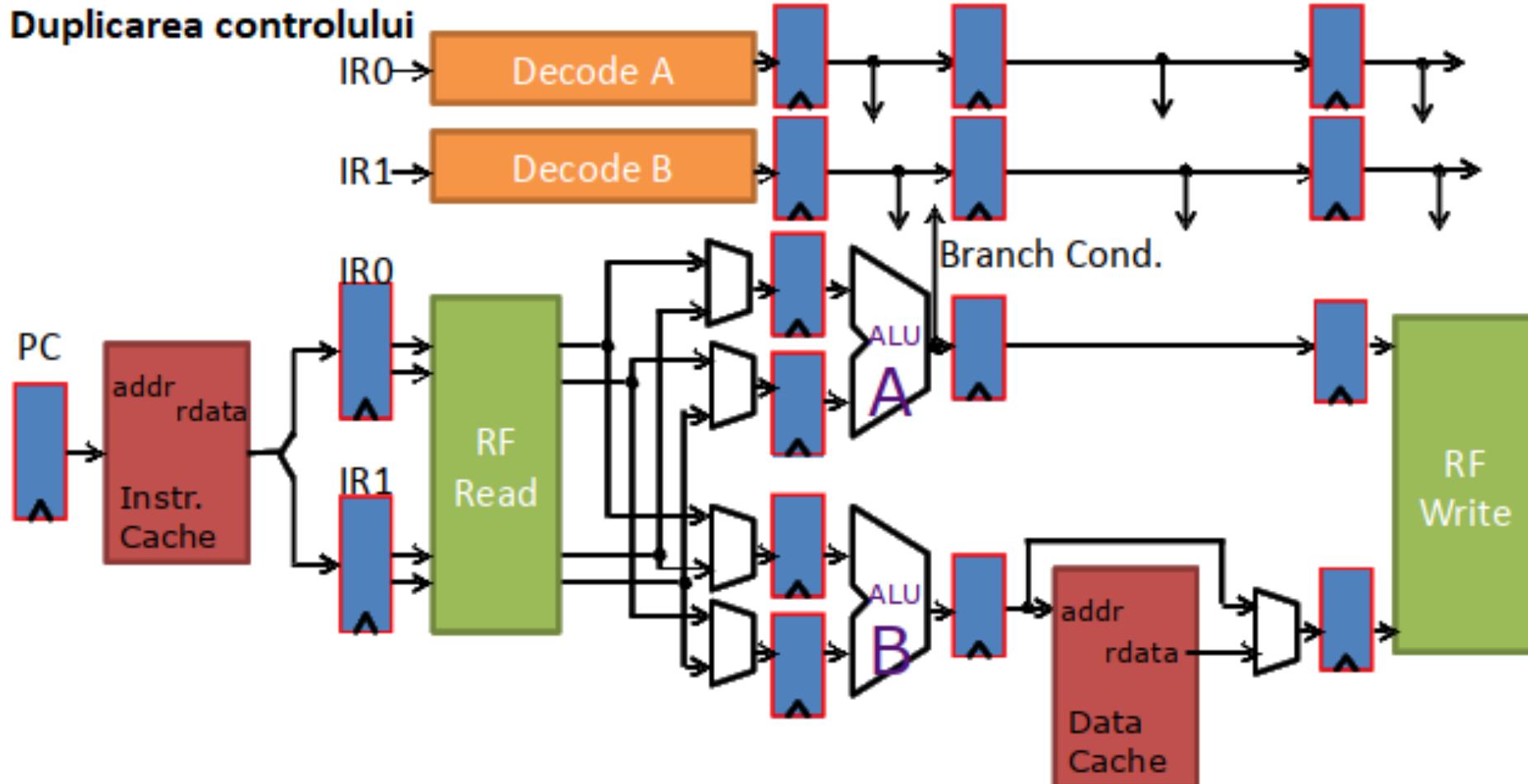
# Superscalar 10 2 căi



Pipe A: Operații întregi și salturi (branch-uri)

Pipe B: Operații întregi și Memoria

# Superscalar 10 2 căi



Pipe A: Operații întregi și salturi (branch-uri)

Pipe B: Operații întregi și Memoria

# Pipe pentru logica de issue

OpA	F	D	A0	A1	W		
OpB	F	D	B0	B1	W		
OpC		F	D	A0	A1	W	
OpD		F	D	B0	B1	W	
OpE			F	D	A0	A1	W
OpF			F	D	B0	B1	W

CPI = 0.5 (IPC = 2)

Pipeline cu 2 stagii issue  
Avem 2 instrucțiuni în același stagiu la același moment de timp

ADDIU	F	D	A0	A1	W	
LW	F	D	B0	B1	W	
LW	F	D	B0	B1	W	
ADDIU	F	D	A0	A1	W	
LW		F	D	B0	B1	W
LW		F	D	D	B1	W

interschimbare față de posizita naturală

Hazard structural

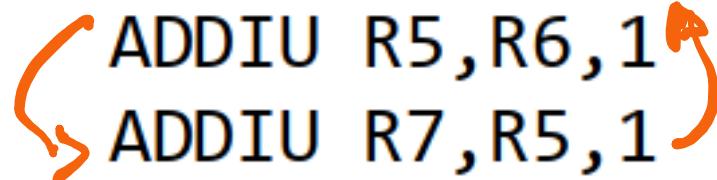
# Hazarduri de date – dual issue

- Fără bypass

ADDIU R1,R1,1	F	D	A0	A1	W	
ADDIU R3,R4,1	F	D	B0	B1	W	
ADDIU R5,R6,1		F	D	A0	A1	W
ADDIU R7,R5,1		F	D	D	D	A0 A1 W

- Full bypass

ADDIU R1,R1,1	F	D	A0	A1	W	
ADDIU R3,R4,1	F	D	B0	B1	W	
ADDIU R5,R6,1		F	D	A0	A1	W
ADDIU R7,R5,1		F	D	D	A0 A1	W



# Hazarduri de date – dual issue

- *Ordinea instrucțiunilor este importantă*
- |               |   |   |       |       |   |
|---------------|---|---|-------|-------|---|
| ADDIU R1,R1,1 | F | D | A0 A1 | W     |   |
| ADDIU R3,R4,1 | F | D | B0 B1 | W     |   |
| ADDIU R7,R5,1 |   | F | D     | A0 A1 | W |
| ADDIU R5,R6,1 |   | F | D     | B0 B1 | W |

- *Hazardul de tip WAR este posibil ?*

# Logica pentru FETCH și alinierea

Cyc	Addr	Instr
0	0x000	OpA
0	0x004	OpB
1	0x008	OpC
1	0x00C	J 0x100
...		
2	0x100	OpD
2	0x104	J 0x204
...		
3	0x204	OpE
3	0x208	J 0x30C
...		
4	0x30C	OpF
4	0x310	OpG
5	0x314	OpH

0x000	0	0	1	1
...				
0x100	2	2		
...				
0x200		3	3	
...				
0x300				4
0x310	4	5		

Citirea mai multor linii de cache este foarte greu de implementat.

Vor trebui adăugate porturi suplimentare

# Logica pentru FETCH și alinierea

Cyc	Addr	Instr
0	0x000	OpA
0	0x004	OpB
1	0x008	OpC
1	0x00C	J 0x100
...		
2	0x100	OpD
2	0x104	J 0x204
...		
3	0x204	OpE
3	0x208	J 0x30C
...		
4	0x30C	OpF
4	0x310	OpG
5	0x314	OpH

**Execuția programului fără constrângeri de aliniere**

OpA	F	D	A0	A1	W
OpB	F	D	B0	B1	W
OpC	F	D	B0	B1	W
J	F	D	A0	A1	W
OpD		F	D	B0	B1 W
J		F	D	A0	A1 W
OpE		F	D	B0	B1 W
J		F	D	A0	A1 W
OpF		F	D	A0	A1 W
OpG		F	D	B0	B1 W
OpH		F	D	A0	A1 W

# Logica pentru FETCH și alinierea

Cu constrângeri de aliniere

Cyc	Addr	Instr
?	0x000	OpA
?	0x004	OpB
?	0x008	OpC
?	0x00C	J 0x100
...		
?	0x100	OpD
?	0x104	J 0x204
...		
?	0x204	OpE
?	0x208	J 0x30C
...		
?	0x30C	OpF
?	0x310	OpG
?	0x314	OpH

0x000	0	0	1	1	
...					
0x100	2	2			
...					
0x200	3	X	3	4	X
...					
0x300			X	5	
0x310	6	6			

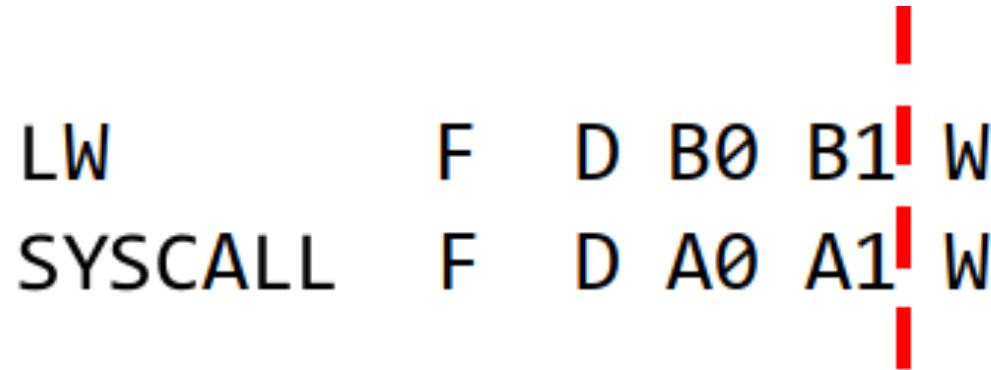
# Logica pentru FETCH și alinierea

Execuția programului cu constrângeri de aliniere

Cyc	Addr	Instr	F	D	A0	A1	W
1	0x000	OpA	F	D	A0	A1	W
1	0x004	OpB	F	D	B0	B1	W
2	0x008	OpC	F	D	B0	B1	W
2	0x00C	J 0x100	F	D	A0	A1	W
3	0x100	OpD	F	D	B0	B1	W
3	0x104	J 0x204	F	D	A0	A1	W
4	0x200	?	F	-	-	-	-
4	0x204	OpE	F	D	A0	A1	W
5	0x208	J 0x30C	F	D	A0	A1	W
5	0x20C	?	F	-	-	-	-
6	0x308	?	F	-	-	-	-
6	0x30C	OpF	F	D	A0	A1	W
7	0x310	OpG	F	D	A0	A1	W
7	0x314	OpH	F	D	B0	B1	W

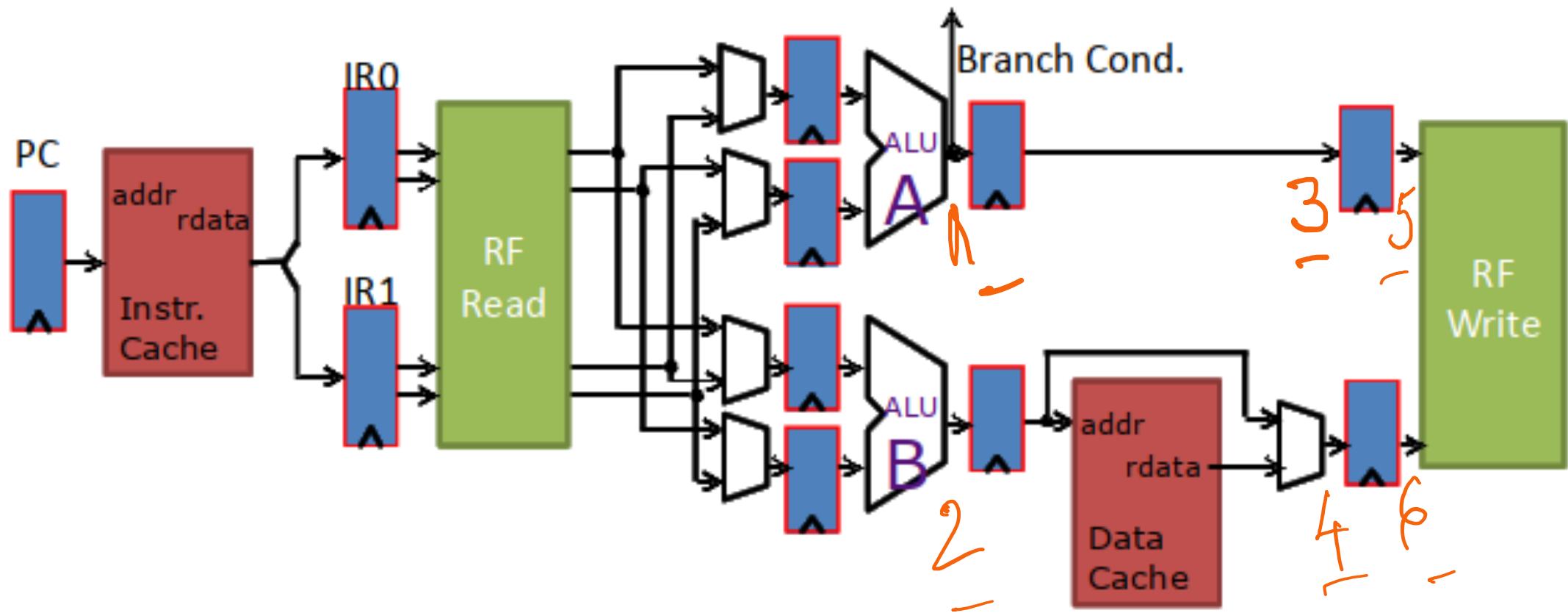
# Excepții în procesoare superscalare

- Este necesar să memorăm ordinea apariției excepțiilor (până acum am memorat ordinea instrucțiunilor din program pentru a identifica dependințele de date)

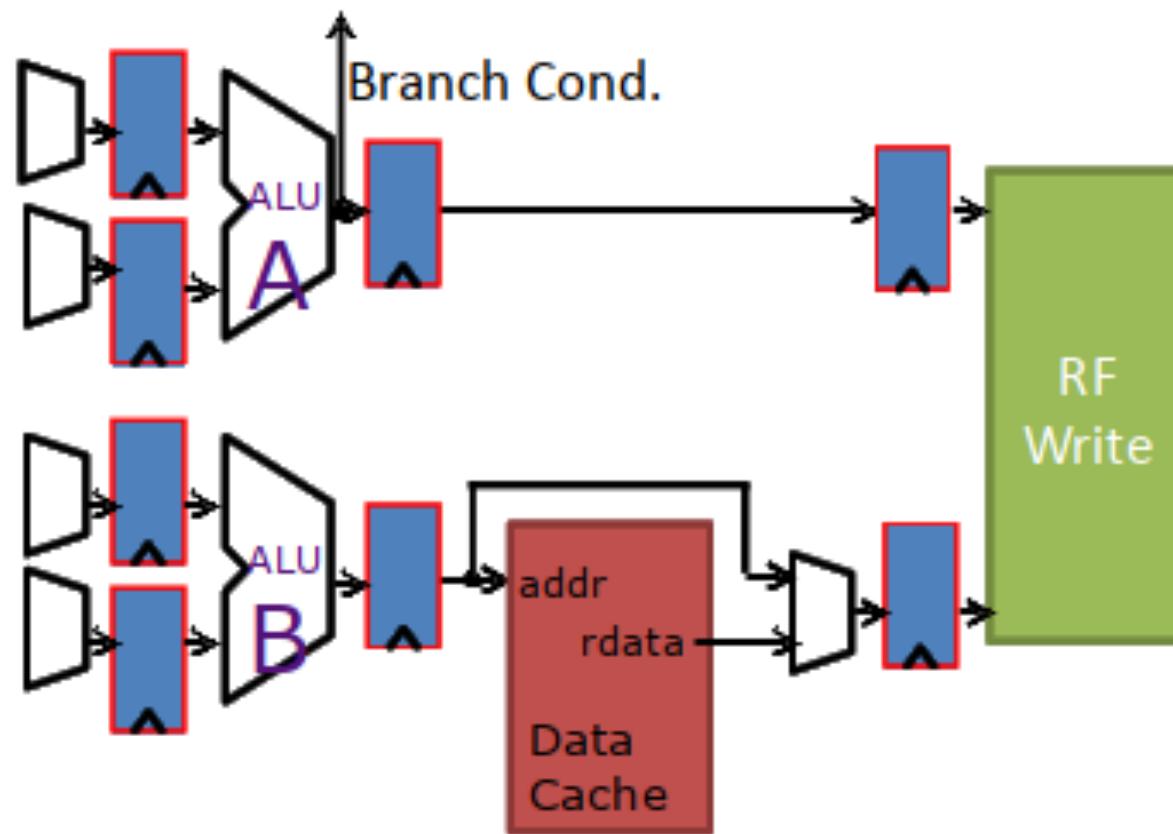


- Instrucțiunea LW este în pipe-ul B, dar comite prima în ordine logică

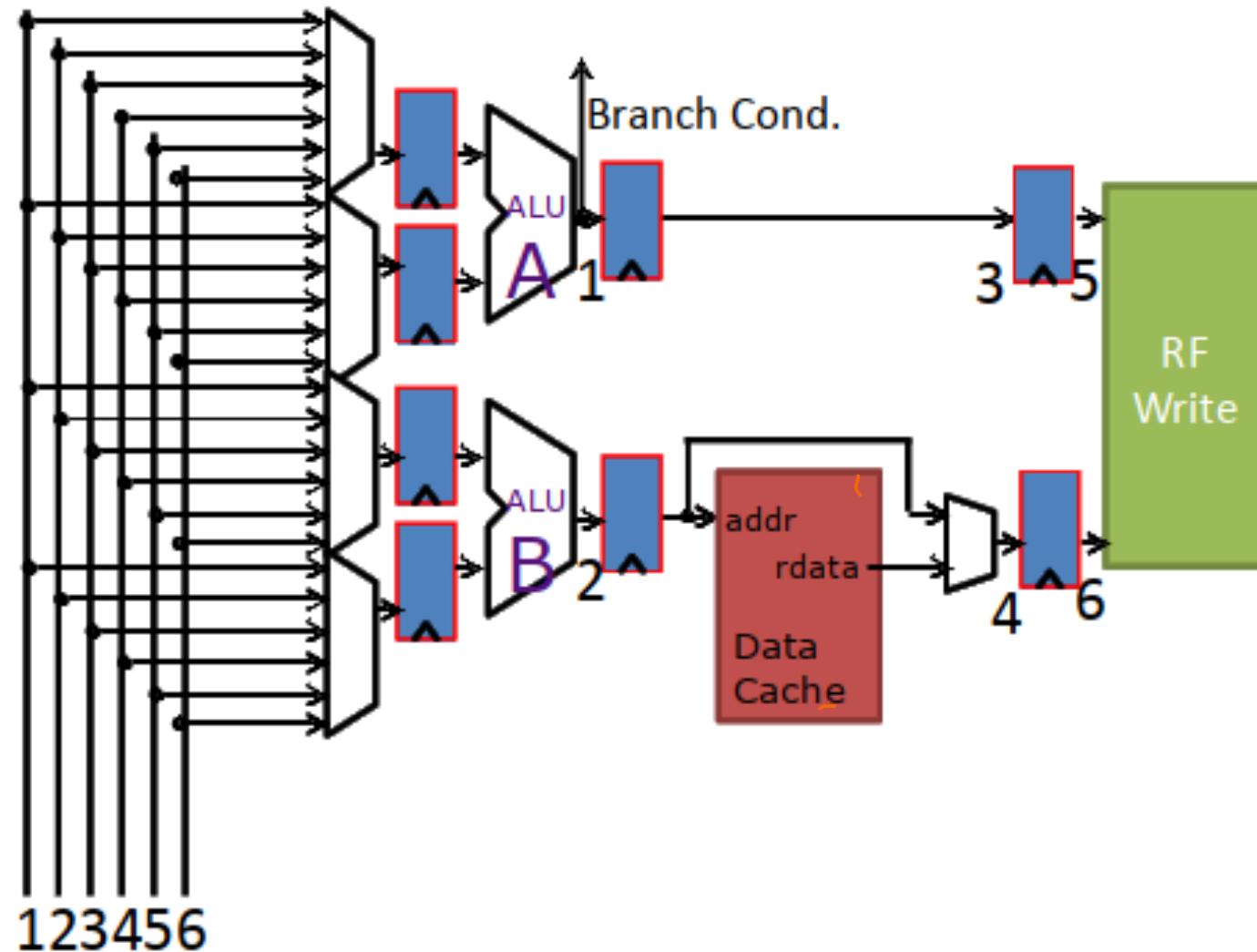
# Bypassing-ul în procesoarele superscalare



# Bypassing-ul în procesoarele superscalare



# Bypassing-ul în procesoarele superscalare



# Împărțirea stagiilor de DECODE și ISSUE

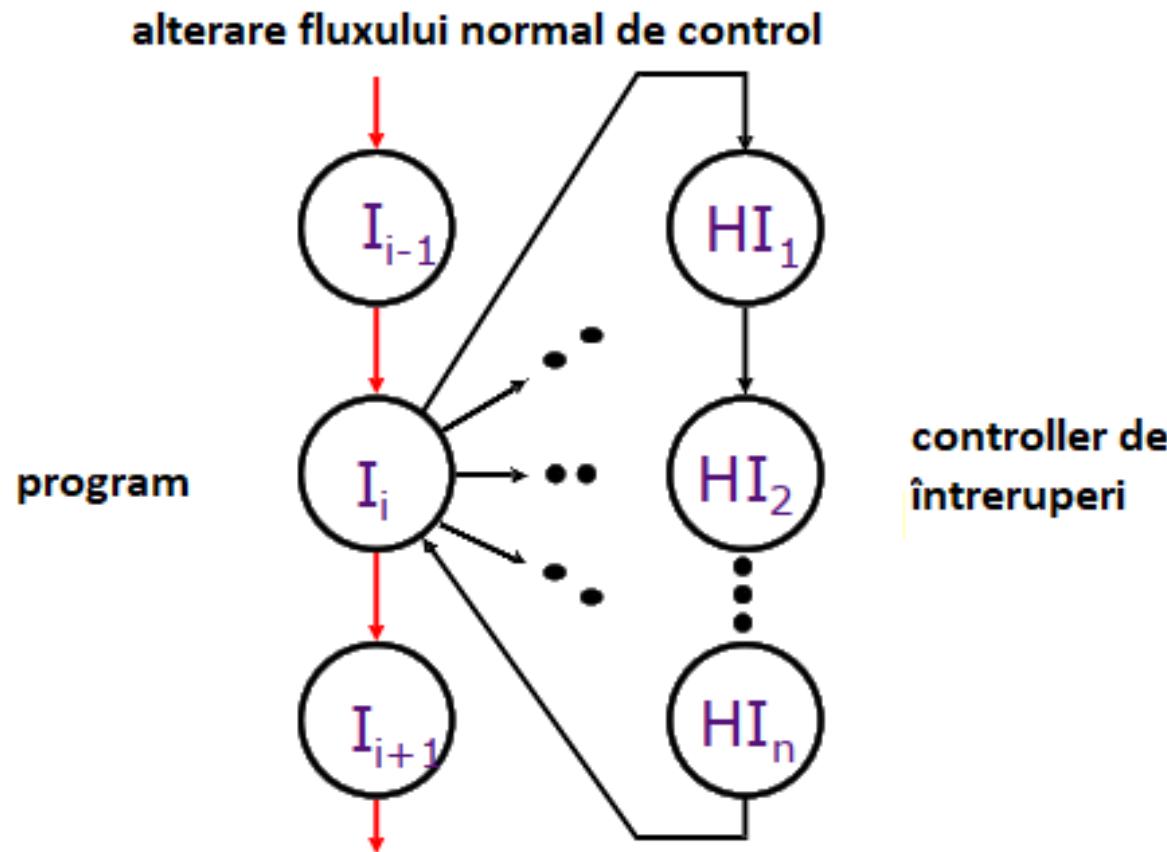
- Rețeaua de bypass poate deveni foarte complexă
- O soluție ar fi să împărțim stagiile DECODE și ISSUE
  - D = DECODE – posibilitate de a rezolva hazardurile structurale
  - I = Citirea fișierului de registre (Register File), bypass, Issue

	OpA	F	D	I	A0	A1	W	
	OpB	F	D	I	B0	B1	W	
	OpC		F	D	I	A0	A1	W
	OpD		F	D	I	B0	B1	W

# Costul BRANCH pentru superscalar

BEQZ	F	D	I	A0	A1	W			
OpA	F	D	I	B0	-	-			
OpB		F	D	I	-	-			
OpC		F	D	I	-	-			
OpD			F	D	-	-			
OpE			F	D	-	-			
OpF			F	-	-	-			
OpG			F	-	-	-			
OpH				F	D	I	A0	A1	W
OpI				F	D	I	B0	B1	W

# Întreruperi



Un eveniment intern sau extern care necesită a fi procesat de către un alt program (de sistem)

Evenimentul este unul neașteptat sau rar din punctul de vedere al programului

# Cauzele exceptiilor

Întreruperile – un **eveniment** care necesită atenția procesorului

- Asincrone – un eveniment extern
  - Dispozitive I/O care trebuie reparate
  - Expirarea timpului
  - Întreruperile de curent
  - Erori hardware
- Sincrone – o excepție internă
  - OPCODE nedefinit, instrucțiuni privilegiate
  - Overflow aritmetic, excepție FPU
  - Accese de memorie nealiniate
  - Excepții la memoria virtuală – page faults, TLB MISS, violări ale protecției
  - Excepții software: acces sistem (de ex salt în kernel)

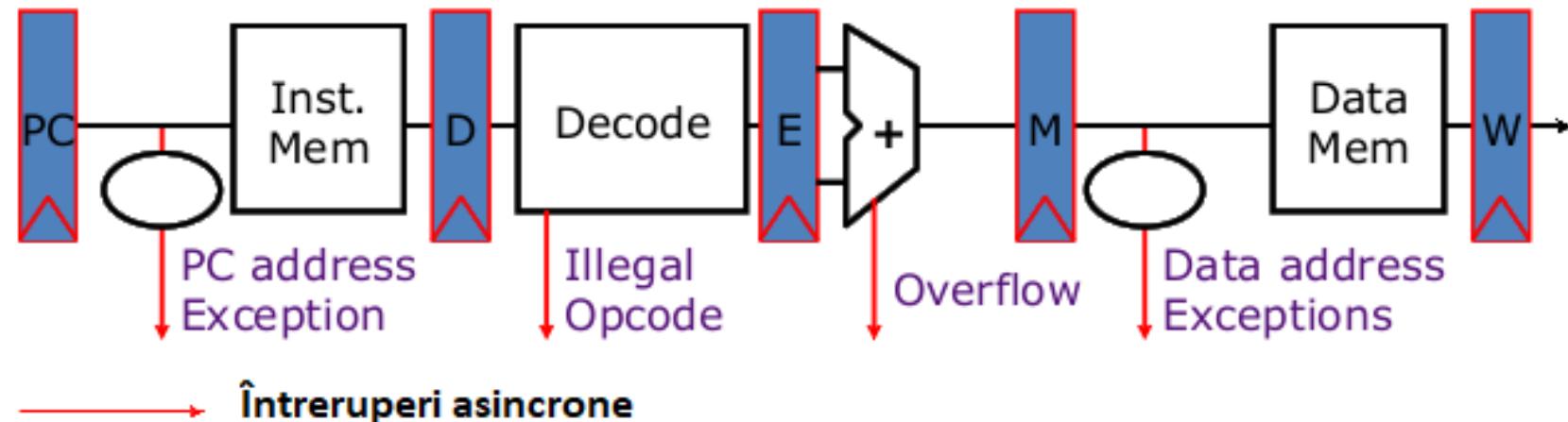
# Întreruperi asincrone

- Un dispozitiv de I/O solicită atenție prin activarea uneia dintre liniile de cerere pentru prioritizarea îնtreruperii
- Când procesorul decide să proceseze înntreruperea
  - Oprește programul curent la instrucțiunea  $I(i)$ , terminând complet execuția tuturor instrucțiunilor până la  $I(i-1)$  – *înntrerupere precisă*.
  - Salvează PC-ul instrucțiunii  $I(i)$  într-un registru special EPC
  - Dezactivează înntreruperile și transferă controlul către o înntrerupere desemnată de controller-ul de înntreruperi care rulează în kernel mode

# Controller-ul de întreruperi

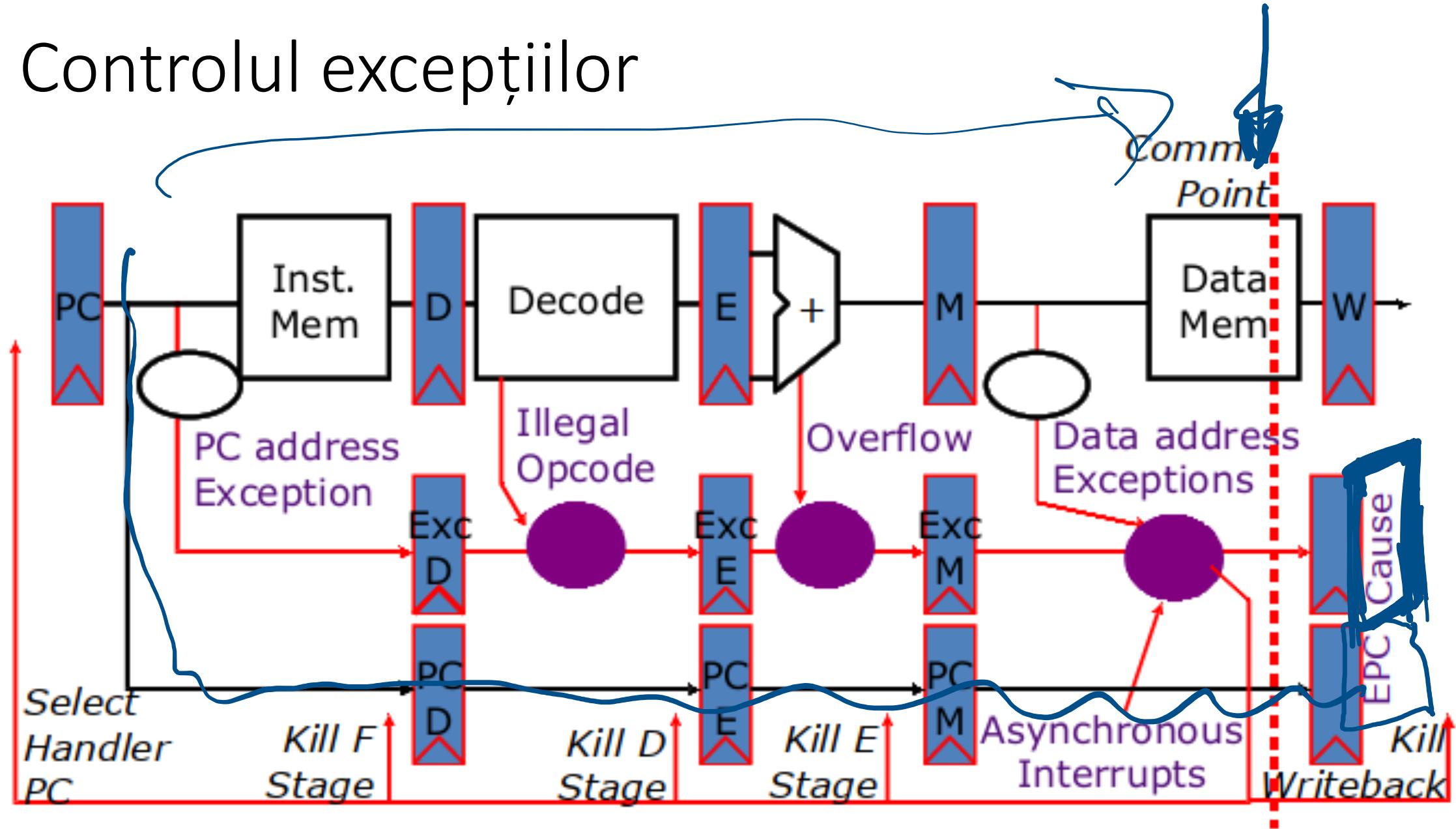
- Salvează registrul EPC înainte de reactivarea întreruperilor pentru a permite întreruperi imbricate
  - Avem nevoie de o instrucțiune pentru a muta EPC în registrele de uz general
  - Avem nevoie de o metodă de a masca întreruperile viitoare până când registrul EPC poate fi salvat
- Avem nevoie să citim un registru de stare care indică cauza întreruperii
- Utilizăm o instrucțiune specială de jump indirect (**Return From Exception**) pentru a ne întoarce la codul de executat:
  - Activarea întreruperilor
  - Restaurarea stării procesorului la modul utilizator
  - Restaurarea stării hardware-ului și stării de control

# Controlul exceptiilor



- Cum controlăm mai multe exceptii simultane dacă apar în diferite stagii de pipe ?
- Cum și unde controlăm îնtreruperile externe asincrone ?

