

# Calculatoare Numerice

– Cursul 11 –

## Banda de asamblare

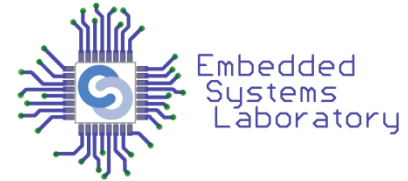
Facultatea de Automatică și Calculatoare  
Universitatea Politehnica București

# Comic of the day



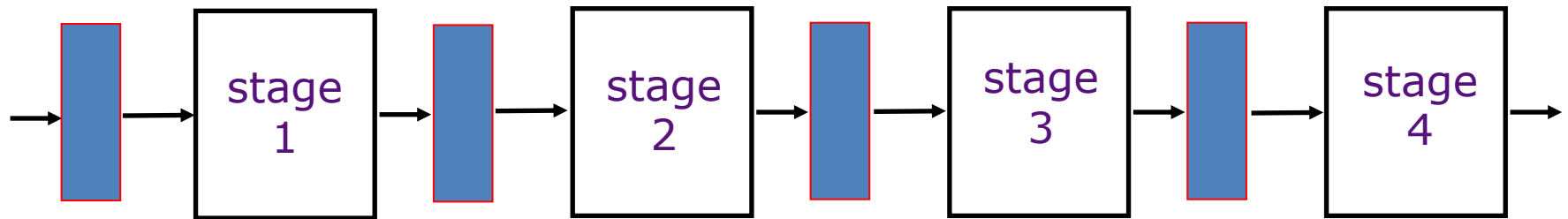
<http://xkcd.com/619/>

# Din cursurile anterioare



- Microcodificarea a devenit din ce în ce mai puțin folosită pe măsură ce diferențele de performanță dintre RAM și ROM s-au micșorat
- Seturile complexe de instrucțiuni sunt dificil de executat în pipeline, așa că este foarte dificil să mărești performanța odată cu creșterea numărului de porți per procesor.
- ISA RISC Load-Store proiectate pentru eficiență maximă și execuție pipeline
  - Foarte similar cu microcod verticalizat
  - Inspirat de mașinile Cray (vom vorbi despre ele mai târziu)
- Legea de fier a performanței descrie destul de eficient spațiul de lucru
  - Echilibru între instrucțiuni/program, cicli/instrucțiune și timp/ciclu

# O bandă de asamblare ideală

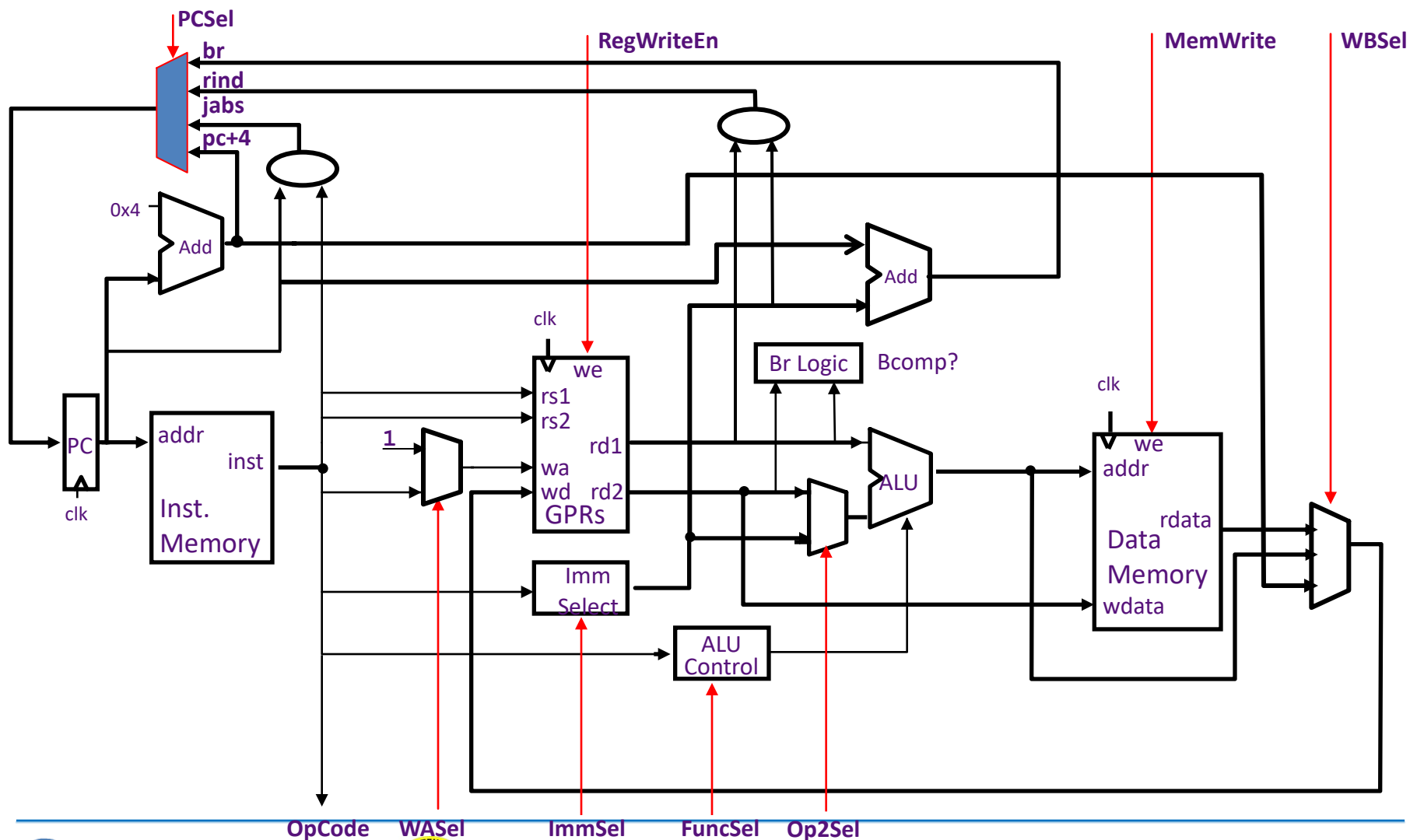


- Toate obiectele trec prin același niveluri
- Fără resurse partajate între oricare două niveluri
- Întârzierea de propagare prin toate etapele benzii de asamblare este egală
- Planificarea intrării unui obiect în pipeline nu este afectată de obiectele din alte etape

*Aceste condiții sunt valabile pentru liniile de asamblare industriale, dar instrucțiunile depind unele de celalalte!*

- Pentru a implementa RISC-V în bandă de asamblare:
- Întâi construim RISC-V fără pipeline cu  $CPI=1$
- Pe urmă, adaugăm registre pentru pipeline pentru a reduce timpul de execuție, menținând  $CPI=1$

# Cursul anterior: implementare fără bandă de asamblare pentru RISC-V



# Cursul anterior: Tabela de semnale pentru unitatea de control a RISC-V

Opcode	ImmSel	Op2Sel	FuncSel	MemWr	RFWen	WBSel	WASel	PCSel
ALU	*	Reg	Func	no	yes	ALU	rd	pc+4
ALUi	IType <sub>12</sub>	Imm	Op	no	yes	ALU	rd	pc+4
LW	IType <sub>12</sub>	Imm	+	no	yes	Mem	rd	pc+4
SW	BsType <sub>12</sub>	Imm	+	yes	no	*	*	pc+4
BEQ <sub>true</sub>	BrType <sub>12</sub>	*	*	no	no	*	*	br
BEQ <sub>false</sub>	BrType <sub>12</sub>	*	*	no	no	*	*	pc+4
J	*	*	*	no	no	*	*	jabs
JAL	*	*	*	no	yes	PC	X1	jabs
JALR	*	*	*	no	yes	PC	rd	rind

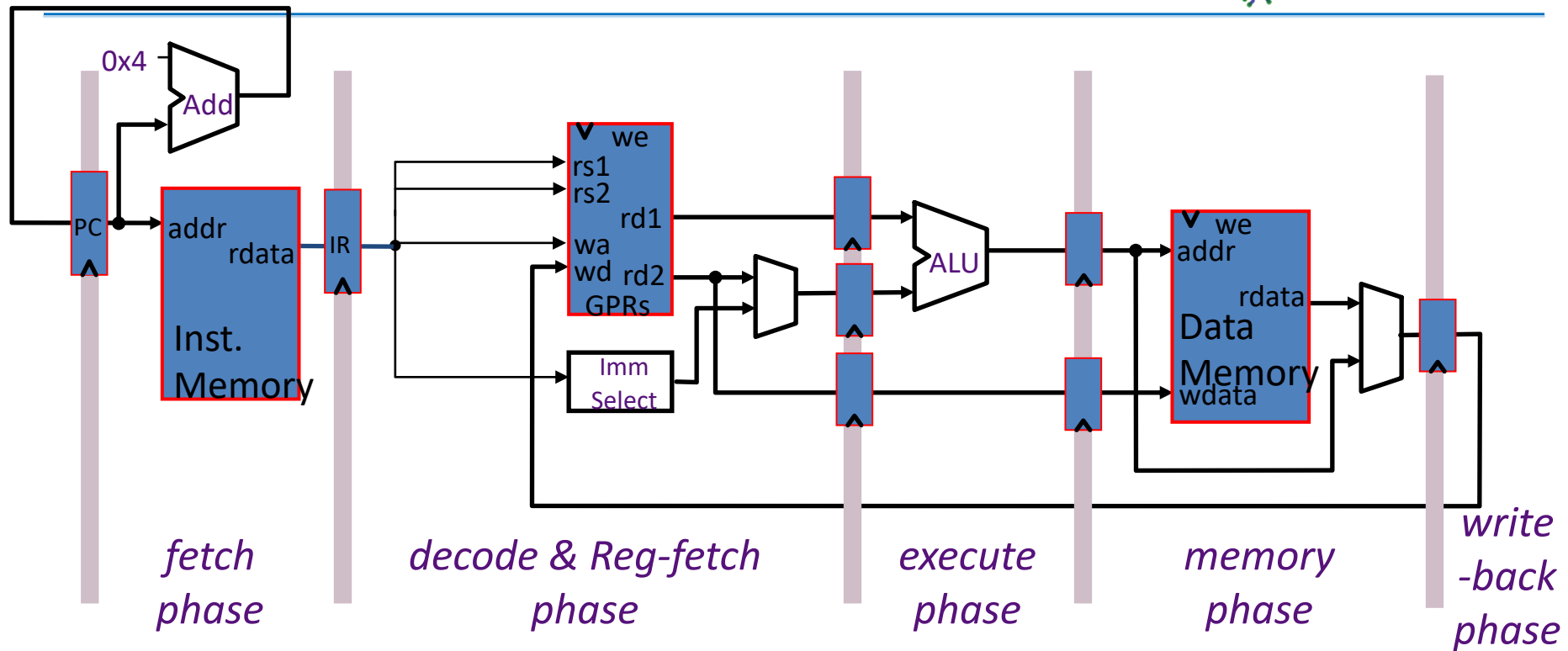
Op2Sel= Reg / Imm  
WASel = rd / X1

WBSel = ALU / Mem / PC  
PCSel = pc+4 / br / rind / jabs





# Magistrală în bandă de asamblare



Perioada ceasului poate fi redusă prin împărțirea execuției unei instrucțiuni în mai multe etape

$$t_C > \max \{t_{IM}, t_{RF}, t_{ALU}, t_{DM}, t_{RW}\} (= t_{DM} \text{ probabil})$$

*Cu toate acestea, CPI va crește dacă instrucțiunile nu sunt executate în pipeline*

# “Legea de fier” a performanței procesoarelor

$$\frac{\text{Time Program}}{\text{Program}} = \frac{\text{Instructions Program}}{\text{Program}} * \frac{\text{Cycles Instruction}}{\text{Instruction}} * \frac{\text{Time Cycle}}{\text{Cycle}}$$

- Instructions per program depinde de codul sursă, compilator și ISA
- Cycles per instructions (CPI) depinde de ISA și  $\mu$ arhitectură
- Time per cycle depinde de  $\mu$ arhitectură și tehnologia de bază în care e implementat procesorul

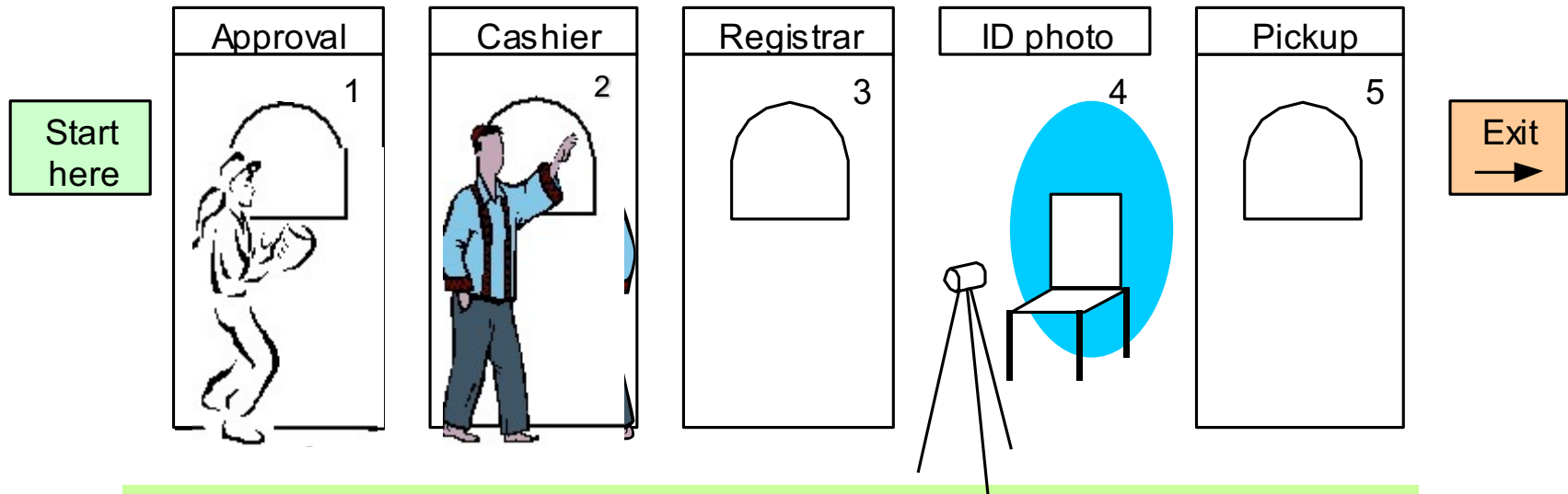
Microarhitectură	CPI	cycle time
Microcodificată	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

# Concepte de pipelining

Strategii pentru îmbunătățirea performanței

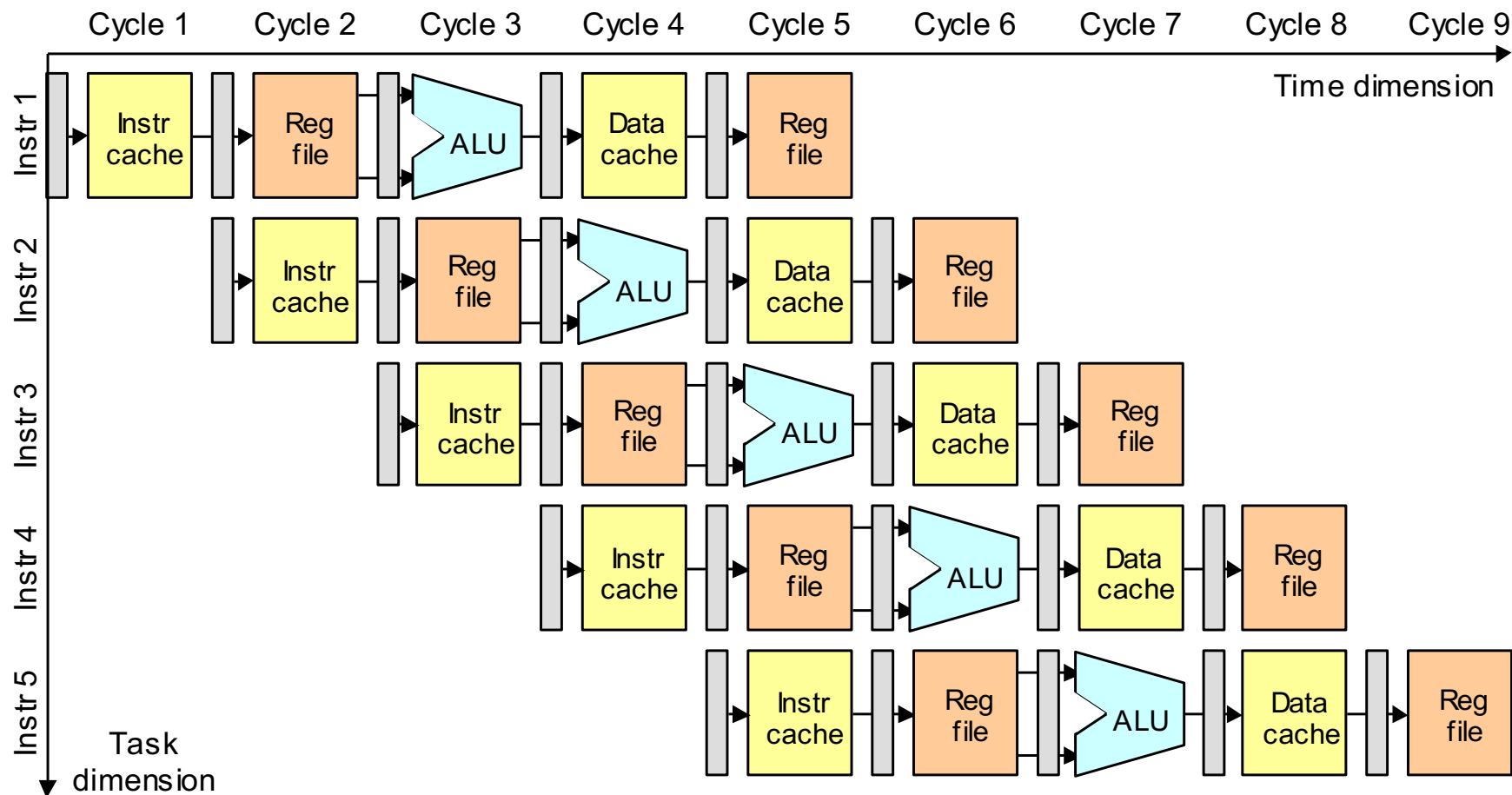
1 – Folosim căi de date independente care acceptă citirea mai multor instrucțiuni în paralel: *multiple-instruction-issue* sau *superscalar*

