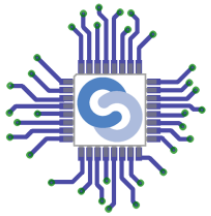


Calculatoare Numerice

– Cursul 11 –

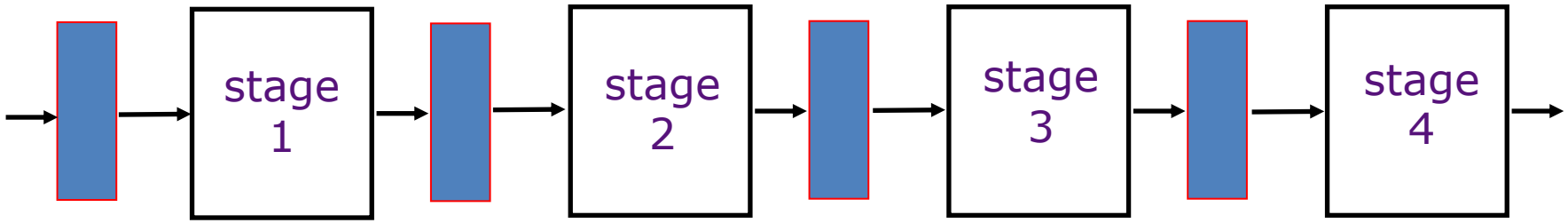
Banda de asamblare

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București

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



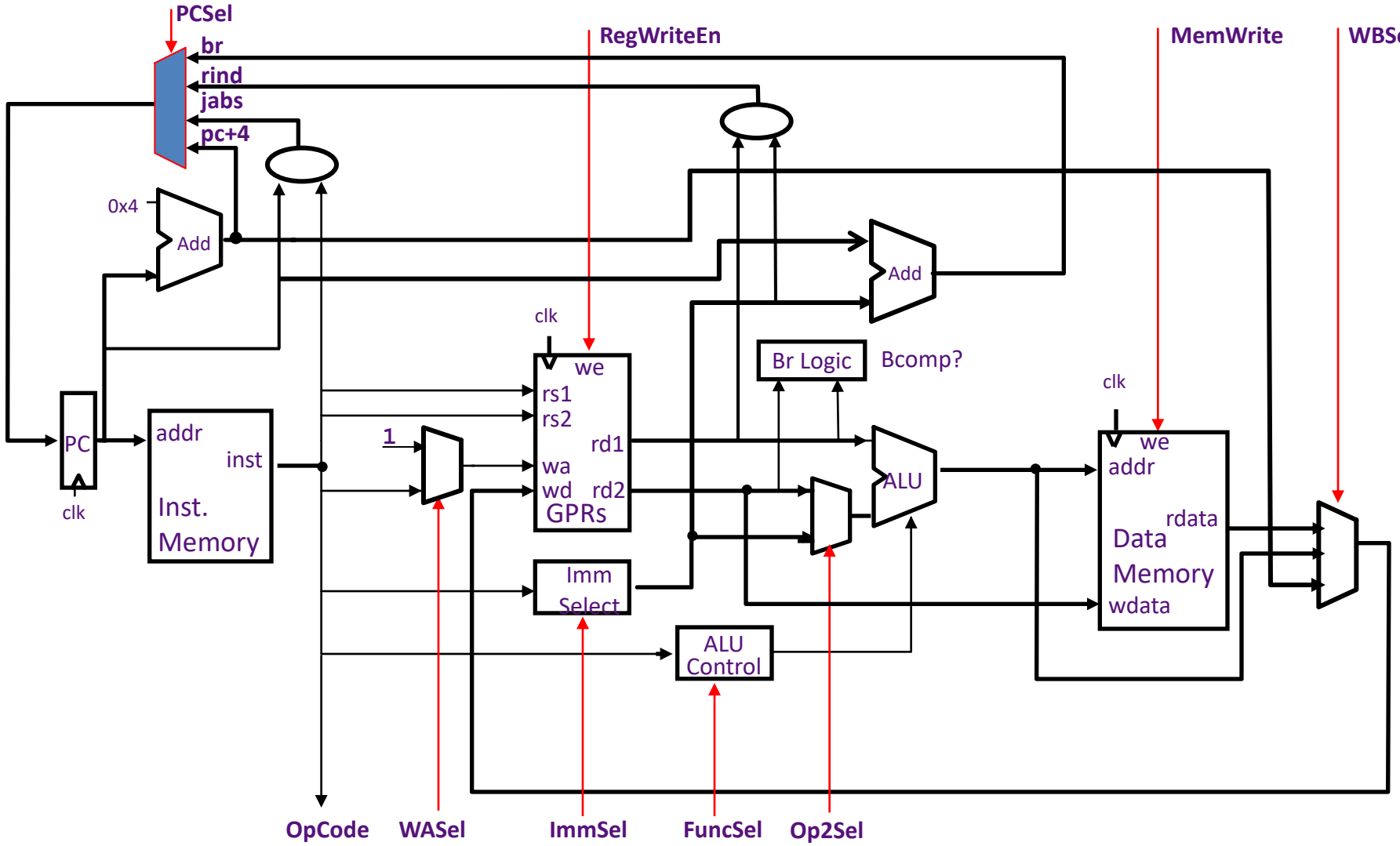
- Toate obiectele trec prin același niveluri
- Fără resurse partajate între oricare două niveluri
- Întârzierea de proapagare prin toate etapele benzii de asamblare este egală
- Planificarea intrării unui obiect în pipeline nu este afectată de obiectele din alte etape

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

RISC-V în bandă de asamblare

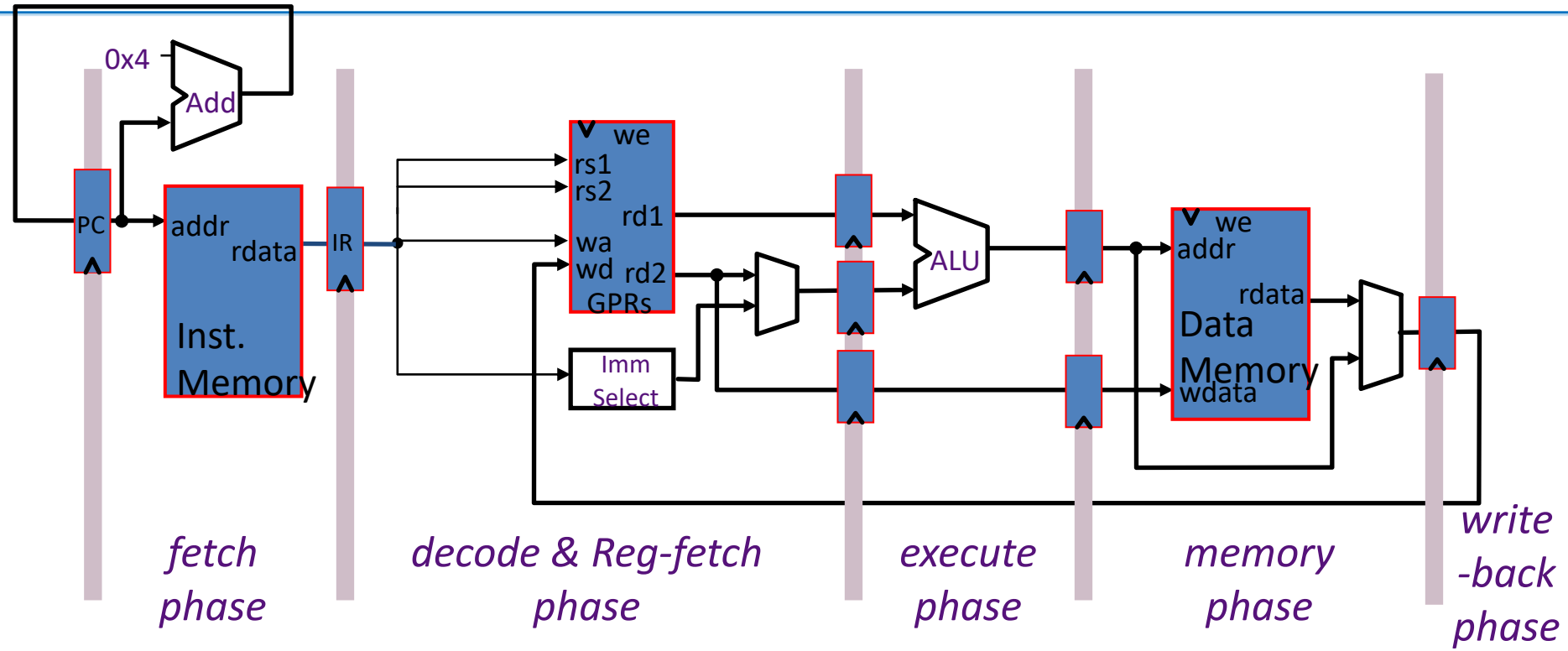
- Pentru a implementa RISC-V în bandă de asamblare:
- Întâi construim RISC-V fără pipeline cu CPI=1
- Pe urmă, adaugăm registre pentru pipeline pentru a reduce timpul de execuție, menținând CPI=1