# Controlul exceptiilor



# Controlul excepțiilor

- Menține un flag de excepție în pipe până la commit point (CP)
- Excepțiile în stagiile anterioare ale pipe-ului vor suprascrie excepțiile mai târzii *pentru o instrucțiune dată*
- Injectarea de intreruperi externe la commit point (CP) – se vor suprascrie celelalte
- Dacă avem o excepție la CP atunci face update la registrele Cause și EPC, eliminăm toate stagiile și injectăm controller-ul PC-ului în stagiul FETCH

# Specularea excepțiilor

- Mecanism de predicție
  - Excepțiile sunt rare, deci cel mai simplu mecanism de predicție – no exceptions – este unul foarte precis
- Verificarea mecanismului de predicție
  - Excepțiile detectate la finalul execuției instrucției în pipe – hardware special pentru numeroase tipuri de excepții
- Mecanismul de recuperare
  - Doar scriem starea arhitecturală la commit point
  - Lansarea controller-ului de excepții după golirea pipe-ului
- Bypass permite utilizarea rezultatelor instrucțiunilor necomise de către instrucțiunile următoare

# Diagramă pipe excepții

	<i>time</i>									
	t0	t1	t2	t3	t4	t5	t6	t7	....	
(I <sub>1</sub> ) 096: ADD	IF <sub>1</sub>	ID <sub>1</sub>	EX <sub>1</sub>	MA <sub>1</sub>	nop					overflow!
(I <sub>2</sub> ) 100: XOR		IF <sub>2</sub>	ID <sub>2</sub>	EX <sub>2</sub>	nop	nop				
(I <sub>3</sub> ) 104: SUB			IF <sub>3</sub>	ID <sub>3</sub>	nop	nop	nop			
(I <sub>4</sub> ) 108: ADD				IF <sub>4</sub>	nop	nop	nop	nop		
(I <sub>5</sub> ) Exc. Handler code					IF <sub>5</sub>	ID <sub>5</sub>	EX <sub>5</sub>	MA <sub>5</sub>	WB <sub>5</sub>	

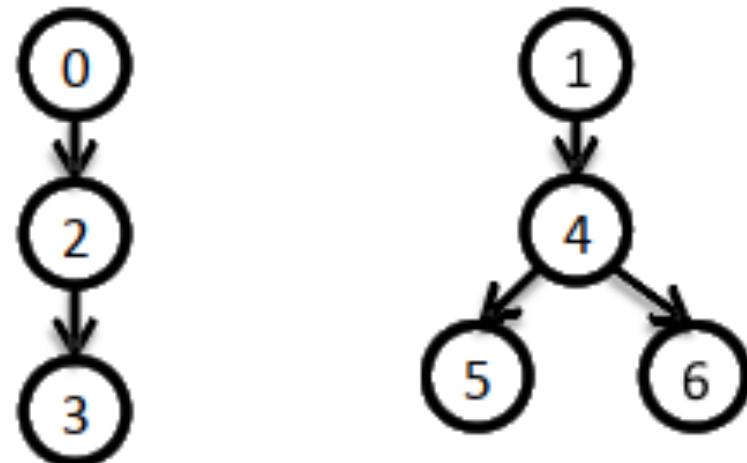
<i>Resource Usage</i>	<i>time</i>									
	t0	t1	t2	t3	t4	t5	t6	t7	....	
	IF	I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>				
	ID		I <sub>1</sub>	I <sub>2</sub>	I <sub>3</sub>	nop	I <sub>5</sub>			
	EX			I <sub>1</sub>	I <sub>2</sub>	nop	nop	I <sub>5</sub>		
	MA				I <sub>1</sub>	nop	nop	nop	I <sub>5</sub>	
	WB					nop	nop	nop	nop	I <sub>5</sub>

# 000 (out – of order)

Name	Frontend	Issue	Writeback	Commit	
I4	IO	IO	IO	IO	Fixed Length Pipelines Scoreboard
I2O2	IO	IO	000	000	Scoreboard
I2OI	IO	IO	000	IO	Scoreboard, Reorder Buffer, and Store Buffer
I03	IO	000	000	000	Scoreboard and Issue Queue
IO2I	IO	000	000	IO	Scoreboard, Issue Queue, Reorder Buffer, and Store Buffer

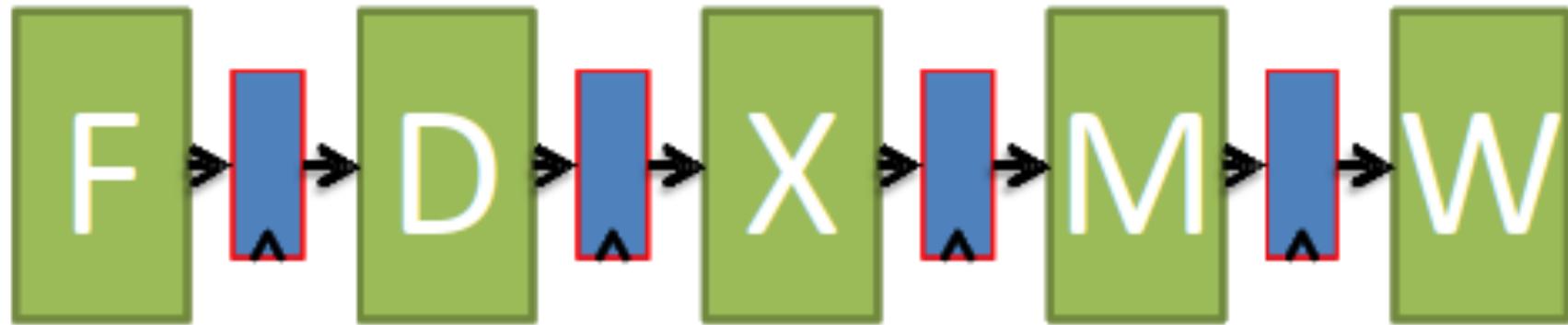
# Motivare 000

```
0 MUL    R1, R2, R3  
1 ADDIU  R11,R10,1  
2 MUL    R5, R1, R4  
3 MUL    R7, R5, R6  
4 ADDIU  R12,R11,1  
5 ADDIU  R13,R12,1  
6 ADDIU  R14,R12,2
```

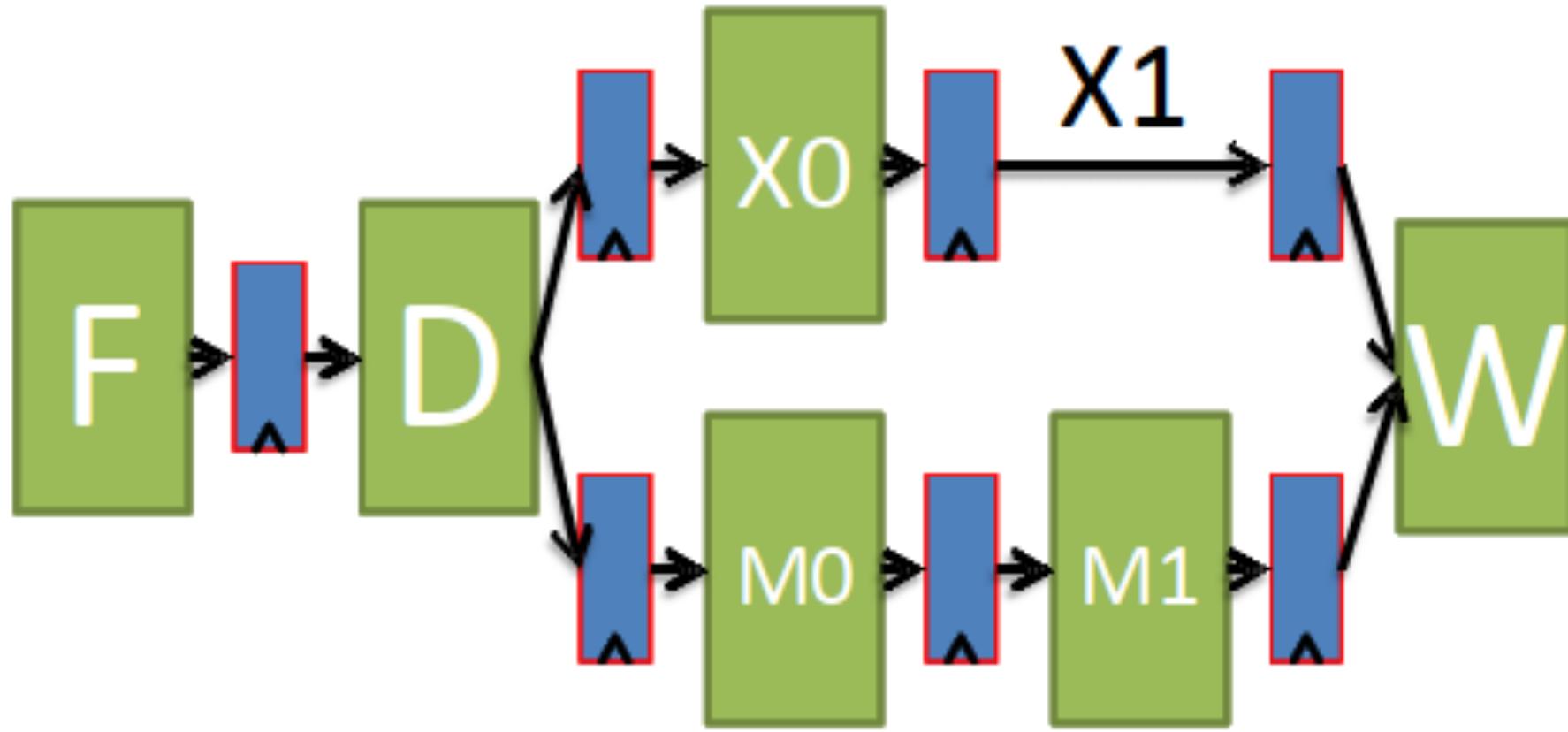


- Două secvențe independente de instrucțiuni oferă flexibilitate în termeni de cum instrucțiunile sunt programate în ordine
- Putem programa instrucțiunile in software (STATIC) sau în hardware (DINAMIC)

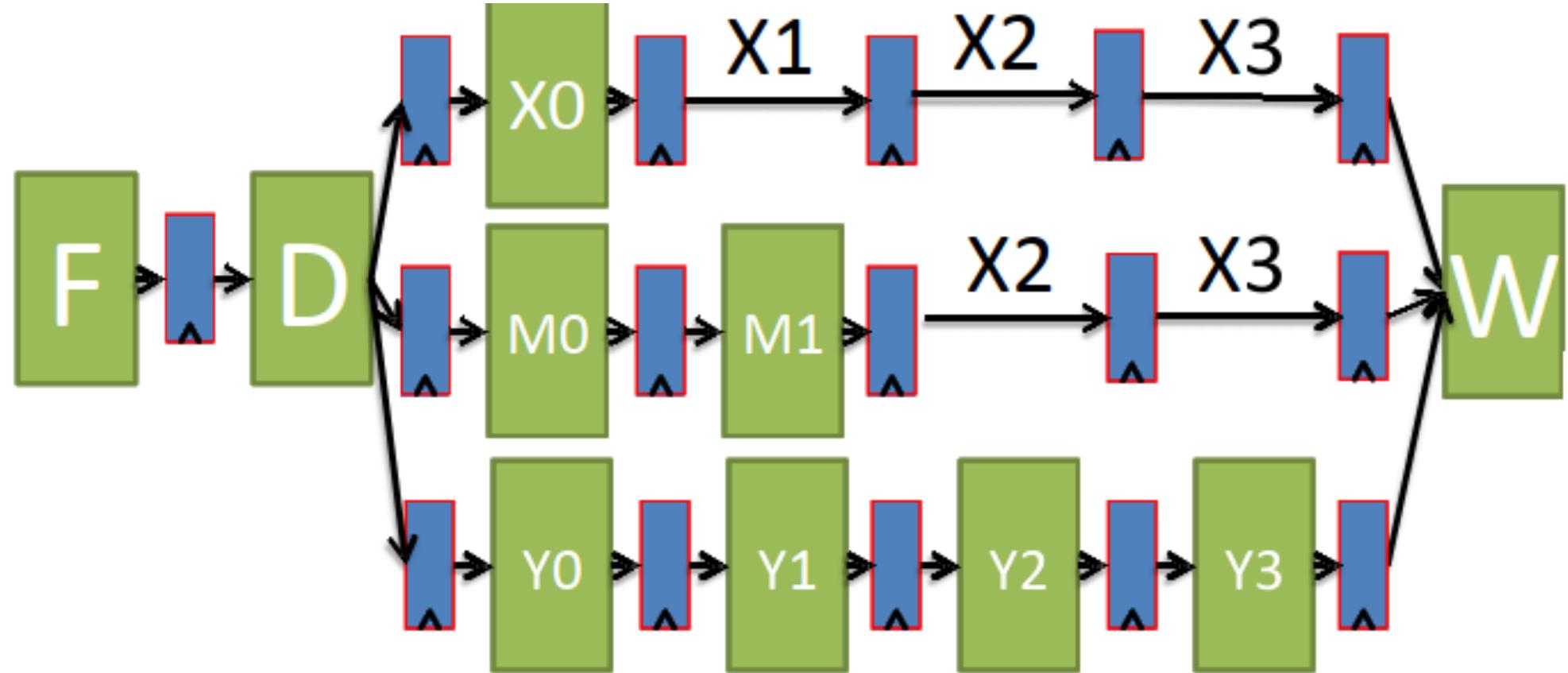
# I4 – In-Order Front-End, Issue, Writeback, Commit



# I4 – In-Order Front-End, Issue, Writeback, Commit



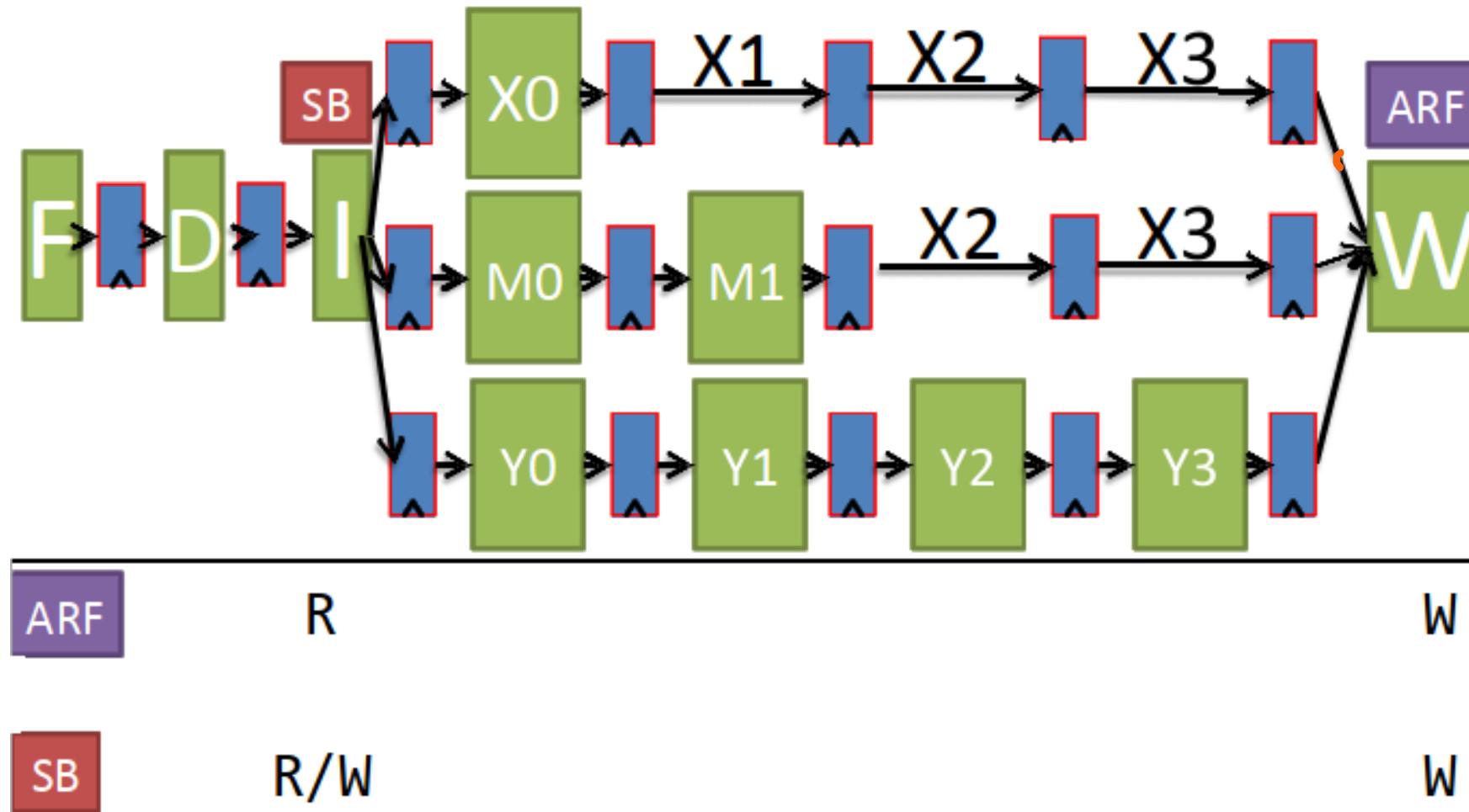
# I4 – In-Order Front-End, Issue, Writeback, Commit (MUL – 4 stagii)



Pentru a evita creșterea CPI este nevoie de full bypassing ceea ce poate conduce la costuri mari

Adăugăm stagiul ISSUE în care se citește fișierul de registre și instrucțiunea este trimisă către unitatea funcțională

# I4 – In-Order Front-End, Issue, Writeback, Commit (MUL – 4 stagii)



# SCOREBOARD - SB

	DATE DISPONIBILE						
	P	F	4	3	2	1	0
R1							
R2							
R3							
...							
R31							

P : Pending : scrie registrul destinație în timpul execuției

F : Care unitate funcțională scrie registrul

DD – Unde sunt scrise datele în unitatea funcțională din pipe

- O valoare de „1” în DD – înseamnă că data rezultat este în stagiul I al unității funcționale F
- Putem utiliza câmpurile F și DD pentru a determina când avem un bypass și unde facem bypass-ul
- O valoare „1” în coloana 0 înseamnă că ciclul unității funcționale este în stagiul WB
- Biții din DD – la fiecare ciclu SHR

# SCOREBOARD - SB

	P	F	4	3	2	1	0
R1				1			
R2							
R3							
...							
R31							

**DATE DISPONIBILE**

The diagram illustrates a scoreboard with 31 rows, labeled R1 through R31. Each row contains a vertical bar divided into three segments: P, F, and 4. A red arrow points from the number 1 in the fourth column of the first row (R1) to the right end of the bar for each row, indicating that all rows are available at the same date.

# SCOREBOARD - SB

0	MUL	R1, R2, R3	F D I	Y0 Y1 Y2 Y3 W
1	ADDIU	R11, R10, 1	F D I	X0 X1 X2 X3 W
2	MUL	R5, R1, R4	F D I I	Y0 Y1 Y2 Y3 W
3	MUL	R7, R5, R6	F D D I I I	Y0 Y1 Y2 Y3 W
4	ADDIU	R12, R11, 1	F F F D D D D I	X0 X1 X2 X3 W
5	ADDIU	R13, R12, 1	F F F F D I	X0 X1 X2 X3 W
6	ADDIU	R14, R12, 2	F D I	X0 X1 X2 X3 W

Cyc D I      4 3 2 1 0      Dest Regs

1 0

2 1 0

3 2 1

1

4

1 1

5

1 1

6 3 2

1 1

7

1 1 1

8

1 1

9

1

10 4 3

1

11 5 4

1 1

12 6 5

1 1

13 6

1 1 1

14

1 1 1 1

15

1 1 1 1

16

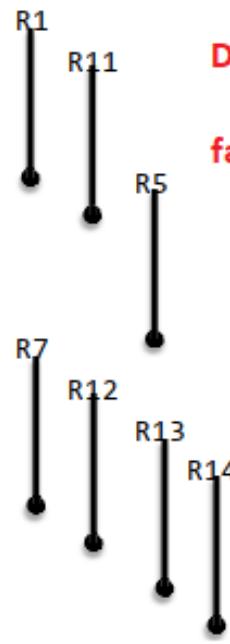
1 1 1

17

1 1

18

1



Dacă ne uităm la stagiu F putem

face bypass pe acest ciclu

# SCOREBOARD - SB

	P	F	4	3	2	1	0
R1			1				
R2							
R3							
...							
R31							

**DATE DISPONIBILE**

The diagram illustrates the availability dates for each row. Red arrows point from the date columns to the right edge of the board, indicating the range of dates for each row.

# **CALCULATOARE NUMERICE 2**

## curs 7

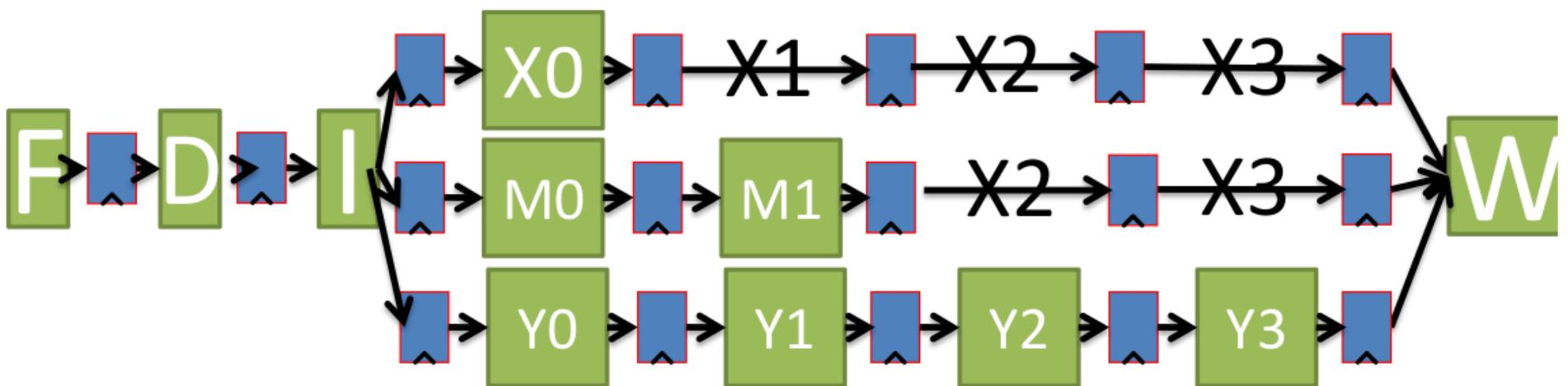
# Conținutul cursului

- **1. Finalizare superscalar**
  - Speculații și salturi
  - Redenumirea regisitrelor
  - Eliminarea ambiguităților din memorie
  - Limitările procesoarelor OOO

# Specularea și salturile în I4

0	MUL	R1, R2, R3	F	D	I	Y0	Y1	Y2	Y3	W
1	ADDIU	R4, R5, 1		F	D	I	X0	X1	X2	X3 W
2	MUL	R6, R1, R4		F	D	I	I	I	Y0	Y1 Y2 Y3 W
3	BEQZ	R6, Target		F	D	D	D	I	I	I X0 X1 X2 X3 W
4	ADDIU	R8, R9 ,1		F	F	F	D	D	D	I -- -- -- --
5	ADDIU	R10,R11,1		F	F	F	F	D	-- -- -- --	--
6	ADDIU	R12,R13,1							F	-- -- -- --
T									F D I . . .	

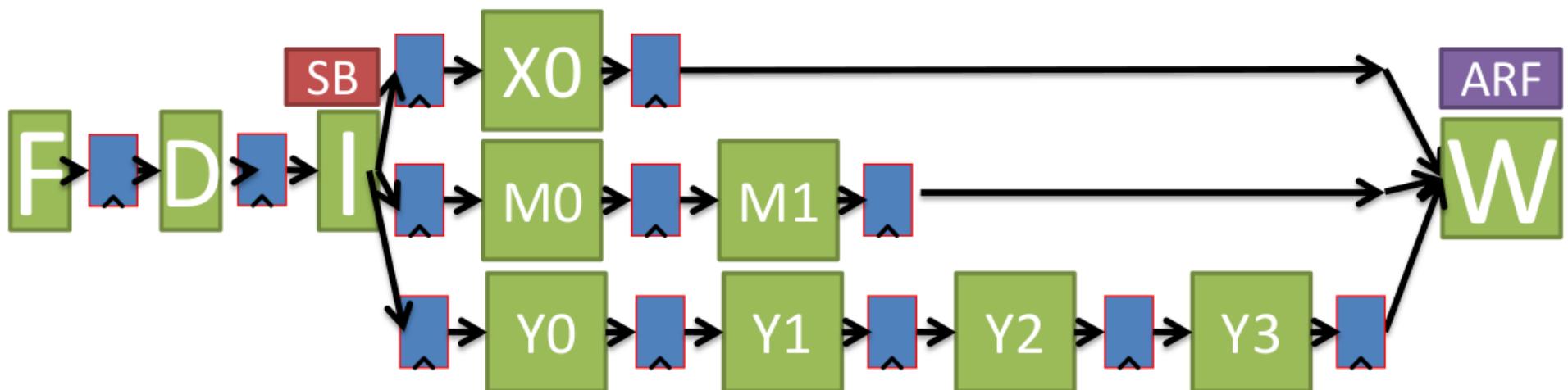
Nu avem instrucțiuni speculative în stare commit



# Specularea și salturile în I2O2

0 MUL R1, R2, R3 F	D	I	Y0	Y1	Y2	Y3	W		
1 ADDIU R4, R5, 1	F	D	I	X0	W				
2 MUL R6, R1, R4	F	D	I	I	Y0	Y1	Y2	Y3	W
3 BEQZ R6, Target	F	D	D	D	I	I	I	I	X0 W
4 ADDIU R8, R9 ,1	F	F	F	D	D	D	D	I	-- --
5 ADDIU R10,R11,1	F	F	F	F	D	--	--	--	--
6 ADDIU R12,R13,1	F	--	--	--	--	--	--	--	--
T					F	D	I	.	.

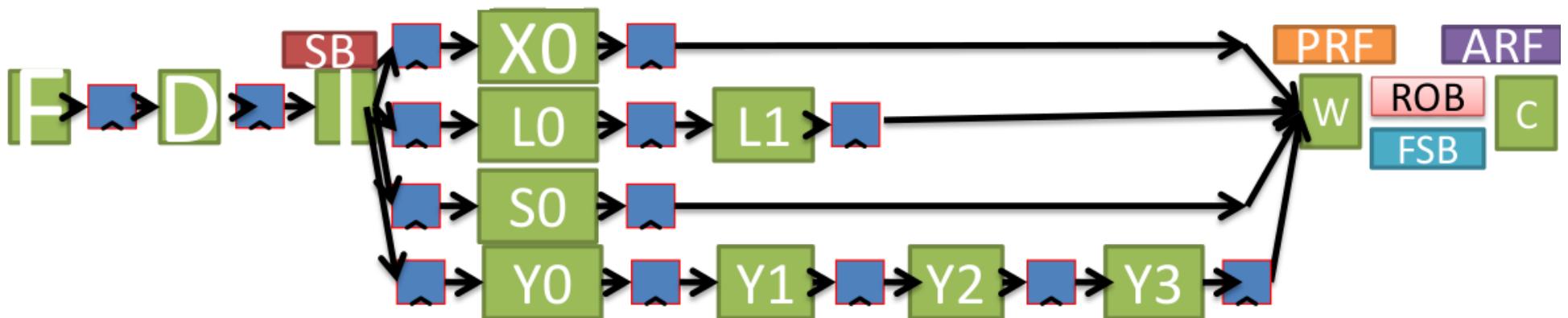
Nu avem instrucțiuni speculative în starea de commit



# Specularea și salturile în I2OI

0	MUL	R1, R2, R3	F	D	I	Y0	Y1	Y2	Y3	W	C			
1	ADDIU	R4, R5, 1		F	D	I	X0	W	r		C			
2	MUL	R6, R1, R4		F	D	I	I	I	Y0	Y1	Y2	Y3	W	C
3	BEQZ	R6, Target		F	D	D	D	I	I	I	I	X0	W	C
4	ADDIU	R8, R9 ,1		F	F	F	D	D	D	D	I	--	--	--
5	ADDIU	R10,R11,1			F	F	F	F	D	--	--	--	--	
6	ADDIU	R12,R13,1				F	--	--	--	--	--			
T						F	D	I	.	.	.			

1. Instrucțiunile după branch trebuie eliminate pentru a preveni scrierea PRF-ului
2. Putem elimina din ROB imediat sau așteptăm până commitim



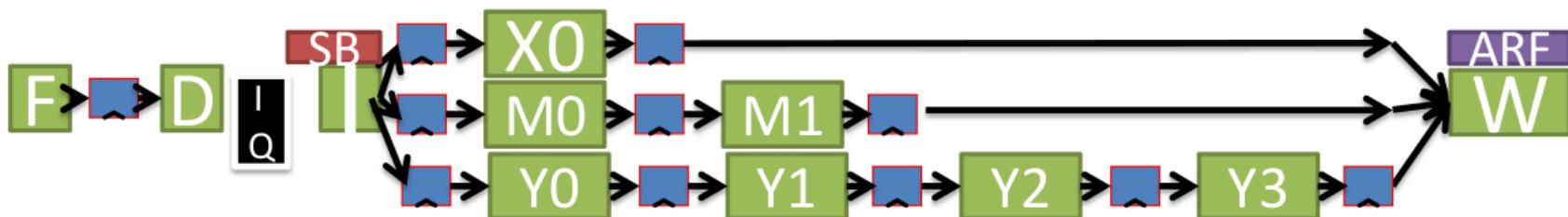
# Specularea și salturile în IO3

0 MUL R1, R2, R3	F D I Y0 Y1 Y2 Y3 W
1 ADDIU R4, R5, 1	F D I X0 W
2 MUL R6, R1, R4	F D i I Y0 Y1 Y2 Y3 W
3 BEQZ R6, Target	F D i I X0 W
4 ADDIU R8, R9 ,1	F D i I X0 W
5 ADDIU R10,R11,1	F D i I X0 W
6 ADDIU R12,R13,1	F D i I X0 W
7 ???	F D
8 ???	F D
9 ???	F D
10???	F D
11???	F D
T	F D I . . .

Instrucțiunile speculative scriu ARF-ul

Nu avem control pentru speculații în IO3

Putem face stall în caz de branch

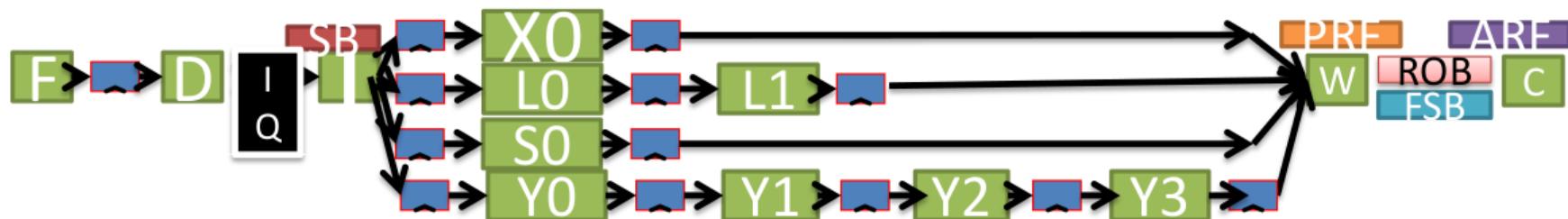


# Specularea și salturile în IO2I

				F	D	I	Y0	Y1	Y2	Y3	W	C	
0	MUL	R1, R2, R3		F	D	I	Y0	Y1	Y2	Y3	W	C	
1	ADDIU	R4, R5, 1		F	D	I	X0	W	r		C		
2	MUL	R6, R1, R4		F	D	i		I	Y0	Y1	Y2	Y3	W C
3	BEQZ	R6, Target		F	D	i				I	X0	W	C
4	ADDIU	R8, R9 ,1		F	D	i	I	X0	W	r	--		
5	ADDIU	R10,R11,1		F	D	i	I	X0	W	--			
6	ADDIU	R12,R13,1		F	D	i					--		
7	???				F	D					--		
8	???				F	D					--		
9	???				F	D					--		
10	???				F						--		
11	???				--						D		
T											F	D	I . . .

Este necesar să ștergem starea "Speculativ" din PRF

Necesită întoarcere înapoi selectivă

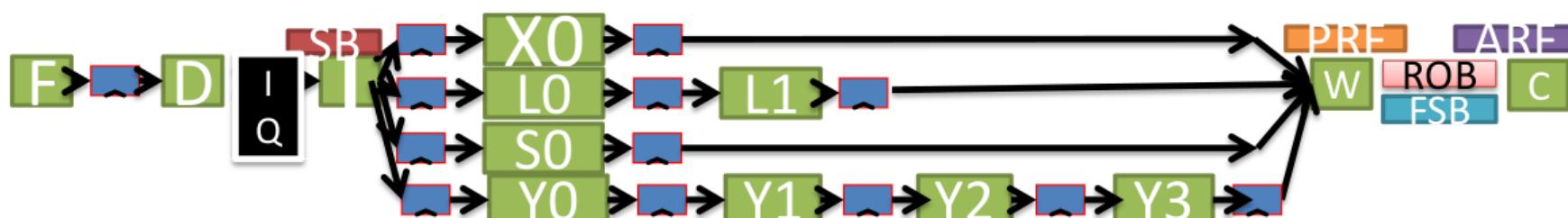


# Specularea și salturile în IO2I

0	MUL	R1, R2, R3	F	D	I	Y0	Y1	Y2	Y3	W	C		
1	ADDIU	R4, R5, 1	F	D	I	X0	W	r		C			
2	MUL	R6, R1, R4	F	D	i		I	Y0	Y1	Y2	Y3	W	C
3	BEQZ	R6, Target	F	D	i				I	X0	W	C	
4	ADDIU	R8, R9 ,1	F	D	i	I	X0	W	r				
5	ADDIU	R10,R11,1	F	D	i	I	X0	W	r				
6	ADDIU	R12,R13,1	F	D	i		I	X0					
7	???		F	D									
8	???		F	D									
9	???		F	D									
10	???		F	D									
11	???		F	D									
12	???		F										
13	???		F										
T										F	D	I	. . .