2 – Suprapunem execuția diferitelor instrucțiuni prin pornirea următoarei instrucțiuni înainte de terminarea celei în curs de execuție: *(super)pipelined*



Banda de asamblare aplicată unui proces de înregistrare persoane

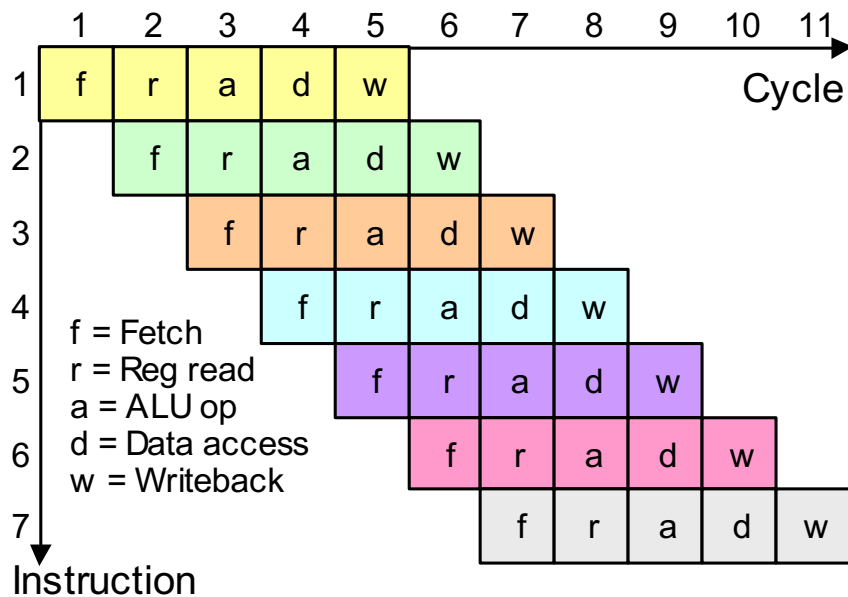
# Execuția instrucțiunilor în banda de asamblare



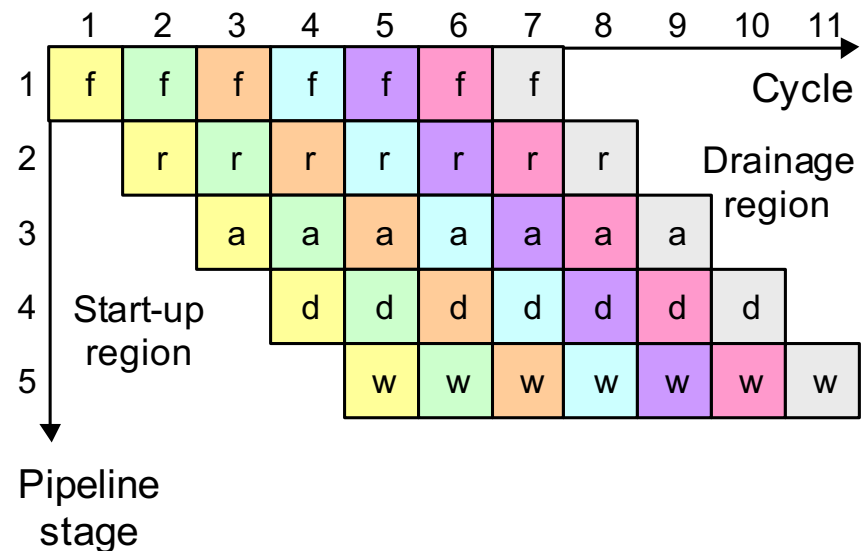
Exemplu de execuție în b.a. pentru un procesor general

# Reprezentări alternative ale benzii de asamblare

O bandă de asamblare poate să execute de obicei o instrucțiune pe front de ceas. IPS este dictată de frecvența ceasului sistemului.



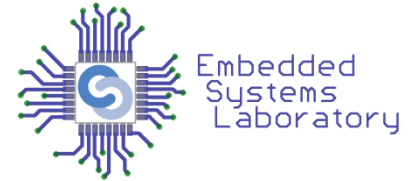
(a) Task-time diagram



(b) Space-time diagram

Două reprezentări grafice abstracte a unei benzi de asamblare în cinci etape ce execută șapte instrucțiuni.

# Exemplu de b.a. într-un fotocopiator



Un fotocopiator cu o tavă de  $x$  foi de hârtie copiază prima foaie în 4s și apoi celelalte foi în câte 1s. Calea pentru copiator este o b.a. în 4 etape, în care fiecare etapă are o latență de 1s. Prima pagină trece prin toate cele 4 etape ale b.a. Și iese după 4s. Fiecare pagină următoare iese la 1s după anterioară. Cum variază productivitatea copiatorului în funcție de  $x$ , presupunând că încărcarea și descărcarea copiatorului durează 15s.

## Soluție

Fiecare teanc de  $x$  foi este copiat în  $15 + 4 + (x - 1) = 18 + x$  secunde.

Un copiator fără b.a. necesită  $4x$  secunde ca să copieze  $x$  foi.

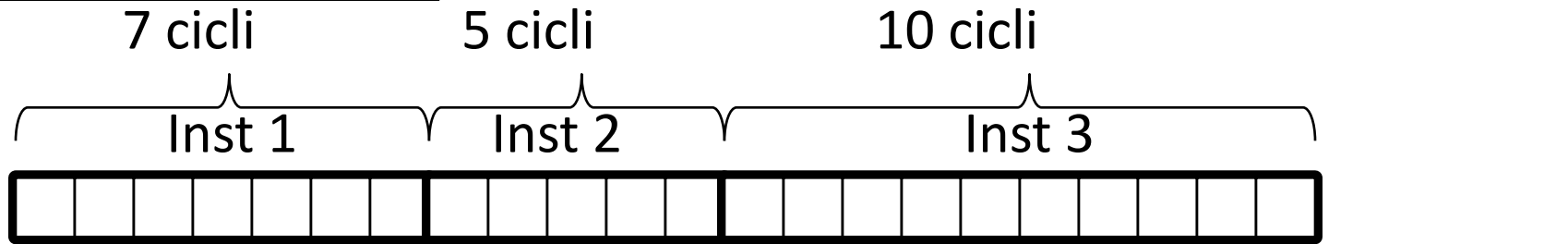
Pentru  $x > 6$ , versiunea cu b.a. are o performanță mai bună.

Dacă  $x = 50$ , îmbunătățirea de performanță este

$$(4 \times 50) / (18 + 50) = 2.94.$$

# Exemple CPI

## Mașină cu microcod



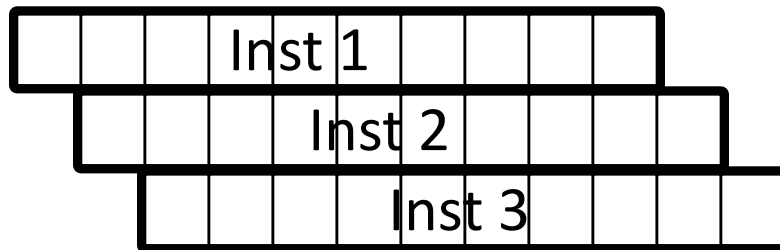
3 instrucțiuni, 22 cicli,  $CPI=7.33$

## Mașină fără pipeline



3 instrucțiuni, 3 cicli,  $CPI=1$

## Mașină cu pipeline



3 instrucțiuni, 3 cicli,  $CPI=1$

**5 niveluri de pipeline  $CPI=5!!!$**

- Memorie mică dar foarte rapidă (cache) susținută de o memorie lentă dar semnificativ mai mare
- UAL rapid (cel puțin pentru calcul pe întregi)
- Tabele de registre multiport (mai lente!)

Prin urmare, următoarea presupunere este rezonabilă

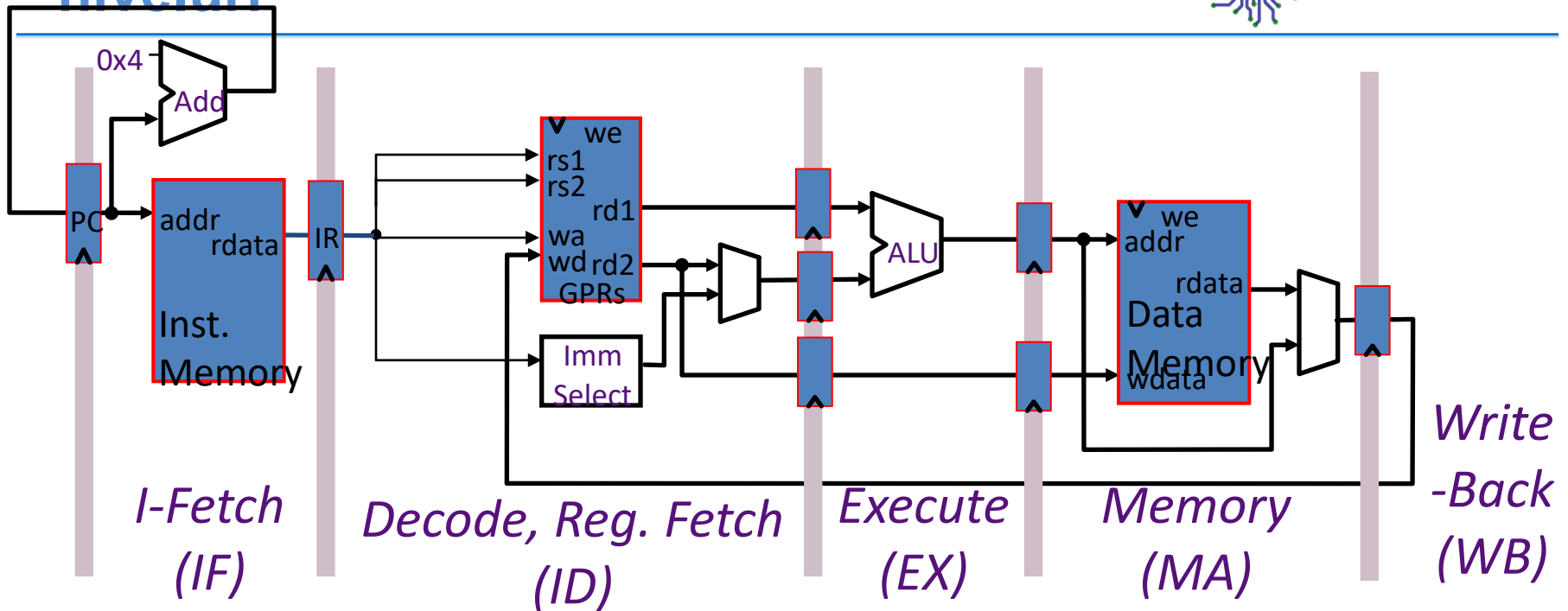
$$t_{IM} \approx t_{RF} \approx t_{ALU} \approx t_{DM} \approx t_{RW}$$

Vom proiecta o bandă de asamblare în 5 etape

*- unele design-uri comerciale de procesor au peste 30 de etape pentru a face o singură adunare de întregi!*

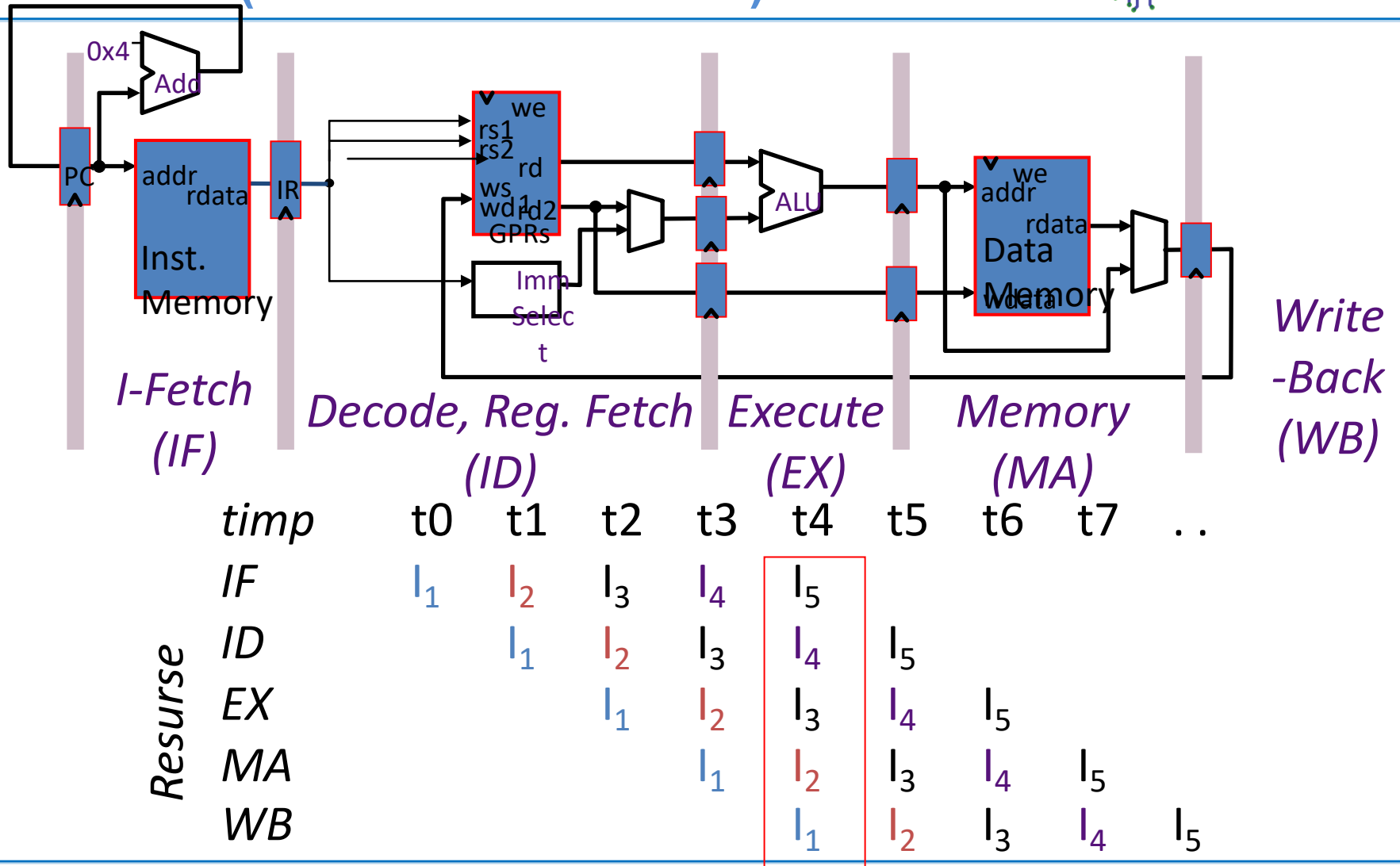


# Execuție în bandă de asamblare cu 5 niveluri

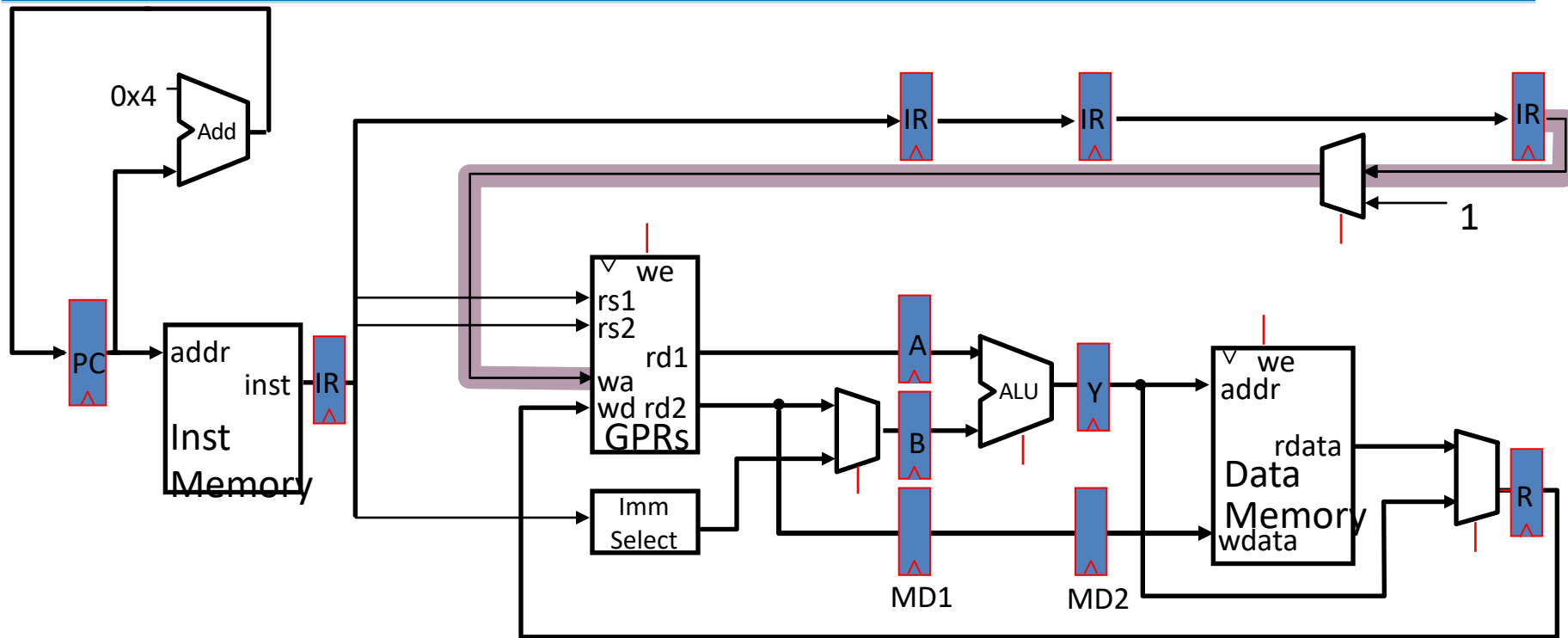


<i>timp</i>	t0	t1	t2	t3	t4	t5	t6	t7	....
instruction1	IF <sub>1</sub>	ID <sub>1</sub>	EX <sub>1</sub>	MA <sub>1</sub>	WB <sub>1</sub>				
instruction2		IF <sub>2</sub>	ID <sub>2</sub>	EX <sub>2</sub>	MA <sub>2</sub>	WB <sub>2</sub>			
instruction3			IF <sub>3</sub>	ID <sub>3</sub>	EX <sub>3</sub>	MA <sub>3</sub>	WB <sub>3</sub>		
instruction4				IF <sub>4</sub>	ID <sub>4</sub>	EX <sub>4</sub>	MA <sub>4</sub>	WB <sub>4</sub>	
instruction5					IF <sub>5</sub>	ID <sub>5</sub>	EX <sub>5</sub>	MA <sub>5</sub>	WB <sub>5</sub>