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




Fetch
Reg Read
ALU
Data Memory
Reg Write

Magistrală în bandă de asamblare



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

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

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

“Legea de fier” a performanței procesoarelor

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

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

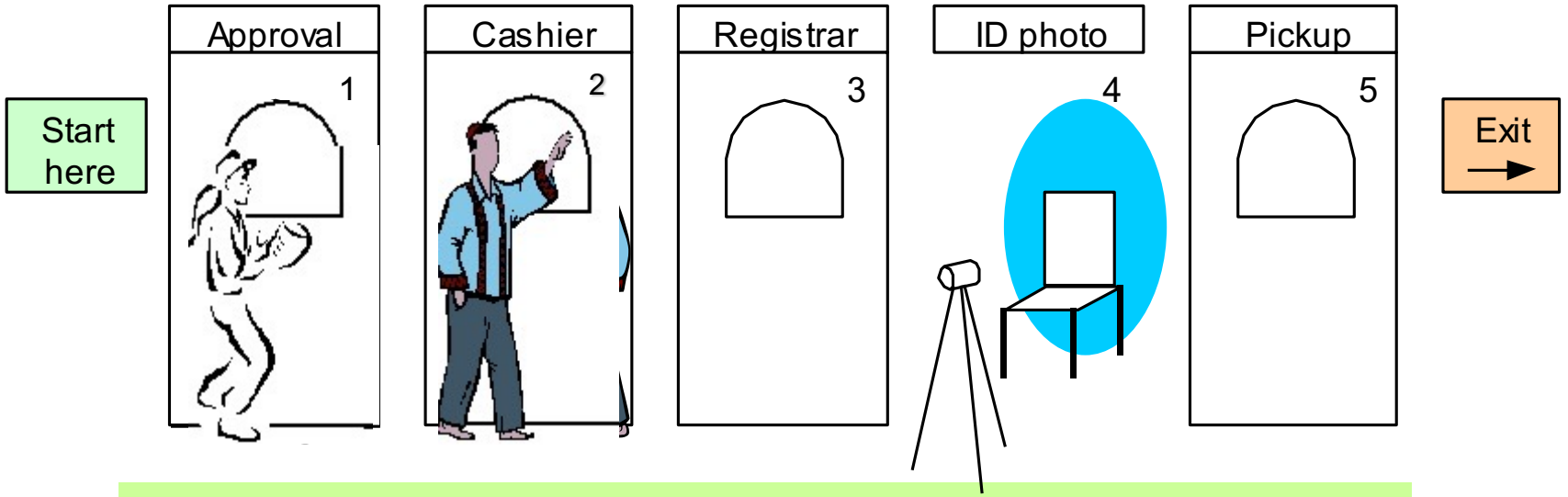
Microarchitectură	CPI	cycle time
Microcodificată	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

Concepte de pipelining

Strategii pentru îmbunătățirea performanței

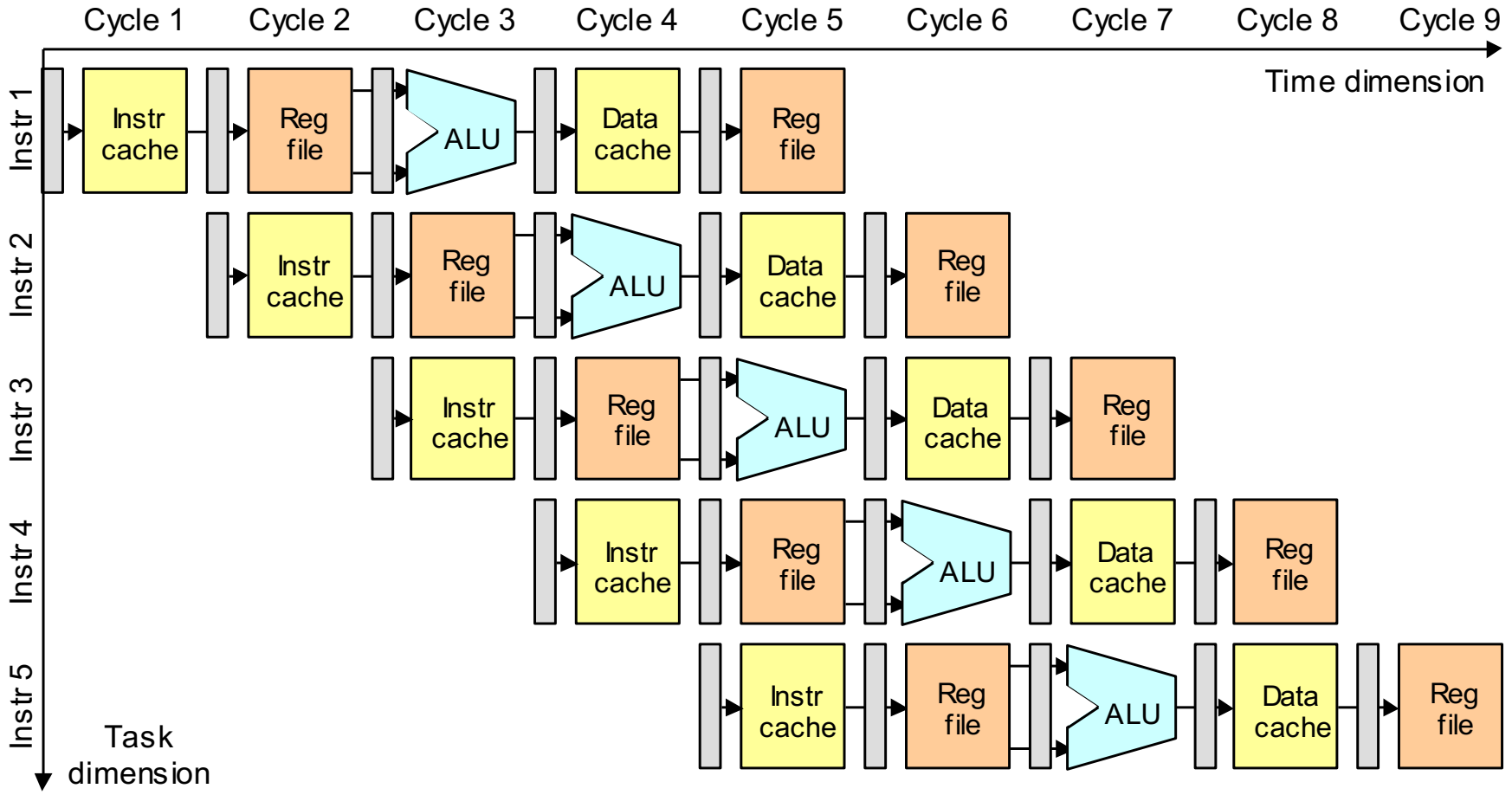
1 – Folosim căi de date independente care acceptă citirea mai multor instrucțiuni în paralel: *multiple-instruction-issue* sau *superscalar*