Instrucțiunile speculative vor fi scrise în PRF nu în ARF

În caz de predicție greșită suprascriem conținutul lui PRF cu conținutul lui ARF



## Redenumirea regisrelor

- WAW și WAR nu sunt depindețe de date adevărate
- Dependințele RAW sunt adevărate deoarece citirea necesită terminarea unei scrieri
- WAW și WAR există deoarece avem un număr limitat de specificatori de registre sau adrese de memorie

# Redenumirea regisrelor

0	MUL	R1, R2, R3	F D I	Y0 Y1 Y2 Y3	W C
1	MUL	R4, R1, R5	F D i	I Y0 Y1 Y2 Y3	W C
2	ADDIU	R6, R4, 1	F D i		I X0 W C
3	ADDIU	R4, R7, 1	F D i	I X0 W r	C

0	MUL	R1, R2, R3	F D I	Y0 Y1 Y2 Y3	W C
1	MUL	R4, R1, R5	F D i	I Y0 Y1 Y2 Y3	W C
2	ADDIU	R6, R4, 1	F D i		I X0 W C
3	ADDIU	R4, R7, 1	F D i	I X0 W r	C

0	MUL	R1, R2, R3	F D I	Y0 Y1 Y2 Y3	W C
1	MUL	R4, R1, R5	F D i	I Y0 Y1 Y2 Y3	W C
2	ADDIU	R6, R4, 1	F D i		I X0 W C
3	ADDIU	R4, R7, 1	F D i	I X0 W r	C

# Adăugarea de noi registre

Ruperea tuturor dependințelor care nu sunt adevărate

0	MUL	R1, R2, R3	F	D	I	Y0 Y1 Y2 Y3	W	C
1	MUL	R4, R1, R5	F	D	i	I Y0 Y1 Y2 Y3	W	C
2	ADDIU	R6, R4, 1	F	D	i	I X0 W	X0 W	C
3	ADDIU	R4, R7, 1	F	D	i	I X0 W r		C

Microarhitectura IO2I păstrează stall-urile

0	MUL	R1, R2, R3	F	D	I	Y0 Y1 Y2 Y3	W	C
1	MUL	R4, R1, R5	F	D	i	I Y0 Y1 Y2 Y3	W	C
2	ADDIU	R6, R4, 1	F	D	i	I X0 W	X0 W	C
3	ADDIU	R4, R7, 1	F	D	D	D D D D D	D D D D I X0 W	C

Redenumirea manuală a regisitrelor. Ce se întâmplă dacă utilizăm mai multe registre ?

0	MUL	R1, R2, R3	F	D	I	Y0 Y1 Y2 Y3	W	C
1	MUL	R4, R1, R5	F	D	i	I Y0 Y1 Y2 Y3	W	C
2	ADDIU	R6, R4, 1	F	D	i	I X0 W	X0 W	C
3	ADDIU	<b>R8</b> , R7, 1	F	D	i	I X0 W r		C

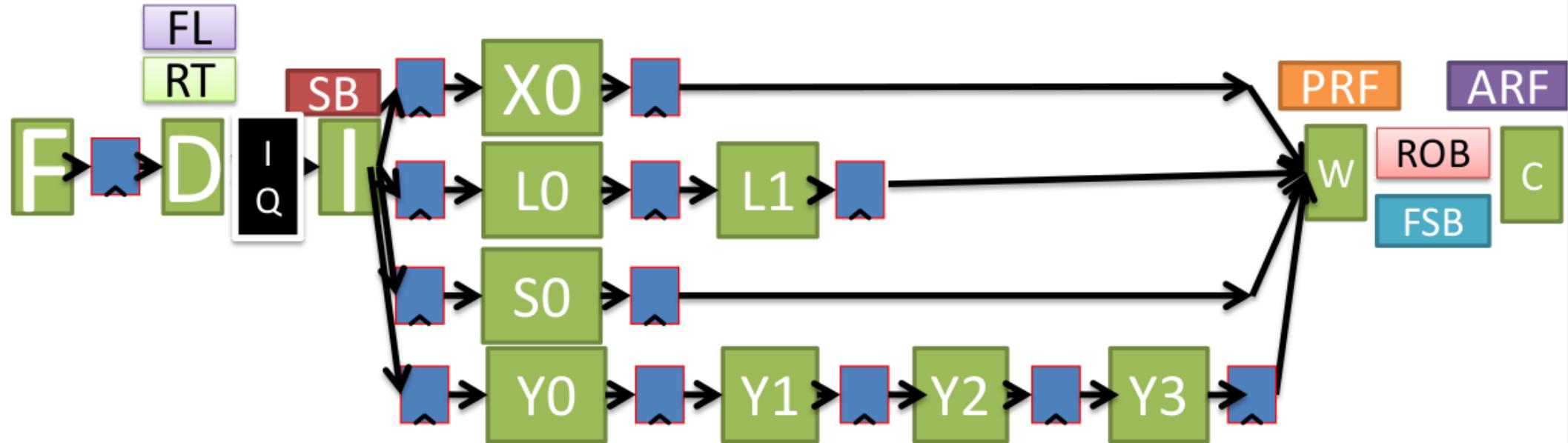
## Redenumirea registrelor

- Adăugarea de mai multe registre/memorii elimină dependințele dar namespațiul unei arhitecturi este limitat
  - Registre: namespace-uri mari necesită mai mulți biți pentru encodarea instrucțiunii – 32 registre necesită 5 biți, 128 de registre necesită 7 biți
- Redenumirea de registre în hardware pentru a elimina hazardurile WAW și WAR

## Redenumirea registrelor

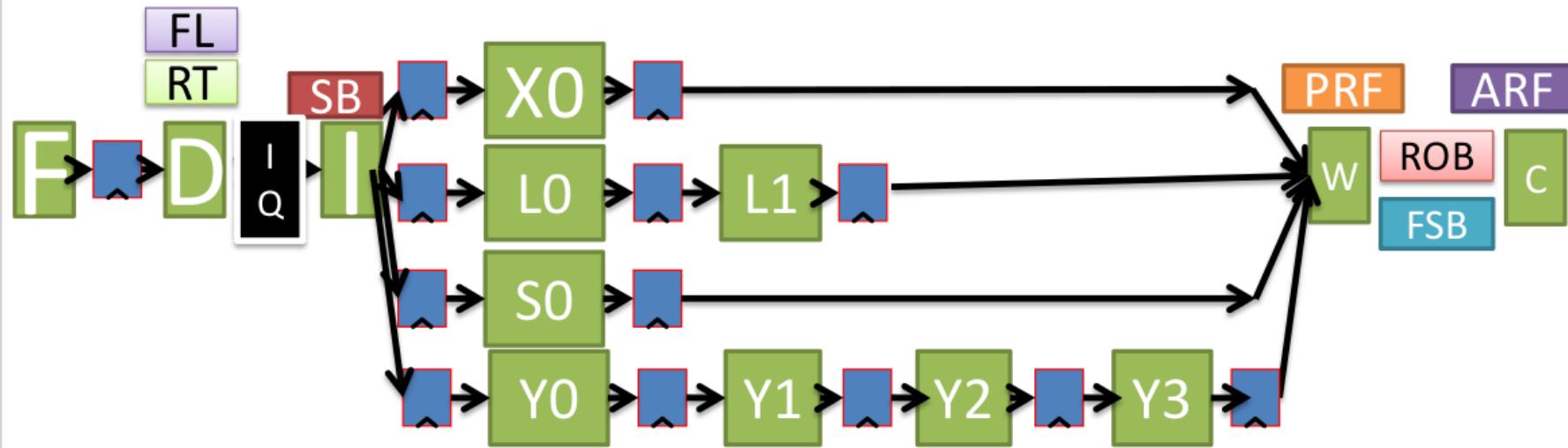
- 2 scheme
  - Utilizarea de pointeri în IQ (Instruction Queue) și/sau ROB(ReOrder Buffer)
    - Utilizarea de valori Values în IQ (Instruction Queue) și/sau ROR(ReOrder Buffer)
- IO2I utilizează pointeri în IQ și ROB

# IO2I – redenumire registre cu pointeri



- Toate structurile de date sunt ca în procesorul IO2I discutat deja cu următoarele excepții:
  - Se adaugă două câmpuri noi la ROB
  - Se adaugă Tabela de Redenumire (RT) și Lista Liberă (FL) a registrelor
- Se mărește dimensiunea PRF-ului pentru a suporta mai multe registre

# IO2I – redenumire registre cu pointeri



ARF		W
SB	R/W	W
PRF	R	W
ROB	R/W	W R/W
FSB		W R/W
IQ	W	W
RT	R/W	W
FL	R/W	W

# ROB modification

State	S	ST	V	Preg	Areg	Ppreg
--						
P						
F						
P						
P						
F						
P						
P						
--						
--						

**State:** {Free, Pending, Finished}

**S:** Speculative

**ST:** Store bit

**V:** Destination is valid

**Preg:** Physical Register File Specifier

**Areg:** Architectural Register File Specifier

**Ppreg:** Previous Physical Register

# Rename Table – RT and Free List (FL)

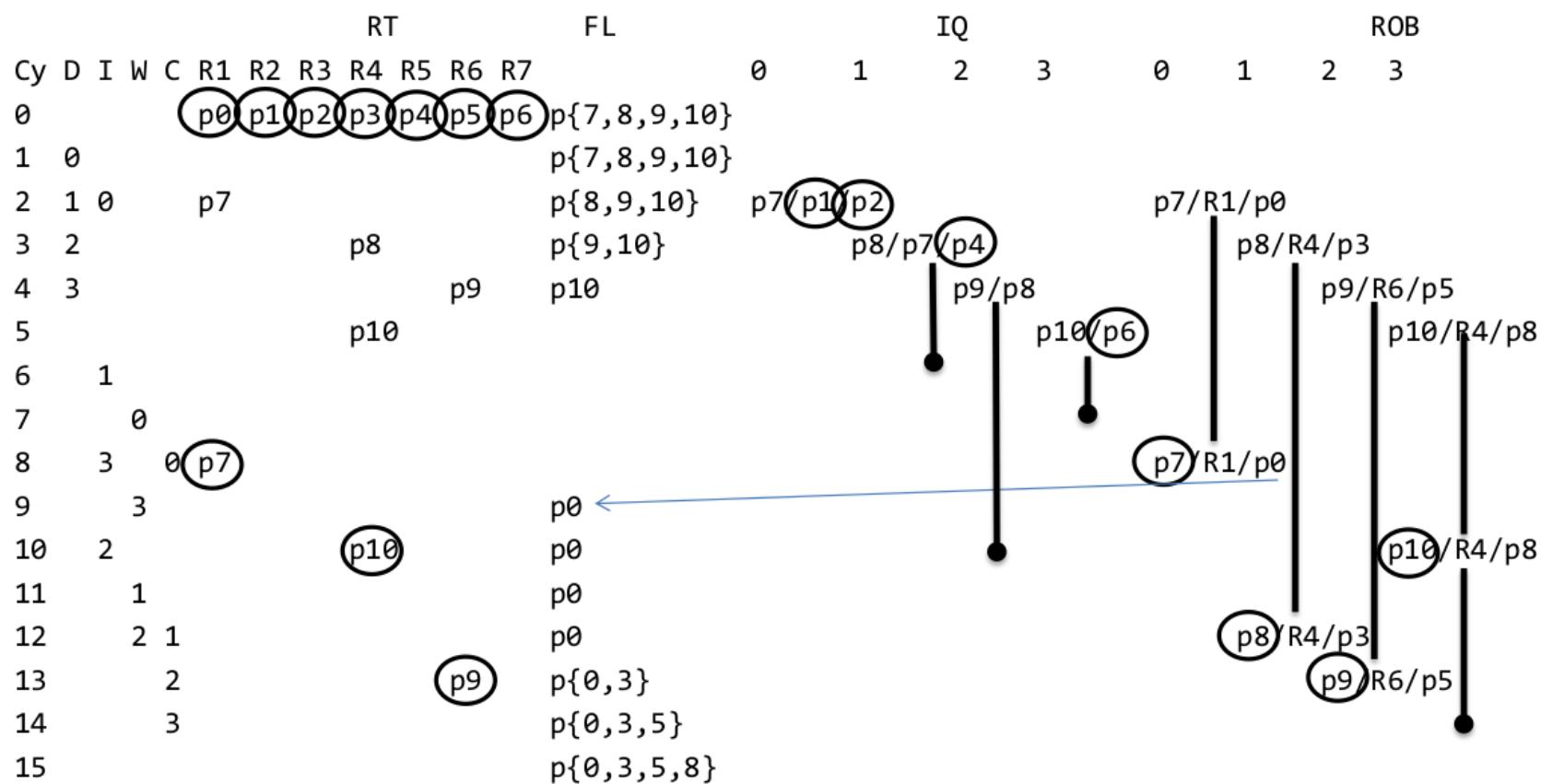
	P	Preg
R1		
R2		
R3		
...		
R31		

- P = Pending, scriere în destinație în modul „in flight”
- Preg = Registrul Arhitectural Fizic
- Free = Registrul este disponibil pentru redenumire
- Dacă Free ==0, PRF este în uz și nu poate fi utilizat pentru redenumire

	Free
p1	
p2	
p3	
...	
pN	

# Execuția programului

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	MUL	R1, R2, R3	F	D	I	Y0	Y1	Y2	Y3	W	C						
1	MUL	R4, R1, R5		F	D	i				I	Y0	Y1	Y2	Y3	W	C	
2	ADDIU	R6, R4, 1		F	D	i				I	X0	W		C			
3	ADDIU	R4, R7, 1		F	D	i			I	X0	W	r		C			



# Eliminarea ambiguităților din memorie

- RAW hazard al sistemului de memorie
- `st R1, 0(R2)`
- `ld R3, 0(R4)`
- Când putem executa LOAD ?

# Coada de memorie IO

- Executăm toate instrucțiunile de LOAD și STORE în ordinea trecută în program
- => LOAD și STORE nu pot părăsi IQ pentru execuție până când toate instrucțiunile de LOAD și STORE anterioare nu și-au terminat complet execuția
- Putem executa instrucțiuni de tipul LOAD și STORE într-o manieră OOO cu respectarea altor instrucțiuni (care nu necesită accesarea memoriei) ?
- Este nevoie o structură de pentru managementul ordonării memoriei

## Specularea adresei

- st R1, 0(R2)
- ld R3, 0(R4)

Ghicim că r4 != r2

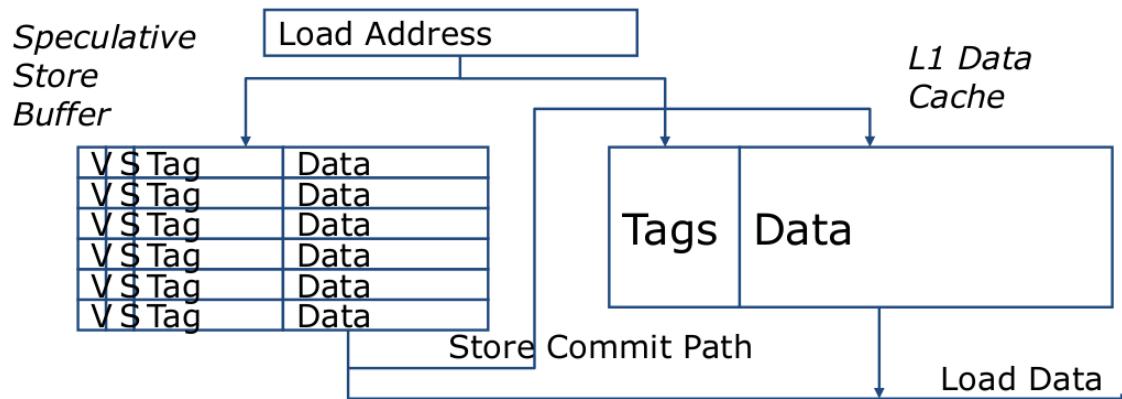
- Executăm LOAD înainte de STORE deoarece adresa este cunoscută
- Este necesar să ținem pe loc toate instrucțiunile LOAD/STORE executate dar necomise adresate conform programului de executat
- Dacă avem un miss pe predicție,  $r4==r2$ , eliminăm instrucțiunile LOAD și toate instrucțiunile următoare

=> Penalitate mare pentru predicție greșită

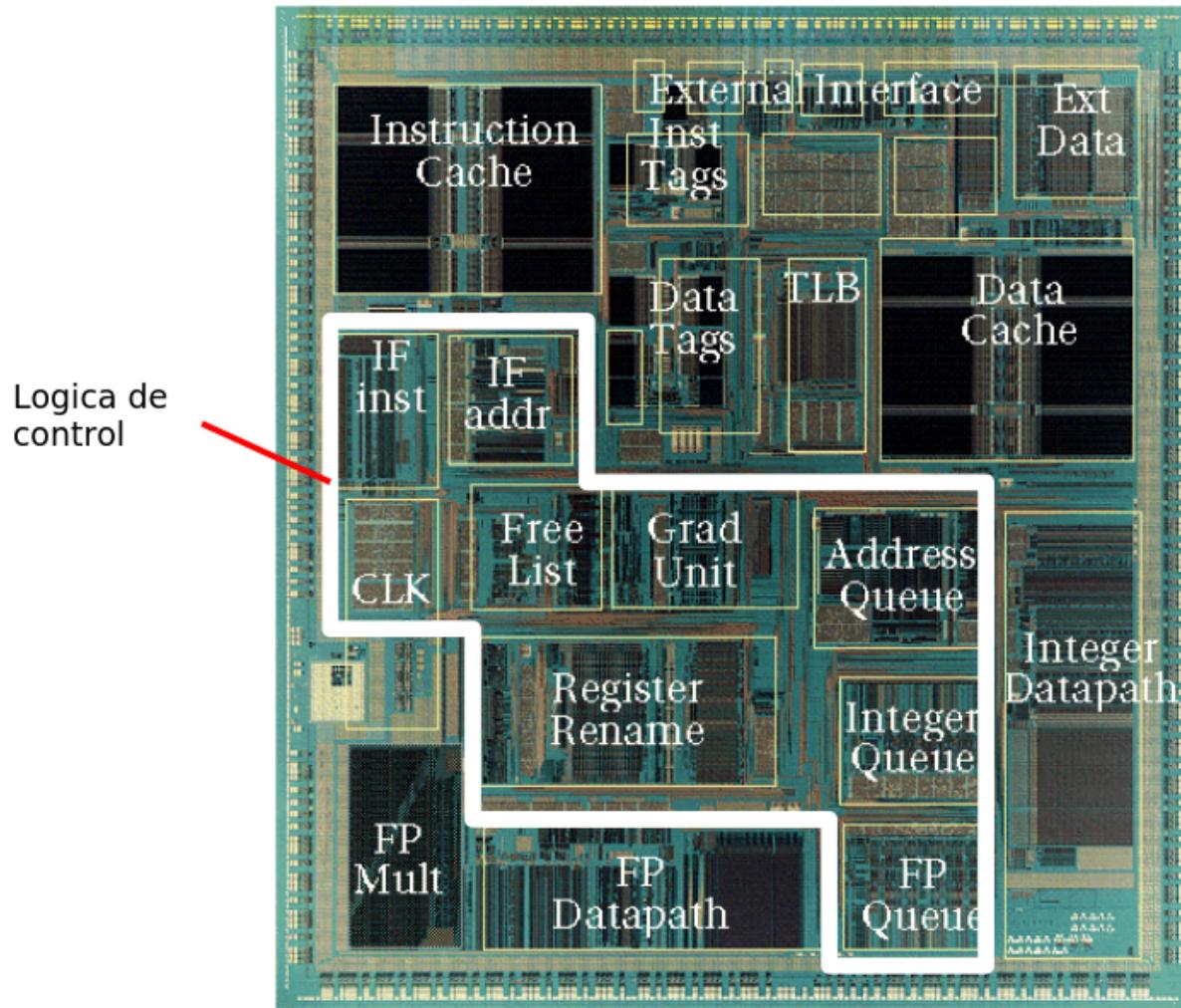
În procesorul ALPHA 21264 a fost introdusă predicția dependenței la memorie.

# LOAD / STORE Speculative

- Ca și updatearea registrelor, instrucțiunile STORE nu trebuie să modifice memoria până ajungem la Commit Point
- Un buffer STORE speculative este introdus pentru a memora datele speculative pentru STORE
  - Executăm STORE:  
marcăm intrarea validă și speculativă și salvăm data și tag-ul instrucțiunii.
  - Comitem:  
ștergem bit-ul speculativ și mutăm data în cache
  - Abort STORE:  
ștergem bitul de validitate

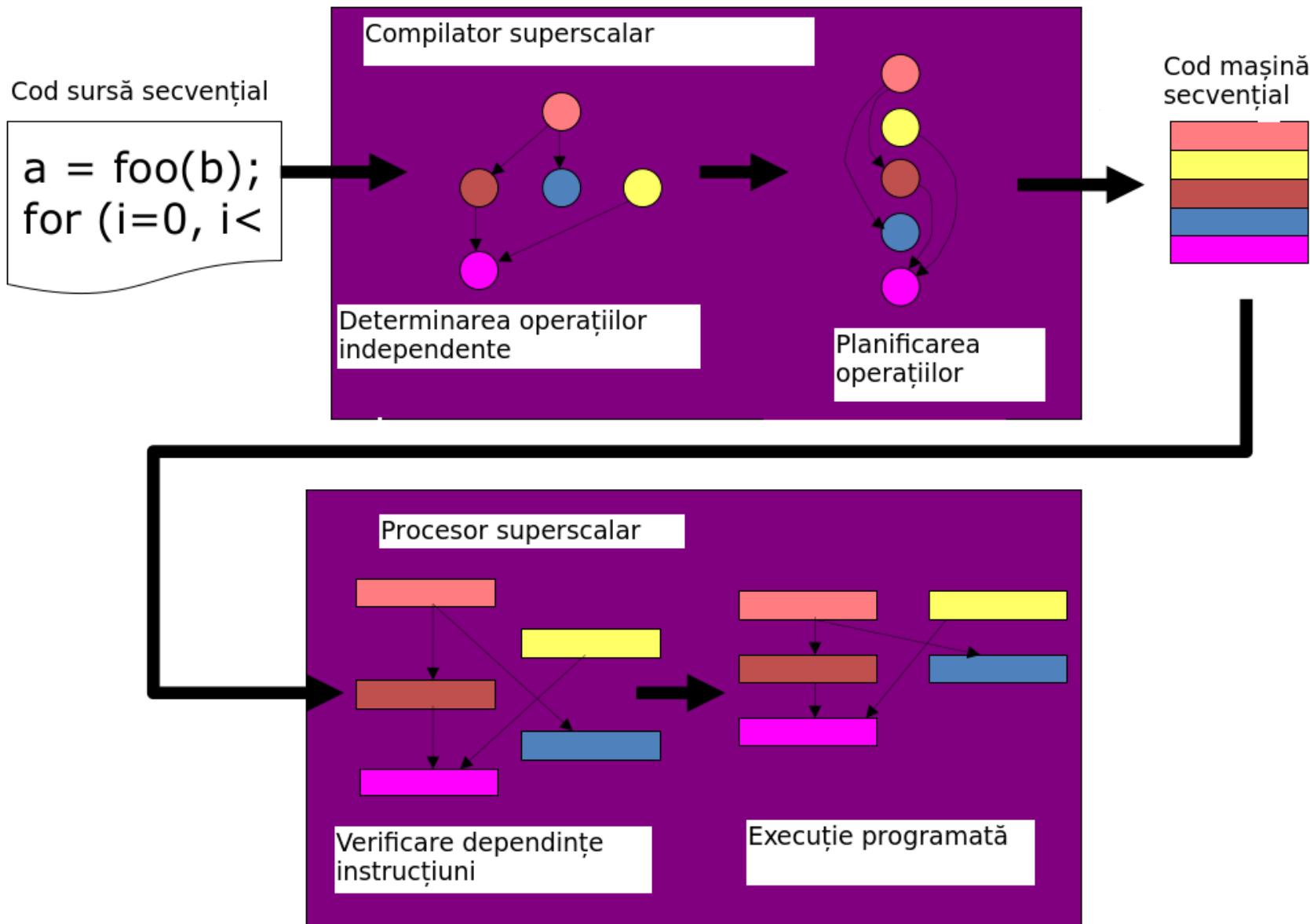


# MIPS R10000



[A. Ahi et al., MIPS R10000 Superscalar Microprocessor, Hot Chips, 1995 ]  
Image Credit: MIPS Technologies Inc. / Silicon Graphics Computer Systems

# Limitări pentru superscalari



## Tema de curs 2

- De făcut referat și de predat pe data de 23.11.2018 pentru :
- 1. H&P 5 – pag. 192 – 196 + Appendix H
- 2. Materialul cu algoritmul lui Tomasulo
- Materialele se vor găsi pe cloud

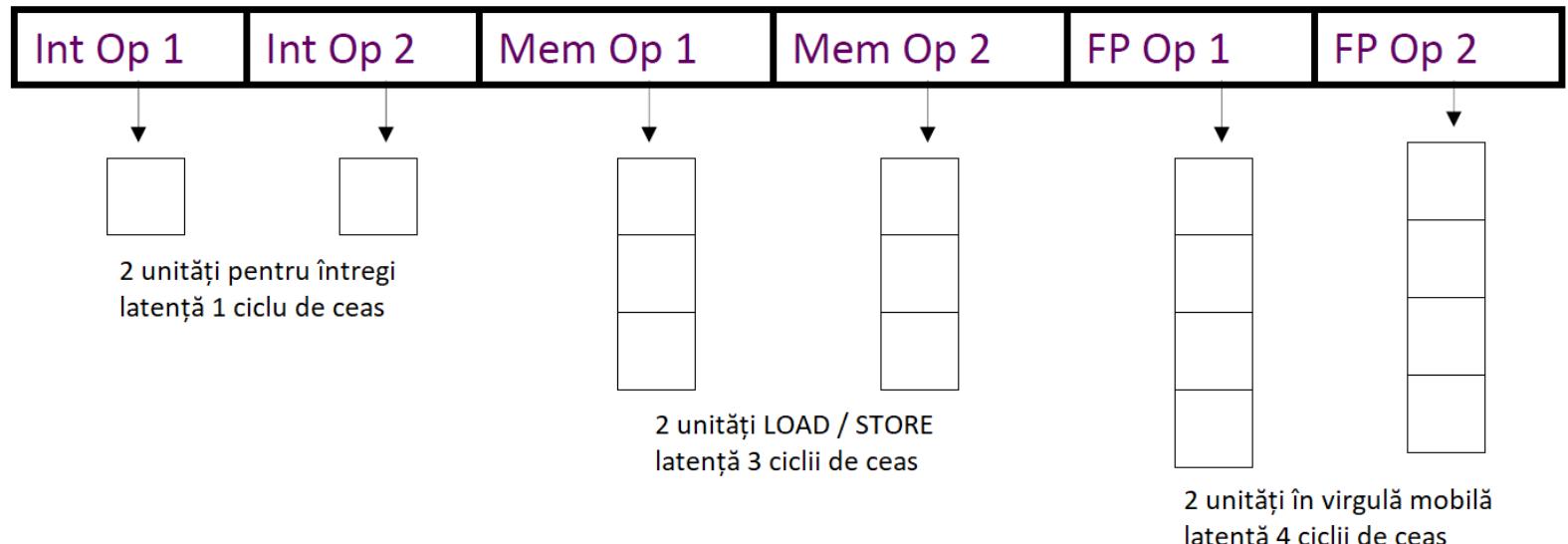
# **CALCULATOARE NUMERICE 2**

## curs 8

# Conținutul cursului

**VLIW – Very long Instruction Word**

***Predictii***



- Operații multiple împachetate într-o singură instrucțiune
  - Fiecare slot de operație este pentru o funcție fixă
  - Latența operațiilor este o constantă și este cunoscută
  - Arhitectura oferă garanții pentru:
    - Nici o dată nu este utilizată înainte de a fi disponibilă => nu există data interlocks
    - În interiorul unei instrucțiuni există paralelism => nu există RAW între instrucțiuni

## VLIW EQ – model de planificare egală

- Fiecare operație durează exact cât este definită latență,
- Utilizarea eficientă a registrelor (mai multe registre)
- Nu este necesară redenumirea registrelor sau buffering-ul
  - Bypass de la ieșirea unei unități funcționale la intrări
  - Registrul este scris oricând după finalizarea execuției unei unități funcționale
- Compilatorul depinde de faptul că registrele nu sunt disponibile mai devreme

## VLIW LEQ – model de planificare mai mică sau egală

- Fiecare operație poate dura mai puțin sau cel mult cât este latența specificată
  - Destinația poate fi scrisă oricând după citirea instrucțiunii
  - Instrucțiunile dependente necesită în continuare planificare după latența instrucțiunii
- Simplificarea întreruperilor precise
- Compatibilitatea binară menținută când latența este redusă

# Compilatorul VLIW – responsabilități

- Planifică operațiile pentru a maximiza execuția paralelă
- Garantează paralelismul în interiorul instrucțiunii
- Planifică pentru evitarea hazardurilor de date (fără interlock)
  - În mod obișnuit, separă operațiile cu NOP-uri explicite

# Execuția ciclurilor

```
for (i=0; i<N; i++)  
    B[i] = A[i] + C;
```

## Compilare

```
loop:    lw F1, 0(R1)
           addiu R1, R1, 4
           add.s F2, F0, F1
           sw F2, 0(R2)
           addiu R2, R2, 4
           bne R1, R3, loop
```

Ciclu

## Planificare

Int1 Int 2 M1 M2 FP+ FPx

- Câte operații FP / ciclu ?
  - $1 \text{ add.s} / 8 \text{ cicli} = 0,125$

# Desfacerea ciclurilor

```
for (i=0; i<N; i++)  
    B[i] = A[i] + C;
```

Desfacerea ciclului pentru a realiza 4 iterații odată

```
for (i=0; i<N; i+=4)  
{  
    B[i]      = A[i] + C;  
    B[i+1]    = A[i+1] + C;  
    B[i+2]    = A[i+2] + C;  
    B[i+3]    = A[i+3] + C;  
}
```

- Este necesar să procesăm valorile lui N care nu sunt multiplu de factorul de desfacere al ciclului la finalul ciclului când eliberăm resursele.