# Execuție în bandă de asamblare cu 5 niveluri (folosirea resurselor)



# Execuție în pipeline: Instrucțiuni UAL

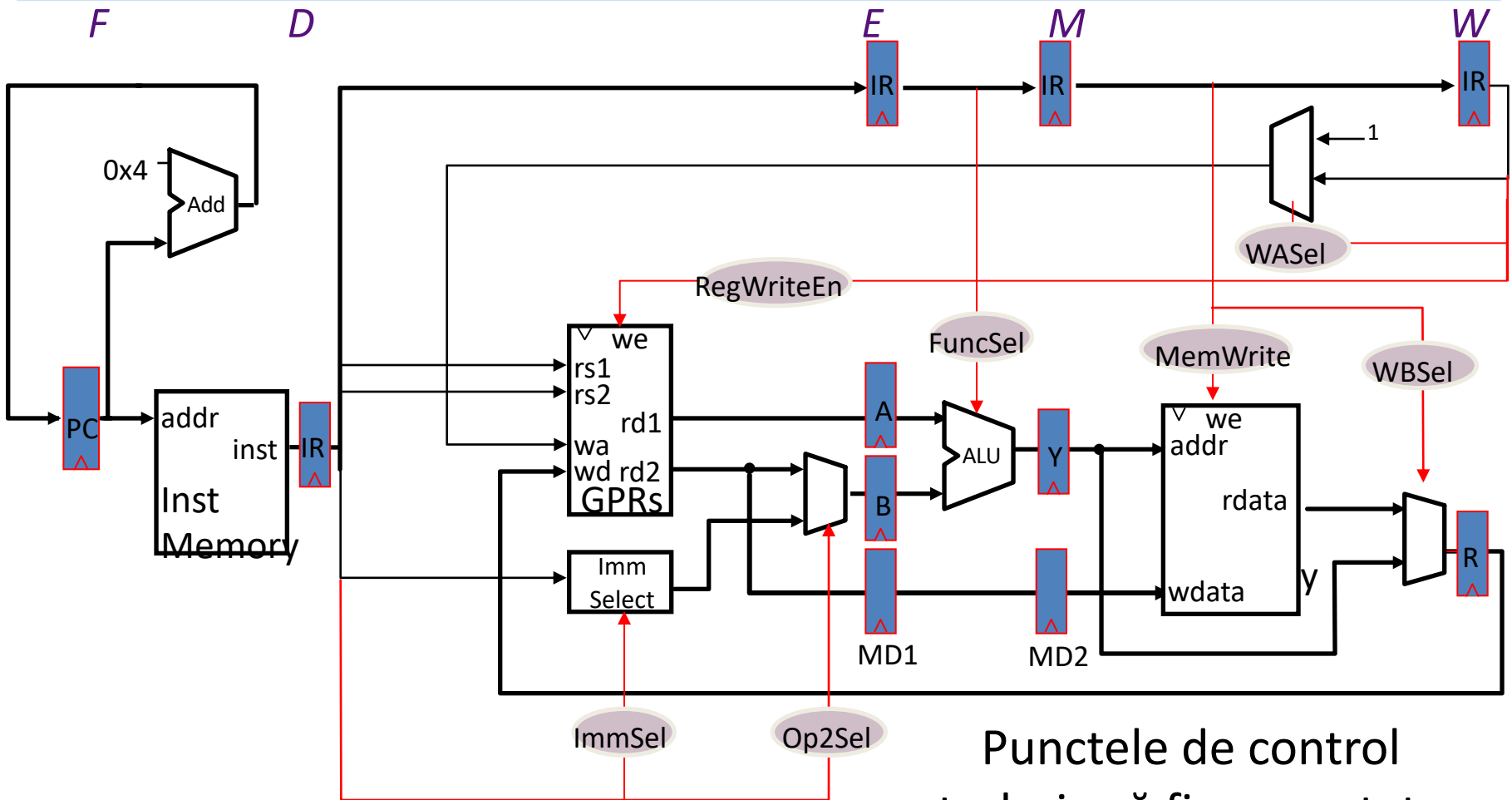
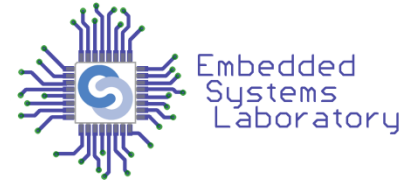


*Nu e foarte corect!*

*Avem nevoie de un Instruction Reg (IR) pentru fiecare etapă*

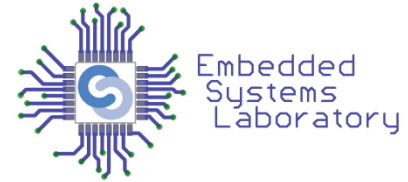
# Magistrala în b.a. Pentru RISC-V

fără instrucțiuni de jump



Punctele de control  
trebuie să fie conectate

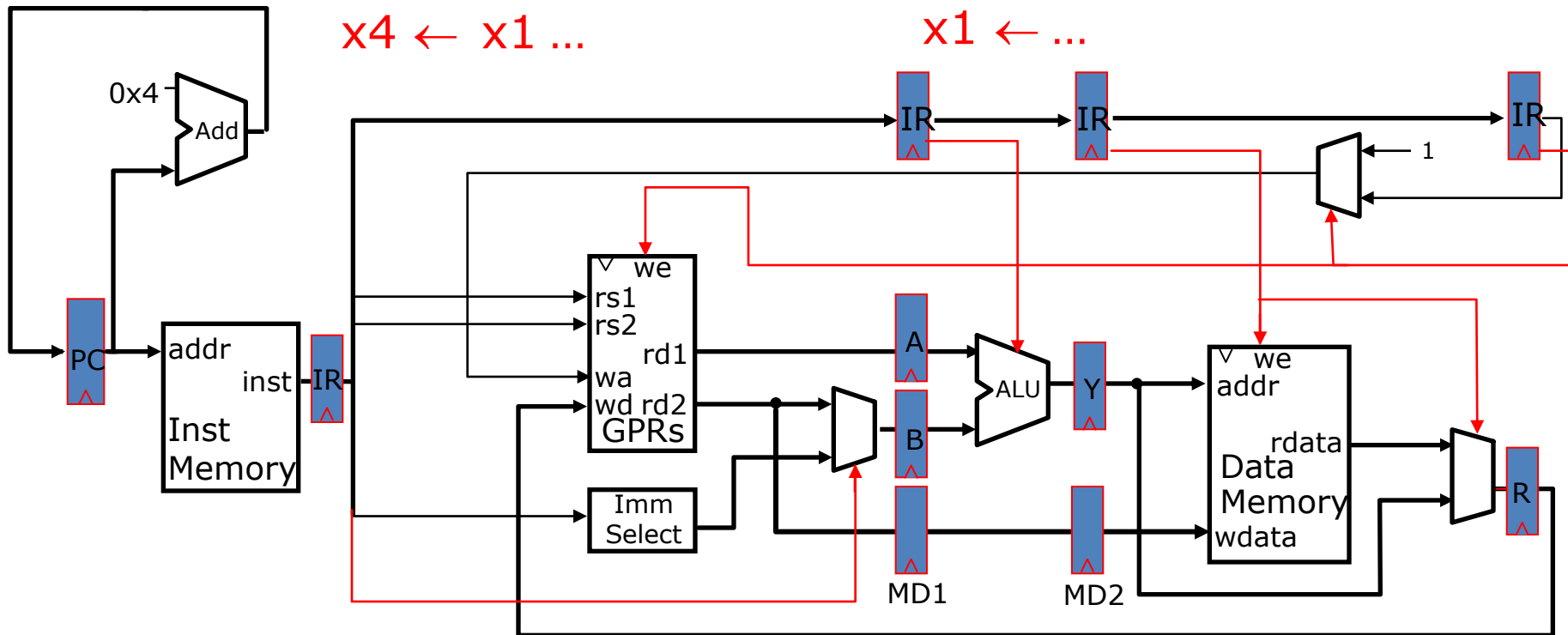
# Instrucțiunile interacționează unele cu celalalte în pipeline



- O instrucțiune din b.a. Poate avea nevoie de o resursă folosită de altă instrucțiune din b.a.  
→ *Hazard structural*
- O instrucțiune poate să depindă de rezultatul unei instrucțiuni anterioare
  - Dependența poate să fie pentru valoarea unei variabile  
→ *hazard de date*
  - Dependența poate să fie pentru adresa următoarei instrucțiuni  
→ *hazard de control (branch-uri, excepții)*

- Hazardele structurale apar atunci când două instrucțiuni necesită aceeași resursă hardware în același timp
  - Poate fi rezolvat în hardware prin oprirea execuției noii instrucțiuni până când instrucțiunea mai veche a eliberat resursa folosită
- Un hazard hardware structural poate fi întotdeauna evitat prin adăugarea de mai mult hardware în design
  - E.g., dacă două instrucțiuni au nevoie de acces la memorie în același timp, se poate soluționa prin folosirea unei memorii bi- sau multi-port
- Implementarea noastră în 5 etape nu are hazarde structurale de la bun început
  - Datorită RISC-V ISA, care a fost proiectată pentru execuția în b.a.

# Hazarduri de date



$x4 \leftarrow x1 \dots$

$x1 \leftarrow \dots$

...

$x1 \leftarrow x0 + 10$

$x4 \leftarrow x1 + 17$

...

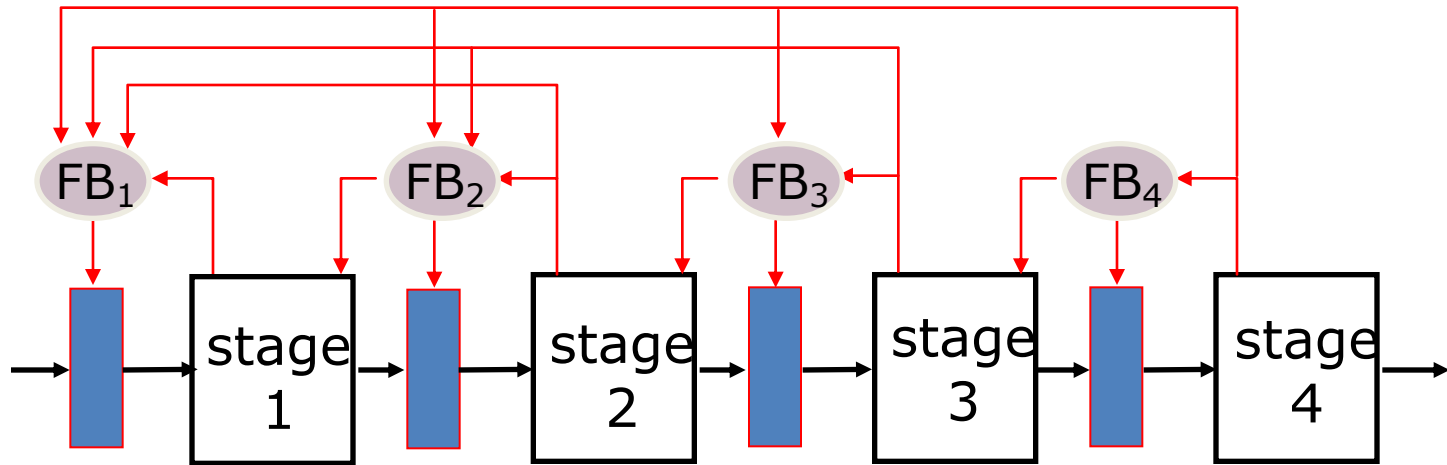
*$x1$  este învechit. Oops!*

## *Strategia 1:*

*Aștept ca rezultatul să fie disponibil prin  
"înghețarea" tuturor etapelor anterioare ale b.a.  
→ **interlocking***



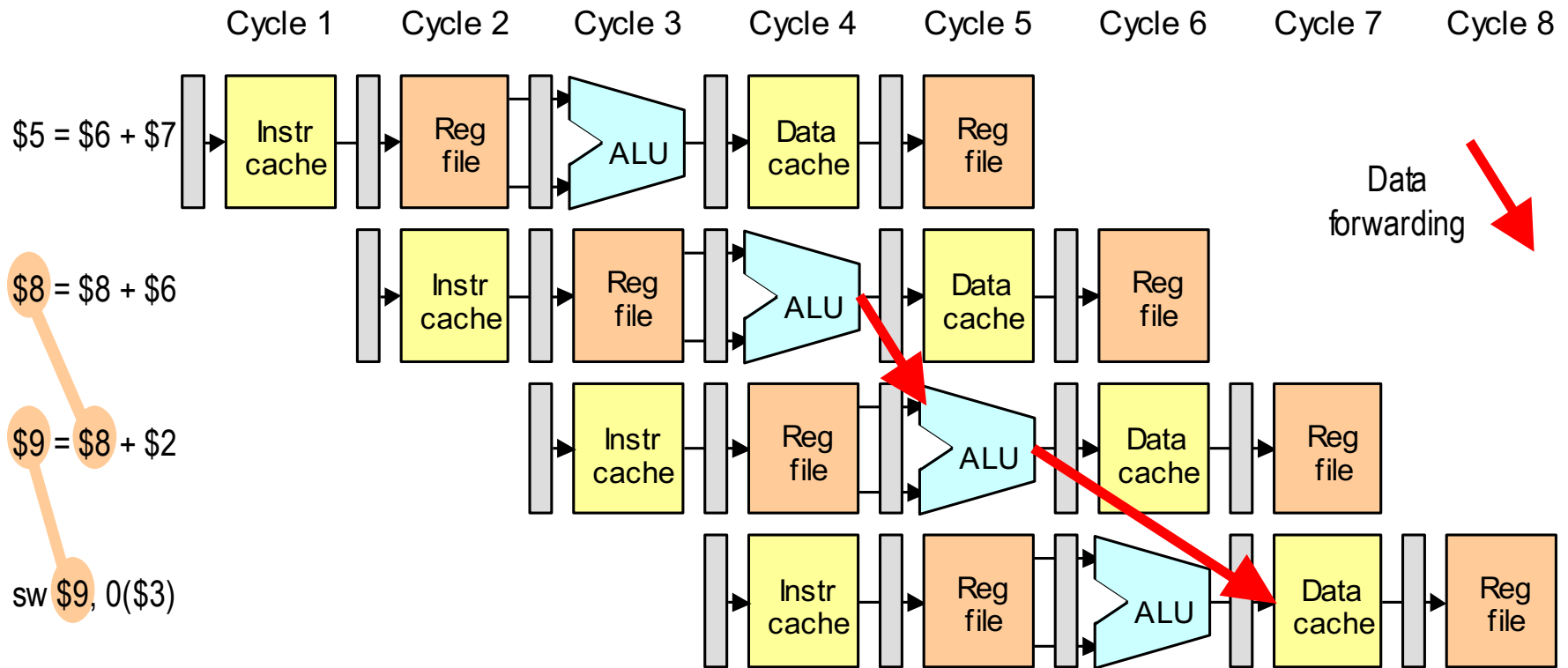
# Feedback pentru soluționarea hazardurilor



- Etapele superioare furnizează informații legate de dependență stagiilor inferioare, care pot încetini sau opri instrucțiunile
- Controlul unei b.a. în acest fel funcționează dacă instrucțiunea de la nivelul  $i+1$  poate să completeze execuția fără nici o interferență cu instrucțiunile din nivelele  $1..i$  (altfel pot apare deadlock-uri)