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

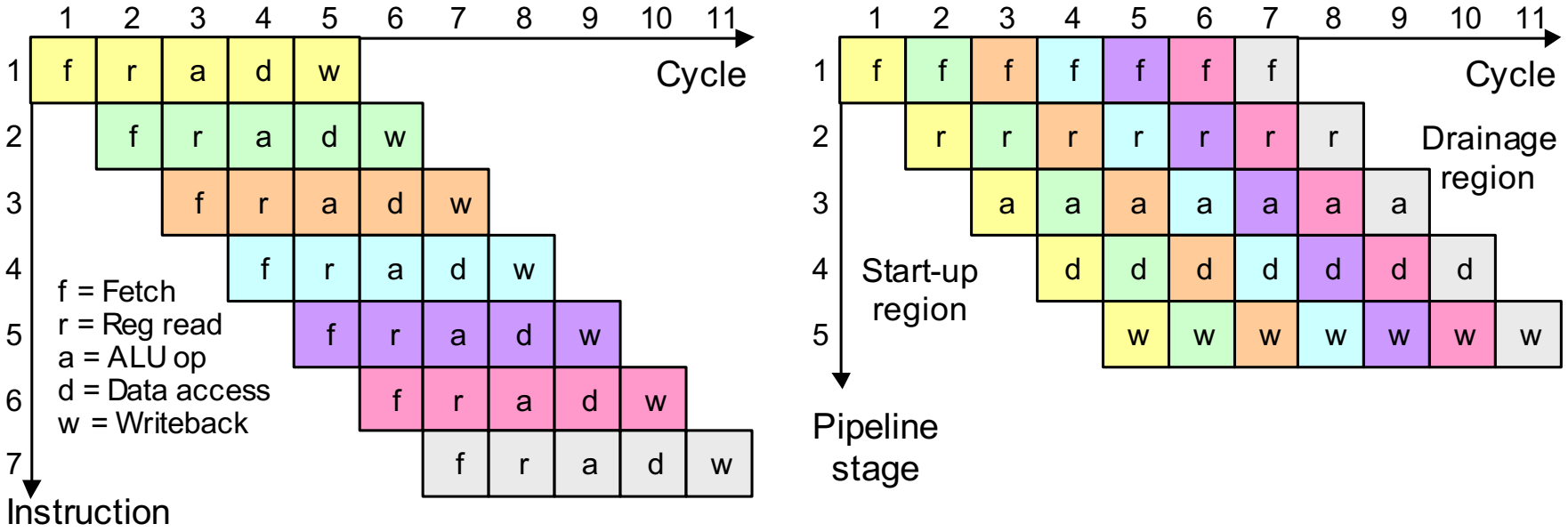
Execuția instrucțiunilor în banda de asamblare



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.

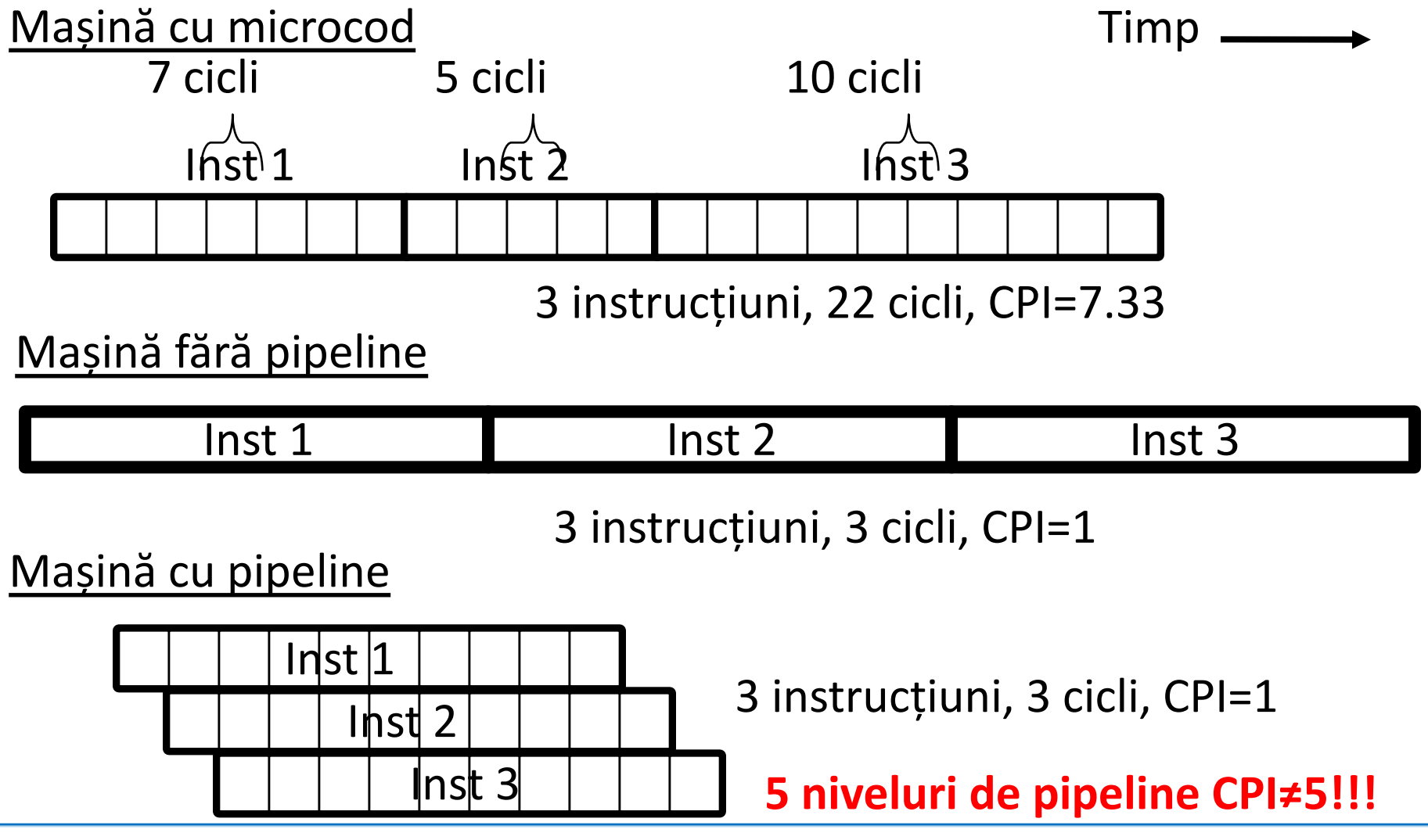
Un copiator fără b.a. necesită $4x$ secunde ca să copieze x foi.

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

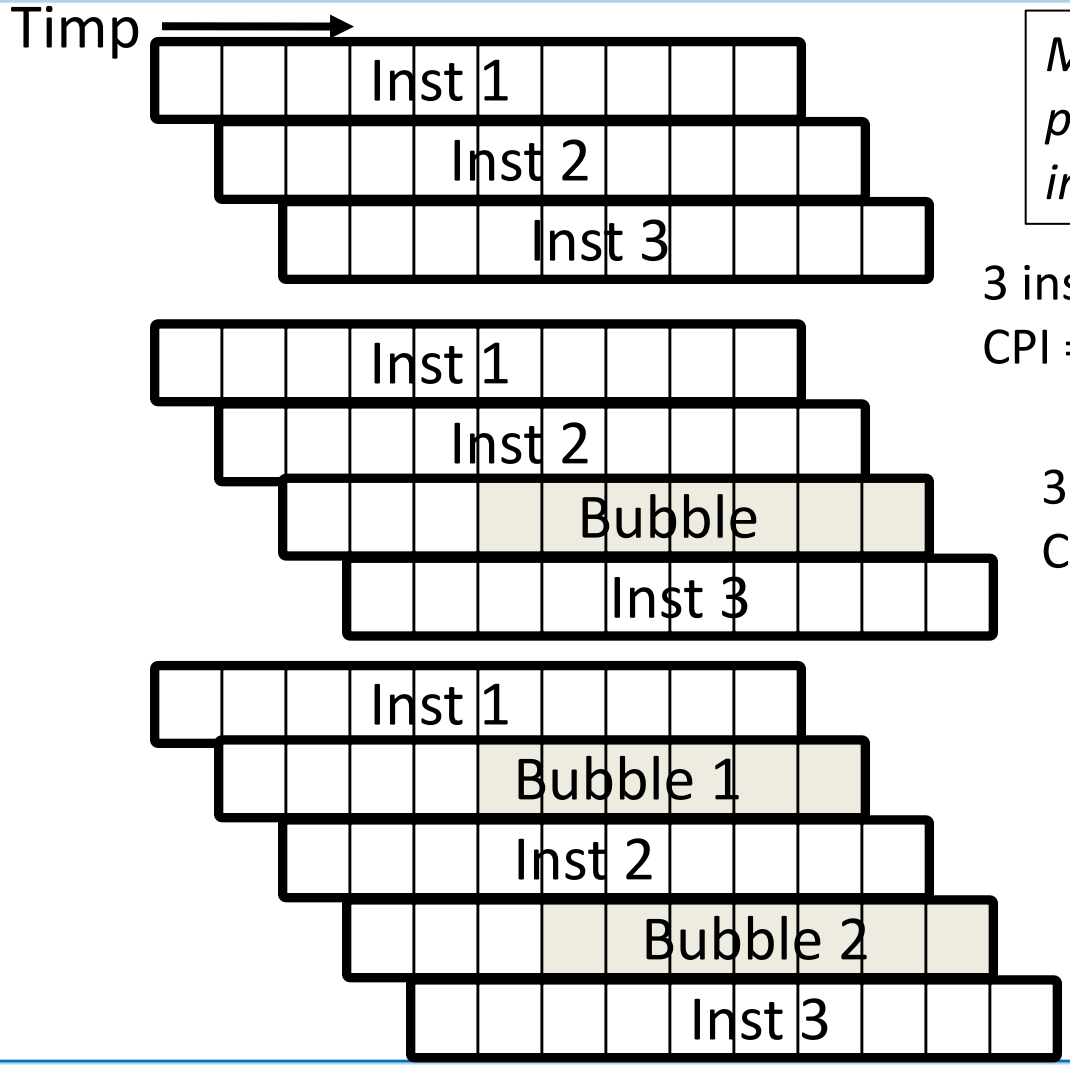
Dacă $x = 50$, îmbunătățirea de performanță este