# Planificarea codului ciclurilor desfăcute

Desfacere pe 4 căi

```
loop: lw F1, 0(r1)
      lw F2, 4(r1)
      lw F3, 8(r1)
      lw F4, 12(r1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
```

loop:

Schedule →

	Int1	Int 2	M1	M2	FP+	FPx
			lw F1			
			lw F2			
			lw F3			
	add R1		lw F4		add.s F5	
					add.s F6	
					add.s F7	
					add.s F8	
			sw F5			
			sw F6			
			sw F7			
	add R2	bne	sw F8			

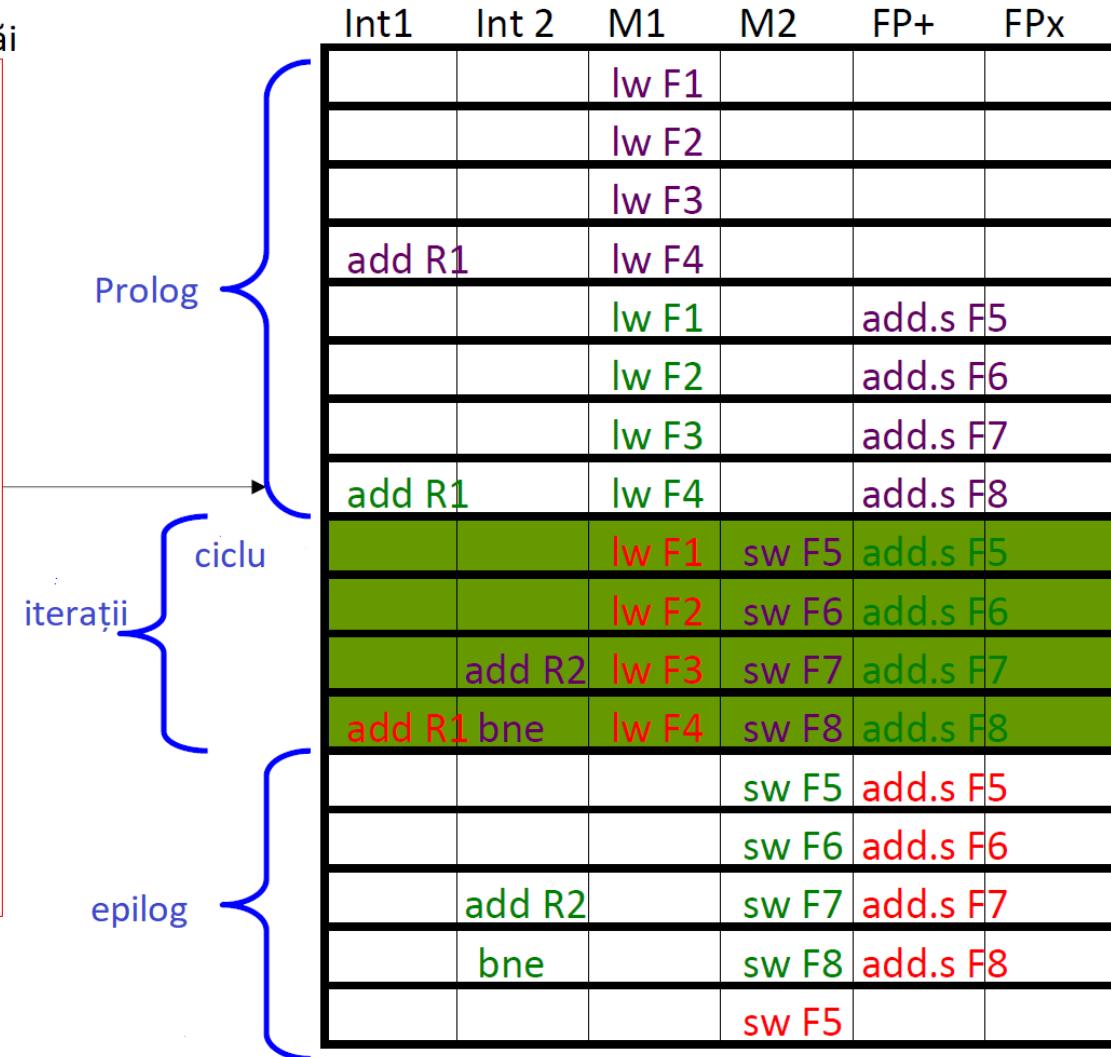
- 4 add.s / 11 ciclui = 0.36 FLOPS / ciclu

# Pipeline software

Prima oară desfacem pe 4 căi

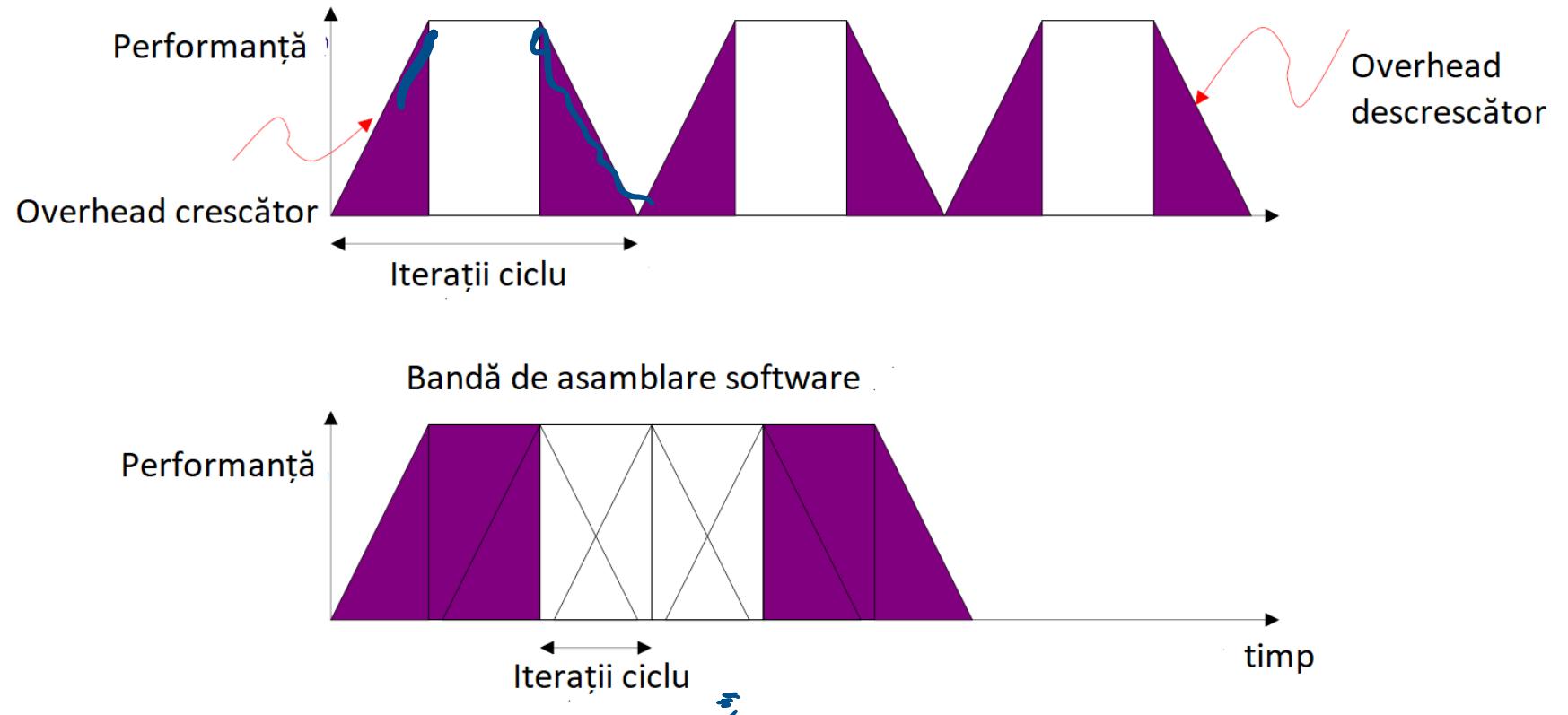
```

loop: lw F1, 0(R1)
      lw F2, 4(R1)
      lw F3, 8(R1)
      lw F4, 12(R1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
  
```



- Câte operații FLOPS/ciclă ? Se execută 4 add.s / 4 ciclă = 1 FLOPS / ciclă

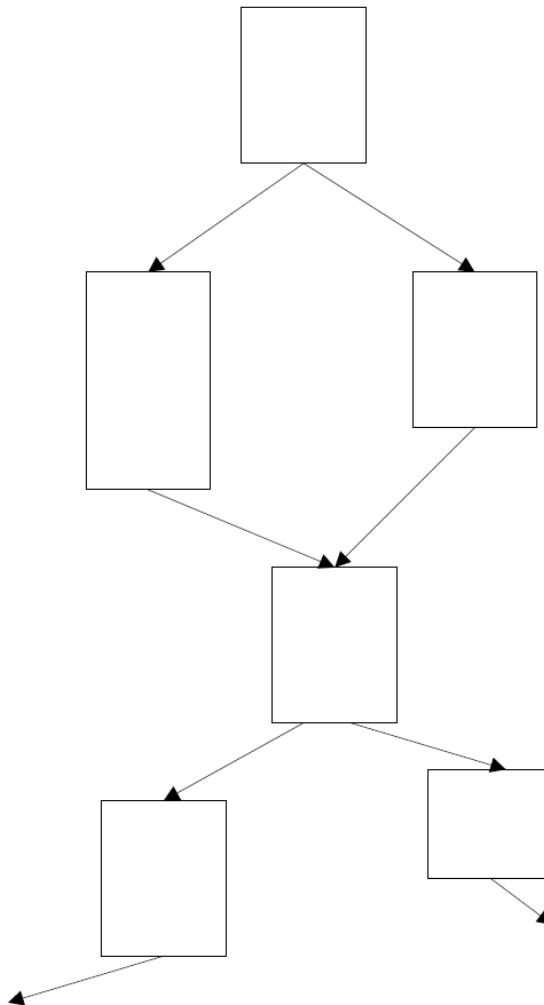
# Pipeline software vs desfacere cilu



- Banda de asamblare software are un cost datorită overhead-ului crescător / descrescător doar odată / ciclu nu odată pe iterație

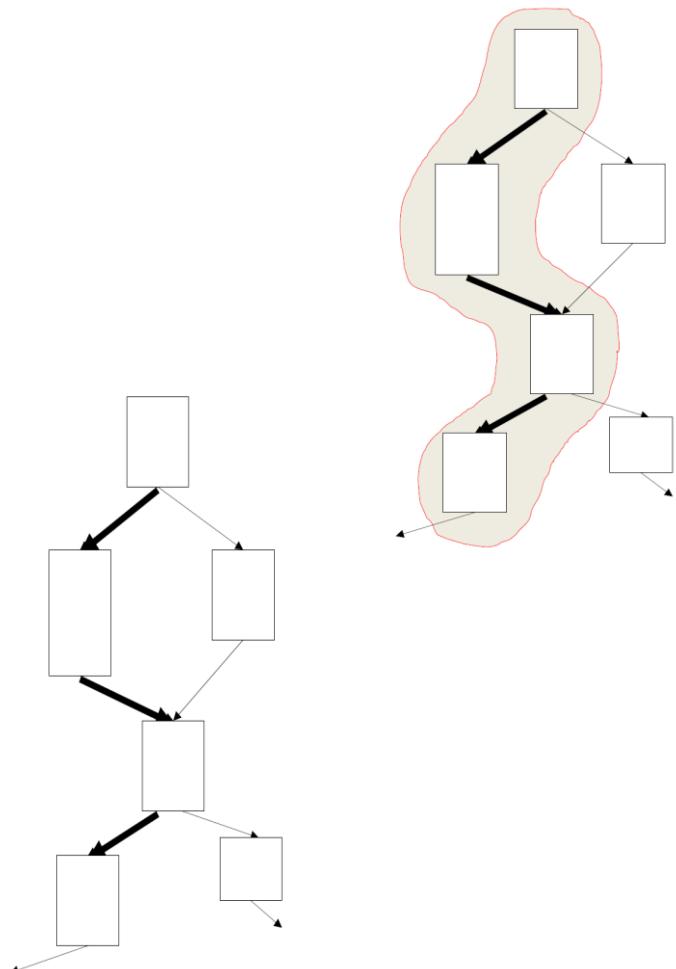
# Ce se întâmplă dacă nu avem cicluri ?

- Blocuri de bază
  - O singură intrare
  - O singură ieșire



- Salturile limitează dimensiunea blocurilor de bază în fluxul de controlul intensiv al unui cod irregular
- Dificilă găsirea ILP în blocurile de bază individuale

## Urmărirea planificării

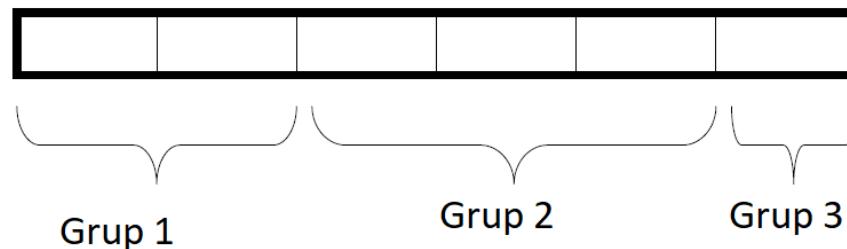


- Alegem sirul blocurilor de bază, **urma**, care reprezintă calea cea mai frecventă folosită de un salt
  - Utilizăm profile pentru feedback sau euristici de compilator pentru a identifica cele mai folosite căi de către branch-uri
  - Planificăm o întreagă „urmă” la un moment de timp
  - Adăugăm cod suplimentar pentru a face față branch-urilor care „sar” în afara „urmei”.

# Problemele VLIW-ului clasic

- Compatibilitate cod-obiect
  - Trebuie recompilat tot codul pentru fiecare mașină, chiar pentru 2 mașini din aceeași generație
- Dimensiunea cod-obiect
  - Instrucțiunile în plus consumă memorie / cache
  - Desfacerea de cicluri / banda de asamblare software replică cod
- Planificarea variabilă a latenței pentru operațiile cu memoria
  - Conflictele de memoria cache impun o variabilitate statică nepredictibilă
- Cunoașterea probabilităților de branch
  - Profilele necesită un pas extra semnificativ în procesul de construire a lor
- Planificare pentru branch-urile statice nepredictibile
  - Planificare optimă variază cu calea branch-ului
- Execuția întreruperilor precise poate fi provocatoare

# Encodarea instrucțiunilor VLIW



- Scheme de reducere a efectului câmpurilor neutilizate
  - Format comprimat în memorie, expandare pe I-cache
    - Utilizat în urmele cu flux multiplu
    - Introduce instrucțiuni de adresare
  - Marcarea grupurilor paralele
    - TMS320C6x DSP, Intel IA-64
  - Oferirea de instrucțiuni VLIW cu o singură operație
    - Cydra-5

# Predictii

**Problema:** Predictiile greșite pentru branch-uri limitează ILP

**Soluție:** Eliminarea branch-urilor greu de prezis cu execuții pre-dedicate

Pre-dedicarea ajută prin folosirea regiunilor de salt (de dimensiuni mici) și / sau transformarea fluxului de control într-un flux de date

Cea mai folosită formă de prezicere: mutări condiționale

- movz rd, rs, rt if ( R[rt] == 0 ) then R[rd] <- R[rs]
- movn rd, rs, rt if ( R[rt] != 0 ) then R[rd] <- R[rs]

```
if (a<b)      slt R1, R2, R3    slt R1, R2, R3
  x=a          beq R1, R0, L1    movz R4, R2, R1
else           move R4, R2       movn R4, R3, R1
  x=b          j  L2
L1:move R4, R3
L2:
```

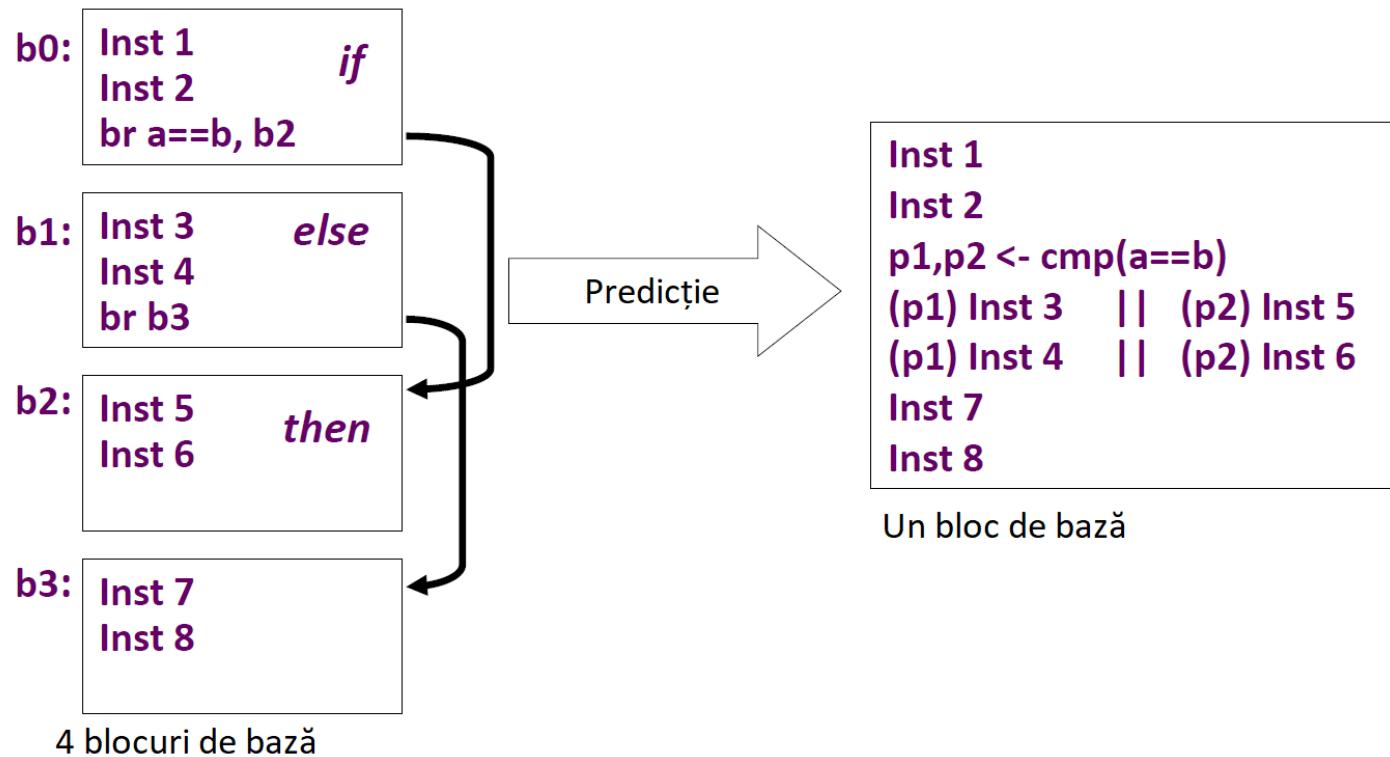
Ce se întâmplă dacă ciclul  
**if-then-else**

are mai multe instrucțiuni ?

Dacă este nebalansat ?

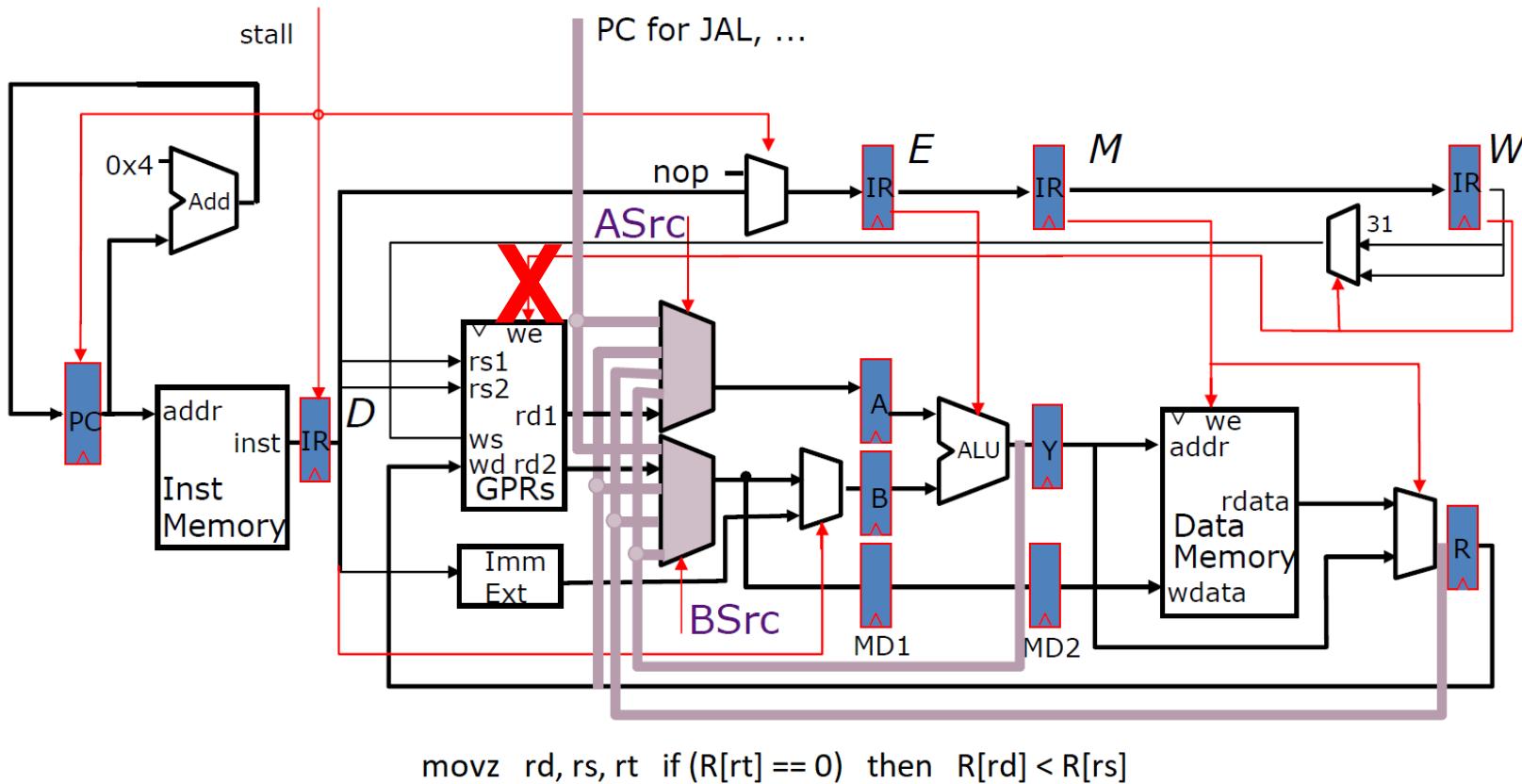
# Predictia completa

- Aproape toate instructiunile pot fi executate conditonal sub predictie
  - Instructiunea devine NOP daca predictia registrului este false



# Implementarea predictorilor

# Predictia și calea de date



- Suprimarea stagiului de WB
- Nu merge bypass (avem nevoie de o valoare inițială – extra port pe RF, port de citire)

## Probleme legate de predictia completa

- Adăugare de alte registre
- Necesitatea de a face bypass predictiilor
- Necesita o valoare originală pentru a utiliza valorile de date bypass-ate.

# Folosirea execuției speculative și reacția la evenimente dinamice în VLIW

- Specularea
  - Mutarea instrucțiunilor în afara branch-urilor
  - Mutarea operațiilor de memorie peste alte operații de memorie
- Evenimente dinamice
  - Miss la cache
  - Exceptii
  - Predictii false pentru branch

## Mutarea codului

Înainte de mutarea codului

MUL R1, R2, R3  
ADDIU R11,R10,1  
MUL R5, R1, R4  
MUL R7, R5, R6  
SW R7, 0(R16)  
ADDIU R12,R11,1  
LW R14, 0(R9)  
ADD R13,R12,R14  
ADD R14,R12,R13  
BNEQ R16, target

După mutarea codului

LW R14, 0(R9)  
ADDIU R11,R10,1  
MUL R1, R2, R3  
ADDIU R12,R11,1  
MUL R5, R1, R4  
ADD R13,R12,R14  
MUL R7, R5, R6  
ADD R14,R12,R13  
SW R7, 0(R16)  
BNEQ R16, target

## Planificare și împachetare

Înainte de împachetare

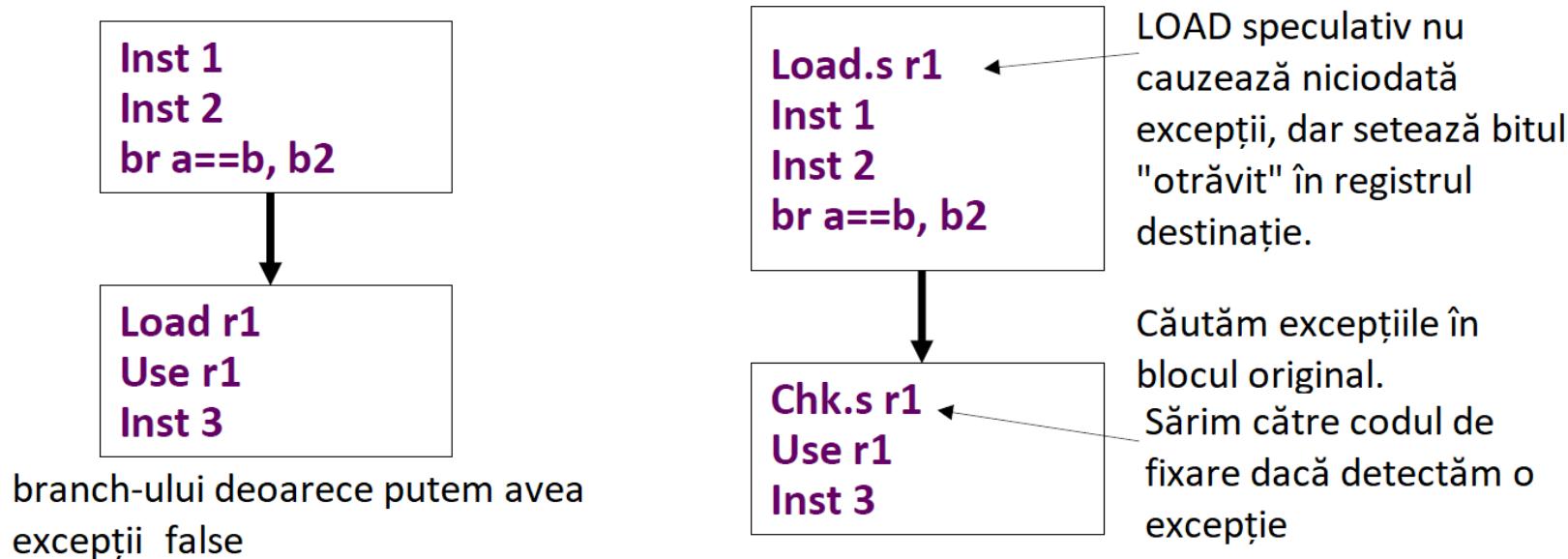
LW R14, 0(R9)  
ADDIU R11,R10,1  
MUL R1, R2, R3  
ADDIU R12,R11,1  
MUL R5, R1, R4  
ADD R13,R12,R14  
MUL R7, R5, R6  
ADD R14,R12,R13  
SW R7, 0(R16)  
BNEQ R16, target

După împachetare

{ LW R14, 0(R9)  
ADDIU R11,R10,1  
MUL R1, R2, R3 }  
{ ADDIU R12,R11,1  
MUL R5, R1, R4 }  
{ ADD R13,R12,R14  
MUL R7, R5, R6 }  
{ ADD R14,R12,R13  
SW R7, 0(R16)  
BNEQ R16, target }

# Execuția speculativă în VLIW

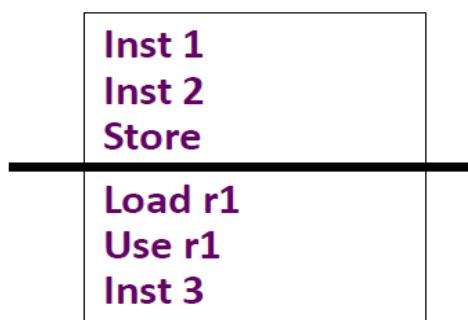
- Problema: Branch-urile restricționează compilatorul să mute codul
- Soluție: Operații speculative care nu pot cauza excepții



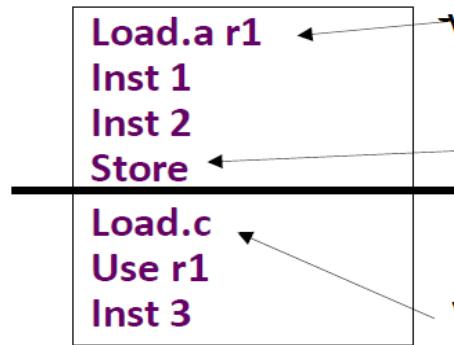
- Util pentru planificarea timpurie a LOAD-urilor cu latență mare

# Specularea datelor în VLIW

- Problema: Posibilele hazarduri de memorie limitează mutarea codului
- Soluție: Hardware pentru a verifica pointer-ul la hazard



Nu putem muta LW  
deasupra STORE  
deoarece STORE poate fi  
la aceeași adresă



LOAD speculativ adaugă  
adresa la tabelul de  
verificare a adreselor

STORE invalidează  
orice potrivire cu  
LOAD în TVA

Verificăm dacă LOAD  
este invalid salt la codul  
de fix-up

- Necesită hardware asociativ în ALAT

# ALAT – Advanced Load Address Table

Register Number	Address	Size

Chk.a/Id.c  
CAM pe registru  
Număr

STORE  
CAM pe adresă

- Load.a adaugă o intrare în ALAT
- STORE șterge o intrare dacă adresa / dimensiunea se potrivesc
- Instrucțiunea Check (chk.a sau Id.c) verifică că adresa este încă în ALAT și STORE-ul intermediar nu pune adresa în altă parte
- Dacă nu este găsită, se rulează codul de recuperare (LOAD se re-execută)

## Branch-uri cu mai multe căi VLIW

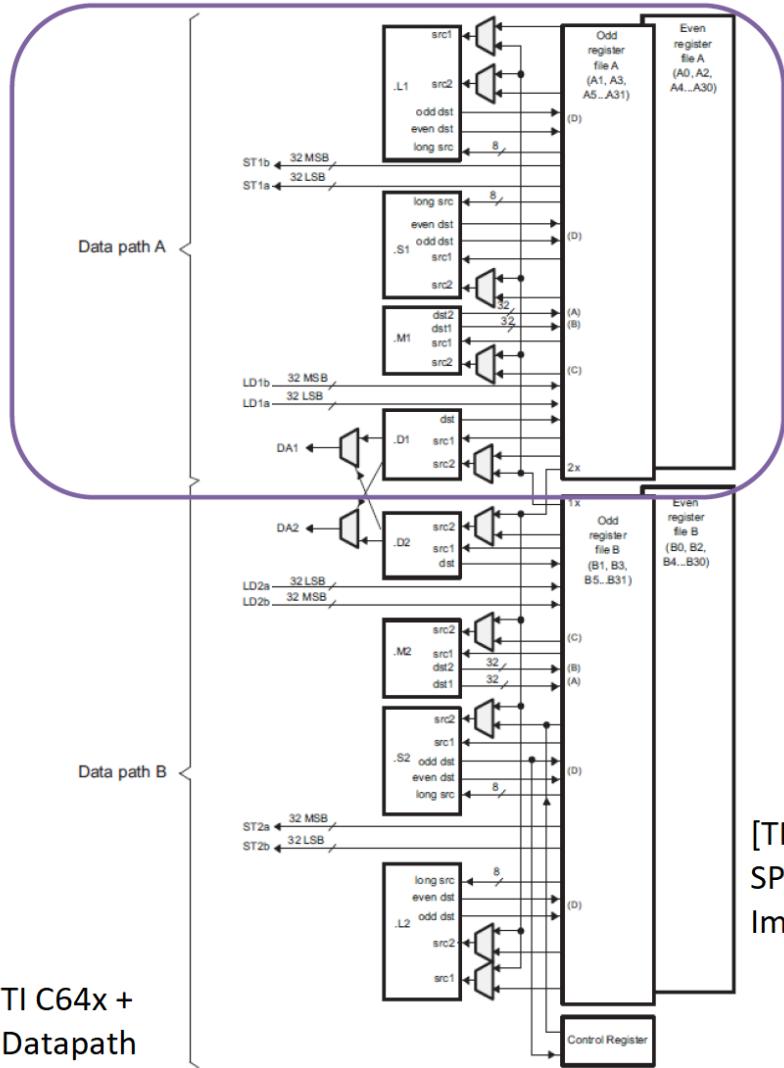
- Problema: Instrucțiunile lungi oferă puține oportunități pentru branchuri
- Soluție: permitem ca o instrucțiune să facă salt către mai multe direcții

```
{ .mii
    cmp.eq P1, P2 = R1, R2
    cmp.ne P3,P4 = R4, R5
    cmp.lt P5,P6, R8, R9
}
{ .bbb
(P1) br.cond label1
(P2) br.cond label2
(P5) br.cond label3
}
// eliminare cod
```

## Planificarea evenimentelor dinamice

- MISS la cache
  - Informare LOADs – NU există implementare comercială. Ideea este că nu se execută instrucțiunile subsecvente
  - Procesorul Elbrus are branch-uri pentru cazul de MISS – dacă MISS la cache dute și execută o bucată de cod planificată diferit
- Predicție greșită branch
  - Sloturi de întârziere pt. branch. În setul de instrucțiuni și utilizăm predicția în aceste slot-uri.
- Excepțiile – imposibil de prezis în VLIW

# VLIW în cluster



- Mașina este împărțită în clustere – register files și unitățile funcționale sunt locale
- Lățime de bandă mică între clustere
- Latență mare între clustere
- Utilizat în
  - Procesorul HP/ST Lx (imprimante)
  - DSP-urile TI din seria C6x

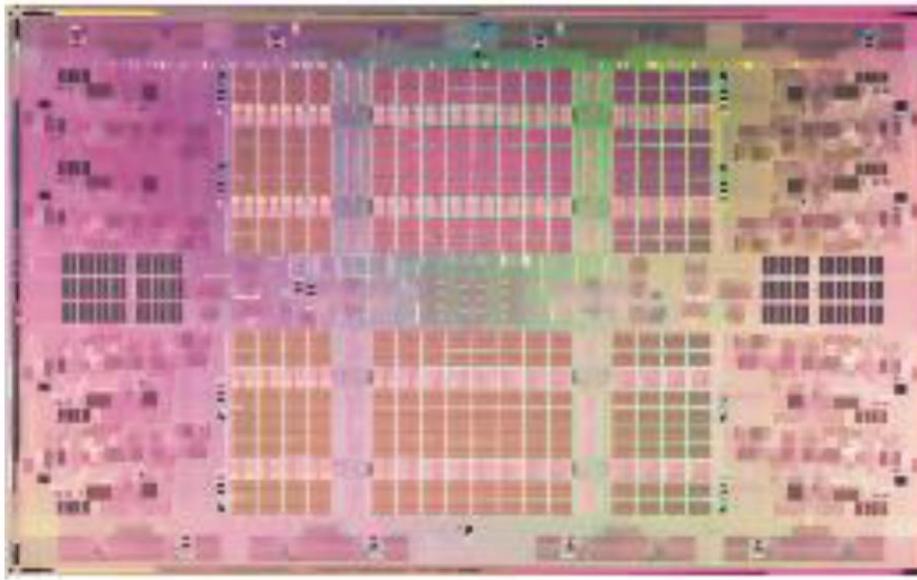
[TMS320C64x / C64x + DSP CPU și ISA, TI, 2010,  
SPRU732J]

Imagine conform - Texas Instruments Incorporated

## Intel ITANIUM – EPIC IA64

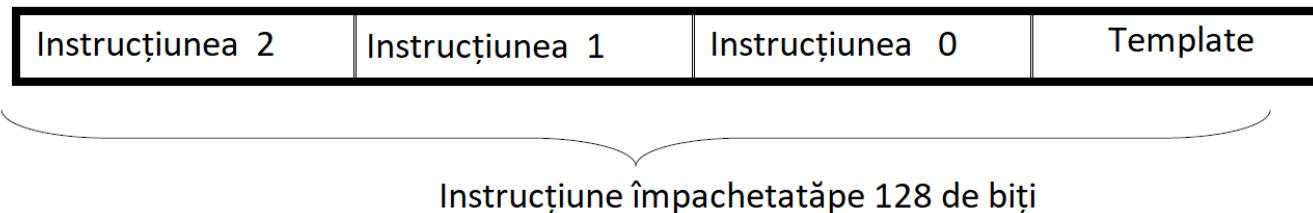
- EPIC este stilul arhitecturii (CISC, RISC)
  - EPIC = Explicitly Parallel Instruction Computing
- IA-64 este ISA aleasă de Intel (x86, MIPS)
  - IA-64 = Intel Architecture 64 biți
  - Un cod obiect compatibil cu VLIW
- MERCED a fost prima implementare de Itanium (8086)
  - Prima livrare așteptată pentru 1997 (actual 2001)
  - A doua implementare McKinley, livrată în 2002
  - Versiunea recentă – Poulson – 8 core-uri anunțată în 2011
  - 2017 ultima versiune de ITANIUM - Kittson

# ITANIUM Poulson

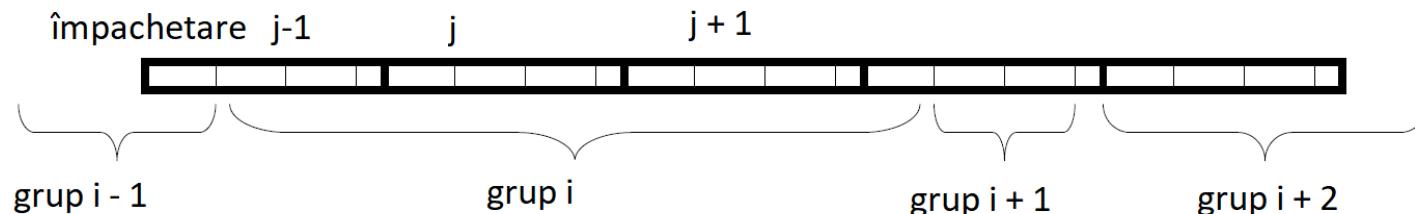


- 8 core-uri
- 1-cycle 16KB L1 I&D caches
- 9-cycle 512KB L2 I-cache
- 8-cycle 256KB L2 D-cache
- 32 MB shared L3 cache
- 544mm<sup>2</sup> in 32nm CMOS
- **Peste 3 miliarde de transzistori**
- Coreurile sunt 2-căi multithread
- Se citesc 6 instrucțiuni/ciclu
  - 2 instrucțiuni 128 biti împachetate
- Până la 12 instanțe / ciclu

# Formatul de instrucțiuni IA 64



- Biții de template descriu gruparea acestor instrucțiuni cu altele din împachetările adiacente
- Fiecare grup conține instrucțiuni care pot fi executate în paralel

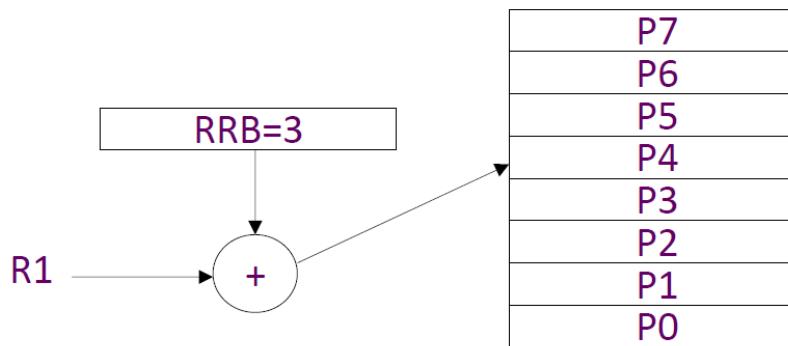


## Registrele IA 64

- 128 de registre cu scop general. Registrele de întregi pe 64 biți iar registrele pentru virgulă mobilă au 64 / 80 biți,
- 64 registre de predicție pe 1 bit
- GPRs rotiți pentru a reduce mărimea codului pentru ciclurile în BA software
- Rotirea este o formă simplă de redenumire a registrelor permitând unei instrucțiuni să adreseze diferite registre fizice pentru fiecare iterație

## Rotirea fișierului de registre IA 64

- Problema: Planificarea de cicluri necesită multe nume de registre și duplicarea codului în prolog și epilog
- Soluție: Alocăm un nou set de registre pentru fiecare iterație a ciclului.