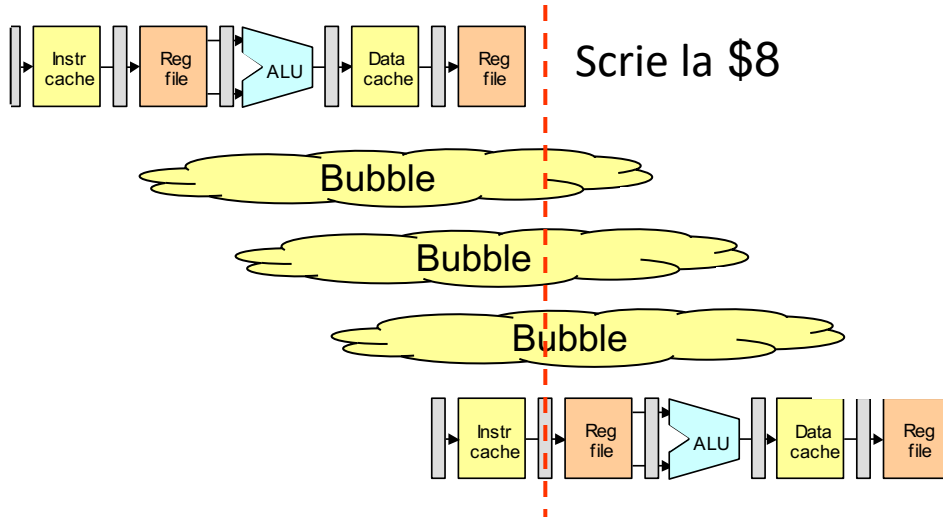
# Hazarduri în b.a. Stalls & Bubbles

Generate din cauza dependențelor de date

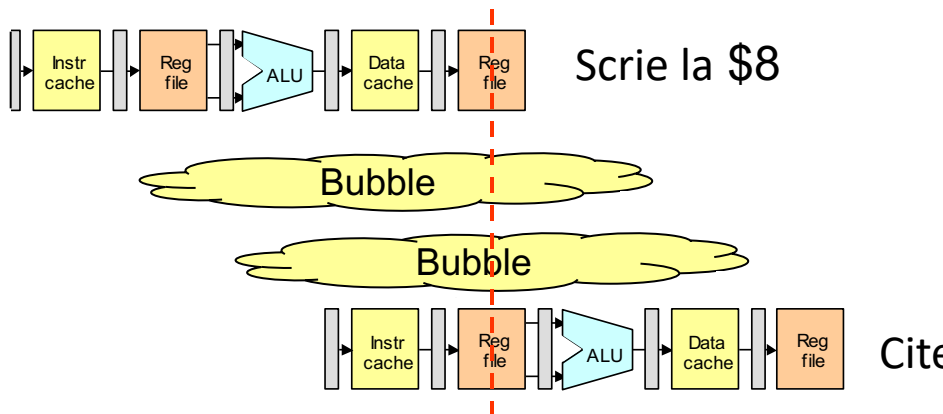


Dependență Read-After-Write și soluționarea ei prin data forwarding

# Inserarea de bule într-o b.a.

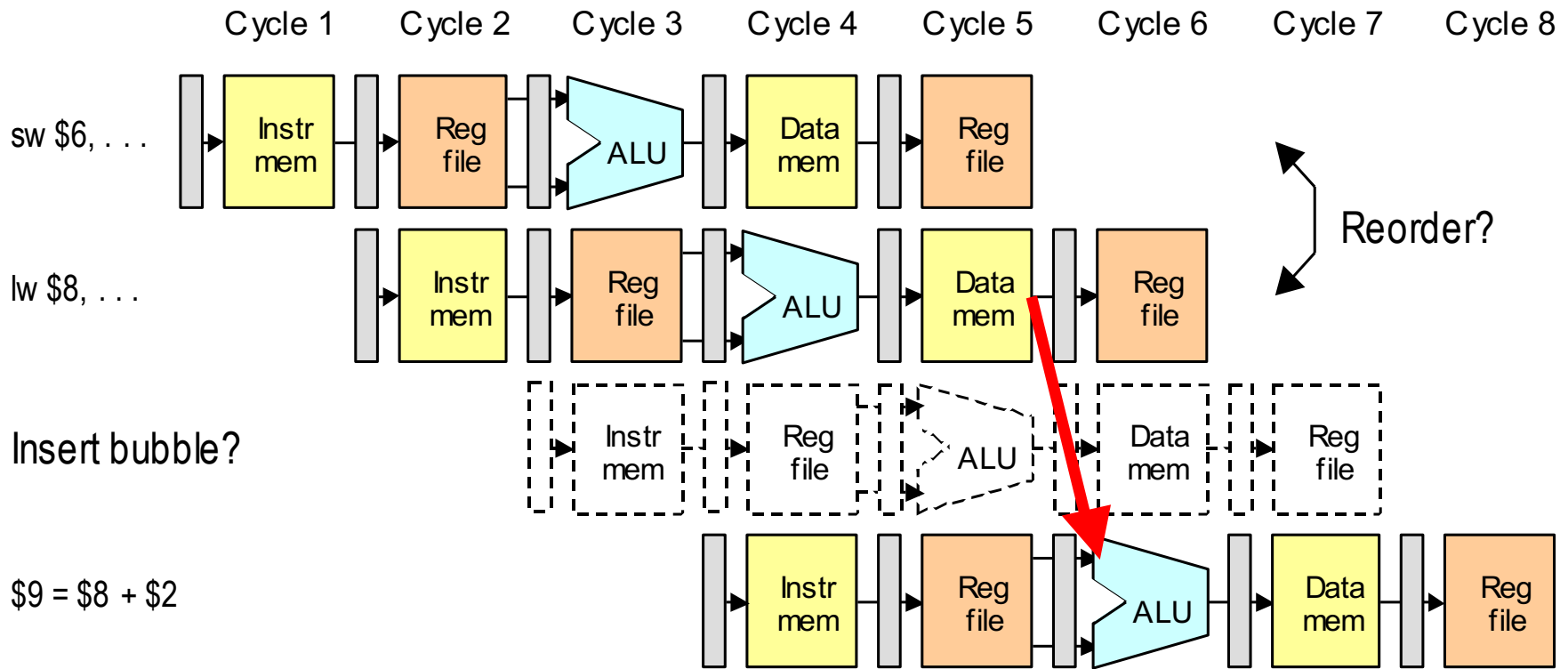


Fără data forwarding avem nevoie de aceste trei “bule” pentru a soluționa dependența read-after-write



Doar două bule dacă presupunem că un registru poate fi scris și citit într-un singur ciclu de ceas.

# Al doilea tip de dependență de date

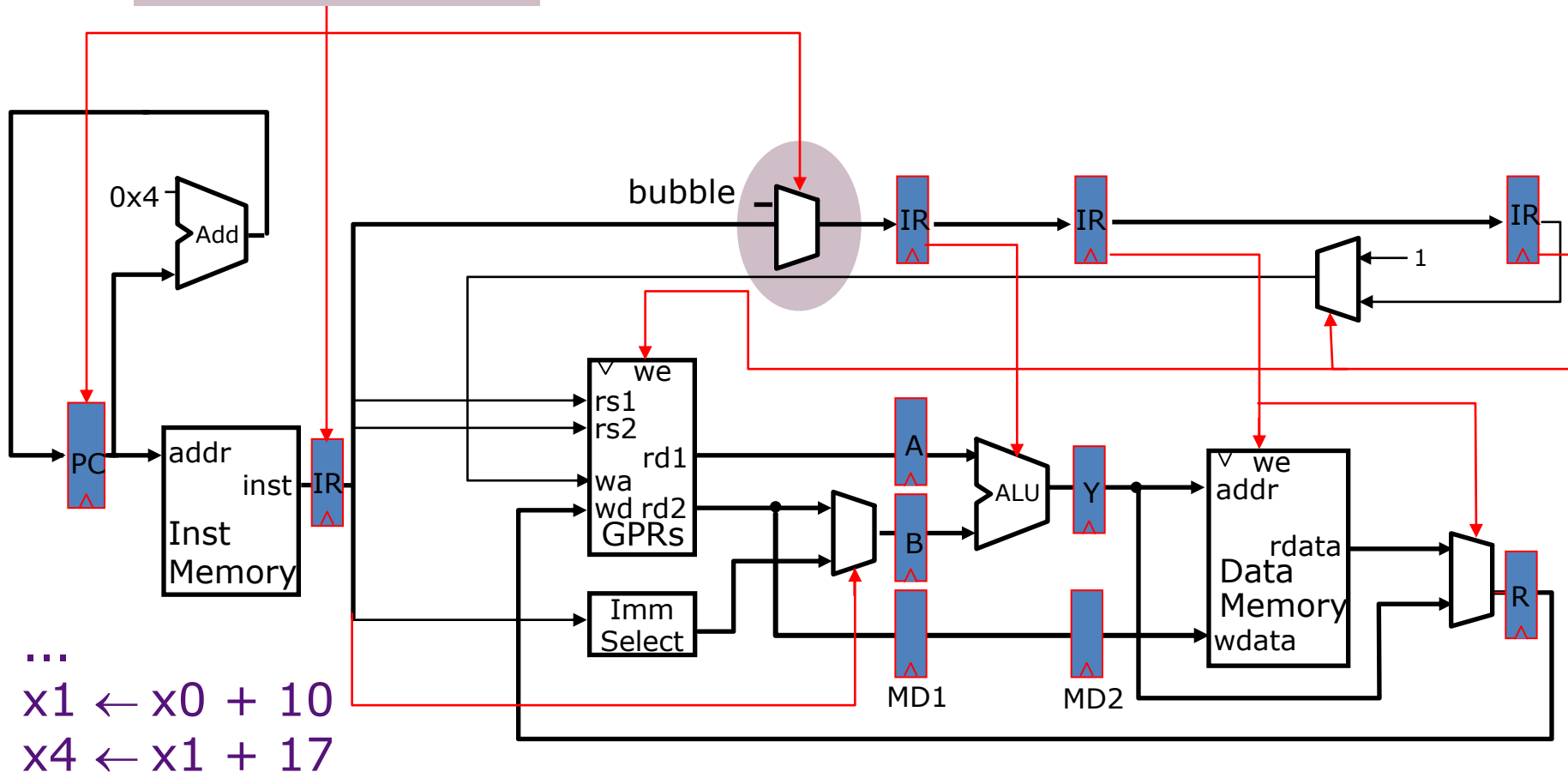


Fără data forwarding, este nevoie de trei (două) bule pentru a rezolva un conflict read-after-load.

Dependența Read-After-Load și soluționarea ei prin inserarea de bule și data forwarding

# Interlocking pentru a rezolva hazardurile de date

*Condiție de stall*



# Stalled Stages & Pipeline Bubbles

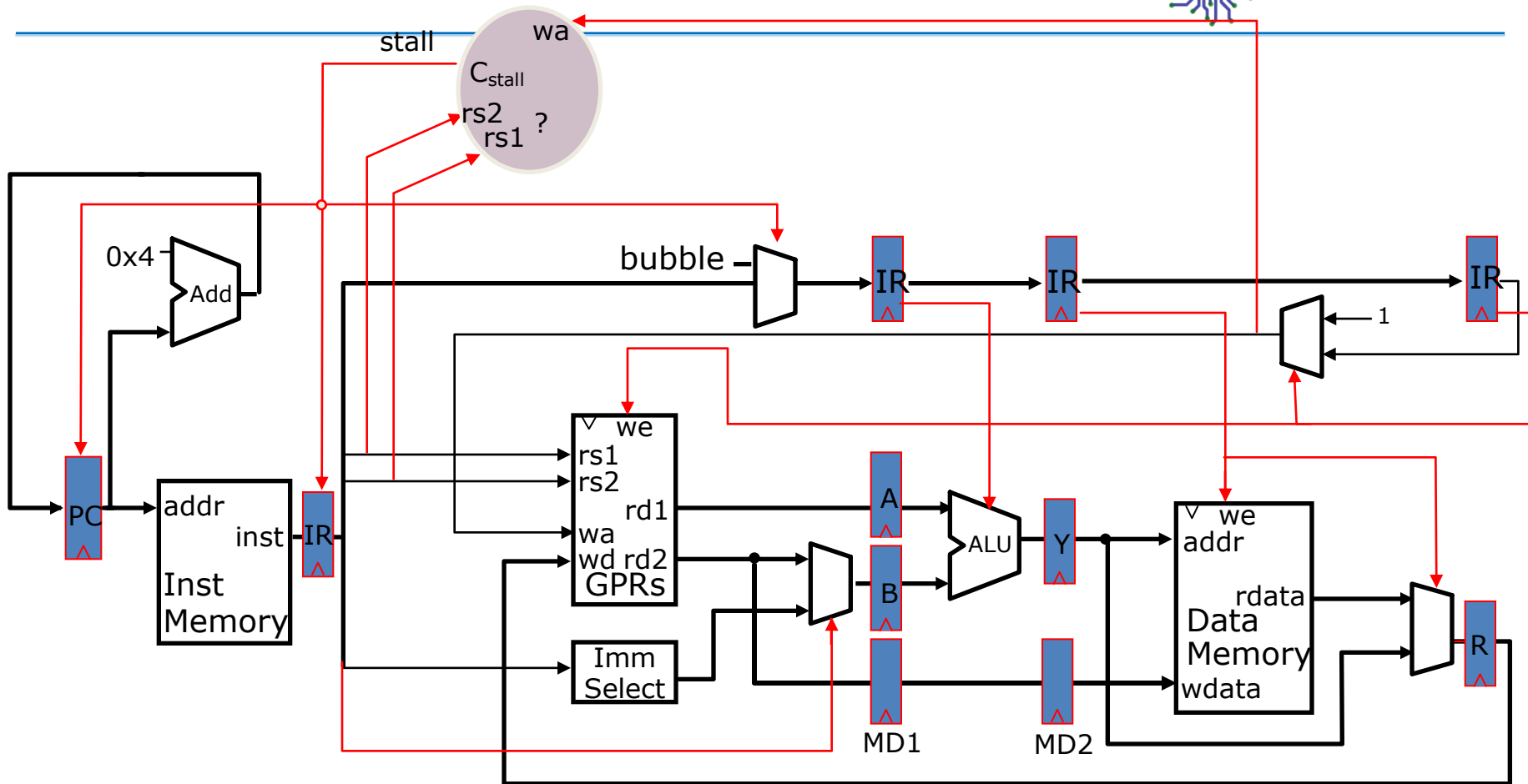
	<i>time</i>											
	t0	t1	t2	t3	t4	t5	t6	t7	...	...		
(I <sub>1</sub> ) $x1 \leftarrow (x0) + 10$	IF <sub>1</sub>	ID <sub>1</sub>	EX <sub>1</sub>	MA <sub>1</sub>	WB <sub>1</sub>							
(I <sub>2</sub> ) $x4 \leftarrow (x1) + 17$		IF <sub>2</sub>	ID <sub>2</sub>	ID <sub>2</sub>	ID <sub>2</sub>	ID <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>	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>	ID <sub>4</sub>	EX <sub>4</sub>	MA <sub>4</sub>	WB <sub>4</sub>	
(I <sub>5</sub> )								IF <sub>5</sub>	ID <sub>5</sub>	EX <sub>5</sub>	MA <sub>5</sub>	WB <sub>5</sub>

*stalled stages*

	<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>3</sub>	I <sub>3</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>				
ID		I <sub>1</sub>	I <sub>2</sub>	I <sub>2</sub>	I <sub>2</sub>	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>			
EX			I <sub>1</sub>	-	-	-	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>		
MA				I <sub>1</sub>	-	-	-	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>	
WB					I <sub>1</sub>	-	-	-	I <sub>2</sub>	I <sub>3</sub>	I <sub>4</sub>	I <sub>5</sub>

- ⇒ *pipeline bubble*

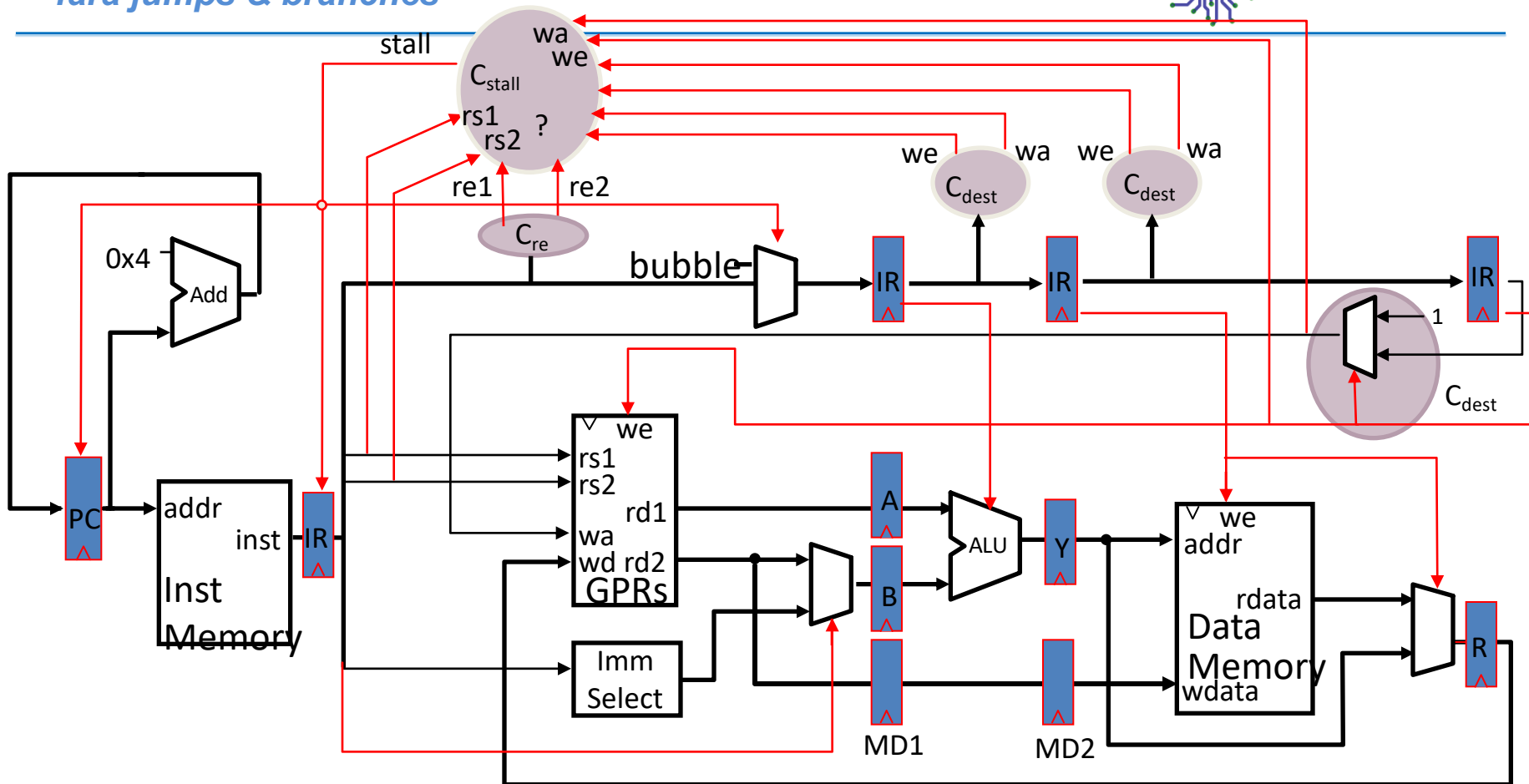
# Logica de control pentru interlocking



Compară registrele sursă ale instrucțiunii din faza de *decode* cu registrul destinație al instrucțiunilor în *curs de execuție*