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

Exemple CPI


Mașină cu microcod
Timp
7 cicli
5 cicli
10 cicli
Inst 1
Inst 2
Inst 3
3 instrucțiuni, 22 cicli, CPI=7.33
Mașină fără pipeline
Inst 1
Inst 2
Inst 3
3 instrucțiuni, 3 cicli, CPI=1
Mașină cu pipeline
Inst 1
Inst 2
Inst 3
3 instrucțiuni, 3 cicli, CPI=1
5 niveluri de pipeline CPI≠5!!!

Exemple de CPI pentru b.a.



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

3 instrucțiuni termină în 3 cicli
CPI = 3/3 =1

3 instrucțiuni termină în 4 cicli
CPI = 4/3 = 1.33

3 instrucțiuni termină în 5 cicli
CPI = 5/3 = 1.67

Presupuneri legate de tehnologie

- 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!)

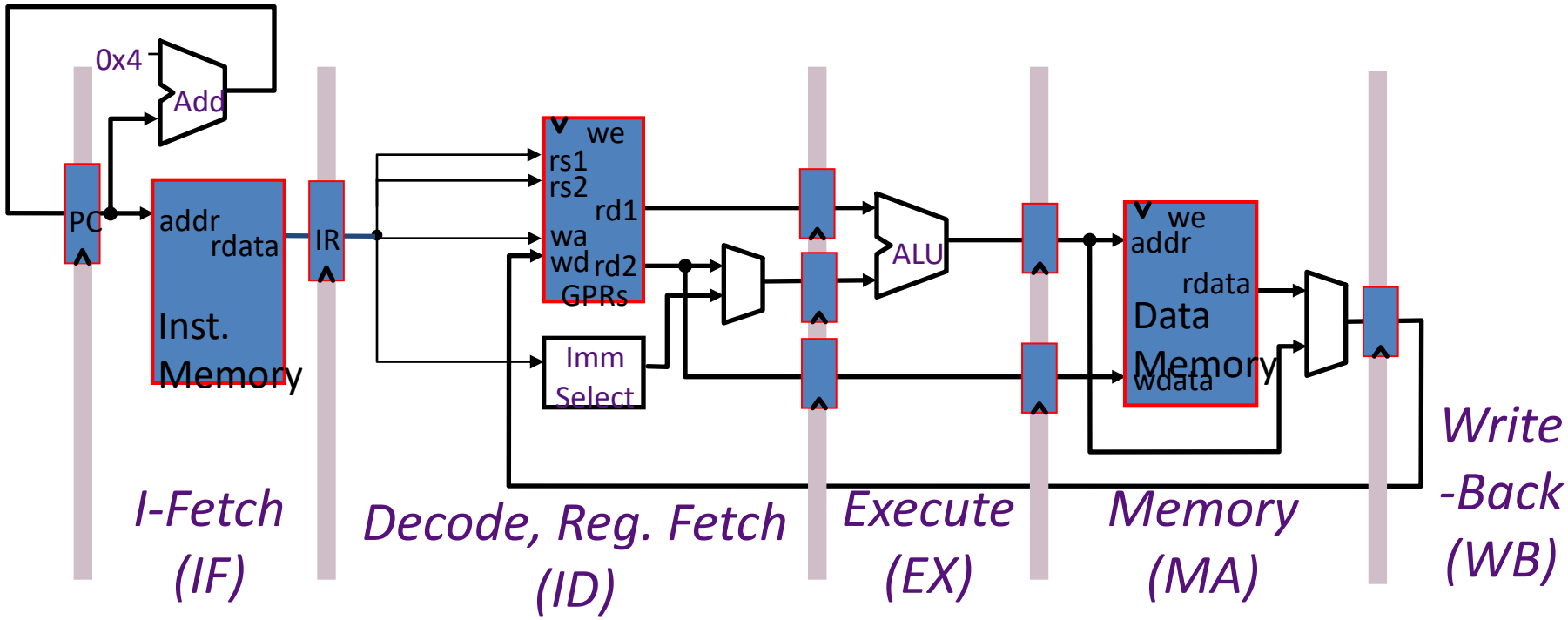
Prin urmare, următoarea presupunere este rezonabilă

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

Vom proiecta o bandă de asamblare în 5 etape

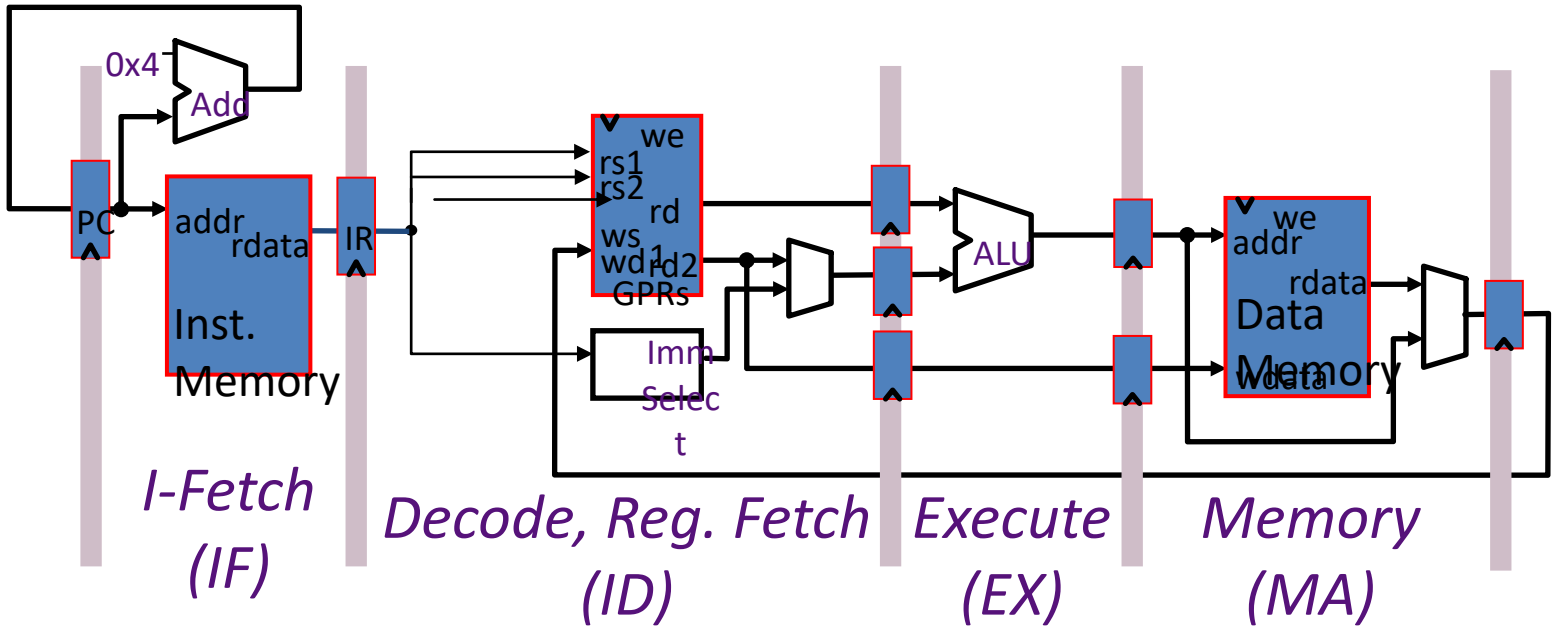
- unele design-uri comerciale de procesor au peste 30 de etape pentru a face o singură adunare de întregi!