- Registrul RRB (Rotating Register Base) pointează către baza setului de registre curent. Valoarea adăugată la specificatorul registrului logic este folosită pentru a obține numărul registrului fizic. În mod uzual, registrele se împart în registre de rotație și registre care nu pot fi rotite.

# Rotirea fișierului de registre IA 64

3 ciclii de latență LOAD

encodeați ca diferență de 3 în  
numărul specificatorului de  
registru ( $f_4 - f_1 = 3$ )

4 ciclii de latență pentru add.s encodeați  
 $f_9 - f_5 = 4$

lw f1, ()	add.s f5, f4, ...	sw f9, ()	bloop
-----------	-------------------	-----------	-------

lw P9, ()	add.s P13, P12,	sw P17, ()	bloop
lw P8, ()	add.s P12, P11,	sw P16, ()	bloop
lw P7, ()	add.s P11, P10,	sw P15, ()	bloop
lw P6, ()	add.s P10, P9,	sw P14, ()	bloop
lw P5, ()	add.s P9, P8,	sw P13, ()	bloop
lw P4, ()	add.s P8, P7,	sw P12, ()	bloop
lw P3, ()	add.s P7, P6,	sw P11, ()	bloop
lw P2, ()	add.s P6, P5,	sw P10, ()	bloop

RRB=8

RRB=7

RRB=6

RRB=5

RRB=4

RRB=3

RRB=2

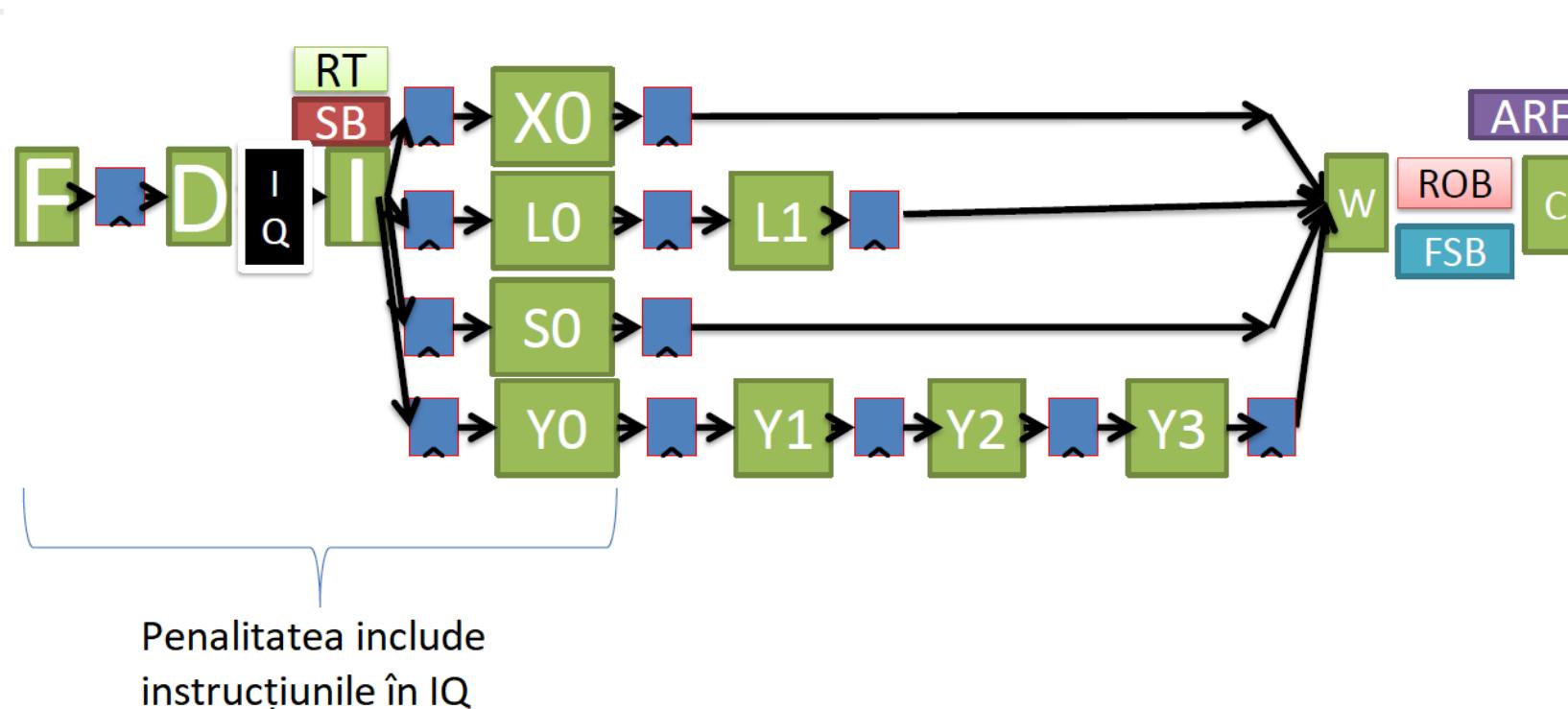
RRB=1

## De ce nu a fost un succes IA 64 – opinia lui David Wentzlaff

- ISA este dependentă de proiectarea micro-arhitecturii
  - ALAT
  - Predicția (crește inter-dependența instrucțiunilor)
  - Rotația
- Prima implementare a avut rata ceasului foarte mică
- Encodare complexă
- Complexitate mare a compilatorului
  - Este necesară realizarea de profile pentru o performanță bună
- Nu sunt soluționate problemele planificării dinamice
- ILP limitat (Static Instruction Level Parallelism)
- AMD 64
- Aria nu a fost cea mai critică constrângere (compatibilitatea codului și puterea sunt critice).
- Construirea de procesoare superscalare mai complexe

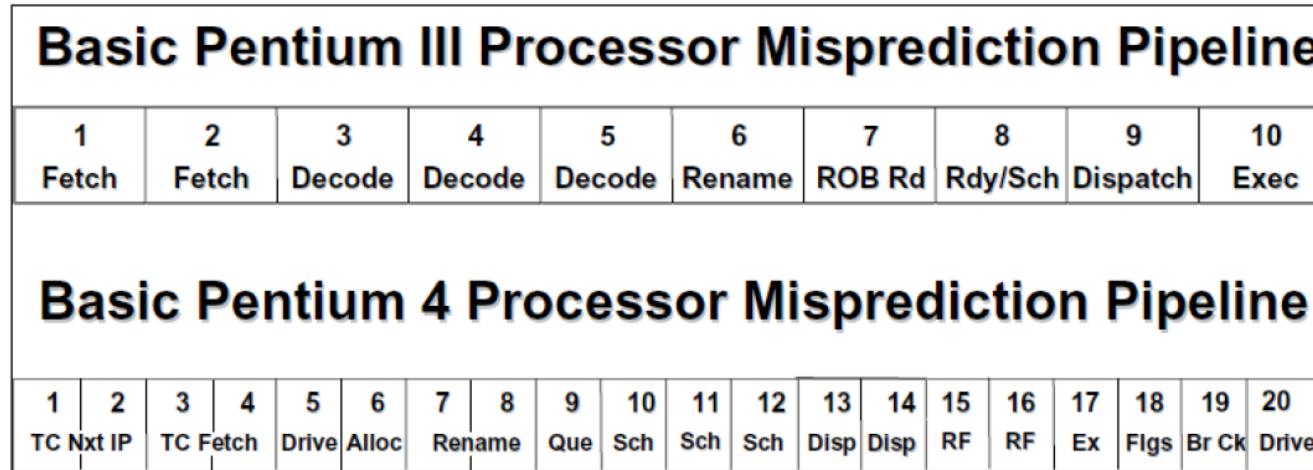
# Predictia branch-urilor

- Frontend cât mai mare în BA amplifică costul branch-urilor



# Predictia branch-urilor

- Frontend mai mare în BA amplifică costul branch-urilor

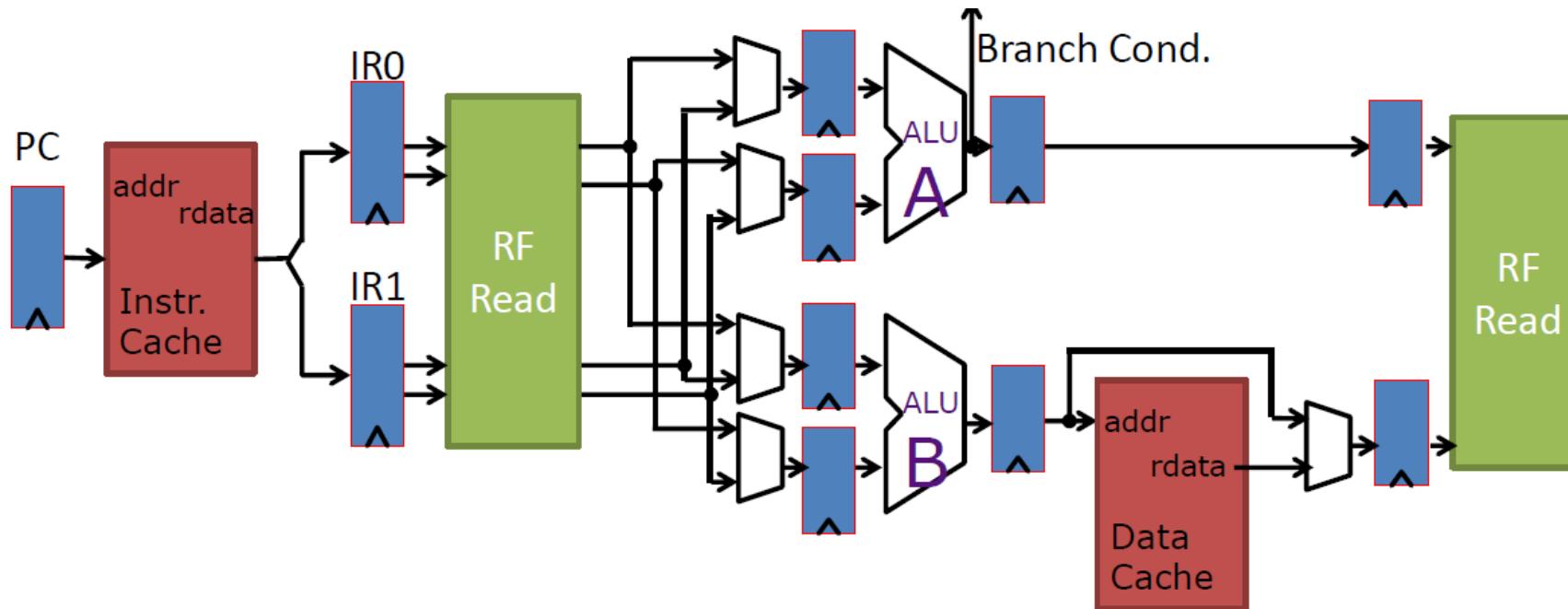


Pentium 3: 10 cicli de penalitate pentru branch

Pentium 4: 20 cicli de penalitate pentru branch

Preluare: The Microarchitecture of the Pentium 4 Processor by Glenn Hinton et al.  
Appeared in Intel Technology Journal Q1, 2001. Image courtesy of Intel

# Stagiu de ISSUE dual și costul branch-ului



# Superscalar – costul branch-ului multiplu

	BEQZ	F	D	I	A0	A1	W	
OpA		F	D	I	B0	-	-	
OpB			F	D	I	-	-	
OpC				F	D	I	-	-
OpD					F	D	-	-
OpE						F	D	-
OpF							F	-
OpG								F
OpH								F D I A0 A1 W
OpI								F D I B0 B1 W

Procesor cu  
stagiul ISSUE  
dual are o  
penalitate  
dublă în cazul  
unei predicții  
greșite

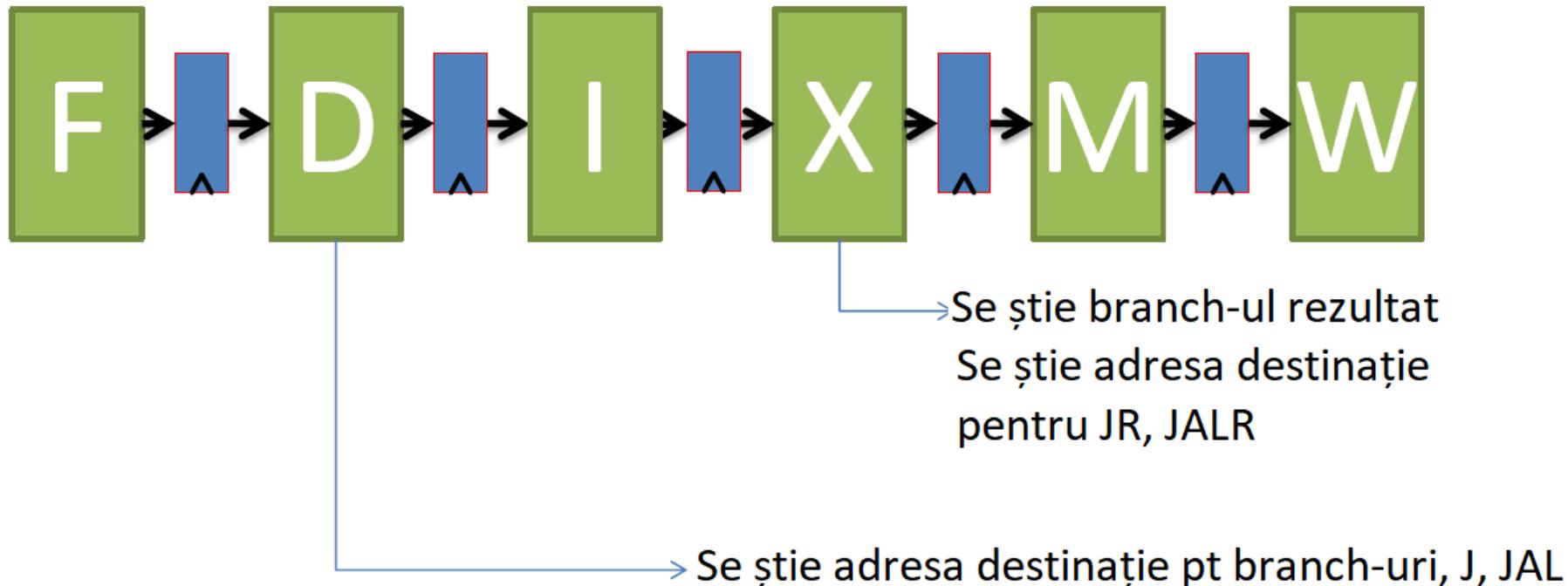
- Cât este penalitatea dacă BA-ul nu urmărește corect instrucțiunile din program ?

~ lungimea BA \* penalitatea la branch

## Predictia Branch-urilor

- Predictia branch-urilor este esentiala in procesoarele moderne pentru a diminua latentele branch-urilor
- Avem 2 tipuri de predictii
  - Predictia branch-ului rezultat
  - Predictia adresei de branch / jump

# Unde este cunoscută informația despre branch ?



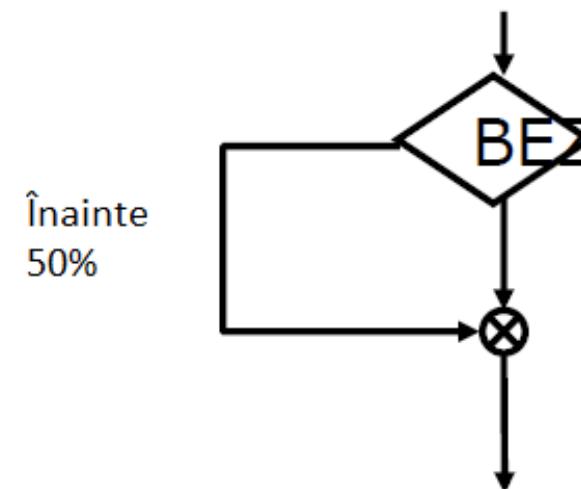
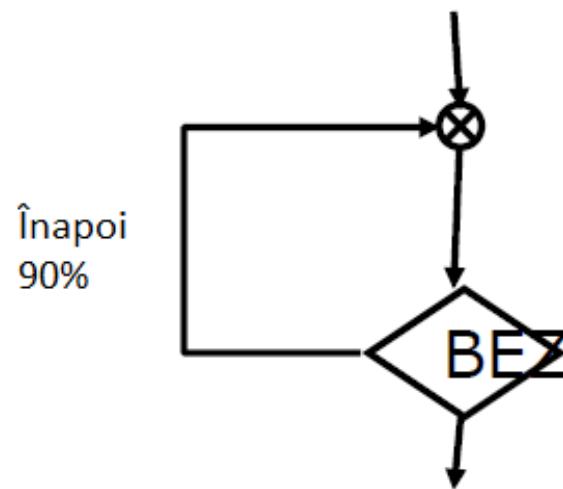
## Predictia branch-ului rezultat - STATICĂ

- Sloturi de întârziere pentru branch-uri – controlul hazardurilor de către software
- Modificăm semantica ISA astfel că instrucțiunile care urmează după un jump sau branch sunt executate întotdeauna
  - Oferim compilatorului flexibilitatea de a pune o instrucțiune utilă unde în mod normal am introduce un STALL în BA

I <sub>1</sub>	096	ADD	
I <sub>2</sub>	100	BEQZ r1 +200	Instrucțiunile din slot-ul de
I <sub>3</sub>	104	ADD	întârziere executate
I <sub>4</sub>	108	ADD	nu se deosebesc branch-urile
I <sub>5</sub>	304	ADD	rezultat

## Predictia branch-ului rezultat - STATICĂ

- Presupunem că probabilitatea de a lua un branch este de 60-70%
- Nu este o probabilitate egal distribuită conform figurii de mai jos. De ce ?



## Predictia branch-ului rezultat - STATICĂ

- Extindem ISA cu extra biți pentru a permite compilatorului să trimită un mesaj arhitecturii atunci când un branch este luat sau nu

	BR.T	F	D	X	M	W
OpA		F	-	-	-	-
Targ			F	D	X	M
BR.NT				F	D	X
OpB					F	D
OpC						F

Ce se întâmplă dacă predicția lui if este falsă ?

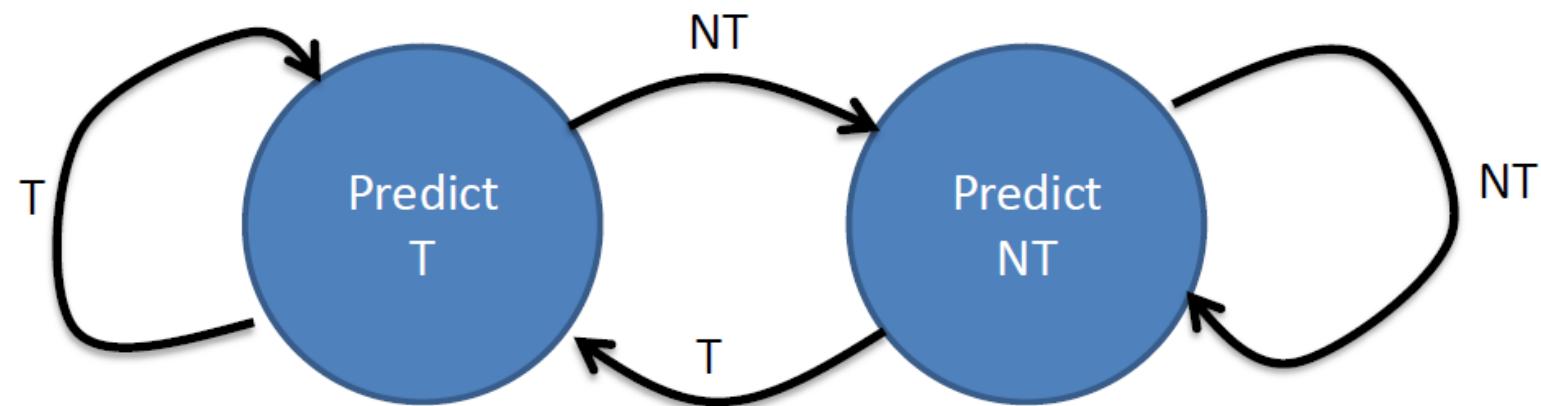
	BR.T	F	D	X	M	W
OpA		F	-	-	-	-
Targ			F	-	-	-
OpA				F	D	X
					M	W

## Predictia branch-ului hardware statică

- Întotdeauna predictia nu este luată
  - Ușor de implementat
  - Se știe din faza de Fetch că valoarea PC-ului este greșită
  - Acuratețe scăzută în special pe branch-urile înapoi
- Întotdeauna predictia este luată
  - Acuratețe scăzută pentru if-then-else
  - Dificil de implementat deoarece nu știm destinația până în faza de DECODE
- Branch-ul înapoi luat, branch-ul înainte nu este luat
  - Acuratețe bună
  - Dificil de implementat deoarece nu știm destinația până în faza de DECODE

## Predictia branch-ului rezultat - DINAMICĂ

- Exploatăm structura programului: Calea de a rezolva un branch poate fi un bun indicator al metodei în care se va rezolva un branch următoarea dată (corelare temporară)
- Schema de predicție cu 1 bit



# Predictia branch-ului rezultat - DINAMICĂ

Iteration	Prediction	Actual	Mispredict?
1	NT	T	Y
2	T	T	
3	T	T	
4	T	NT	Y
...			
1	NT	T	Y
2	T	T	
3	T	T	
4	T	NT	Y

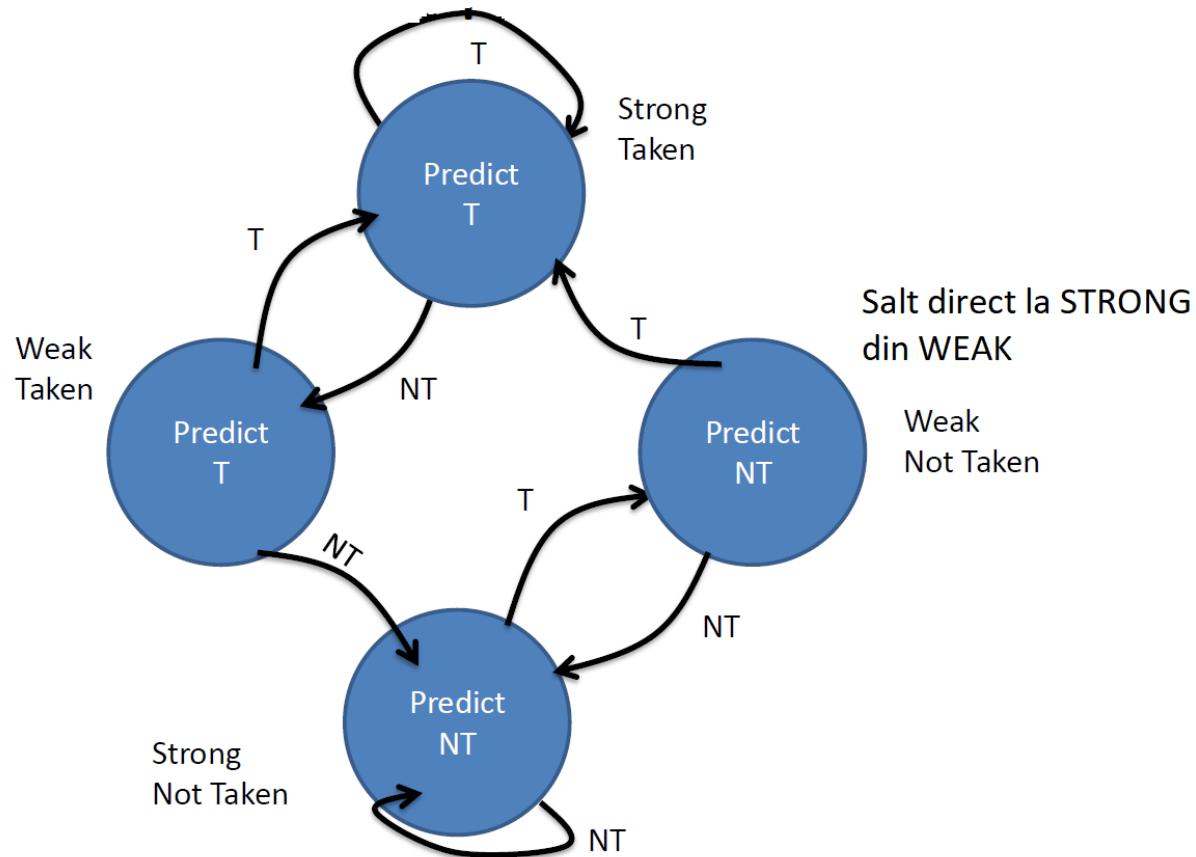
Pentru branch-urile înapoi

- pp 4 iterații
- pp că se execută de mai multe ori

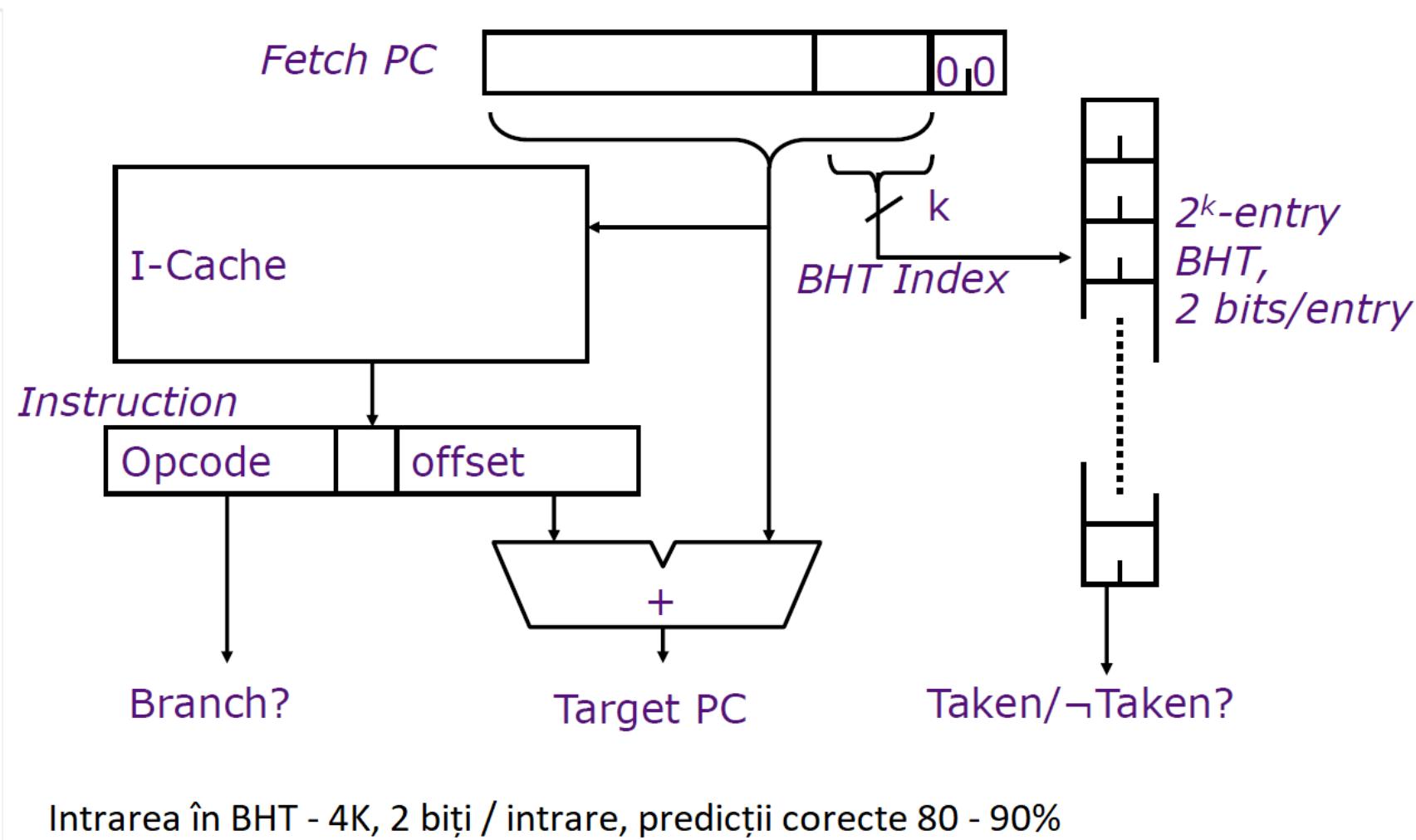
Întotdeauna 2 MISS-uri

# Predictia branch-ului rezultat - DINAMICĂ

- Altă schemă de predicție cu 2 biți
- Câte MISS-uri vom avea ?