# Logica de control pentru interlocking

fără jumps & branches



Trebuie să stagnăm întotdeauna când adresa unui rs este egală cu a unui rd?  
nu orice instrucțiune scrie un registru -> we  
nu orice instrucțiune citește un registru -> re



# Registrele sursă și destinație

rd	rs1	rs2	func10	opcode	ALU
rd	rs1	Imm[11:0]	func3	opcode	ALUI/LW/JALR
Imm[11:7]	rs1	rs2	Imm[6:0]	func3	SW/Bcond
Jump offset[24:0]				opcode	

*source(s)*

*destination*

ALU	rd <- rs1 func10 rs2	rs1, rs2	rd
ALUI	rd <- rs1 op imm	rs1	rd
LW	rd <-M [rs1 + imm]	rs1	rd
SW	M [rs1 + imm] <- rs2	rs1, rs2	-
Bcond	rs1,rs2	rs1, rs2	-
	<i>true:</i> PC <- PC + imm		
	<i>false:</i> PC <- PC + 4		
J	PC <- PC + imm	-	-
JAL	x1 <- PC, PC <- PC + imm	-	x1
JALR	rd <- PC, PC <- rs1 + imm	rs1	rd

# Determinarea semnalului de stall pentru b.a.

$C_{dest}$

$ws = Case\ opcode$

JAL  $\rightarrow X1$

else  $\rightarrow rd$

$we = Case\ opcode$

ALU, ALUi, LW, JALR  $\rightarrow (ws \neq 0)$

JAL  $\rightarrow on$

...  $\rightarrow off$

$C_{re}$

$re1 = Case\ opcode$

ALU, ALUi,

LW, SW, Bcond,

JALR  $\rightarrow on$

J, JAL  $\rightarrow off$

$re2 = Case\ opcode$

ALU, SW, Bcond  $\rightarrow on$

...  $\rightarrow off$

$C_{stall}$

$$\begin{aligned}
 stall = & ((rs1_D = ws_E).we_E + \\
 & (rs1_D = ws_M).we_M + \\
 & (rs1_D = ws_W).we_W) \cdot re1_D + \\
 & ((rs2_D = ws_E).we_E + \\
 & (rs2_D = ws_M).we_M + \\
 & (rs2_D = ws_W).we_W) \cdot re2_D
 \end{aligned}$$

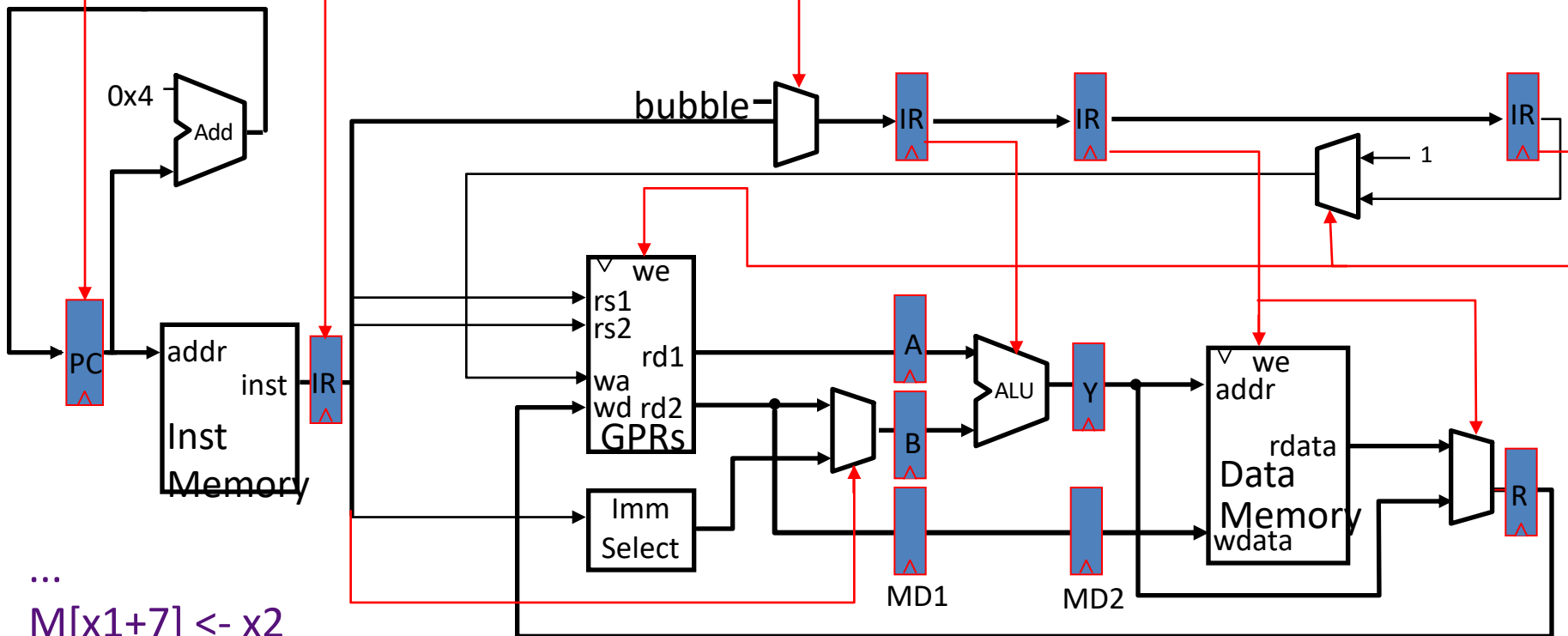
*This is not  
the full story!*

# Hazarduri determinate de instrucțiuni

## Load & Store

Condiție de stall

Dar dacă  
 $x1+7 = x3+5$  ?



...  
 $M[x1+7] \leftarrow x2$   
 $x4 \leftarrow M[x3+5]$   
 ...

*Este posibil vreun hazard de date în cadrul acestei secvențe de instrucțiuni?*

# Hazarde Load & Store

```
...  
M[x1+7] <- x2  
x4 <- M[x3+5]  
...
```

$x1+7 = x3+5 \rightarrow$  *hazard de date*

Cu toate acestea, hazardul este evitat pentru că sistemul de memorie face un write într-un singur ciclu de ceas!

Hazardurile Load/Store sunt câteodată soluționate în pipeline și câteodată în sistemul de memorii.

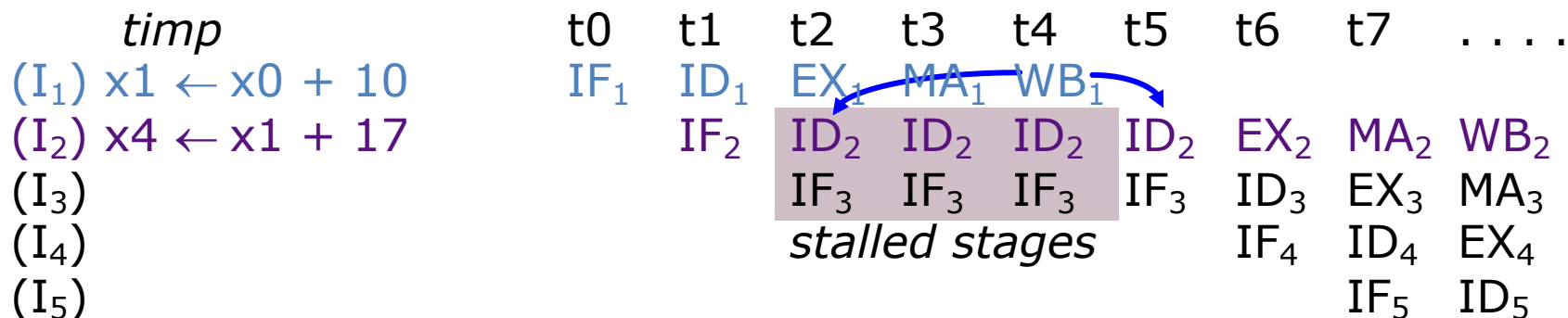
*Vom vorbi mai multe despre asta în curs.*

## Strategia 2:

Rutează datele imediat ce ele sunt calculate către nivelurile inferioare din banda de asamblare

→ *bypass*

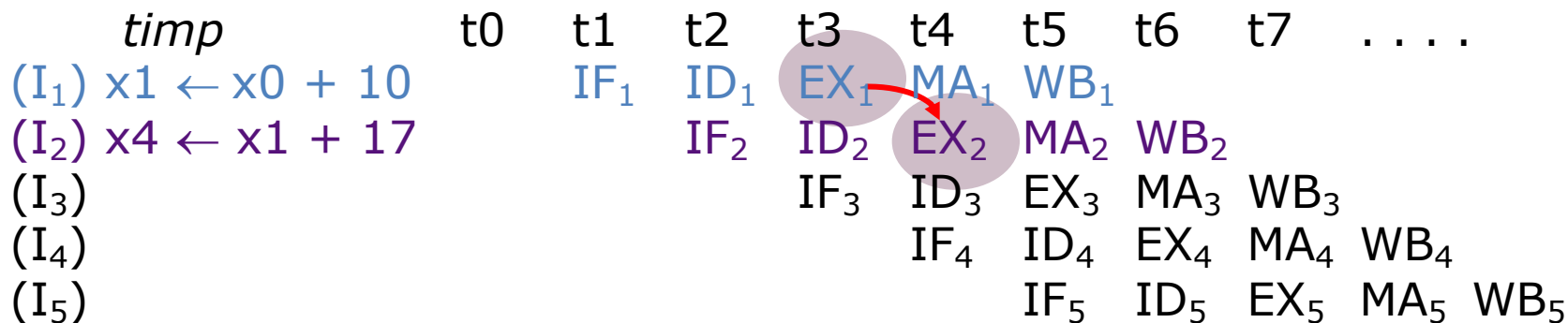
# Bypassing



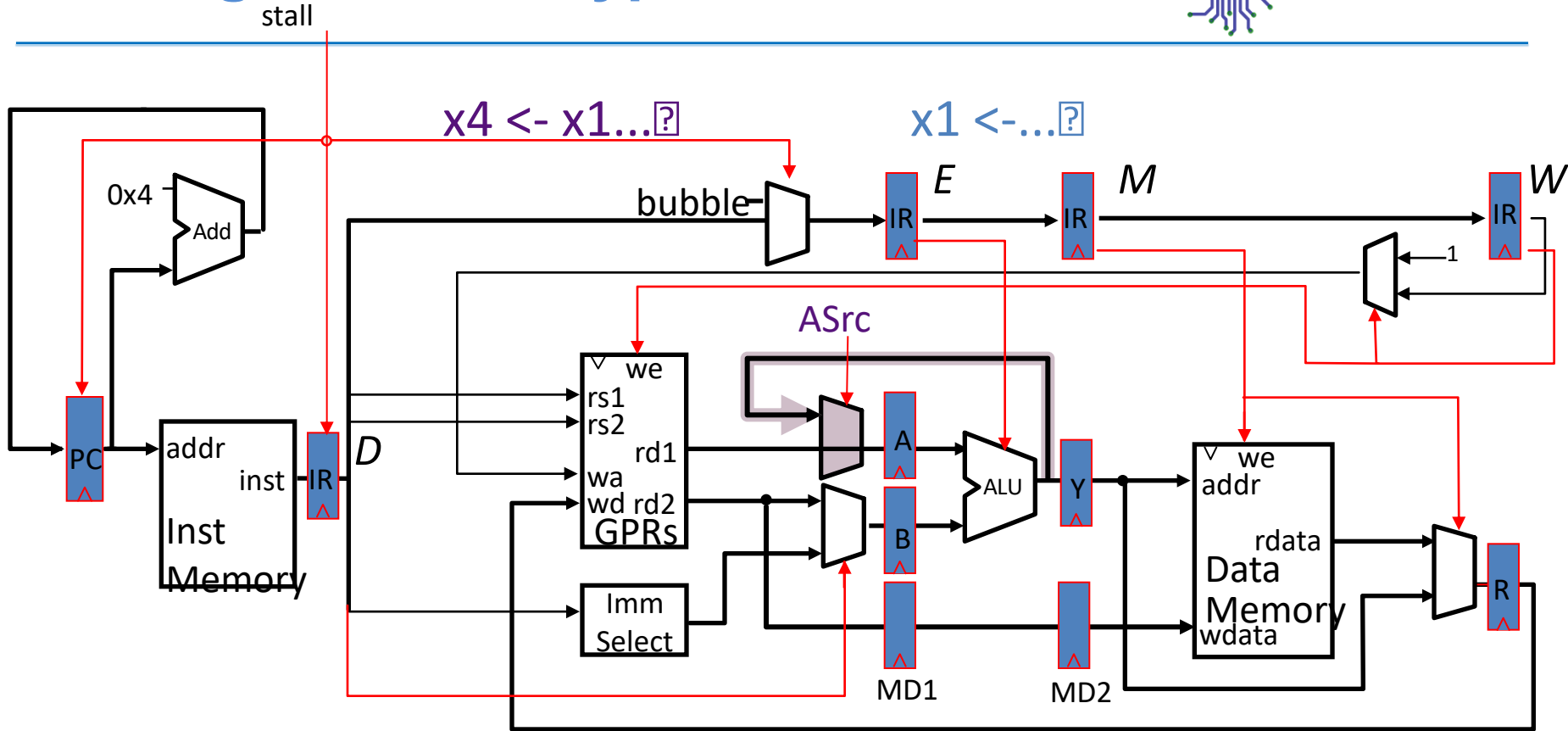
Fiecare *stall sau kill* introduce o bulă în banda de asamblare

$$\Rightarrow CPI > 1$$

O nouă cale de date, i.e., *un bypass*, poate aduce datele de la ieșirea UAL la intrarea acesteia



# Adăugarea unui bypass



Când ajută acest bypass?

...

(I<sub>1</sub>)  $x1 \leftarrow x0 + 10$

(I<sub>2</sub>)  $x4 \leftarrow x1 + 17$

$x1 \leftarrow M[x0 + 10]$

$x4 \leftarrow x1 + 17$

JAL 500

$x4 \leftarrow x1 + 17$

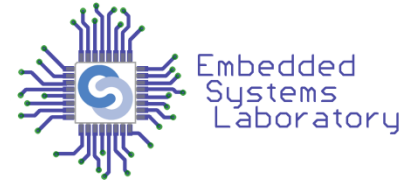
da

nu

nu

# Semnalul de bypass

## Derivarea semnalului din cel pentru Stall



$$\text{stall} = ( (\cancel{rs1_D = ws_E}.we_E + (rs1_D = ws_M).we_M + (rs1_D = ws_W).we_W).re1_D + ((rs2_D = ws_E).we_E + (rs2_D = ws_M).we_M + (rs2_D = ws_W).we_W).re2_D )$$

$ws = \text{Case opcode}$

JAL ->X1

else ->rd

$we = \text{Case opcode}$

ALU, ALUi, LW, JALR ->(ws != 0)

JAL ->on

... ->off

$$ASrc = (rs1_D = ws_E).we_E.re1_D$$

Este corect?