Execuție în bandă de asamblare cu 5 niveluri



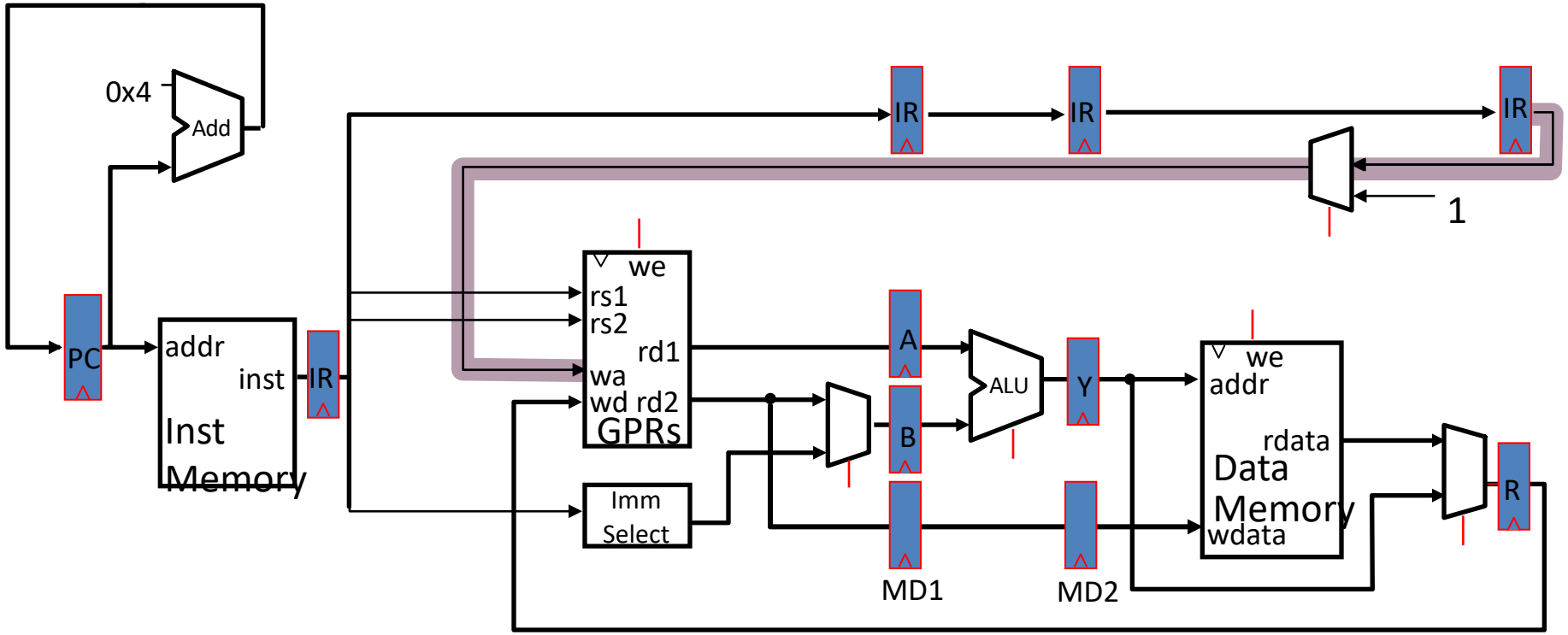
timp	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

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



Resurse	timp	t0	t1	t2	t3	t4	t5	t6	t7	. .
	IF	I_1	I_2	I_3	I_4	I_5				
	ID		I_1	I_2	I_3	I_4	I_5			
	EX			I_1	I_2	I_3	I_4	I_5		
	MA				I_1	I_2	I_3	I_4	I_5	
	WB					I_1	I_2	I_3	I_4	I_5

Execuție în pipeline: Instrucțiuni UAL

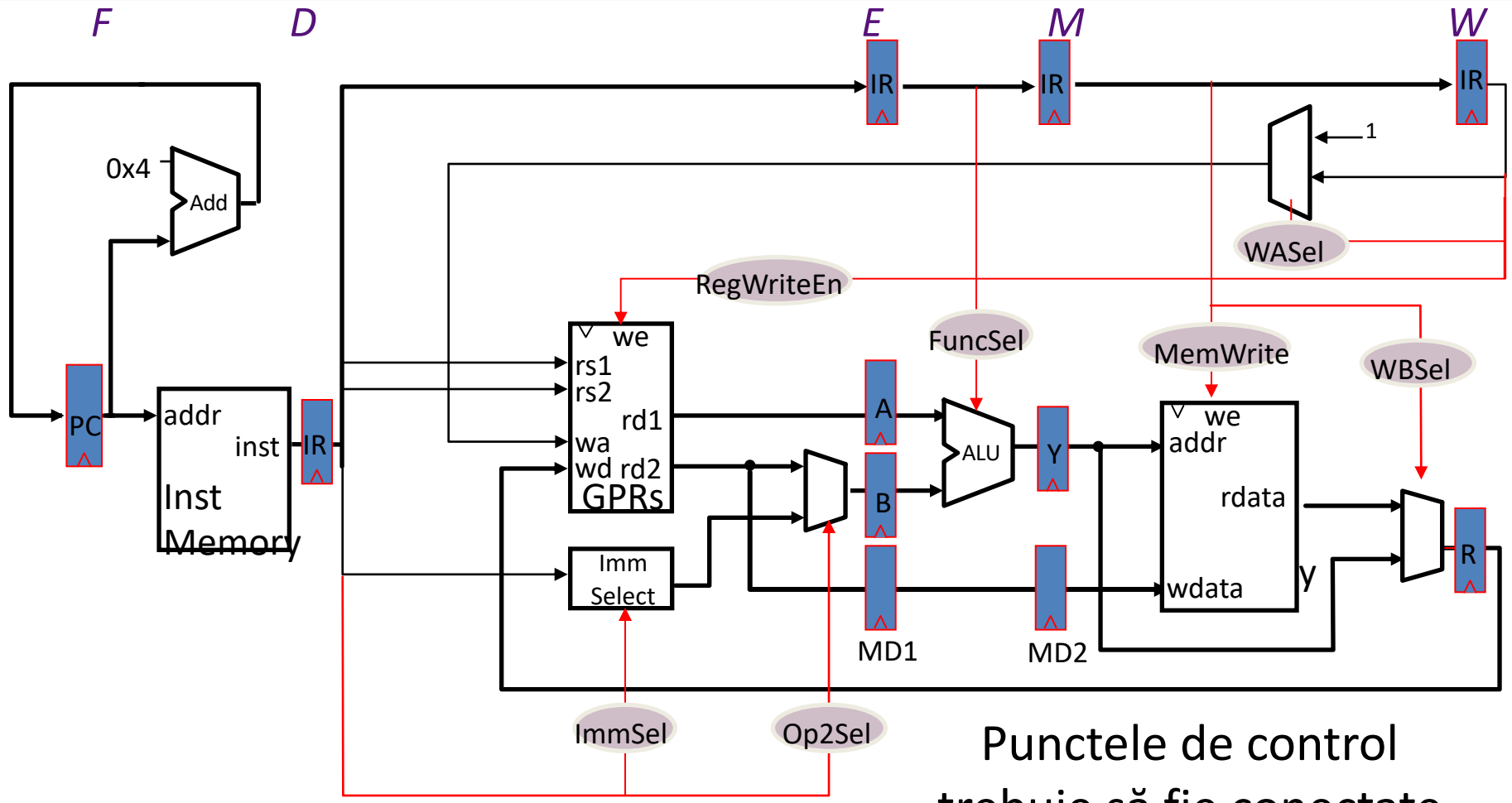


Nu e foarte corect!