## Tabelă de istoric pentru branch (BHT)



# Exploatarea corelării spațiale

```
if (x[i] < 7) then  
    y += 1;  
if (x[i] < 5) then  
    c -= 4;
```

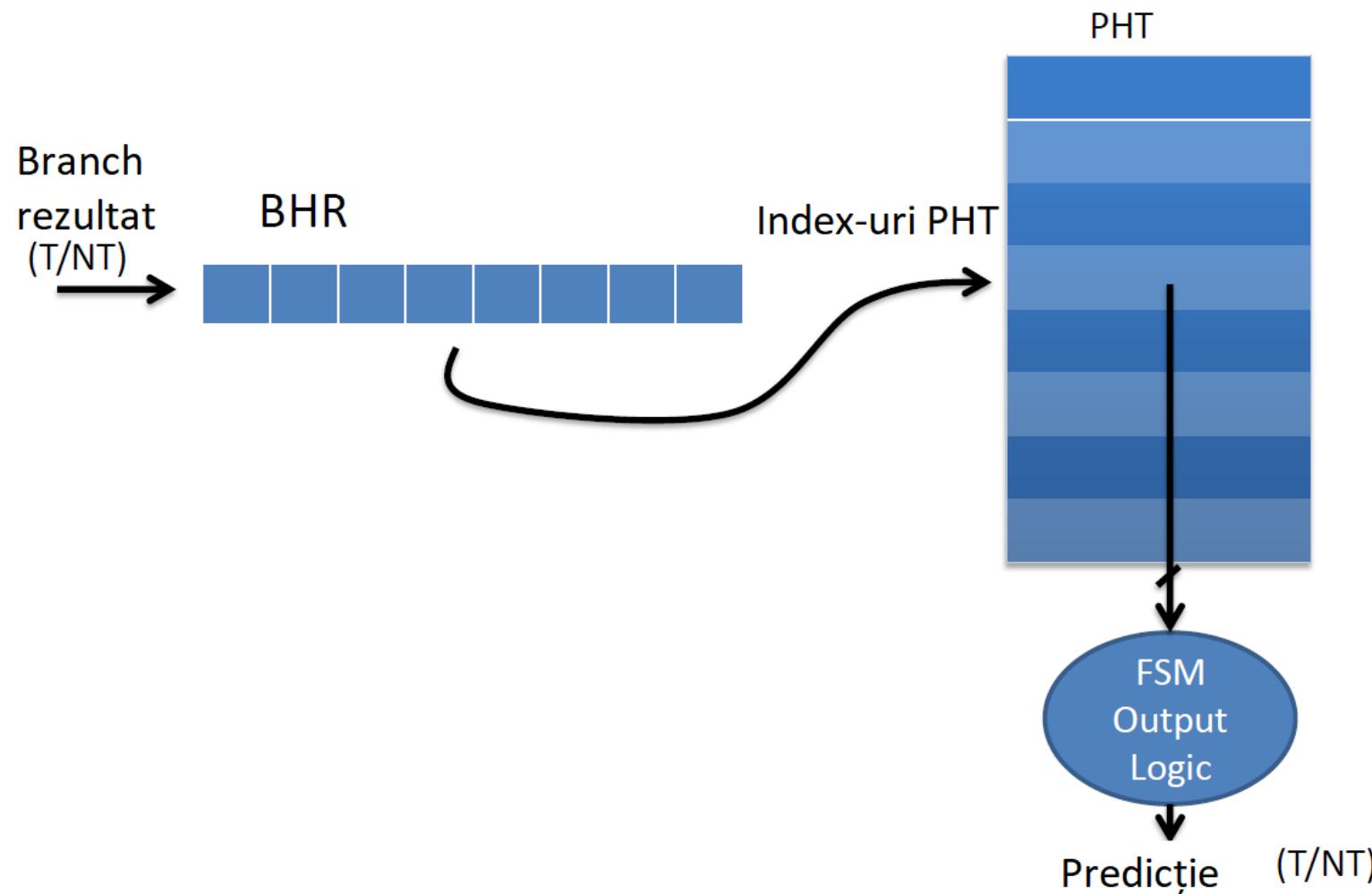
- Dacă prima condiție este falsă, atunci și a doua condiție este falsă
- BHR memorează direcția ultimelor N branch-uri executate de procesor (un registru de deplasare)

Branch  
rezultat  
(T/NT)  
→

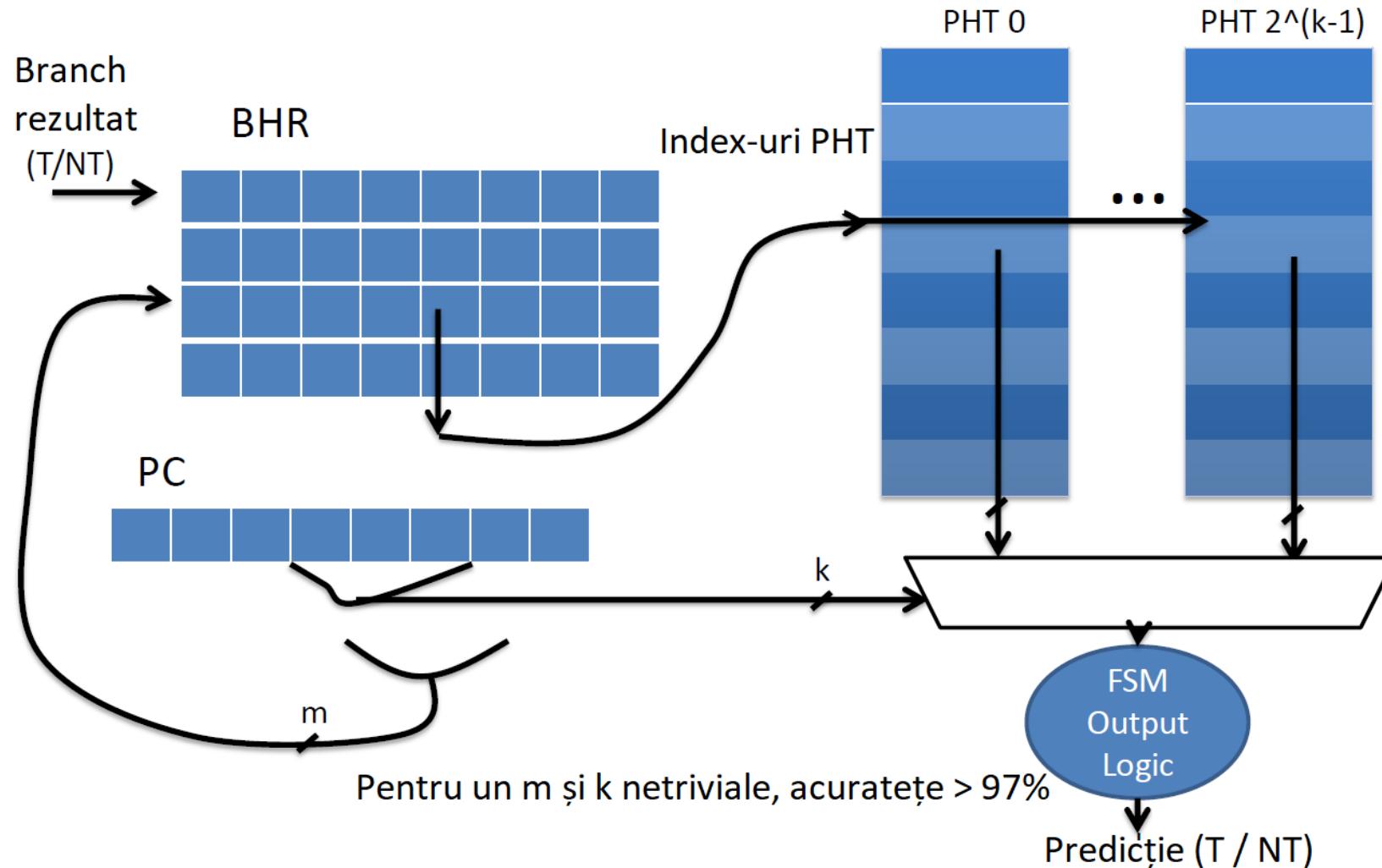
BHR



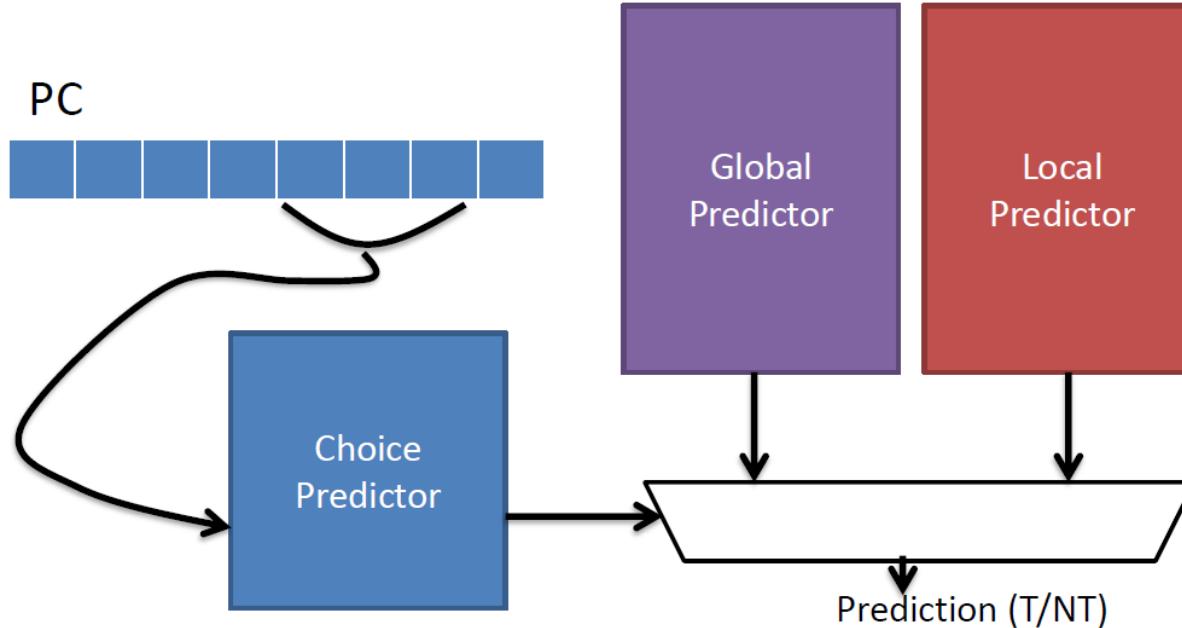
# Tabela de istoric pentru șabloane (PHT)



# Predictie pentru branch pe 2 nivele – caz general

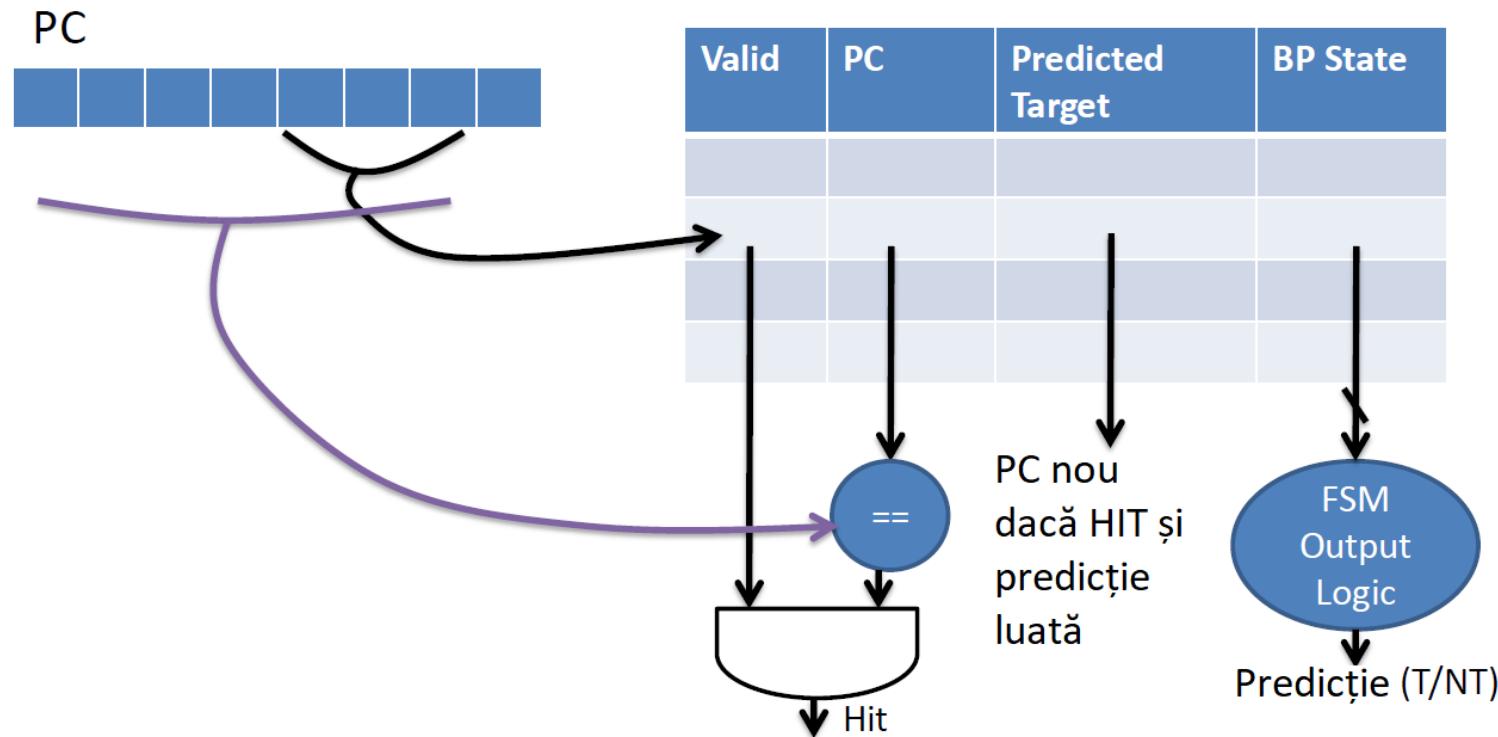


# Predictori Tournament



- Predictorul „Alege” învață când este cel mai bine de utilizat istoricul branch-ului local sau global în predicția următorului branch
- Istoricul global este speculativ reîmprospătat dar restaurare se va face doar în caz de predicție greșită
- Rată de succes între 90 - 100%

# Branch Target Buffer (BTB)



Punem BTB în stagiul FETCH în paralel cu logica de speculare a lui PC+4

## BTB doar pentru controlul instrucțiunilor

- BTB conține informații utile **doar** pentru instrucțiunile de branch și jump
- BTB nu se reîmprospătează pentru alte instrucțiuni
- Pentru toate celelalte instrucțiuni PC este PC+4
- Cum obținem acest efect fără a decodifica instrucțiunea ?

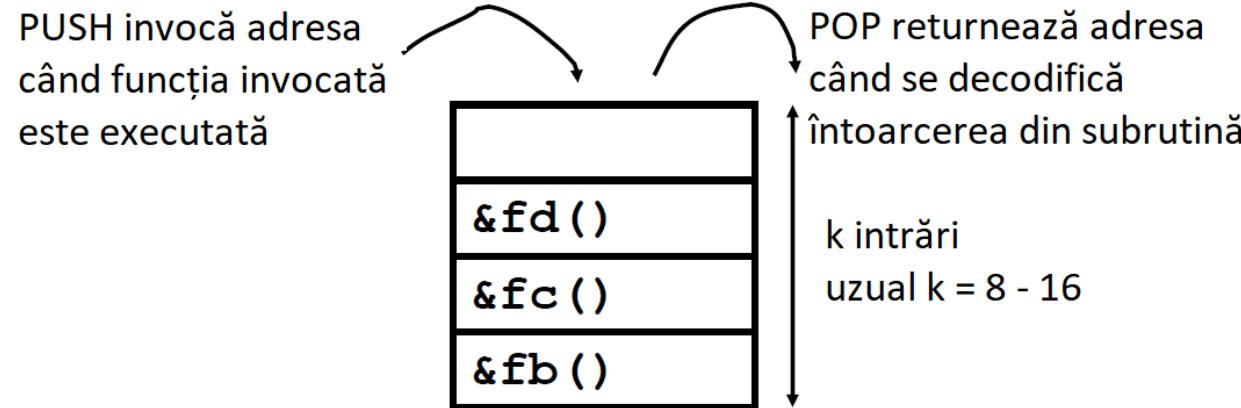
## Utilizarea lui JR (Jump Register)

- Instrucțiunile SWITCH (salt la o adresă în caz de potrivire)
  - BTB lucrează bine dacă același caz este utilizat în mod repetat
- Invocare funcție dinamică (salt la adresa funcției de run-time)
  - BTB lucrează bine dacă aceeași funcție este invocată (în programarea C++ când obiectele au același tip în invocarea funcției virtual)
- Revenirea din subrutine (salt al adresa de întoarcere - CASE)
  - BTB lucrează bine dacă întoarcerea se face în același loc

# Stiva subrutinei de întoarcere

- Structură mică pentru accelerarea instrucțiunii JR pentru subrutina de întoarcere – în mod uzual mult mai precisă decât BTB

```
fa() { fb(); }  
fb() { fc(); }  
fc() { fd(); }
```



# **CALCULATOARE NUMERICE 2**

## curs 10

# Conținutul cursului – memorii cache avansate

**Recapitulare**

**Scrierea în cache pipeline**

**Buffer de scriere**

**Memorii cache multi-nivel**

**Citirea în avans – hardware / software**

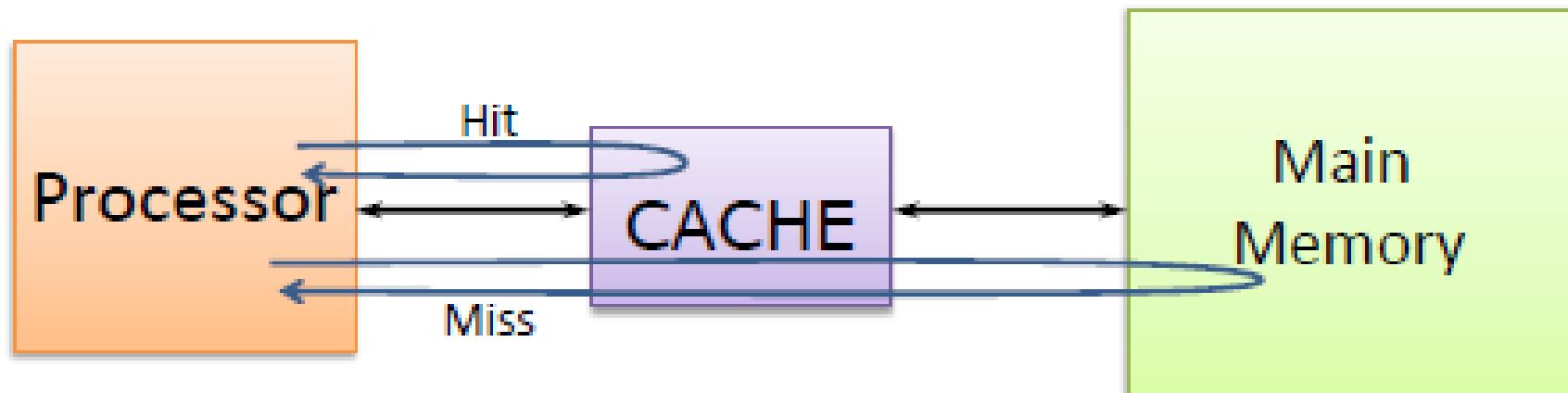
**Bank-uri de memorii cache și memorii cache multi-port**

**Optimizări software**

**Memorii cache neblocante**

**Cuvânt critic**

# Timpul mediu de acces la memoria cache

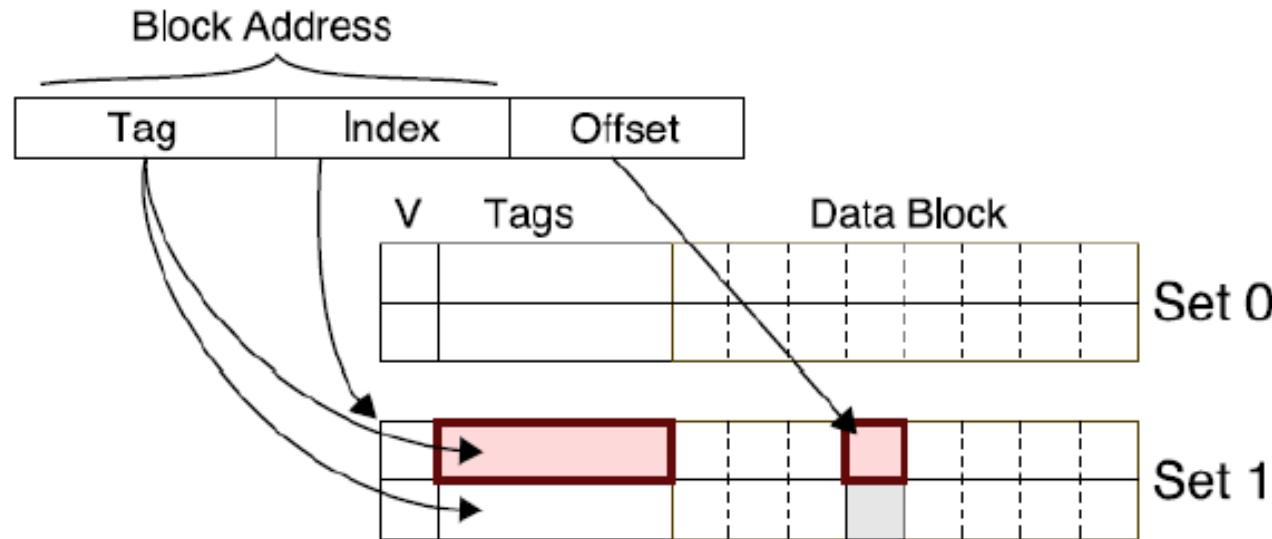


100 iterări

lw	F	D	X	M	W	
addu	F	D	X	M	W	<u>stall</u>
lw		F	D	X	M	M M M W
addu			F	D	X	X X M W
beq				F	D	D D X M W
subu					F	F F F F D X M W

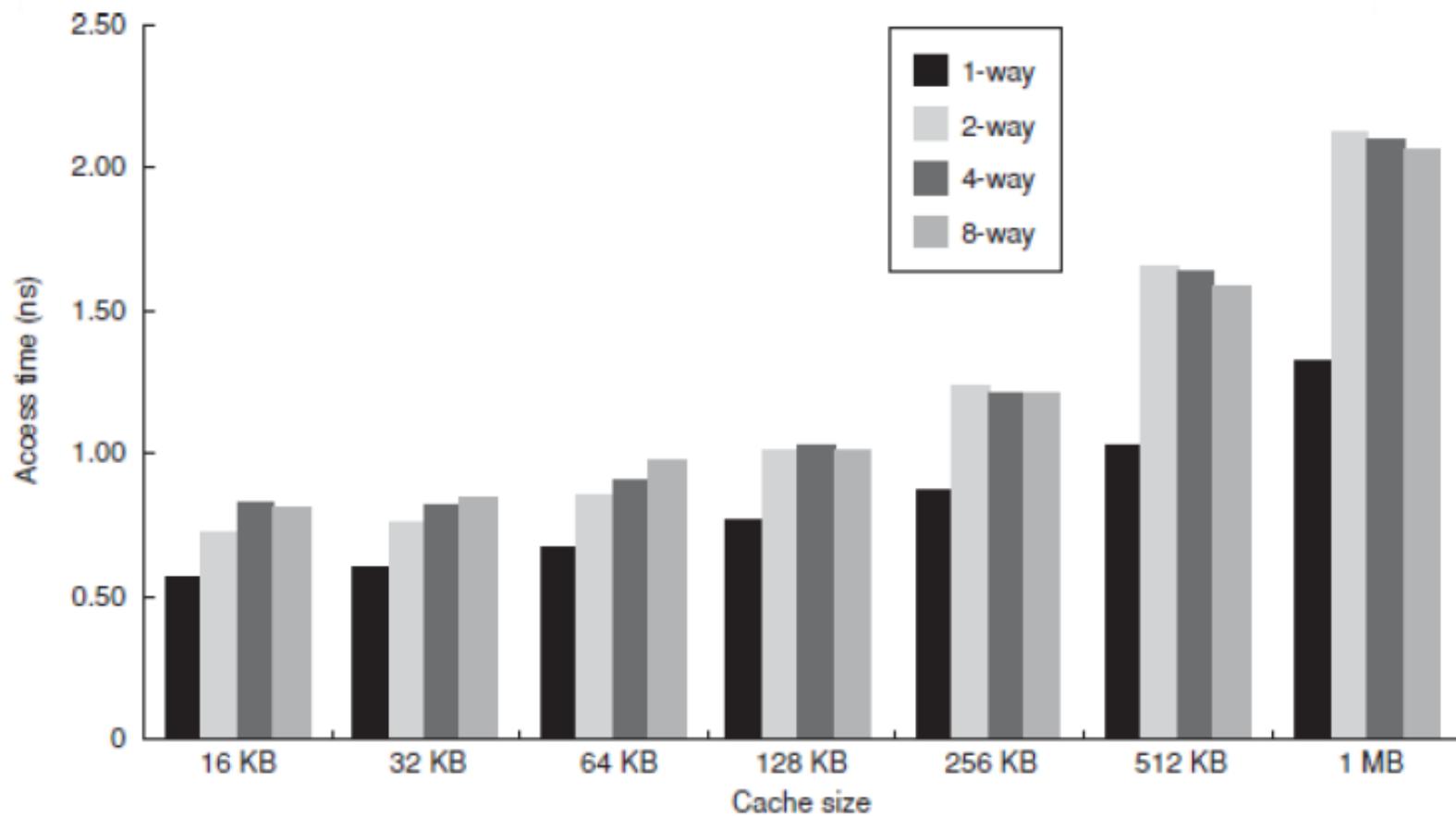
- Timpul mediu de acces = Timp HIT + (Rata de MISS \* Penalitatea la MISS)

# Categorii de MISS-uri: cei 3 C



- Compulsory – prima referință la un bloc apare chiar dacă avem o memorie cache infinită
- Capacity – memoria cache este prea mică pentru a memora toate datele necesare unui program, apare chiar dacă avem o politică perfectă de replasare a datelor în memorie (ciclu peste 5 linii de cache)
- Conflict – MISS-urile apar datorită coliziunilor – nu avem o proiectare full asociativă (ciclu peste 3 linii de cache)

# Reducerea timpului de HIT

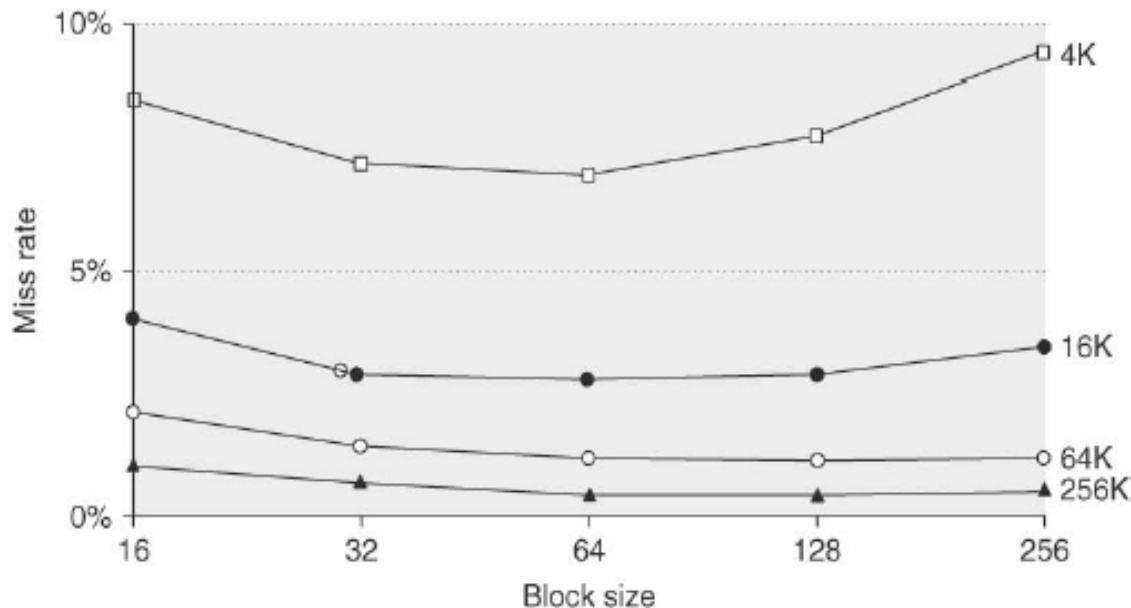


Plot from Hennessy and Patterson Ed. 4

Image Copyright © 2007-2012 Elsevier Inc. All rights Reserved.

- Memorii cache mici și simple ca proiectare

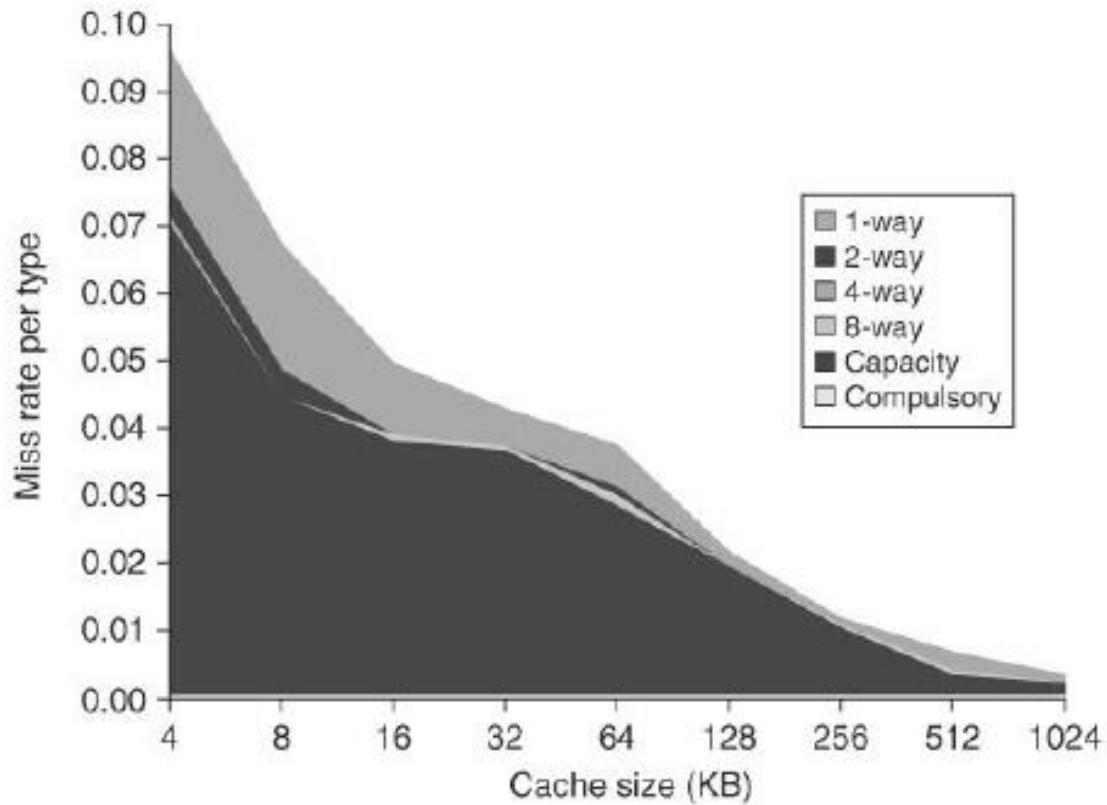
# Reducerea ratei de MISS: Dimensiuni mari de blocuri



Plot from Hennessy and Patterson Ed. 5 Image Copyright © 2011, Elsevier Inc. All rights Reserved.

- Overhead pentru TAG mic
- Exploatarea transferurilor rapide de la DRAM
- Se irosește lățimea benzii magistralei de transfer dacă datele nu sunt utilizate
- Mai puține blocuri înseamnă mai multe conflicte

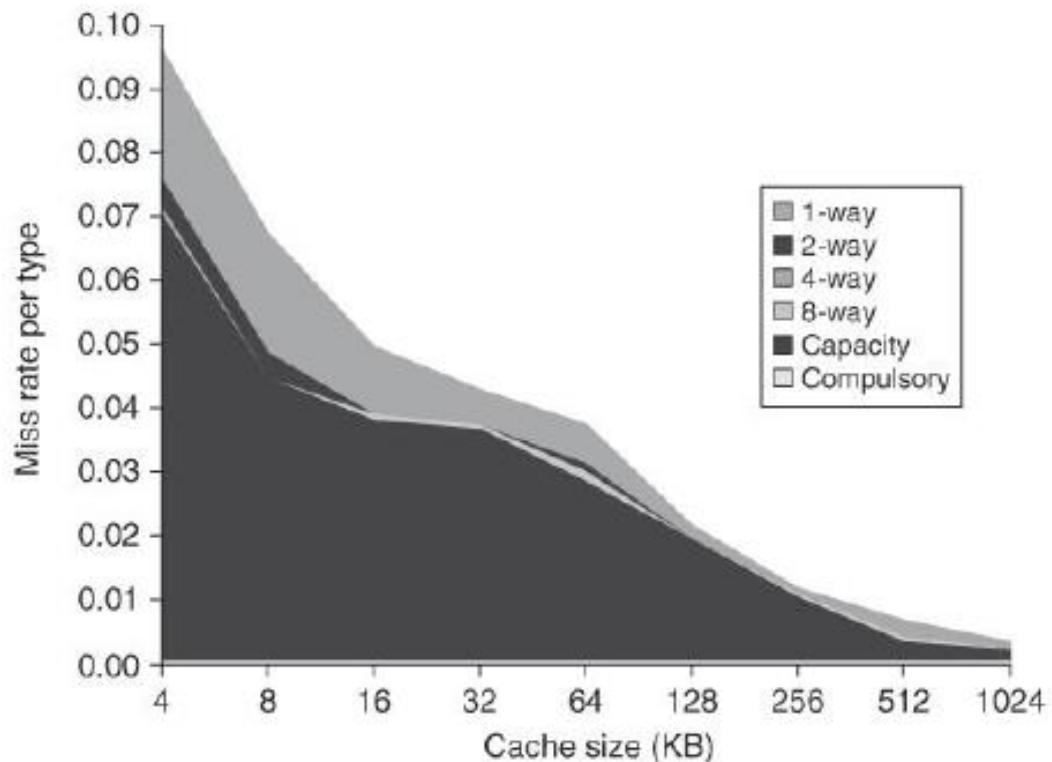
# Reducerea ratei de MISS: Dimensiune mare cache



Plot from Hennessy and Patterson Ed. 5 Image Copyright © 2011, Elsevier Inc. All rights Reserved.

- Dacă dimensiunea memoriei cache este dublă, rata de MISS scade în mod ușor cu  $\sqrt{2}$

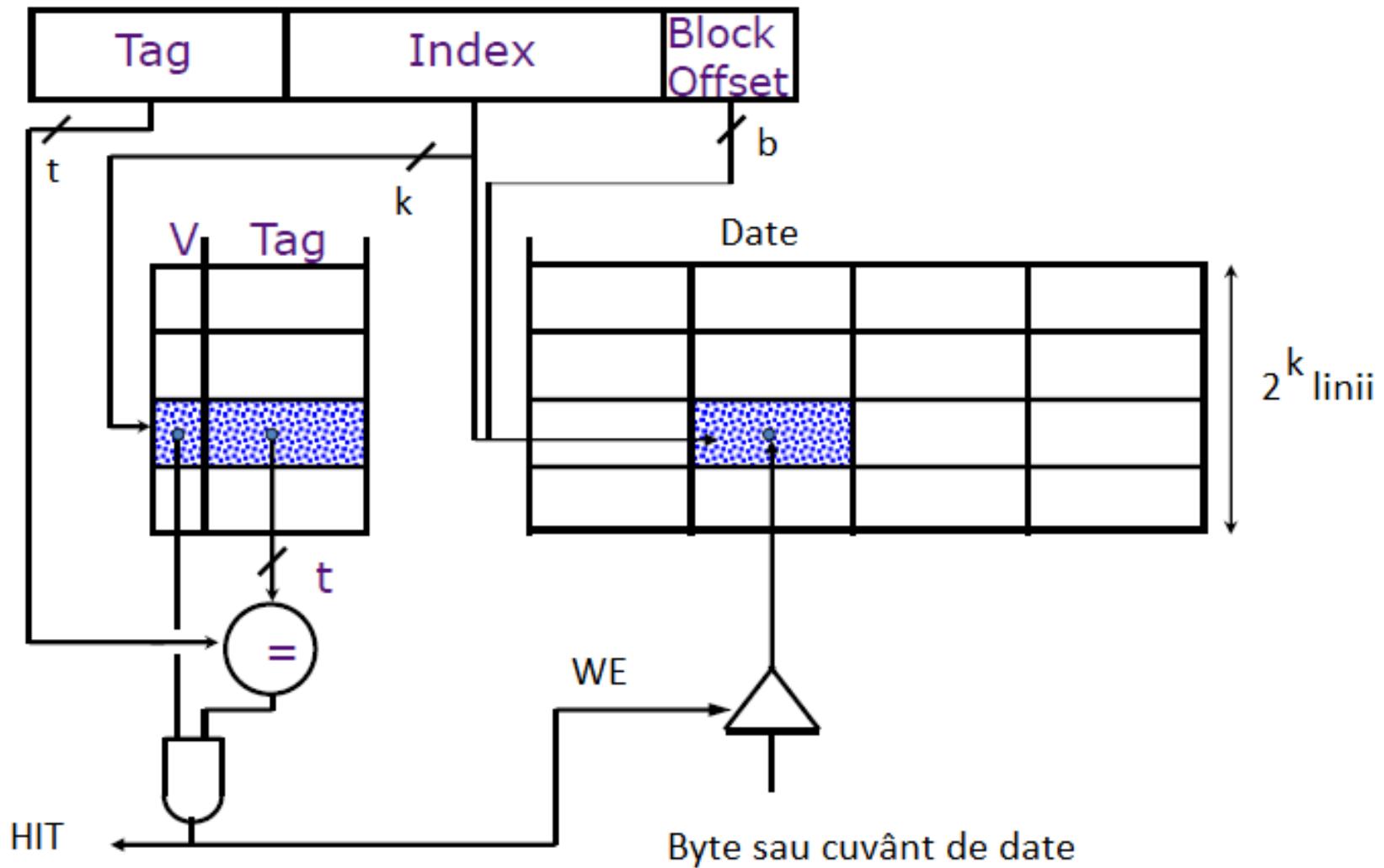
# Reducerea ratei de MISS: Asociativitate mare



Plot from Hennessy and Patterson Ed. 5 Image Copyright © 2011, Elsevier Inc. All rights Reserved.