Nu, pentru că doar instrucțiunile tip ALU și ALUi pot beneficia de un astfel de bypass

Împărțim  $we_E$  în două componente: we-bypass, we-stall



# Semnalele Bypass și Stall

Împărțim  $we_E$  în două componente:  $we$ -bypass,  $we$ -stall

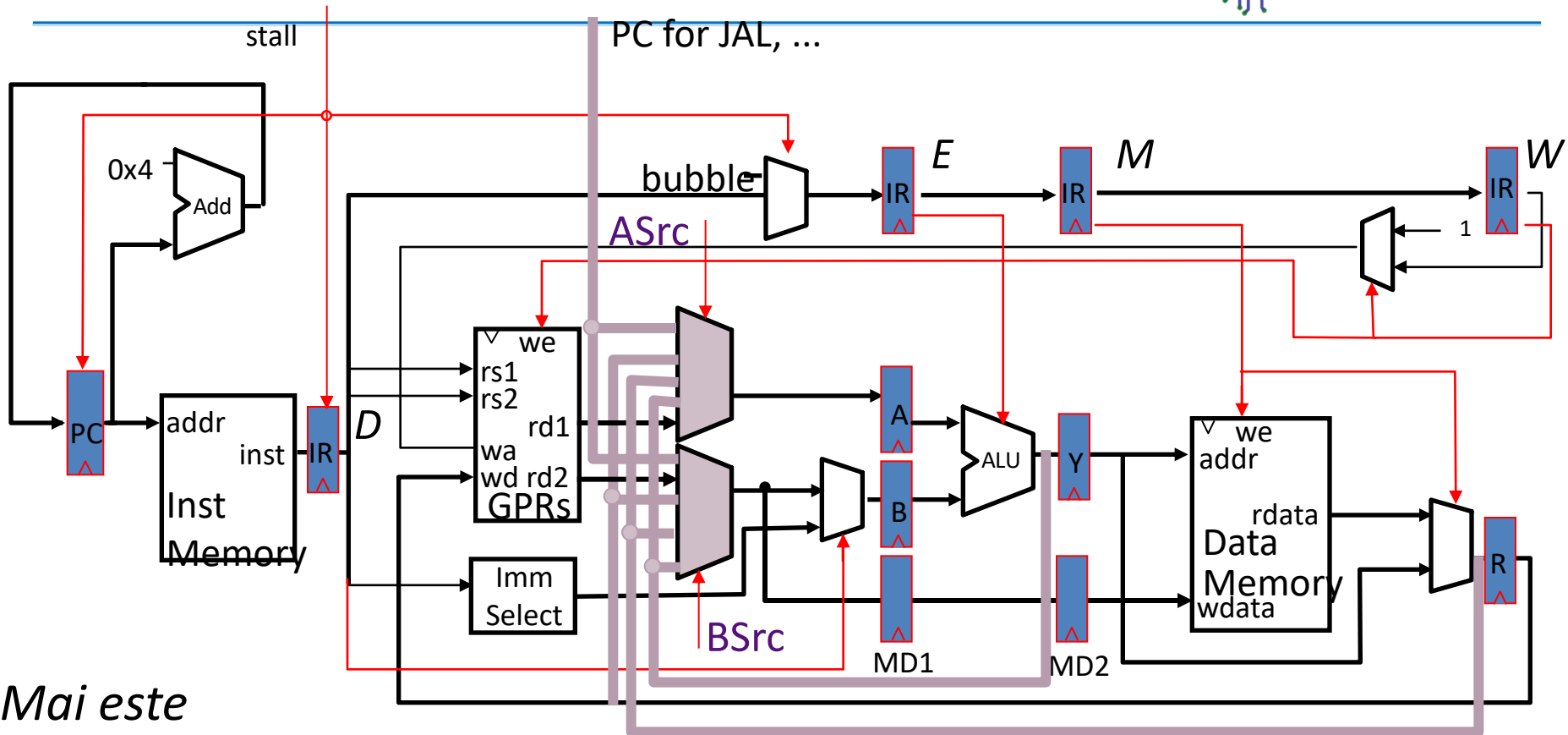
$we$ -bypass<sub>E</sub> = *Case opcode<sub>E</sub>*  
ALU, ALUi → ( $ws \neq 0$ )  
... → off

$we$ -stall<sub>E</sub> = *Case opcode<sub>E</sub>*  
LW, JALR → ( $ws \neq 0$ )  
JAL → on  
... → off

ASrc =  $(rs1_D = ws_E) \cdot we$ -bypass<sub>E</sub> · re1<sub>D</sub>

stall =  $((rs1_D = ws_E) \cdot we$ -stall<sub>E</sub> +  
           $(rs1_D = ws_M) \cdot we_M + (rs1_D = ws_W) \cdot we_W) \cdot re1_D$   
+  $((rs2_D = ws_E) \cdot we_E + (rs2_D = ws_M) \cdot we_M + (rs2_D = ws_W) \cdot we_W) \cdot re2_D$

# Căi de date complet separate

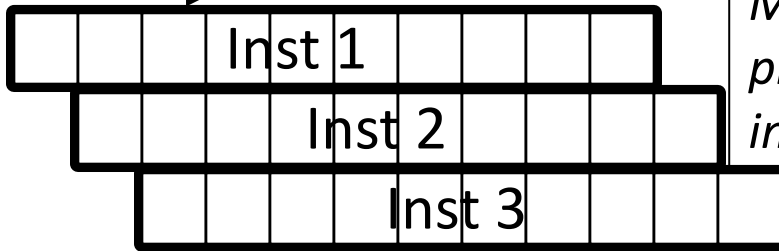


Mai este  
nevoie de un  
semnal de  
stall ?

$$\text{stall} = (rs1_D = ws_E) \cdot (\text{opcode}_E = LW_E) \cdot (ws_E \neq 0) \cdot re1_D + (rs2_D = ws_E) \cdot (\text{opcode}_E = LW_E) \cdot (ws_E \neq 0) \cdot re2_D$$

# Exemple de CPI pentru b.a.

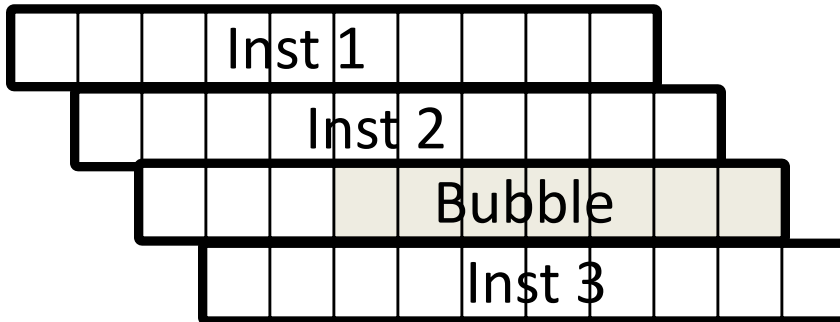
Timp →



*Măsurăm din momentul în care termină prima instrucțiune până când ultima instrucțiune își incheie execuția.*

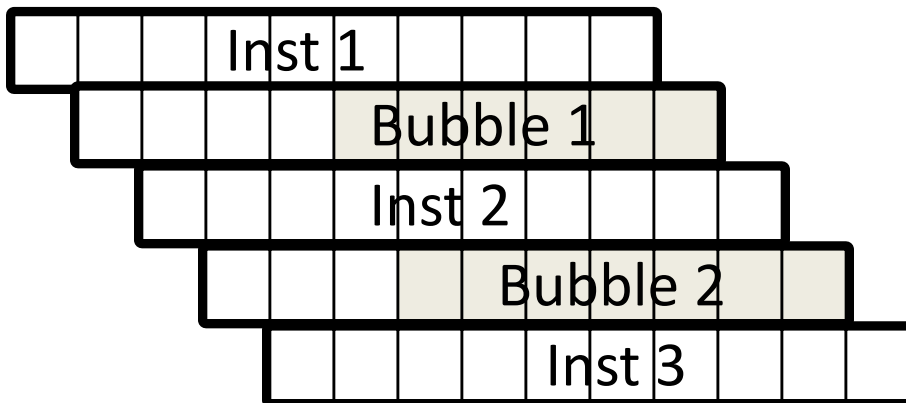
3 instrucțiuni termină în 3 cicluri

$$\text{CPI} = 3/3 = 1$$



3 instrucțiuni termină în 4 cicluri

$$\text{CPI} = 4/3 = 1.33$$



3 instrucțiuni termină în 5 cicluri

$$\text{CPI} = 5/3 = 1.67$$

*Strategia 3: Speculează legat de dependență!*

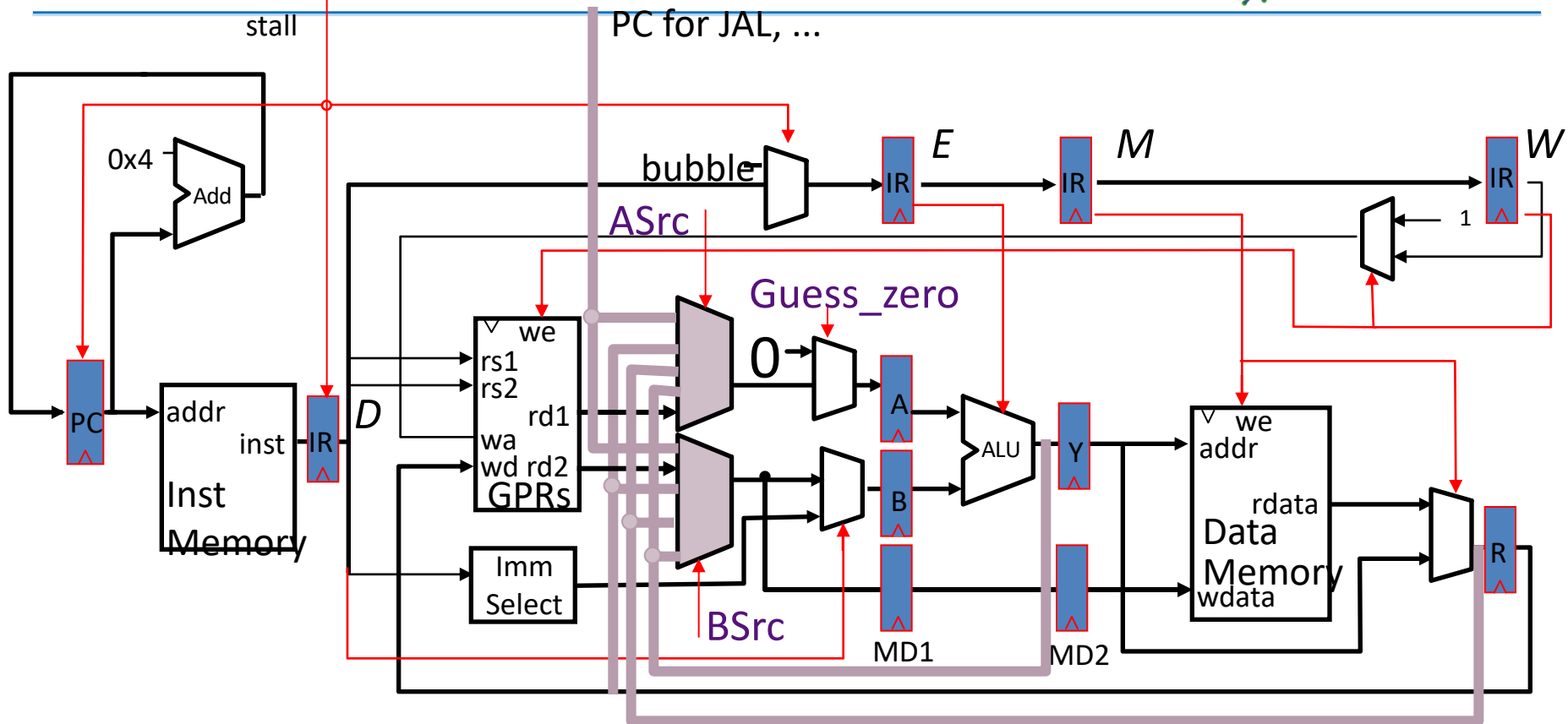
*Două cazuri:*

*Am ghicit corect* → nu face nimic

*Am ghicit incorect* → kill și restart

.... Vom vedea mai târziu exemple de astfel de comportament în procesoare mai complexe.

# Speculăm ca valoarea pentru load=zero



$$\text{Guess\_zero} = (\text{rs1}_D = \text{ws}_E) \cdot (\text{opcode}_E = \text{LW}_E) \cdot (\text{ws}_E \neq 0) \cdot \text{re1}_D$$

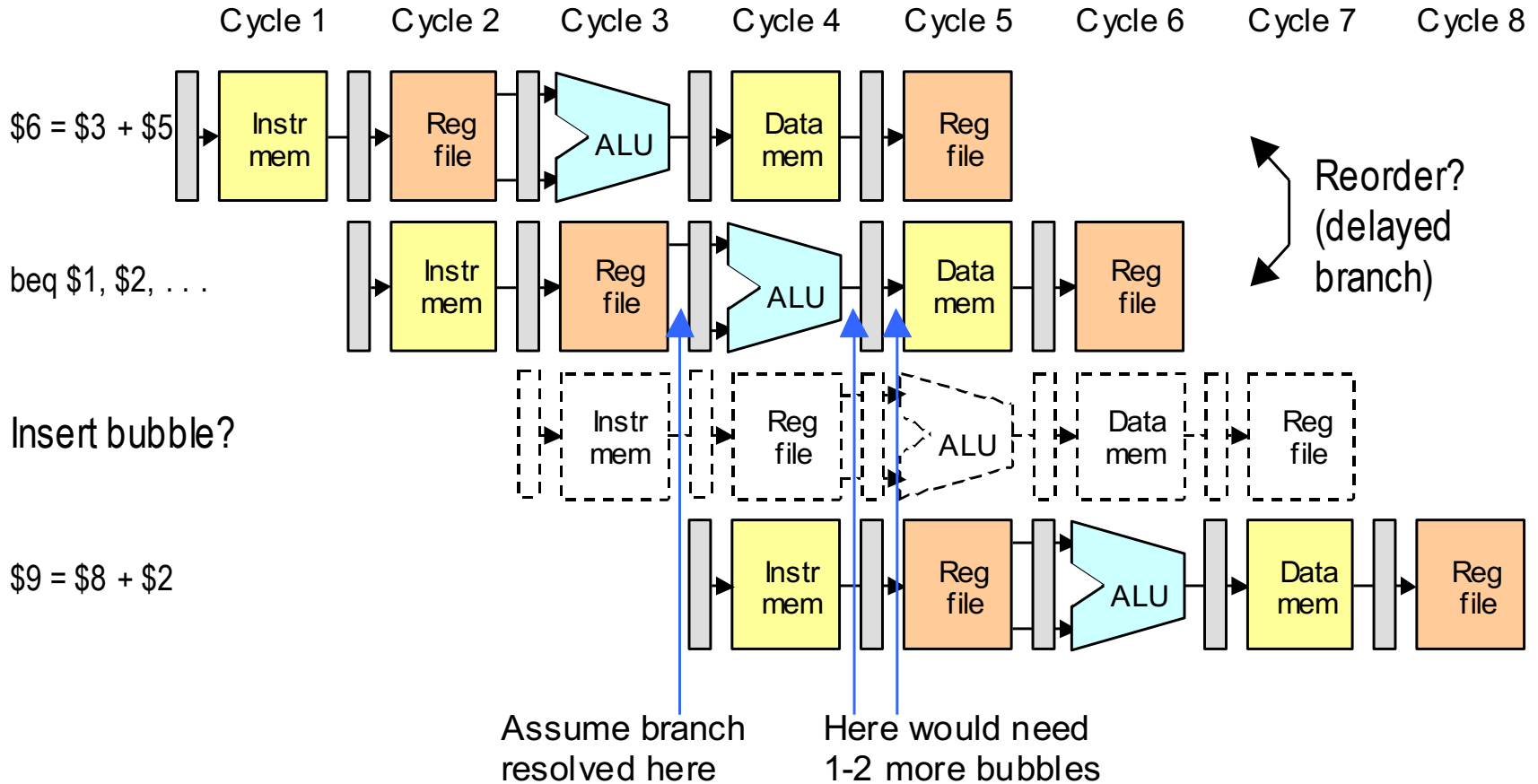
De asemenea, avem nevoie de circuite adiționale pentru a ne aduce aminte că a fost o presupunere și să golim b.a. dacă load != zero!

Nu este indicată o astfel de implementare în practică – de ce?

## De ce avem nevoie pentru a calcula următorul PC?

- Pentru Jump-uri
  - Opcode, PC și offset
- Pentru Jump Register
  - Opcode, Register value și PC
- Pentru Branch-uri Condiționale
  - Opcode, Register (pentru condiție), PC și offset
- Pentru toate celalalte instrucțiuni
  - Opcode și PC ( și trebuie să știm dinainte că nu este una dintre cele de mai sus )

# Hazarduri de control într-o b.a.



Hazard de control datorat unui branch condițional.

# Bule în calculul PC

*time*

t0 t1 t2 t3 t4 t5 t6 t7 . . . .