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.

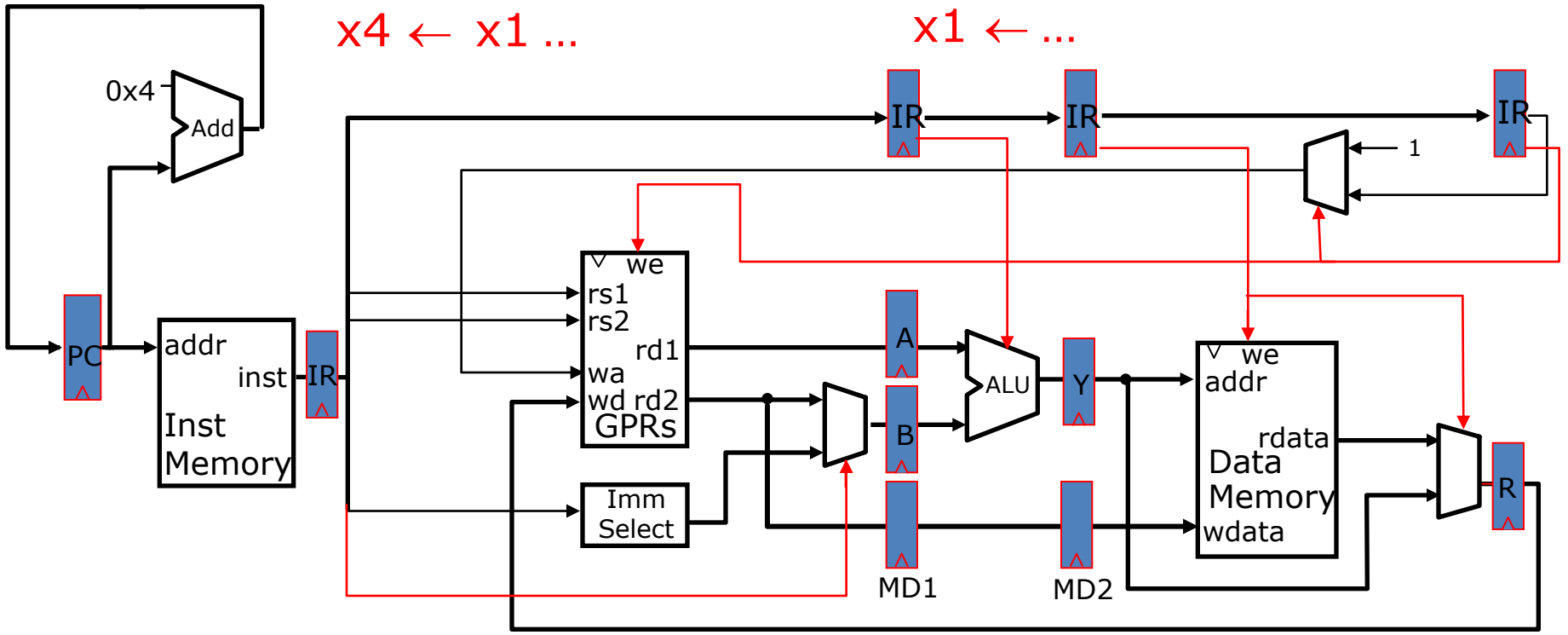
→ *Hazard structural*

- 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)*

Soluționarea hazardurilor structurale

- Hazardurile 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 hazarduri structurale de la bun început
  - Datorită RISC-V ISA, care a fost proiectată pentru execuția în b.a.

Hazarduri de date




```
...
x1 ← x0 + 10
x4 ← x1 + 17
...
```

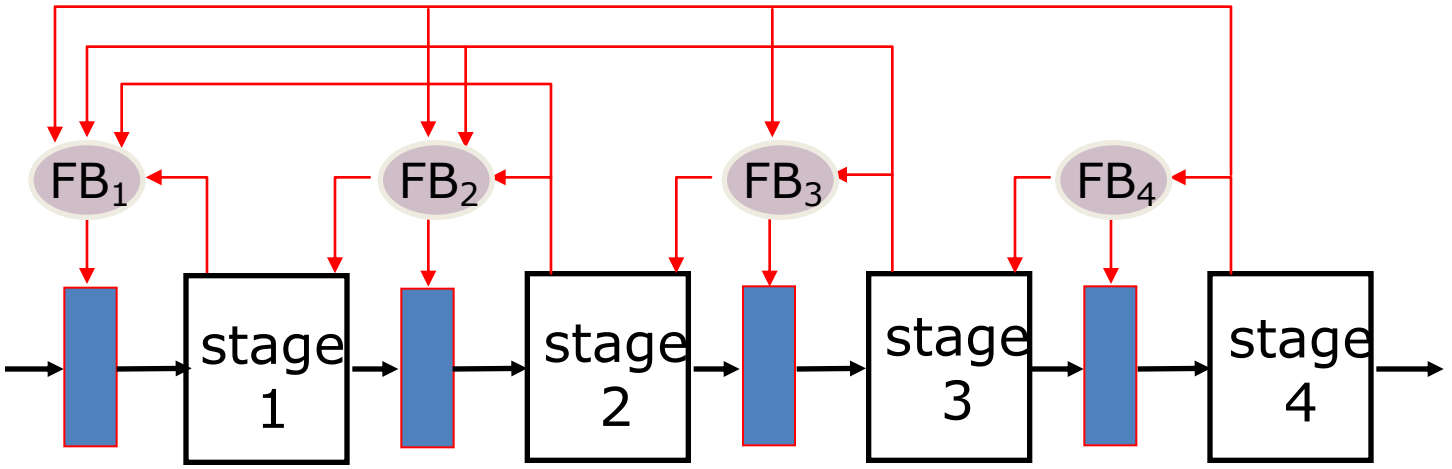
x1 este învechit. Oops!

Soluționarea hazardurilor de date (1)

Strategia 1:

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

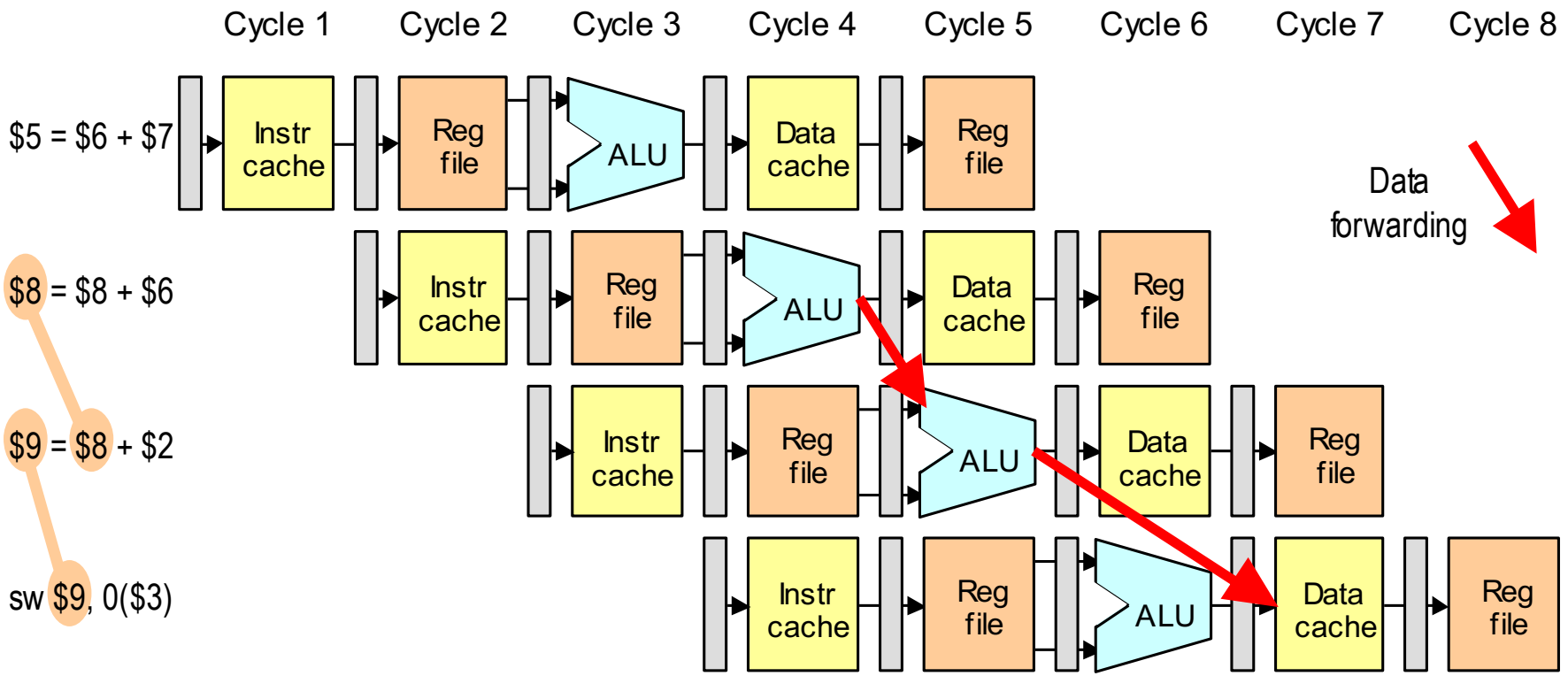
Feedback pentru soluționarea hazardurilor