Maparea directă pentru o memorie cache de dimensiune  $N$  are aproximativ aceeași rată de MISS ca o memorie cache set asociativă pe 2 căi de dimensiune  $\frac{N}{2}$

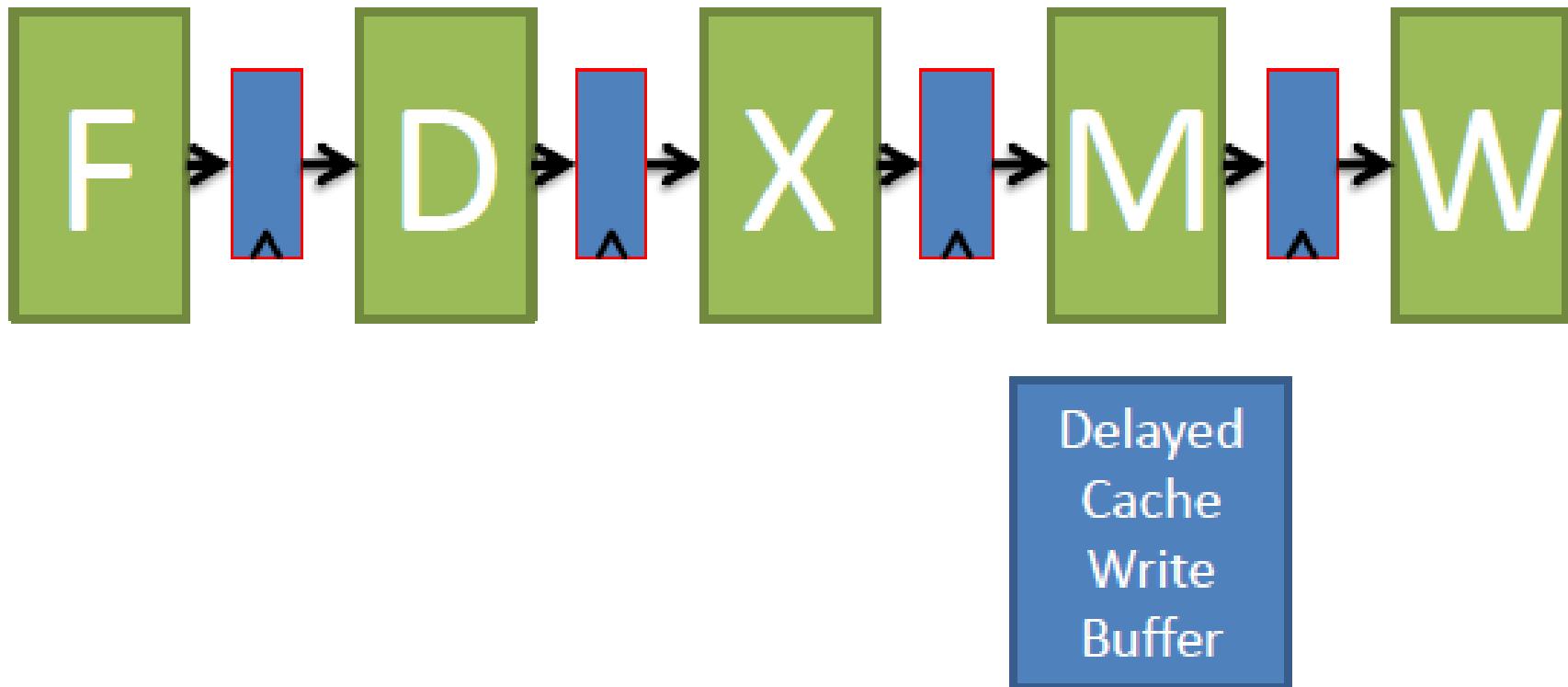
# Performanță scrierii



# Reducerea timpului de HIT pentru WRITE

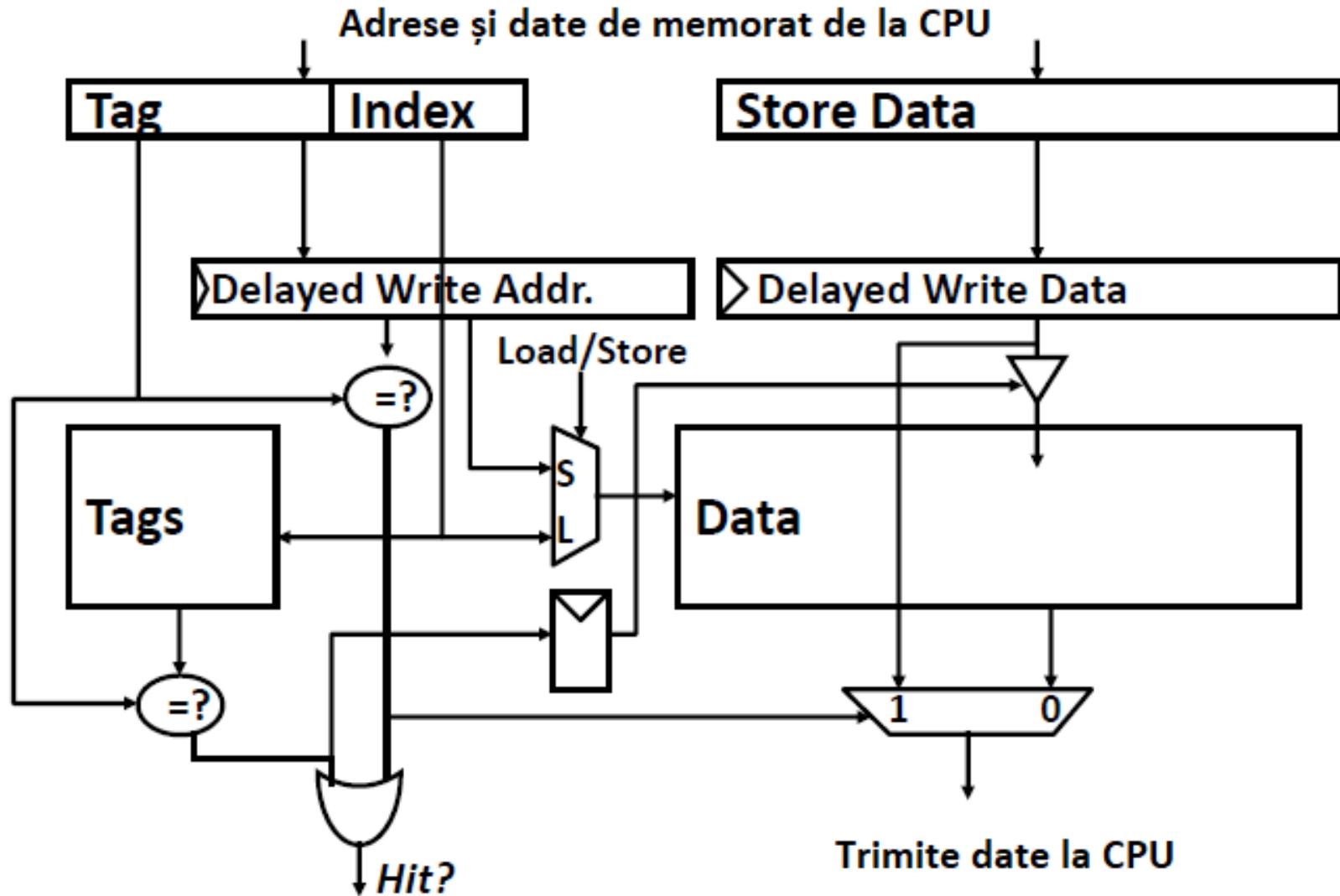
- **Problema :** Scrimerile necesită două cicluri în stagiul de memorie, un ciclu pentru verificarea TAG-ului și un ciclu pentru scrierea datelor dacă avem HIT
- **Soluție :**
  - Proiectarea de memorii de date RAM care pot realiza citiri și scrieri concurente, restaurarea valorii vechi după un MISS la TAG
  - Memorii cache complet asociative (CAM TAG): se activează cuvântul de linie doar dacă avem HIT
  - Scrieri în pipe: păstrarea datelor pentru scriere în vederea memorării într-un singur buffer deasupra memoriei cache – scrierea datelor în cache pe durata următoarei verificări a TAG-ului

# Scrierea pipeline în cache



- *Datele de la hit pentru memorare sunt scrise într-o porțiune de date a memoriei cache pe durata accesării TAG-ului a unei operații de memorare subsecvente.*

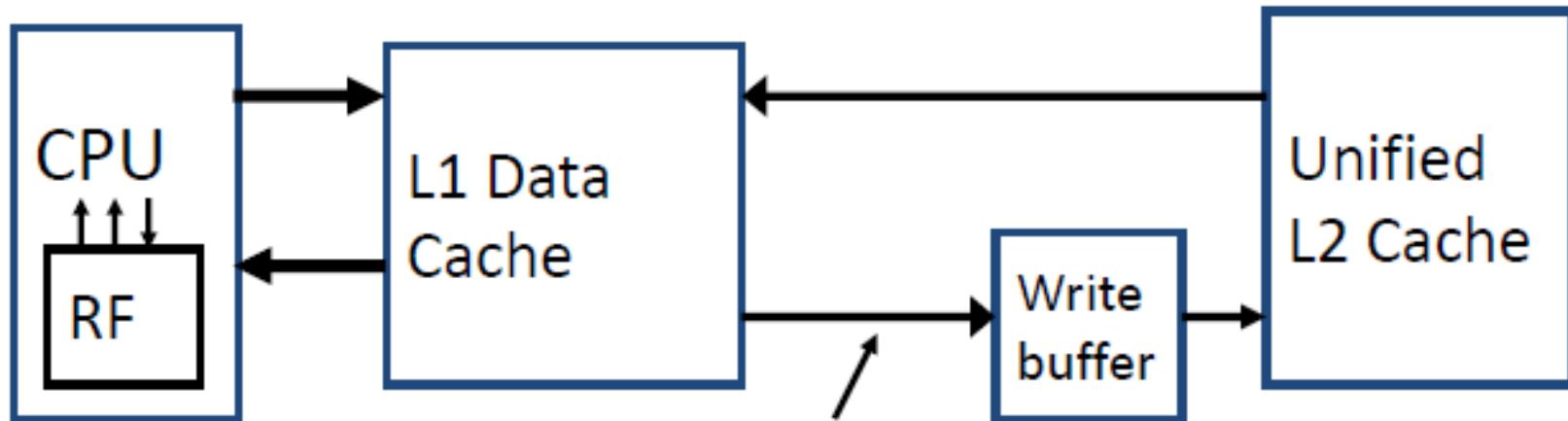
# Scrierea pipeline în cache



# Eficiența unui cache cu pipe

Optimizare cache	Rata de MISS	Penalitatea de MISS	Timpul de HIT	Bandwidth
Scrieri în mod pipe			-	+

# Buffer de scriere pentru reducerea penalății de MISS



Ștergerea liniilor "dirty" pentru memorii cache în mod writeback  
SAU

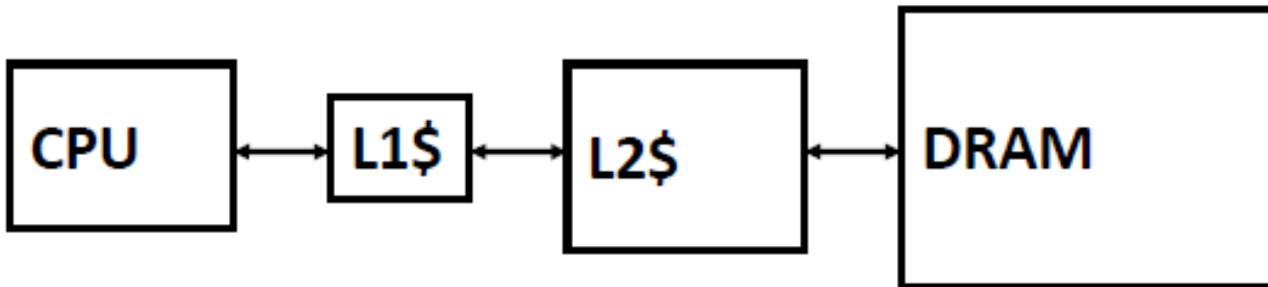
Toate scrierile se fac în memoria cache prin metoda writethrough

- Procesorul nu este oprit pe durata scrierilor și MISS-urile pe read merg mai departe către memoria principală
- **Problema:** Buffer-ul de scriere poate menține valori noi ale locațiilor necesare de un MISS la read
- **Schema simplă:** pe un MISS de read, așteptăm până când buffer-ul de scriere este golit
- **Schema rapidă:** Verificăm adresele din buffer-ul de scriere să nu conțină adrese de MISS pe read, dacă nu avem potrivire, atunci MISS-urile de scriere se execută, altfel returnează valoarea în buffer-ul de scriere

# Eficiență unui buffer de scriere

Optimizare cache	Rata de MISS	Penalitatea de MISS	Timpul de HIT	Bandwidth
Buffer de scriere		+		

# Memorii cache multinivel



- **Problema:** O memorie cache nu poate fi mare și rapidă
- **Soluție:** creșterea dimensiunii memoriei cache pe fiecare nivel
- Rata de MISS locală – MISS în cache / accese la cache
- Rata de MISS globală - MISS în cache / accese CPU la memorie
- MISS-uri pe instrucțiune - MISS în cache / numărul de instrucțiuni

# Prezența lui L2 influențează proiectarea lui L1

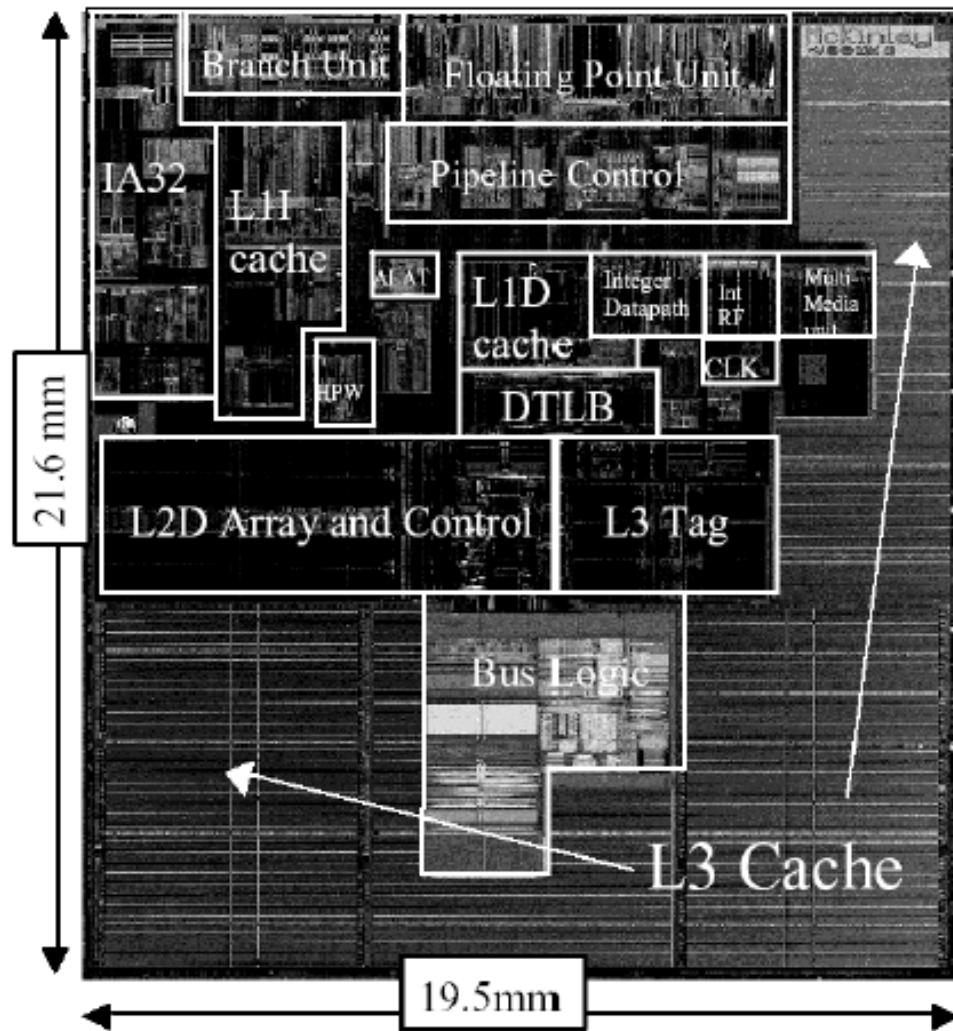
- Dacă există L2 atunci L1 este mic
  - Crește MISS rate pentru L1 și se reduc:
    - timpul de HIT pentru L1 și MISS penalty pentru L1
    - Reducem energia medie de acces
- Utilizăm write-through pentru L1 cu L2 pe același chip
  - Write-back pentru L2 absoarbe traficul de write
  - Cel mult un MISS la L1 pentru accesele la L1 simplifică controlul pipeline
  - Simplificarea problemelor legate de coerență a datelor
  - Simplificarea recuperării erorilor în L1 (putem utiliza doar biți de paritate în L1 și reîncărcăm de la L2 când avem eroare de paritate la citirea lui L1)

# Politica de incluziune

- Cache multinivel inclusiv
  - Memoriile cache interioare mențin copii ale datelor în memoriile cache exterioare
  - Accesul la datele externe coerente necesită doar cache-urile exterioare
- Memoriile cache multinivel exclusive
  - Memoriile cache interioare nu mențin date în cache-urile exterioare
  - În caz de MISS se inter-schimbă liniile memorilor cache interne / exterioare
  - Utilizat în AMD Athlon cu 64KB L1 și 256 KB L2

# Itanium 2 – memorii cache pe chip

(Intel/HP. 2002)



**Level 1:** 16KB, 4-way s.a., 64B line, quad-port (2 load+2 store), single cycle latency

**Level 2:** 256KB, 4-way s.a., 128B line, quad-port (4 load or 4 store), five cycle latency

**Level 3:** 3MB, 12-way s.a., 128B line, single 32B port, twelve cycle latency

Image Credit: Intel

# IBM POWER 7 – memorii cache pe chip

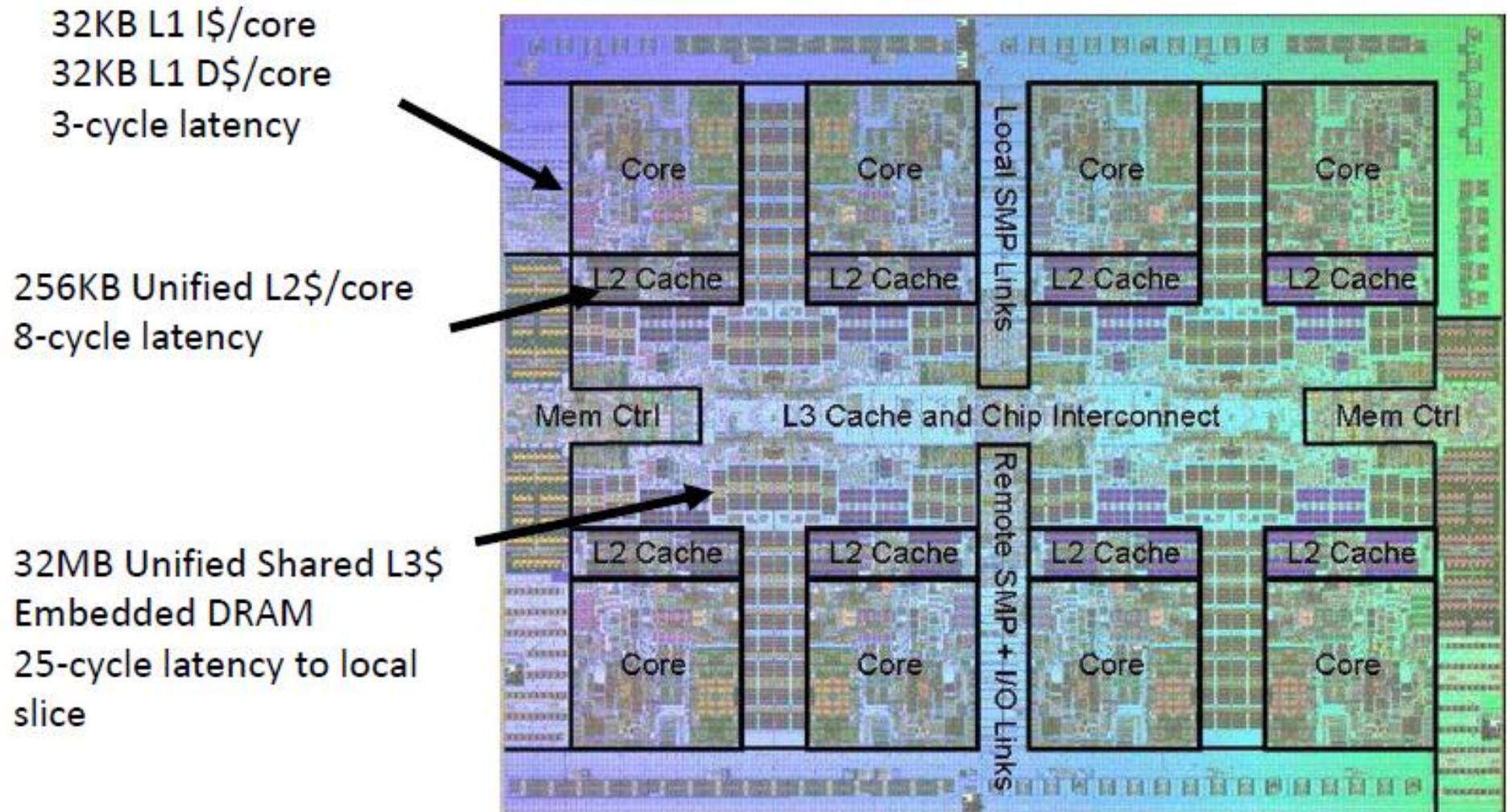


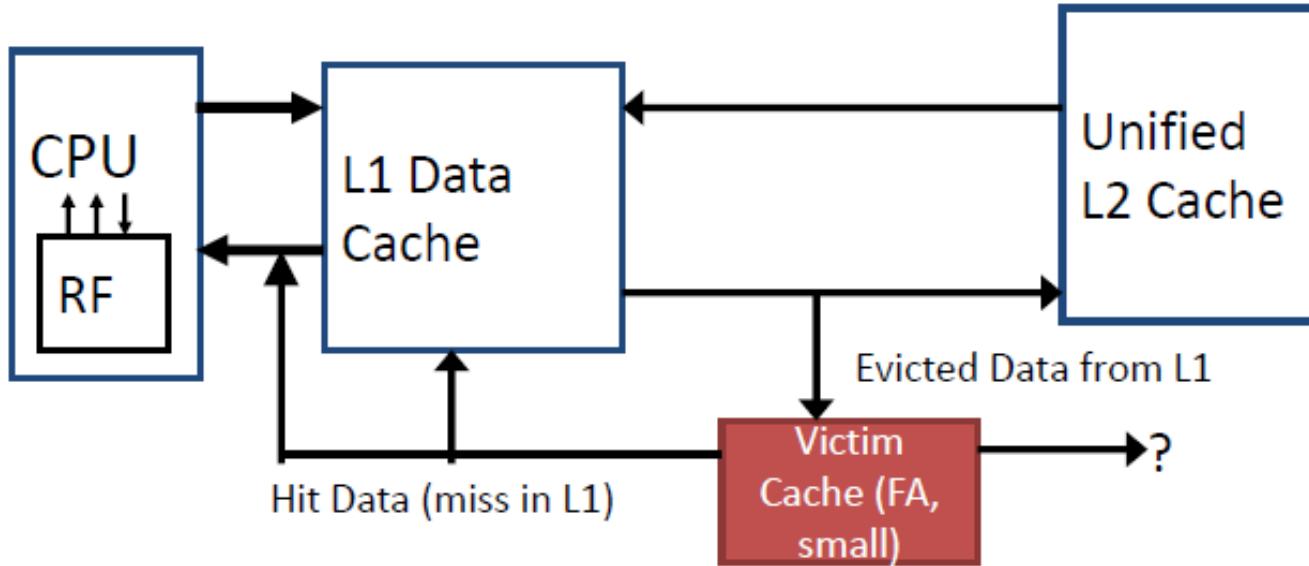
Image Credit: IBM

Courtesy of International Business Machines Corporation.<sup>25</sup>

© International Business Machines Corporation.

# Eficiența memoriei cache multinivel

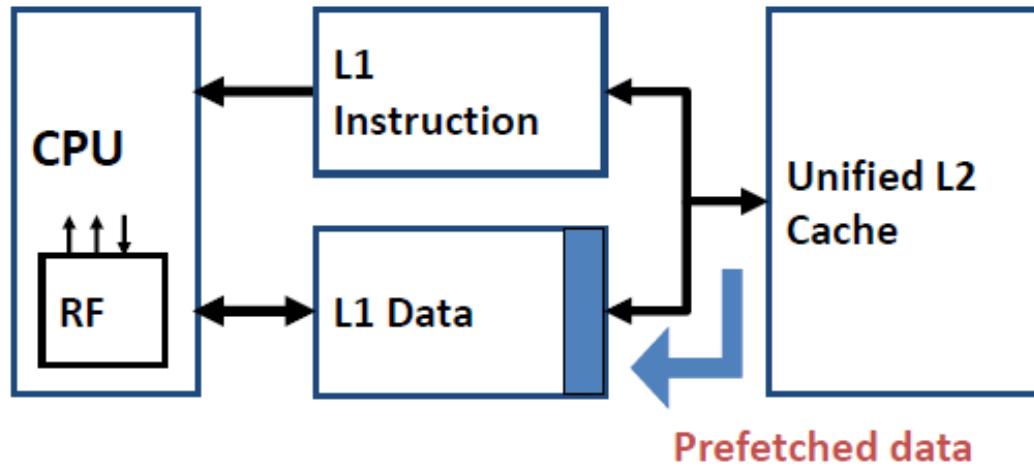
Optimizare cache	Rata de MISS	Penalitatea de MISS	Timpul de HIT	Bandwidth
Memorie cache multinivel	+	+		



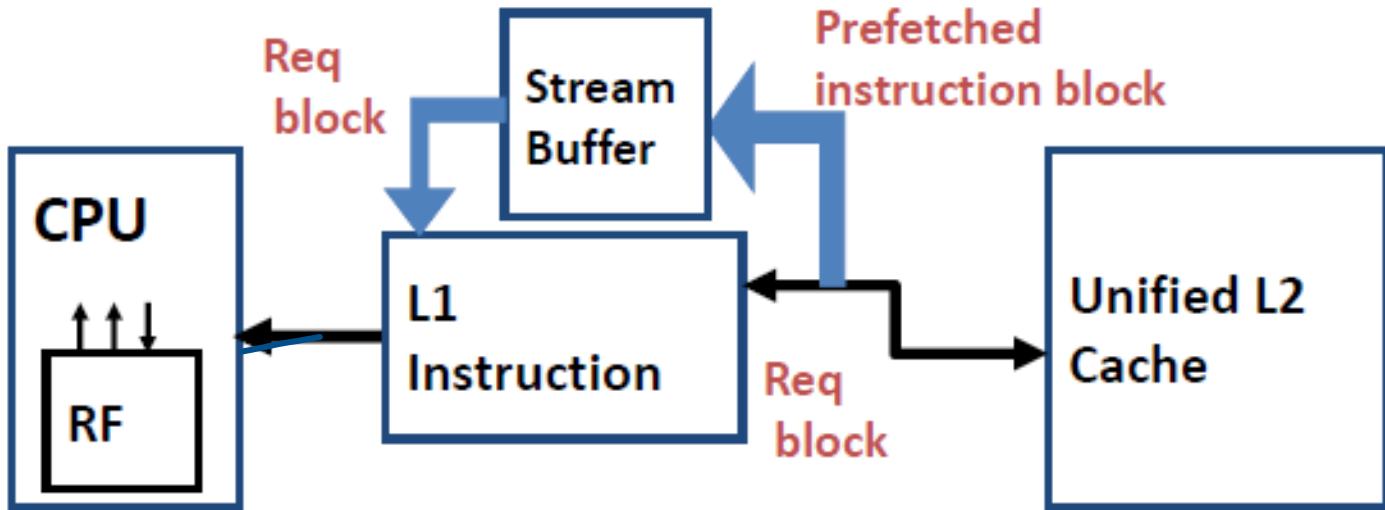
- Cache mic complet asociativ pentru liniile eliminate recent
  - Uzual 4 – 16 blocuri
- Reducerea conflictelor pe MISS
  - Mai multă asociativitate pentru un număr mic de linii
- Poate fi verificat în paralel sau serie cu memoria principală
- MISS în L1, Hit în VC: VC → L1, L1 → VC
- MISS în L1, MISS în VC: L1 → VC, VC → ?

<b>Optimizare cache</b>	<b>Rata de MISS</b>	<b>Penalitatea de MISS</b>	<b>Timpul de HIT</b>	<b>Bandwidth</b>
Victimă cache	+ <u>-</u>	+		

- Utilizăm specularea pentru instrucțiunile și accesele de date viitoare și le citim în cache / cache-uri
  - Accesul instrucțiunilor mai ușor de prezis decât accesul datelor
- Mai multe metode de citire în avans
  - prefetch hardware
  - prefetch software
  - Scheme mixte
- Ce tipuri de MISS afectează prefetch-ul ?