(I<sub>1</sub>) x1 ← x0 + 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>) x3 ← x2 + 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>

*time*

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>

ID

I<sub>1</sub> - I<sub>2</sub> - I<sub>3</sub> - I<sub>4</sub>

EX

I<sub>1</sub> - I<sub>2</sub> - I<sub>3</sub> - I<sub>4</sub>

MA

I<sub>1</sub> - I<sub>2</sub> - I<sub>3</sub> - I<sub>4</sub>

WB

I<sub>1</sub> - I<sub>2</sub> - I<sub>3</sub> - I<sub>4</sub>

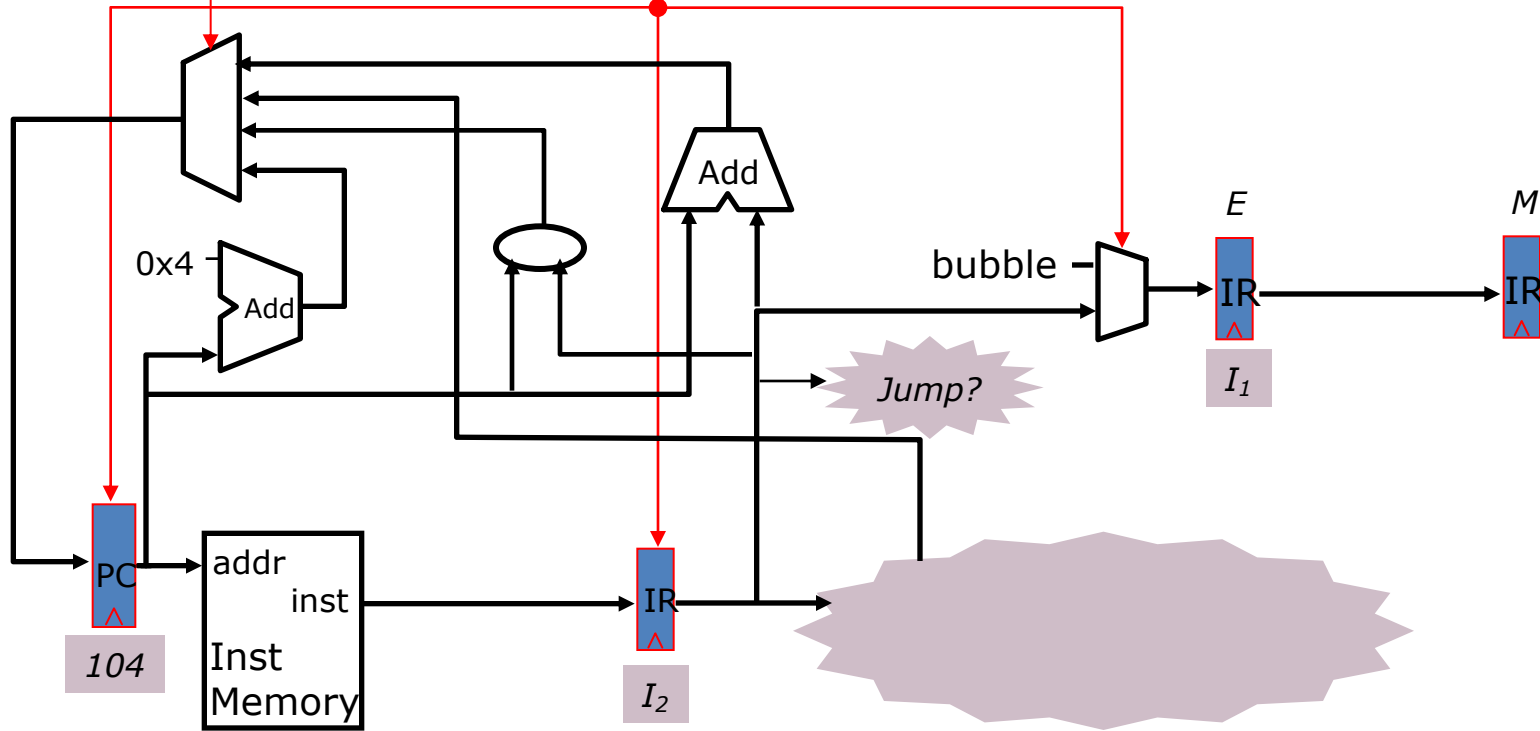
Resource  
Usage

- ⇒ *pipeline bubble*



# Speculăm că următoarea adresă e PC+4

PCSrc (pc+4 / jabs / rind/ br) *stall*

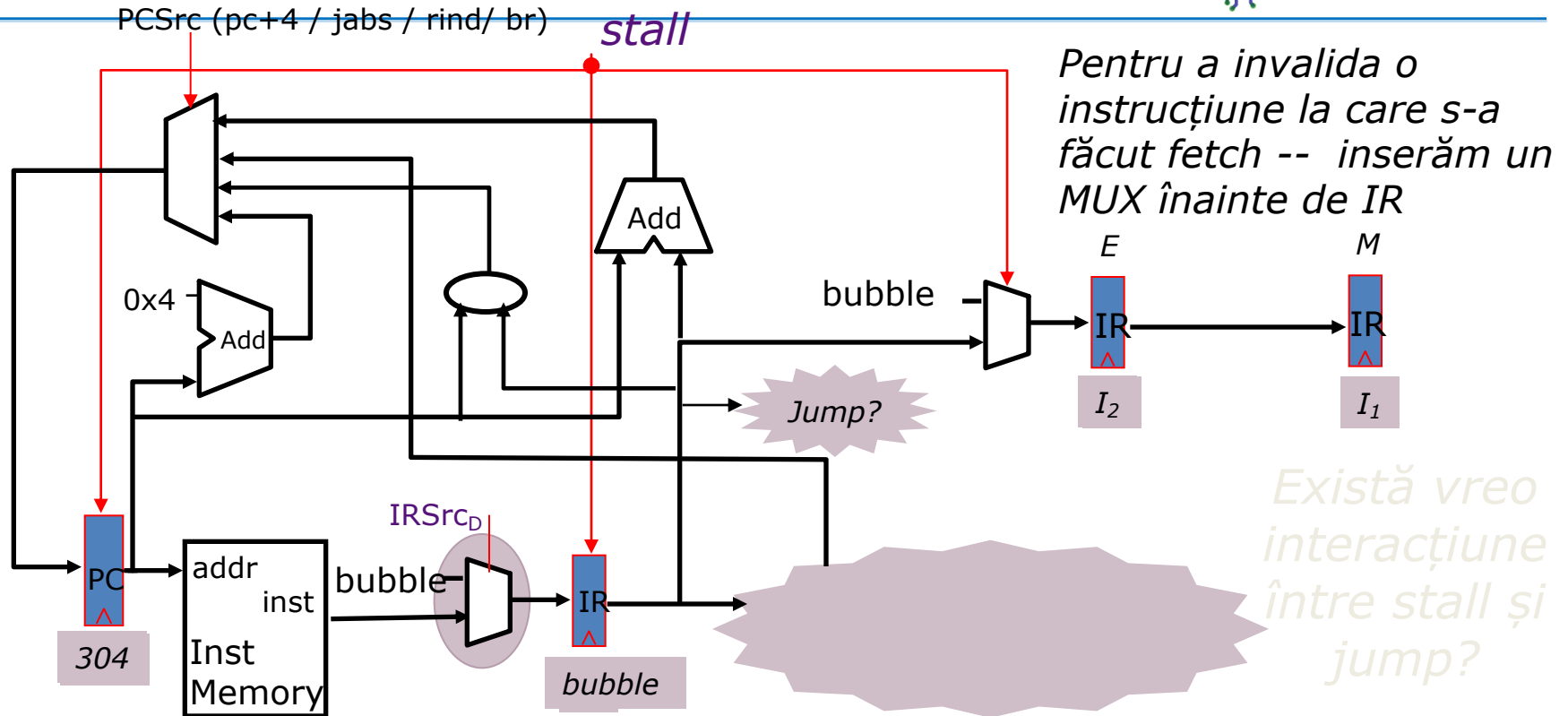


I <sub>1</sub>	096	ADD	
I <sub>2</sub>	100	J 304	
I <sub>3</sub>	<del>104</del>	<del>ADD</del>	<i>kill</i>
I <sub>4</sub>	304	ADD	

O instrucțiune de jump omoară (nu stagnează) următoarea instrucțiune

*Cum?*

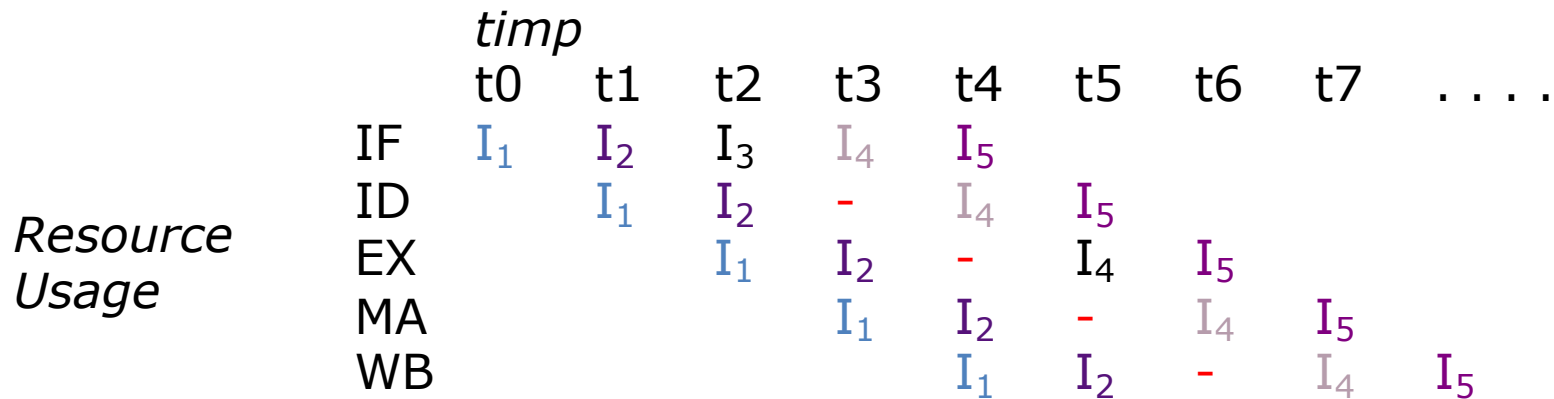
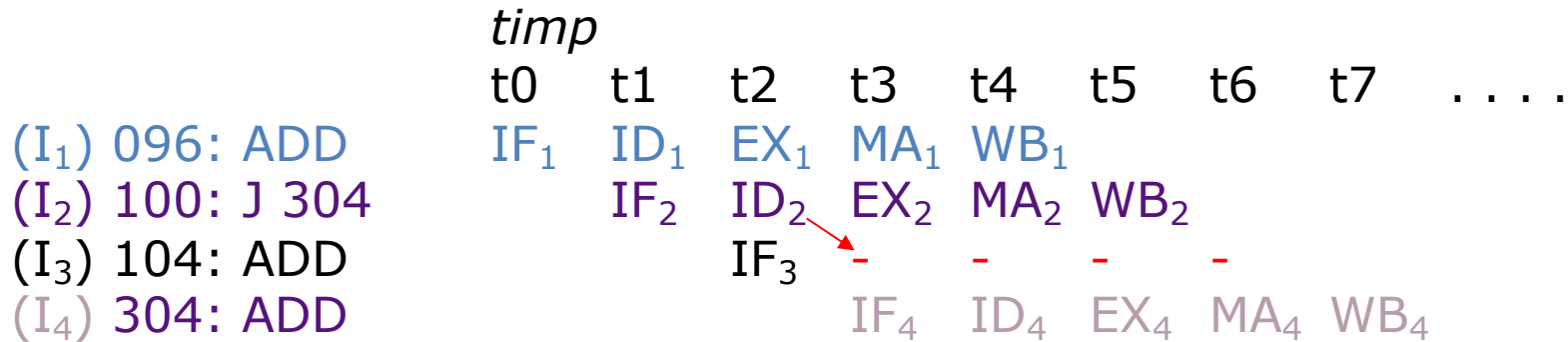
# Implementarea salturilor în b.a.



$I_1$	096	ADD	
$I_2$	100	J 304	
$I_3$	<del>104</del>	<del>ADD</del>	<i>kill</i>
$I_4$	304	ADD	

$IRSrc_D = \text{Case opcode}_D$   
 J, JAL  $\Rightarrow$  bubble  
 ...  $\Rightarrow$  IM

# Diagrame pipeline pentru Jump

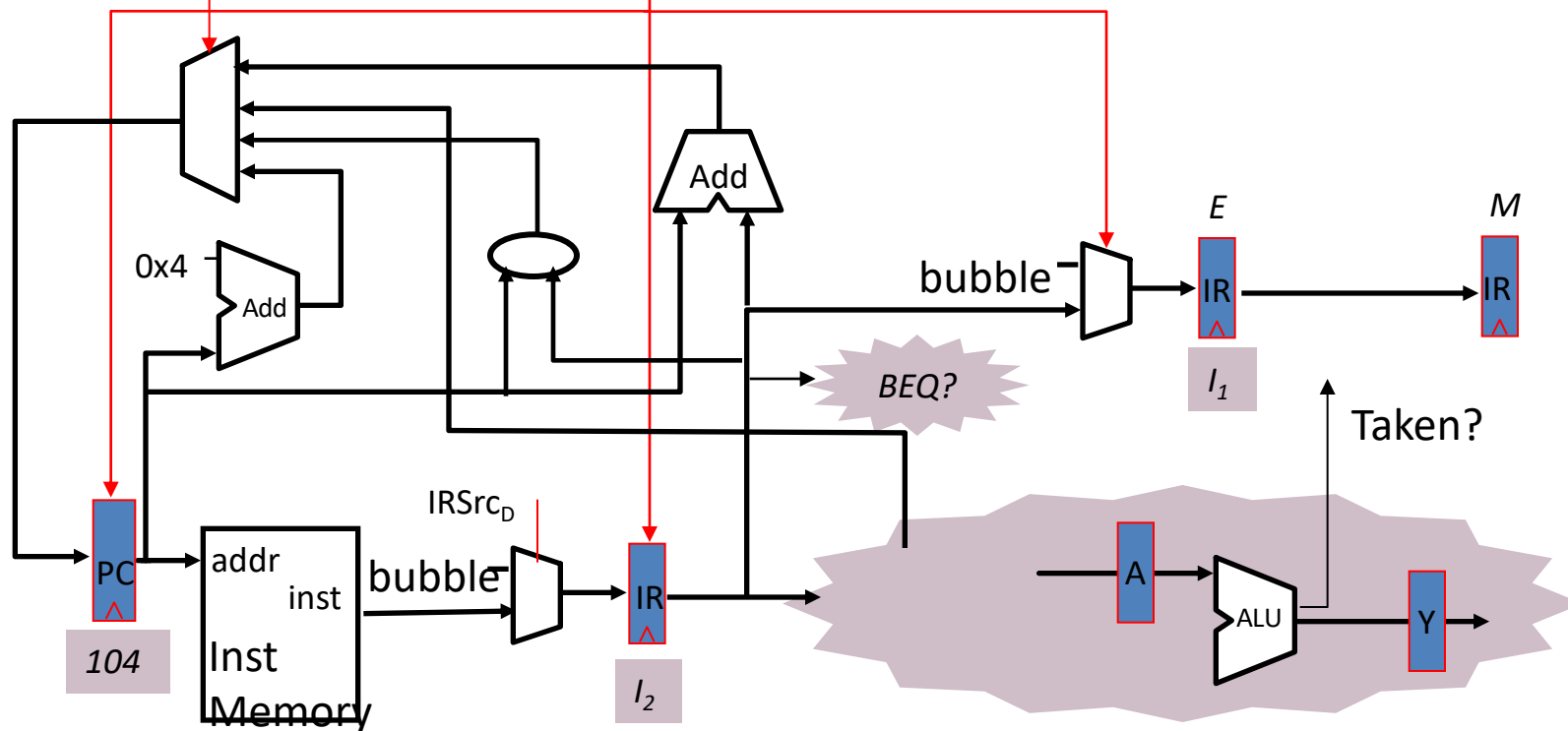


- ⇒ *pipeline bubble*

# Implementarea în b.a. a salturilor condiționale

PCSrc (pc+4 / jabs / rind / br)

*stall*

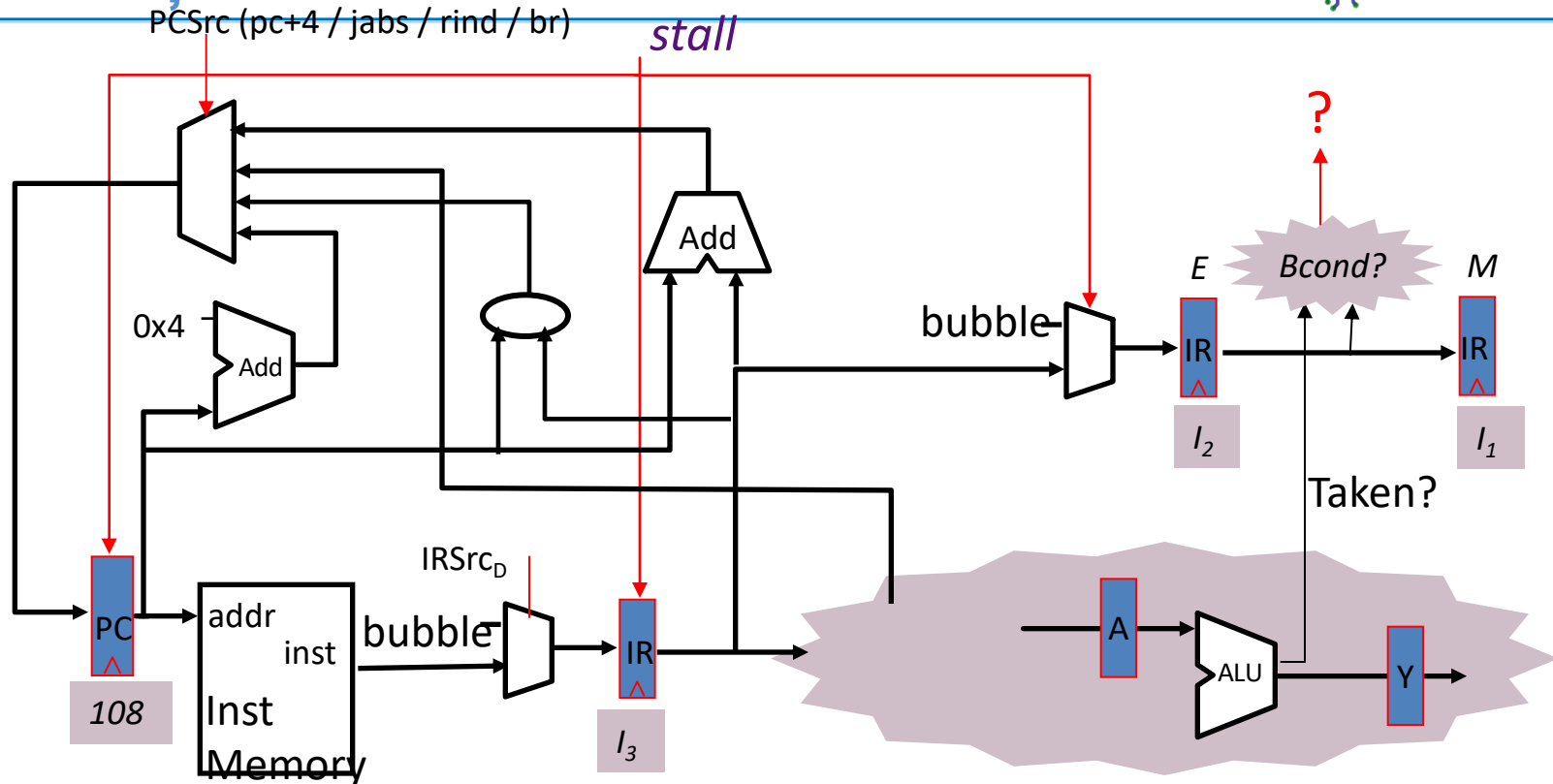


I <sub>1</sub>	096	ADD
I <sub>2</sub>	100	BEQ x1,x2 +200
I <sub>3</sub>	104	ADD
I <sub>4</sub>	304	ADD