- 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 comepleteze execuția fără nici o interferență cu instrucțiunile din nivelele 1..i (altfel pot apare deadlock-uri)

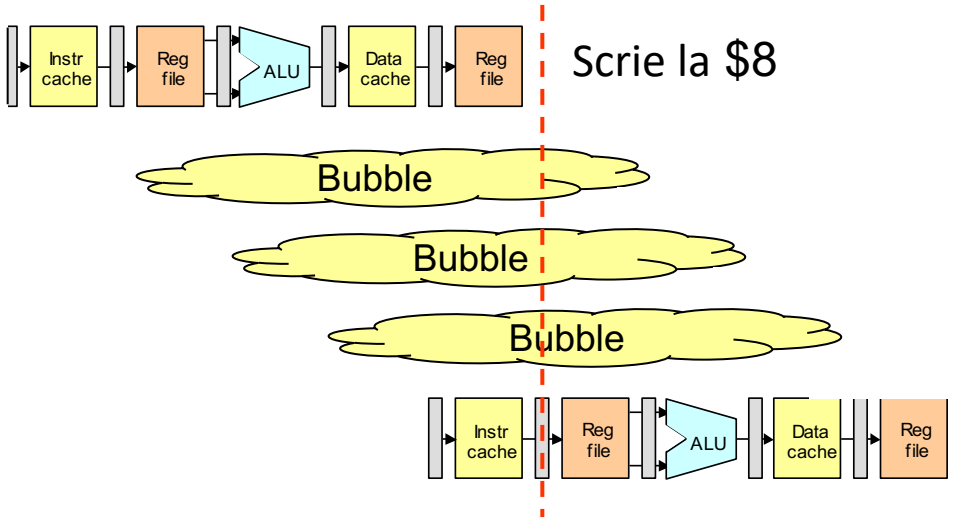
Hazarduri în b.a. Stalls & Bubbles

Generate din cauza dependențelor de date

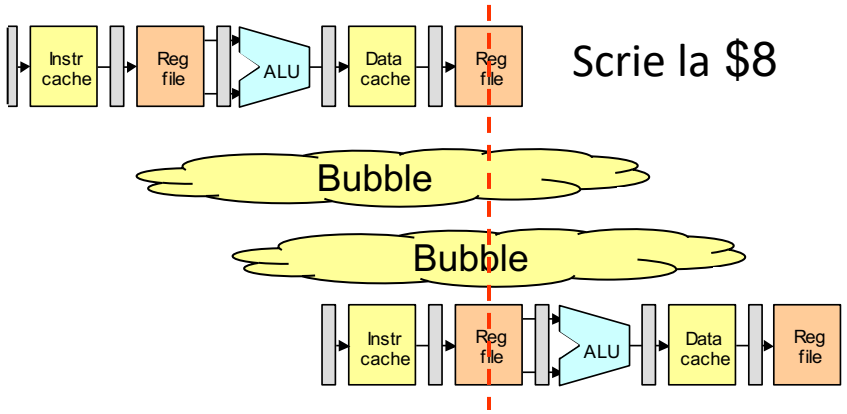


Dependență Read-After-Write și soluționarea ei prin data forwarding

Inserarea de bule într-o b.a.

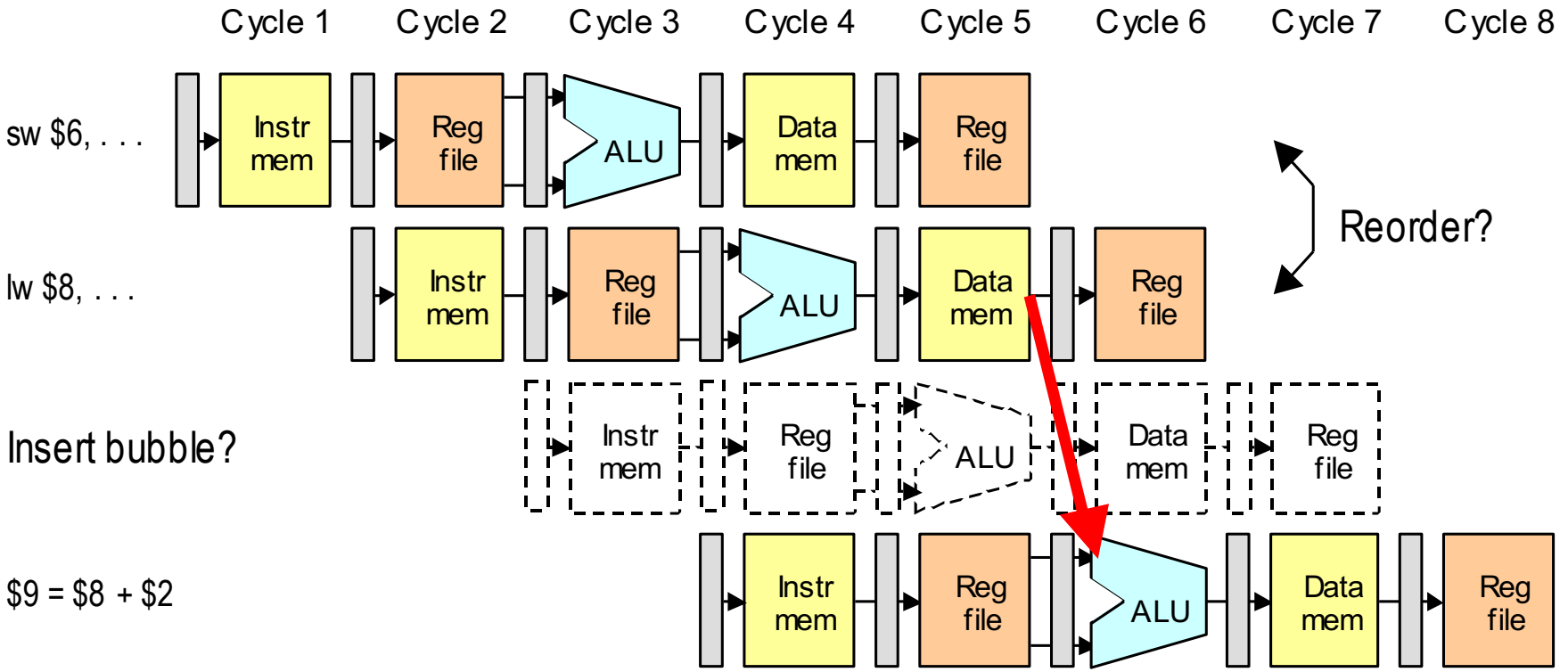


Fără data forwarding avem nevoie de aceste trei “bule” pentru a soluționa dependența read-after-write