- Utilitatea – ar trebui să producă HITS
- Timpul – nici târziu dar nici prea devreme
- Poluarea cache-ului și a lătimii de bandă



- Citim 2 blocuri în caz de MISS: blocul solicitat (i) și următorul bloc consecutiv (i+1)
- Blocul solicitat este plasat în cache iar blocul următor în buffer-ul de instrucțiuni
- Dacă MISS în cache dar HIT în buffer, mutăm blocul din buffer în cache și prefetch la blocul următor (i+2)

- Prefetch pe MISS:
  - Prefectch  $b+1$  pe MISS-ul lui  $b$
  - Schema cu anticiparea unui bloc (OBL)
    - Inițiere prefetch pentru  $b+1$  când accesăm  $b$
    - De ce diferă dimensiunea blocului ?
    - Putem extinde la anticiparea a  $N$  blocuri ?
  - Încălcarea prefetch-ului
    - Dacă observăm secvența de acces  $b, b+N, b+2N$  atunci prefetch  $b+3N$  (IBM Power 5 - 2003)

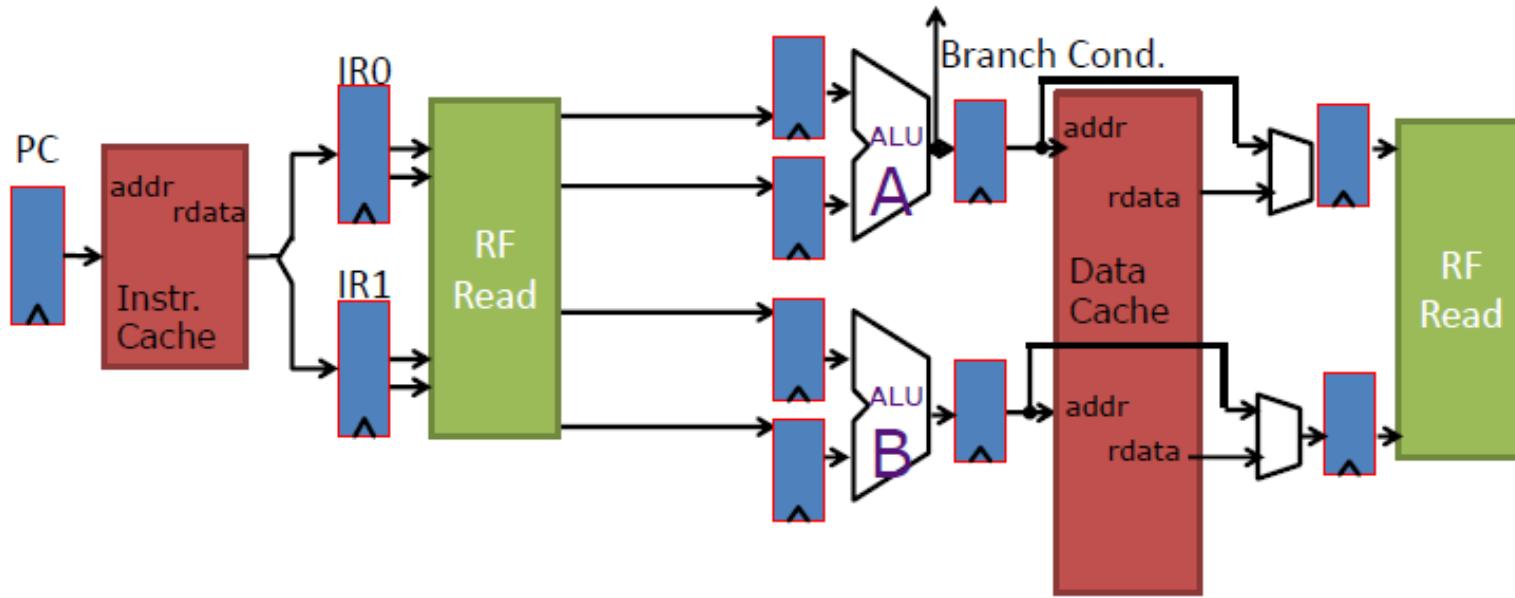
```
for(i=0; i < N; i++) {  
    prefetch( &a[i + 1] );  
    prefetch( &b[i + 1] );  
    SUM = SUM + a[i] * b[i];  
}
```

- Timpul este cel mai mare dezavantaj, nu avem predictibilitate
  - Dacă facem prefetch foarte aproape de momentul în care datele sunt cerute este probabil prea târziu
  - Prefetch prea devreme produce poluare
  - Estimăm cât de mult durează ca datele să ajungă în L1 și apoi setăm P corespunzător

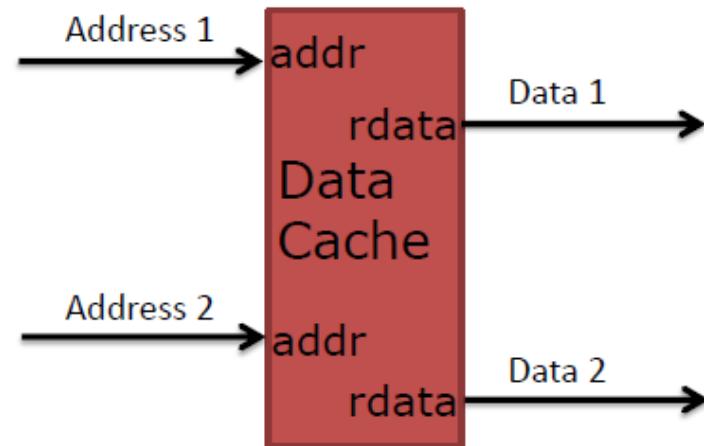
```
for(i=0; i < N; i++) {  
    prefetch( &a[i + P] );  
    prefetch( &b[i + P] );  
    SUM = SUM + a[i] * b[i];  
}
```

- Trebuie luat în considerare costul instrucțiunilor de prefetch

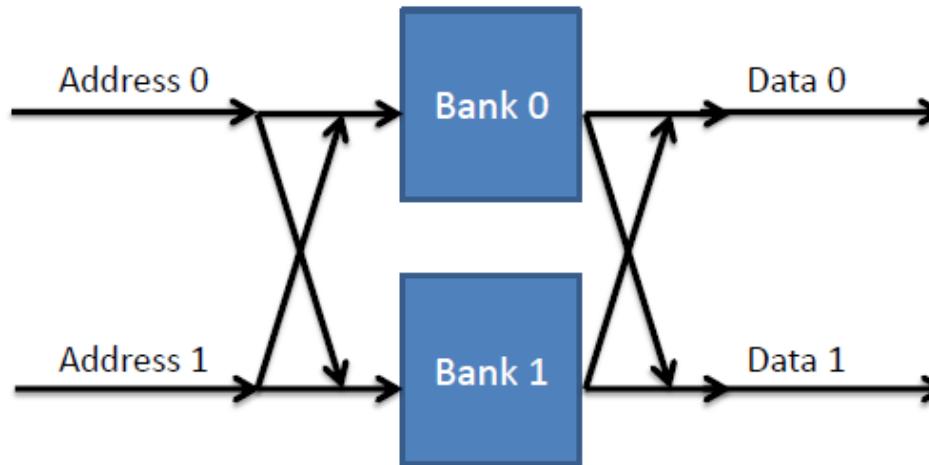
Optimizare cache	Rata de MISS	Penalitatea de MISS	Timpul de HIT	Bandwidth
Prefetch	+	+		



- Complicat – 2 instrucțiuni STORE pe aceeași linie sau LOAD și STORE pe aceeași linie



- Aria crește considerabil – pentru 2 porturi se poate dubla
- Timpul de HIT crește



- Partiționăm spațiul de adrese în mai multe bank-uri
  - Utilizăm porțiuni de adrese
- Beneficii – throughput mărit
- Provocări
  - Conflicte între bank-uri
  - Utilizare când și când
  - Mai multe fire

<b>Optimizare cache</b>	<b>Rata de MISS</b>	<b>Penalitatea de MISS</b>	<b>Timpul de HIT</b>	<b>Bandwidth</b>
Bank memorie cache				+

- Restructurarea codului afectează secvența de acces a blocurilor de date
  - Gruparea acceselor la date pentru a îmbunătăți localizarea spațială
  - Reordonarea acceselor la date pentru a îmbunătăți localizarea temporară
- Prevenirea intrării datelor în cache
  - Utilă pentru variabilele care vor fi accesate doar odată înainte de a fi replasate
  - Este nevoie de un mecanism software care să comunice hardware-ului să nu facă cache de date
- Eliminarea datelor care nu vor mai fi niciodată folosite
  - Stream-ul de date exploatează doar localizarea spațială
  - Replasare în locații de memorie cache moarte

```
for(j=0; j < N; j++) {  
    for(i=0; i < M; i++) {  
        } x[i][j] = 2 * x[i][j];  
    } }
```



```
for(i=0; i < M; i++) {  
    for(j=0; j < N; j++) {  
        } x[i][j] = 2 * x[i][j];  
    } }
```

- Ce tip de localizare nu este îmbunătățită ?

```
for(i=0; i < N; i++)
    a[i] = b[i] * c[i];
```

```
for(i=0; i < N; i++)
    d[i] = a[i] * c[i];
```



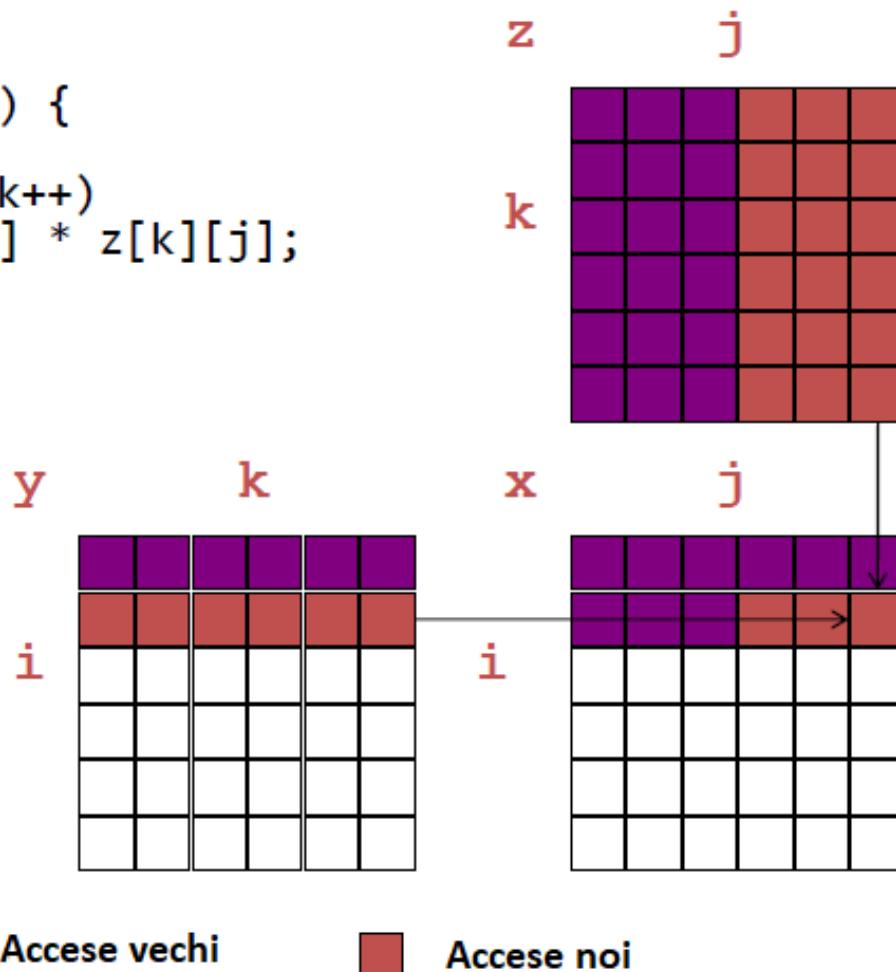
```
for(i=0; i < N; i++)
{
    a[i] = b[i] * c[i];
    d[i] = a[i] * c[i];
}
```

- Ce tip de localizare nu este îmbunătățită ?

```

for(i=0; i < N; i++)
    for(j=0; j < N; j++) {
        r = 0;
        for(k=0; k < N; k++)
            r = r + y[i][k] * z[k][j];
        x[i][j] = r;
    }

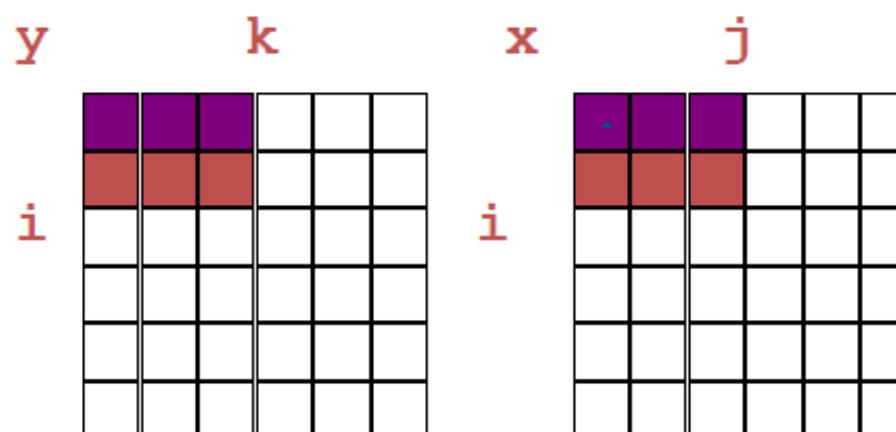
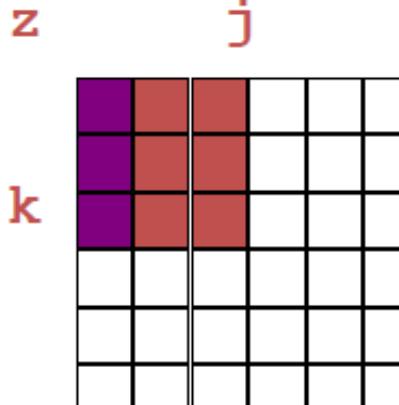
```

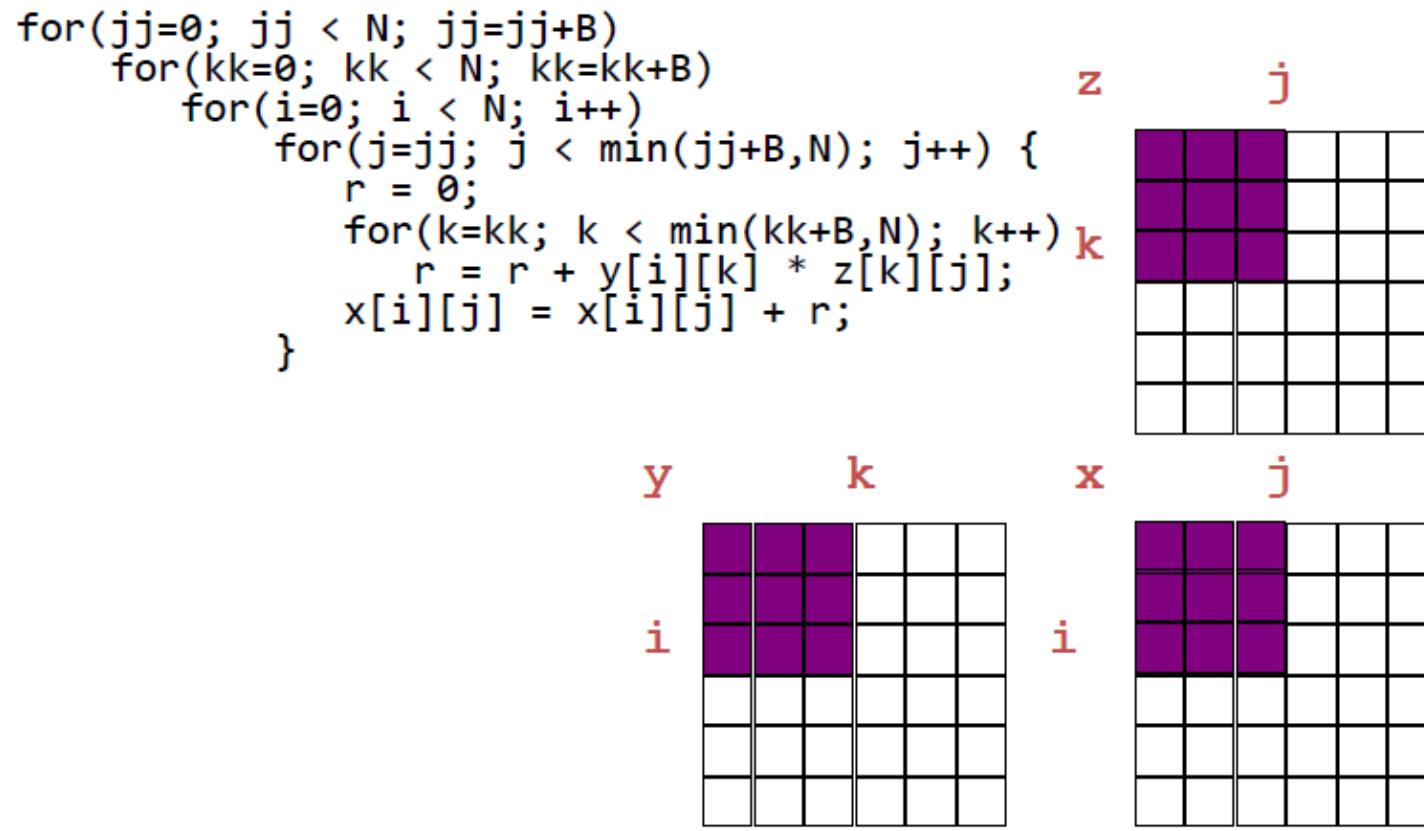


```

for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B,N); j++) {
                r = 0;
                for(k=kk; k < min(kk+B,N); k++) k
                r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
}

```



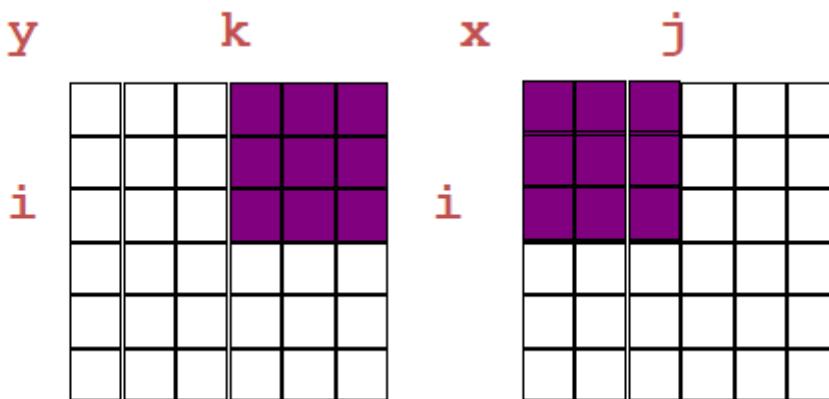
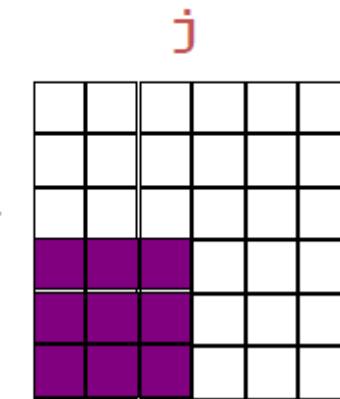


- Ce tip de localizare este îmbunătățită ?

```

for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B,N); j++) {
                r = 0;
                for(k=kk; k < min(kk+B,N); k++) k
                r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
}

```

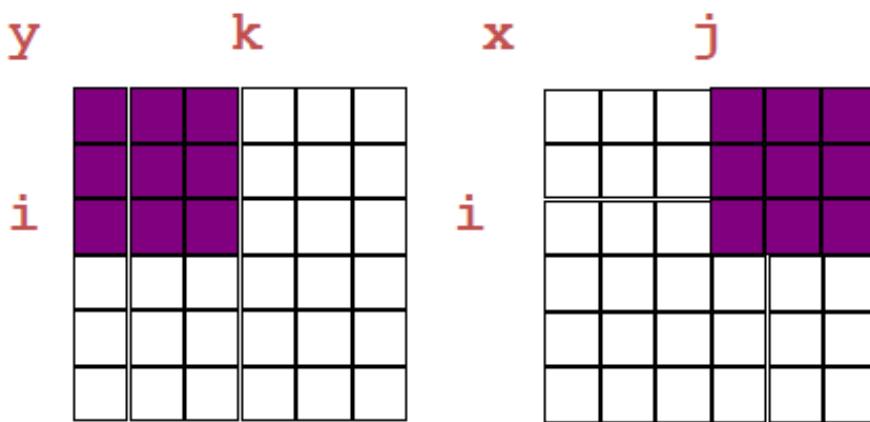
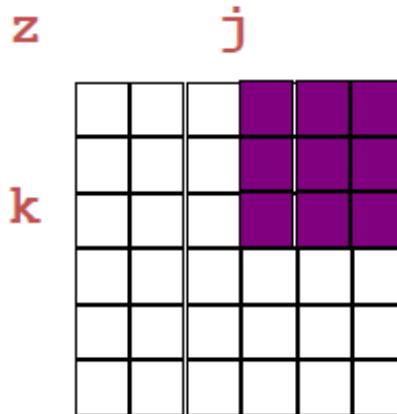


- Ce tip de localizare este îmbunătățită ?

```

for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B,N); j++) {
                r = 0;
                for(k=kk; k < min(kk+B,N); k++)
                    r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
}

```

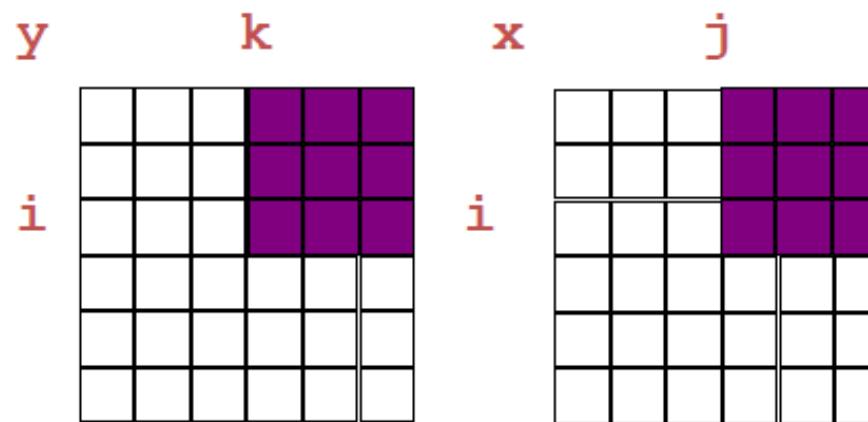
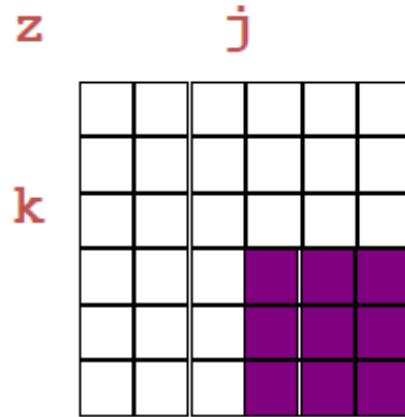


- Ce tip de localizare este îmbunătățită ?

```

for(jj=0; jj < N; jj=jj+B)
    for(kk=0; kk < N; kk=kk+B)
        for(i=0; i < N; i++)
            for(j=jj; j < min(jj+B,N); j++) {
                r = 0;
                for(k=kk; k < min(kk+B,N); k++) k
                r = r + y[i][k] * z[k][j];
                x[i][j] = x[i][j] + r;
            }

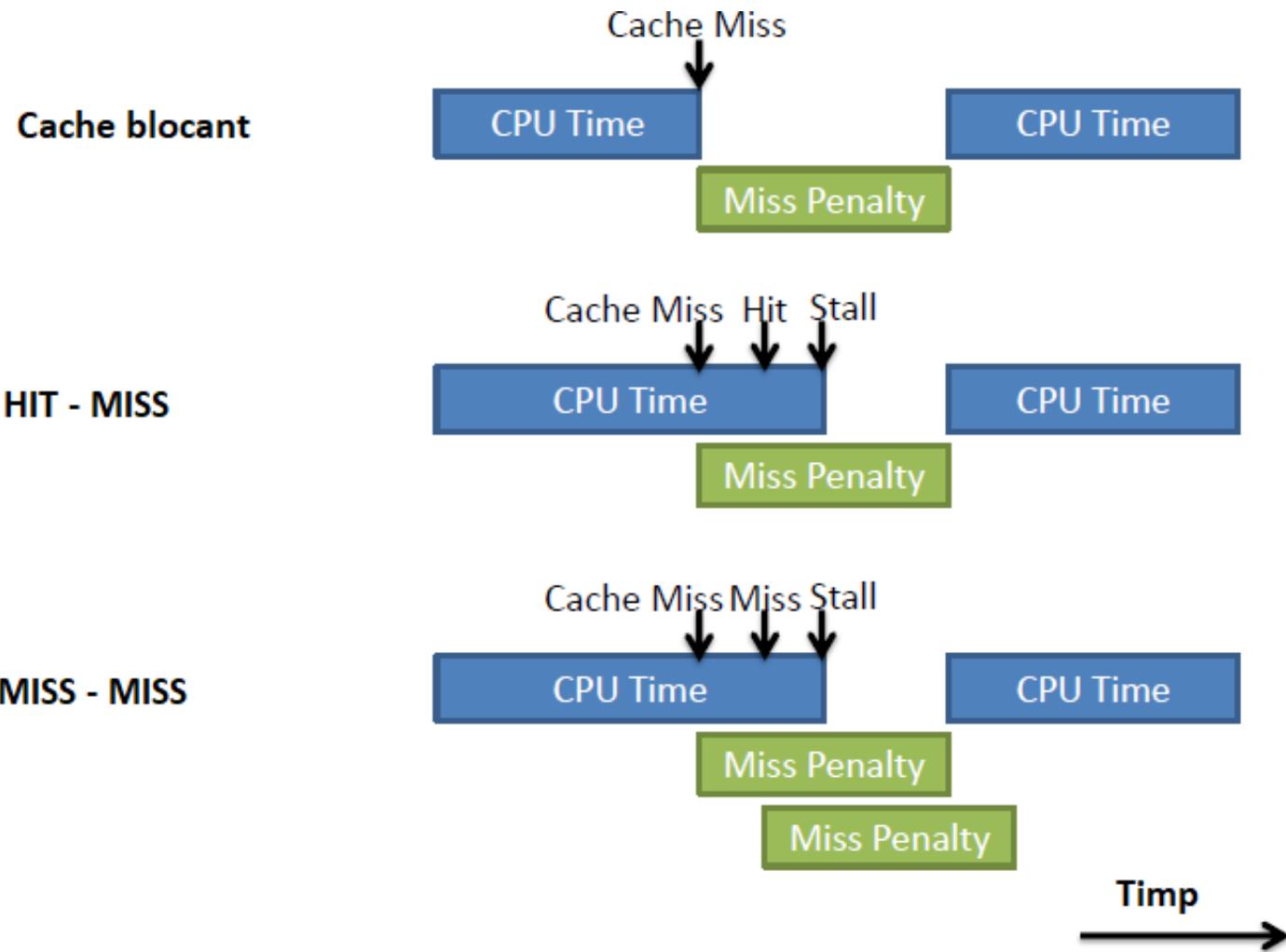
```



- Ce tip de localizare este îmbunătățită ?

Optimizare cache	Rata de MISS	Penalitatea de MISS	Timpul de HIT	Bandwidth
Optimizare compilator		+		

- Se permit accesele la cache-urile subsecvente după apariția unui MISS la cache
  - HIT – MISS
  - MISS – MISS (MISS-uri concurente)
- Se pretează la procesoarele IO sau OOO
- Provocări
  - Menținerea ordinii când avem multiple MISS-uri
  - LOAD sau STORE către o adresă de MISS deja în așteptare (necesită combinarea lor)



**MSHR/MAF (MISS Address File)**

V	Block Address	Issued

**V:** Valid

**Adresa de bloc:** Adresa blocului din cache în memoria principală

**Citire:** Citire din memoria principală / următorul nivel de memorie cache

**Intrare LOAD/STORE**

V	MSHR Entry	Type	Offset	Destination

**V:** Valid

**Intrare MSHR:** număr de intrare

**Tip:** {LW, SW, SH, LB, SB}

**Offset:** Offset-ul din interiorul unui bloc

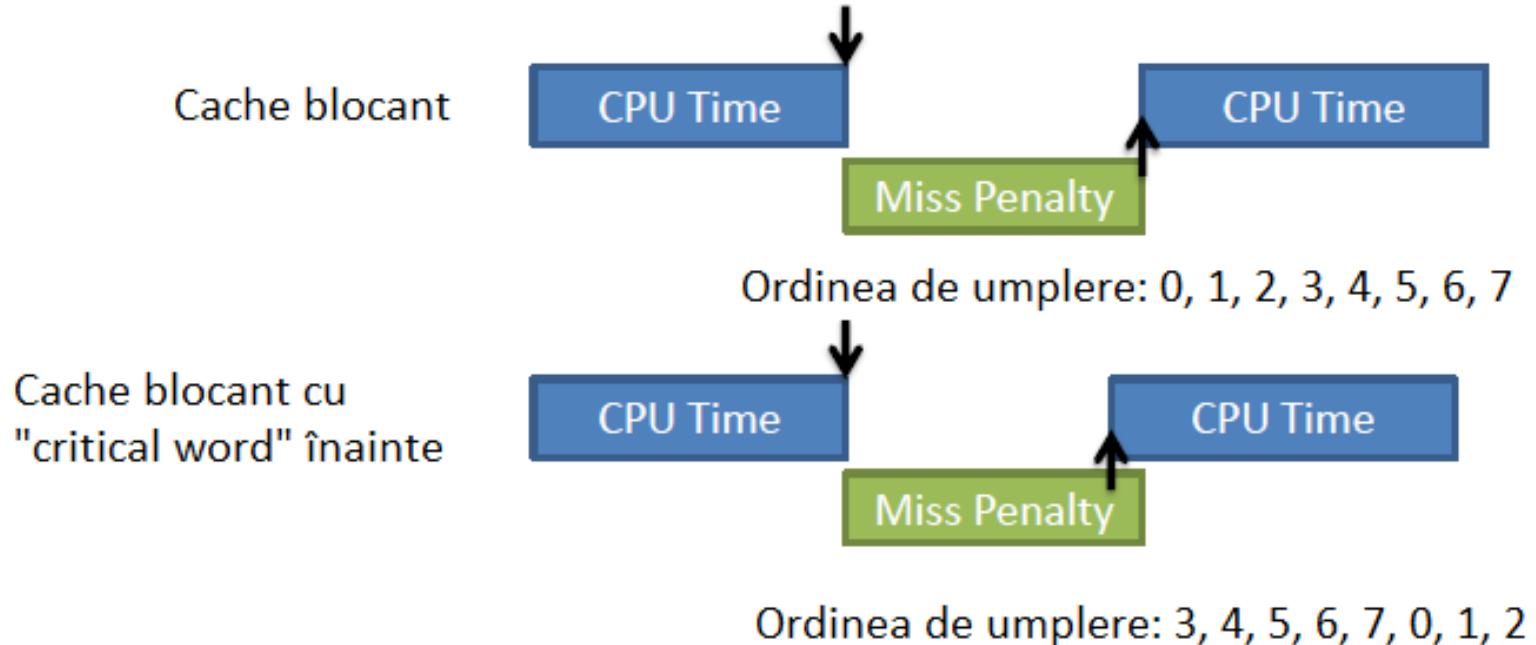
**Destinație:** (LOAD) Registr, (STORE) intrare în buffer-ul de STORE

- În caz de MISS
  - Verificăm MSHR pentru potrivire de adrese
    - Dacă da – alocăm o nouă intrare LOAD/STORE să pointeze către MSHR
    - Dacă nu – Alocăm o nouă intrare în MSHR și o intrare LOAD/STORE
    - Dacă nu mai avem intrări libere în MSHR sau în tabela LOAD/STORE, facem stall pentru a preveni noi LD/ST
  - Pe datele returnate de la memorie
    - Identificăm LOAD/STORE care așteaptă după aceste date
      - Facem forward la datele LOAD către procesor / ștergem buffer-ul STORE
      - Pot fi mai multe LOAD / STORE-uri
    - Scriere date în cache
  - Când liniile de cache sunt complet returnate – dealocăm intrările MSHR

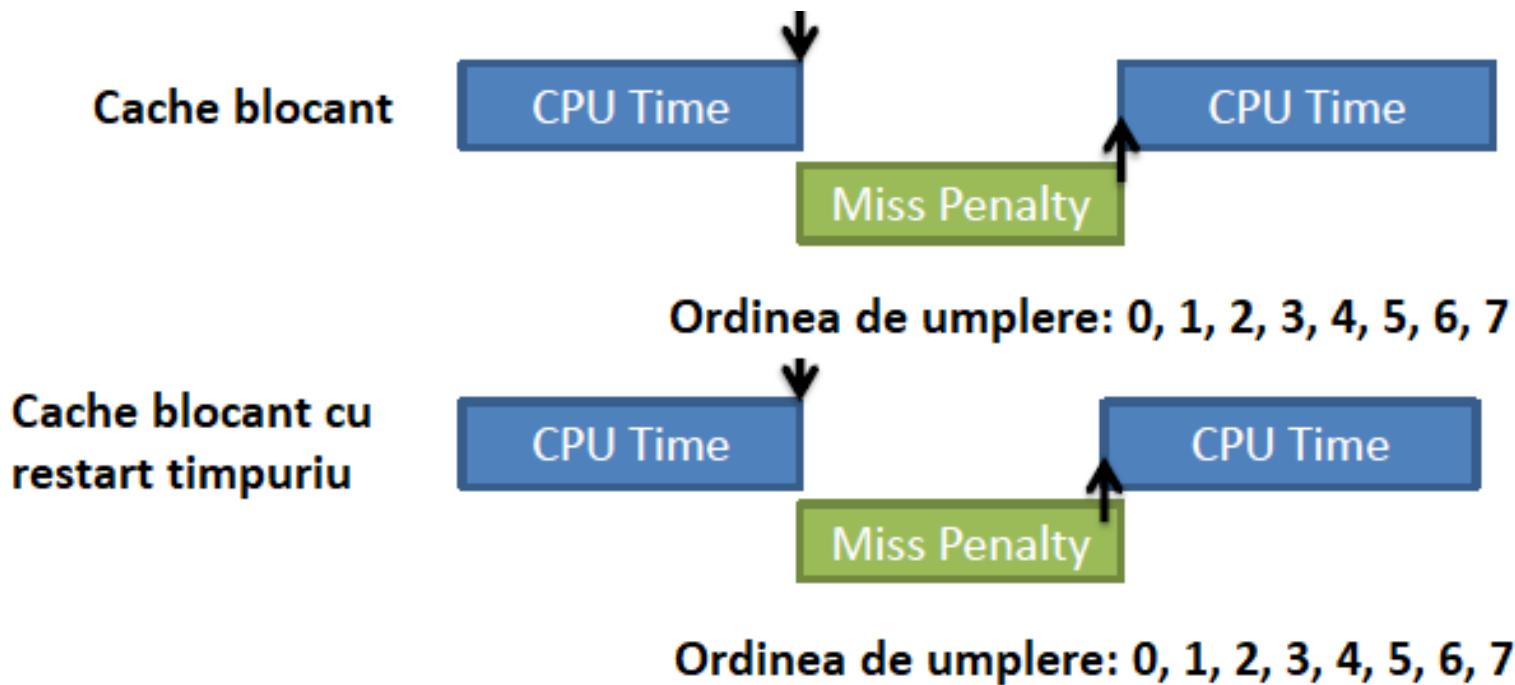
- Este necesar să avem Scoreboard pentru registrele individuale
- LOAD MISS
  - Marcarea registrului destinație ca și ocupat
- Returnare dată pe LOAD
  - Marcarea registrului destinație ca disponibil
- Orice folosire a unui registru ocupat conduce la oprirea execuției procesorului.

<b>Optimizare cache</b>	<b>Rata de MISS</b>	<b>Penalitatea de MISS</b>	<b>Timpul de HIT</b>	<b>Bandwidth</b>
Cache neblocant		+		+

- Se solicită întâi un cuvânt lipsă din memorie
- Restul liniilor cache vin după „cuvântul critic”
  - În mod uzual cuvintele vin înapoi într-o ordine rotită



- Datele se returnează din memorie în ordine
- Procesorul restartează execuția când cuvintele necesare sunt returnate



Optimizare cache	Rata de MISS	Penalitatea de MISS	Timpul de HIT	Bandwidth
cuvântul critic primul și restart timpuriu		+		