Condiția de brach nu e cunoscută până la faza de execuce

*Ce acțiuni putem întreprinde în faza de decode?*

# Implementarea în b.a. a salturilor condiționale



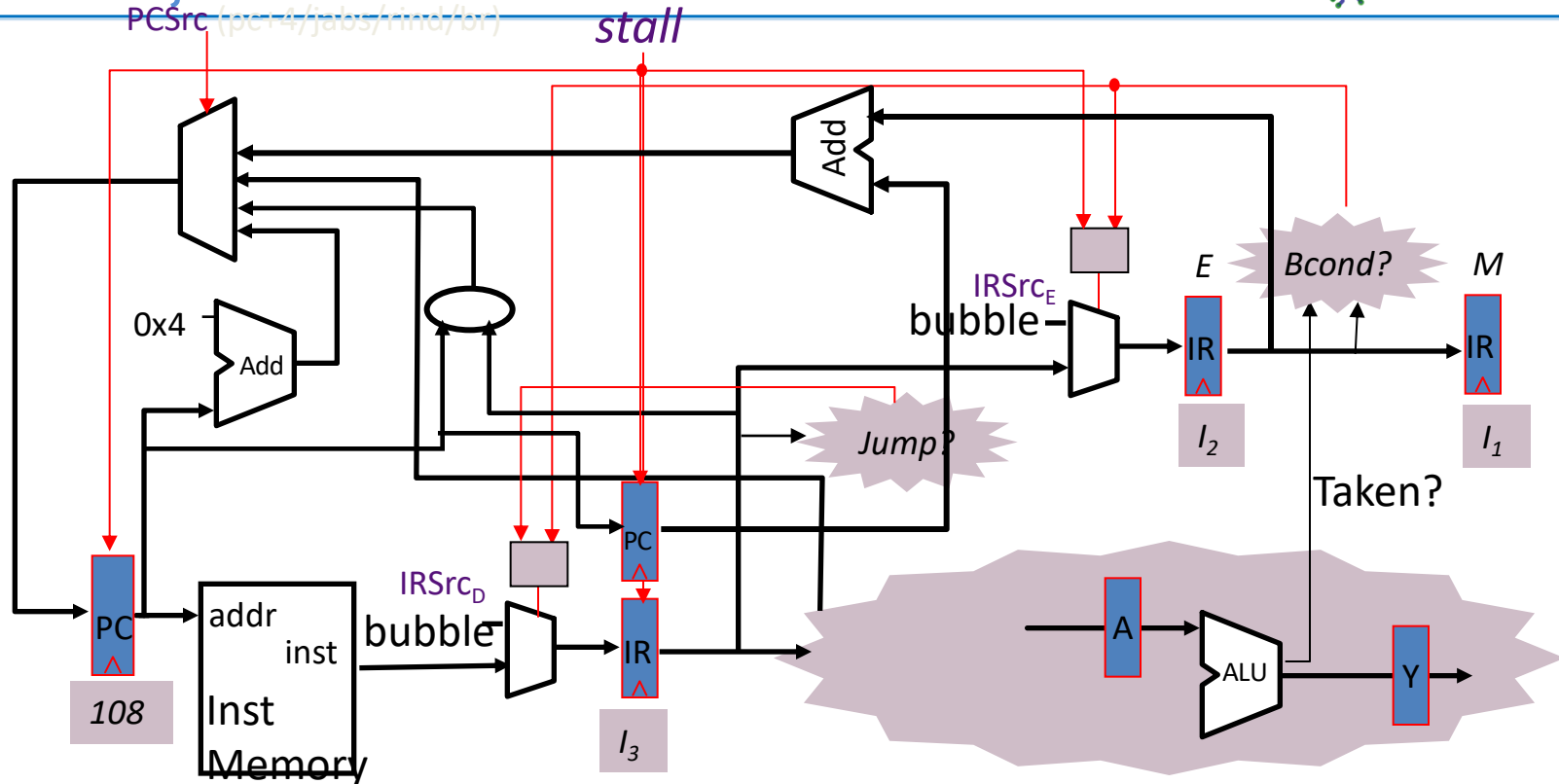
$I_1$	096	ADD
$I_2$	100	BEQ x1,x2 +200
$I_3$	104	ADD
$I_4$	304	ADD

Dacă branch-ul este valid

- Invalidează următoarele 2 instrucțiuni
- Instrucțiunea din faza de decode nu este validă

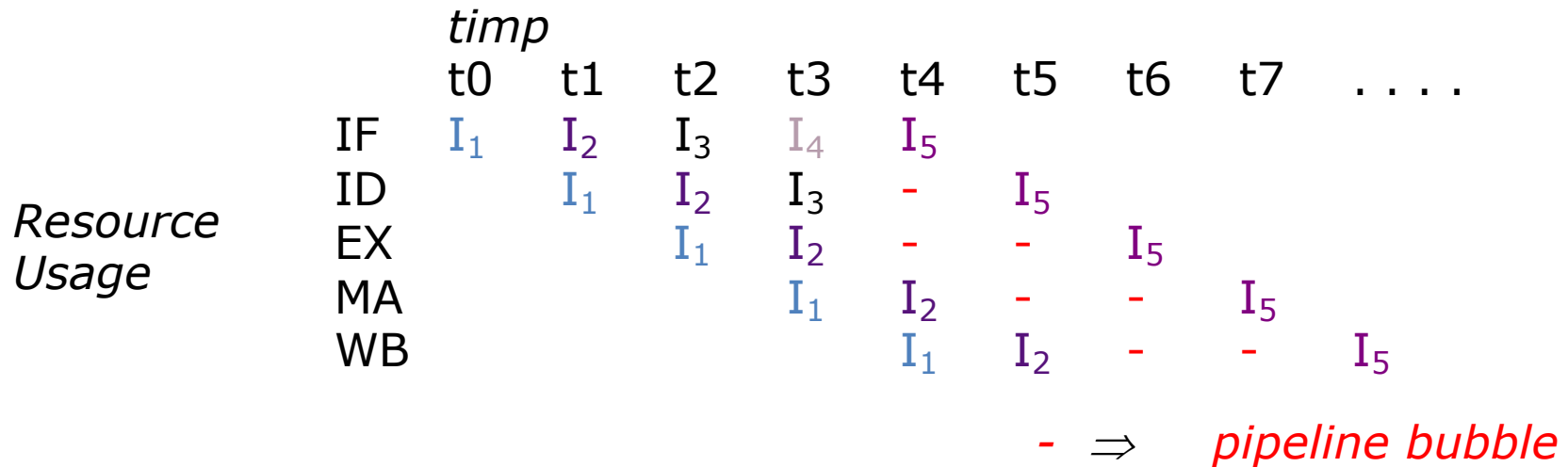
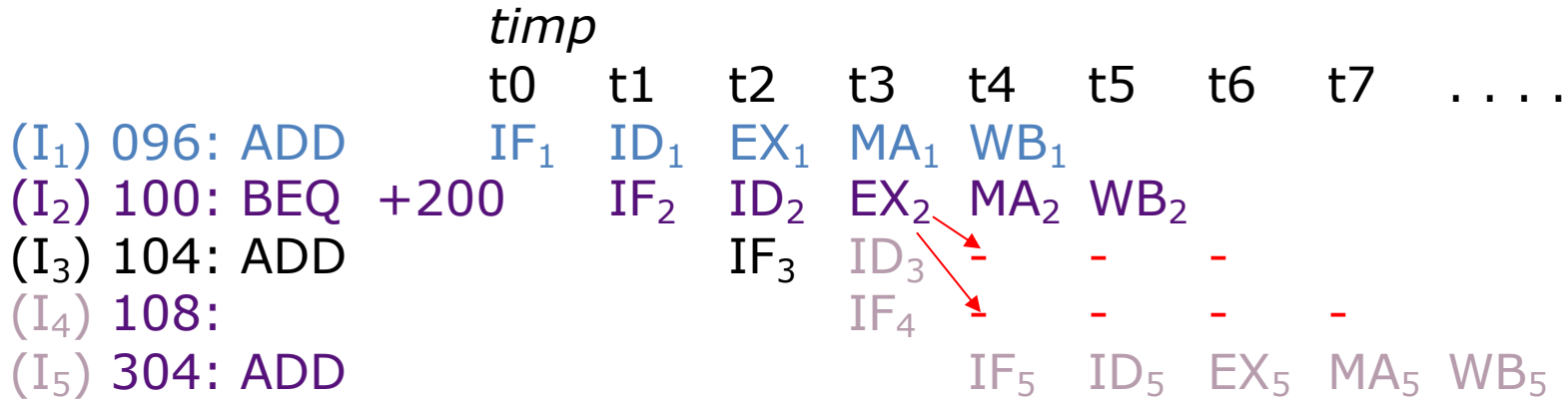
⇒ *semnalul stall nu este valid*

# Implementarea în b.a. a salturilor condiționale

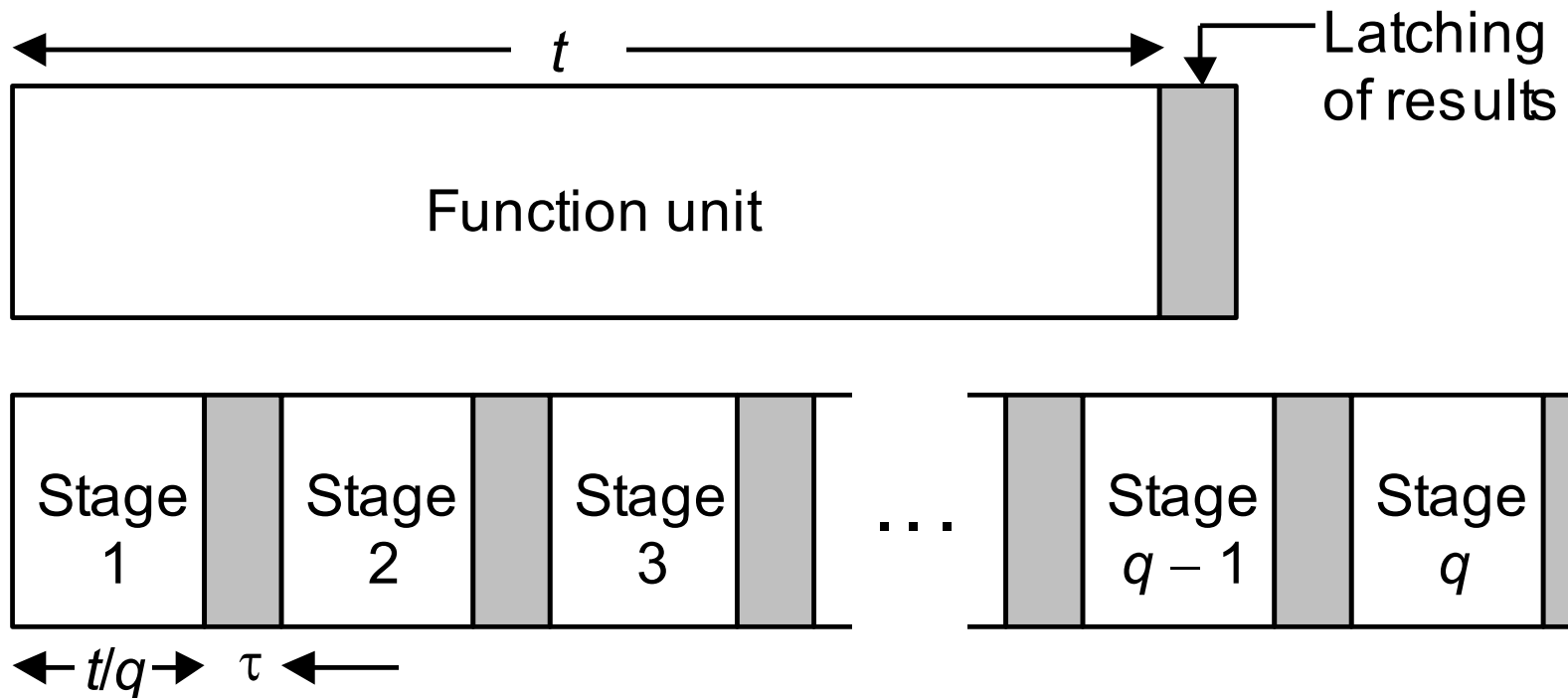


I <sub>1</sub> :	096	ADD	Dacă branch-ul este valid
I <sub>2</sub> :	100	BEQ x1,x2 +200	- Omoară următoarele 2 instrucțiuni
I <sub>3</sub> :	104	ADD	- Instrucțiunea din faza de decode nu este validă
I <sub>4</sub> :	304	ADD	⇒ <i>semnalul stall nu este valid</i>

# Diagrame pentru execuția în b.a. a branch-urilor



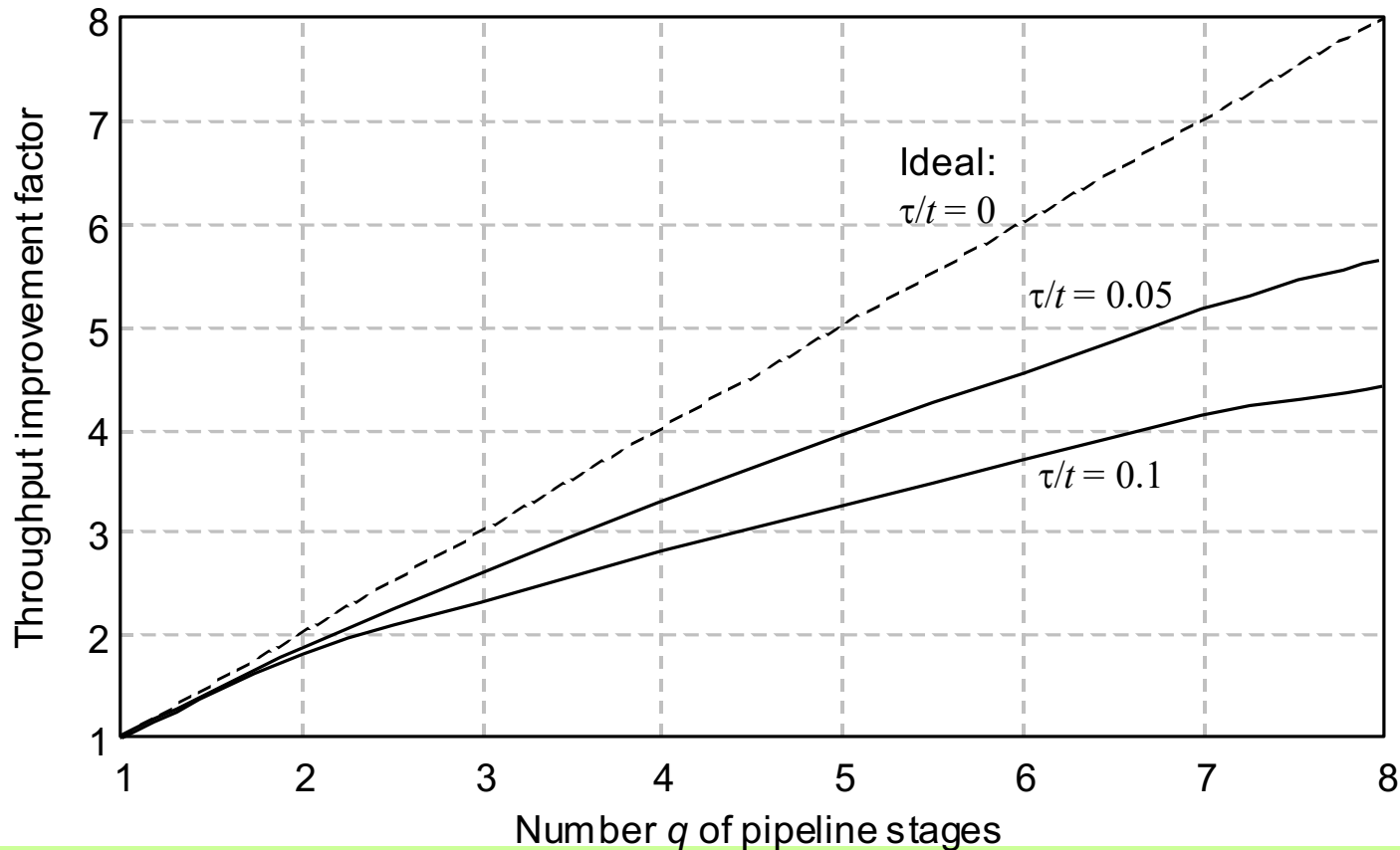
# Temporizarea și performanța unei b.a.



Forma în b.a. a unei unități funcționale cu întârzierile introduse de latch-uri



# Creșterea de productivitate a unei b.a. cu q stagii



$$\frac{t}{t/q + \tau}$$

sau

$$\frac{q}{1 + q\tau/t}$$

Creșterea în productivitate în funcție de numărul de etape din b.a. și de întârzierile datorate propagării semnalelor prin diferitele latch-uri ale b.a.

Presupunem că o bulă trebuie să fie inserată din cauza unei dependențe read-after-load și după un branch.

Notăm cu  $\beta$  procentul de instrucțiuni care sunt urmate de o bulă.

$$\text{Pipeline speedup} = \frac{q}{(1 + q\tau/t)(1 + \beta)}$$

R-type	44%
Load	24%
Store	12%
Branch	18%
Jump	2%

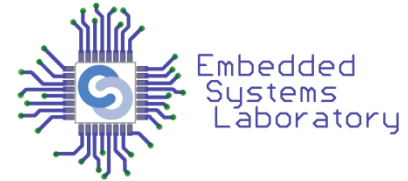
Exemplu:

Calculați CPI efectiv pentru un procesor MIPS, presupunând ca un sfert din toate instrucțiunile de branch și load sunt urmate de bule

**Soluție** Procent de bule:  $\beta = 0.25(0.24 + 0.18) = 0.105$

$\text{CPI} = 1 + \beta = 1.105$  (foarte aproape de valoarea ideală 1)

# Acknowledgements



- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252