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

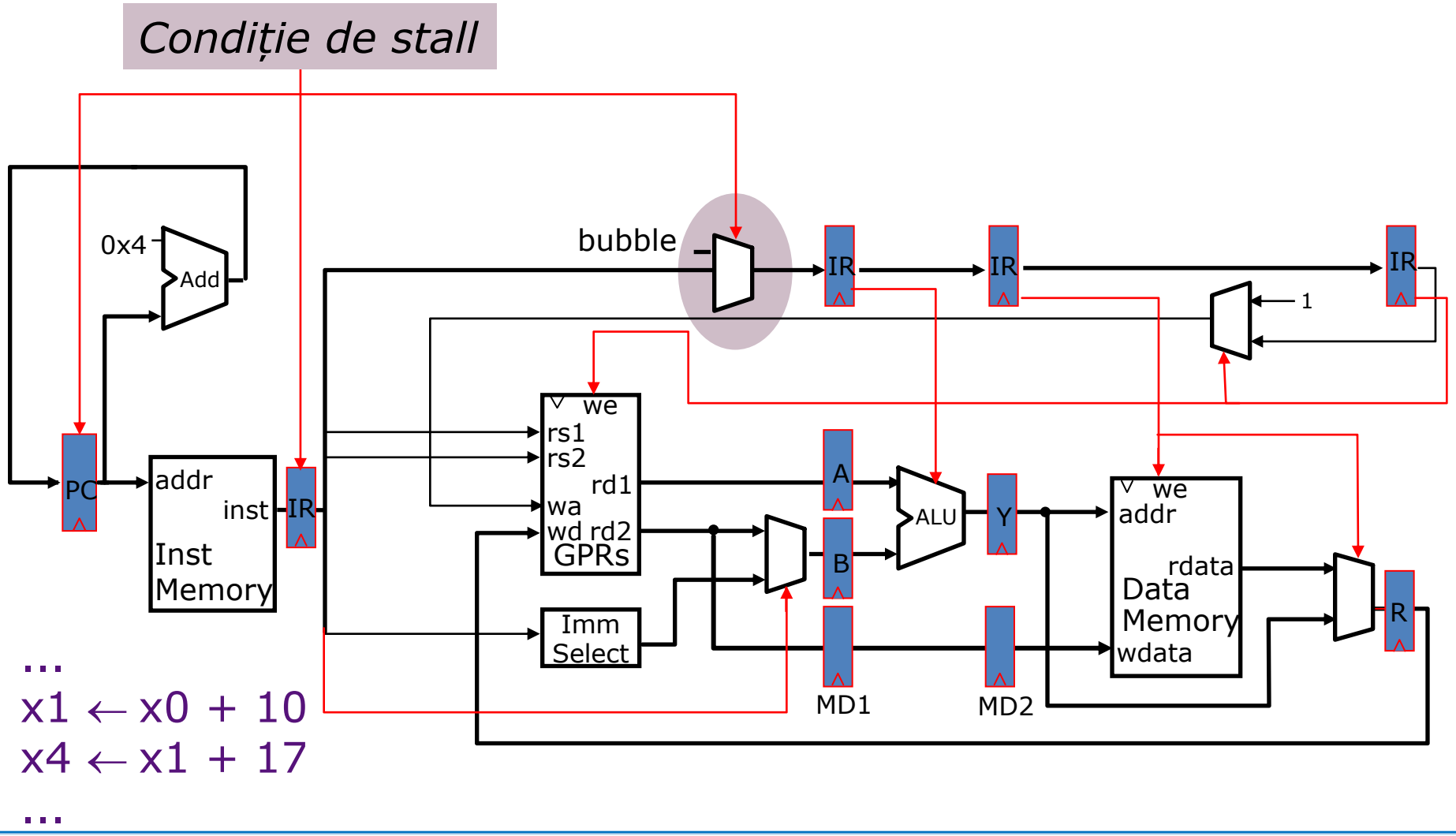
Al doilea tip de dependență de date



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



Stalled Stages & Pipeline Bubbles

timp

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .		
(I_1) x1 ← (x0) + 10	IF_1	ID_1	EX_1	MA_1	WB_1						
(I_2) x4 ← (x1) + 17		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2		
(I_3)			IF_3	IF_3	IF_3	IF_3	ID_3	EX_3	MA_3	WB_3	
(I_4)							IF_4	ID_4	EX_4	MA_4	WB_4
(I_5)								IF_5	ID_5	EX_5	MA_5 WB_5

stalled stages

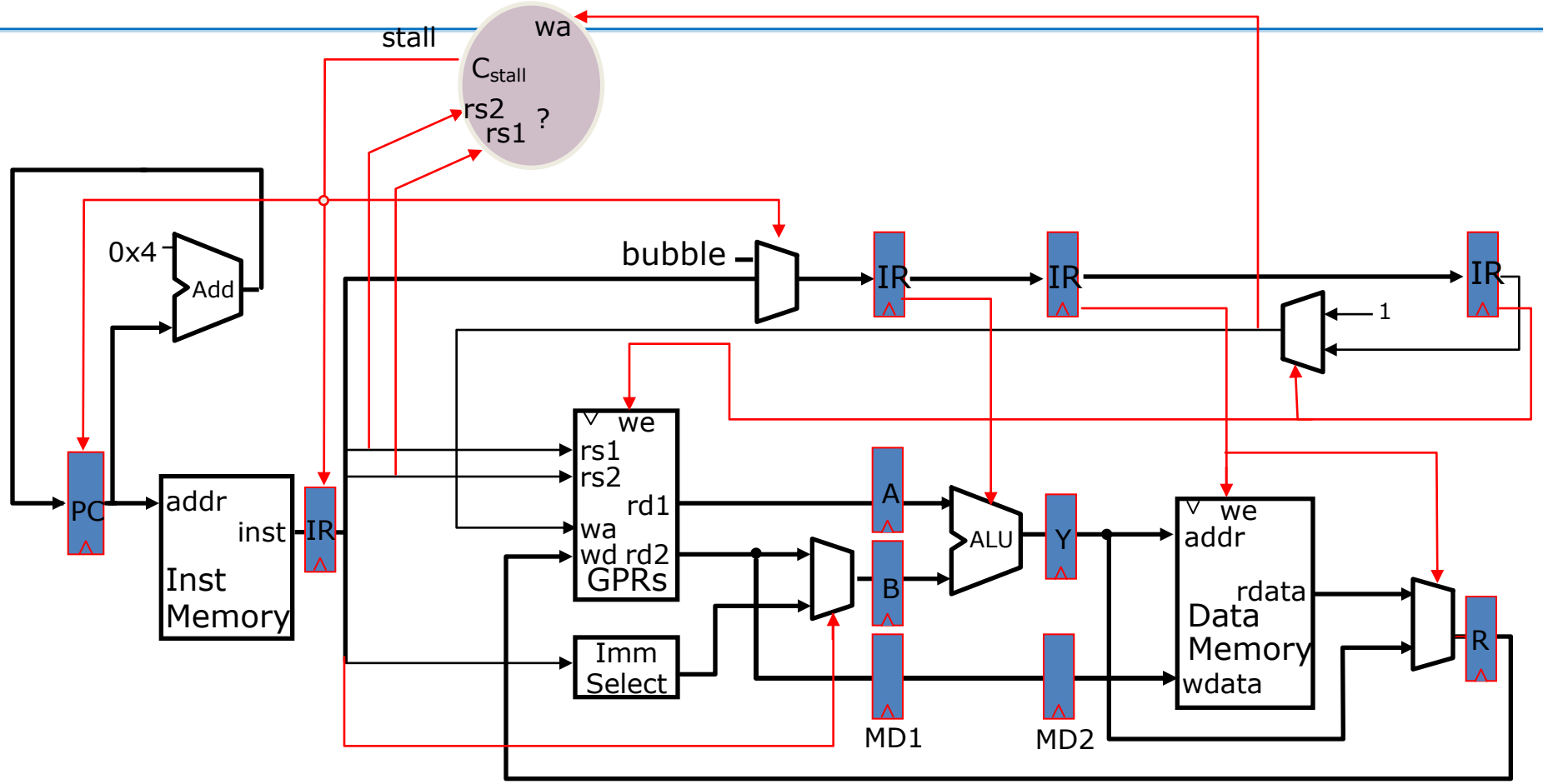
Resource Usage

timp

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .			
IF	I_1	I_2	I_3	I_3	I_3	I_3	I_4	I_5				
ID		I_1	I_2	I_2	I_2	I_2	I_3	I_4	I_5			
EX			I_1	-	-	-	I_2	I_3	I_4	I_5		
MA				I_1	-	-	-	I_2	I_3	I_4	I_5	
WB					I_1	-	-	-	I_2	I_3	I_4	I_5

- ⇒ *pipeline bubble*

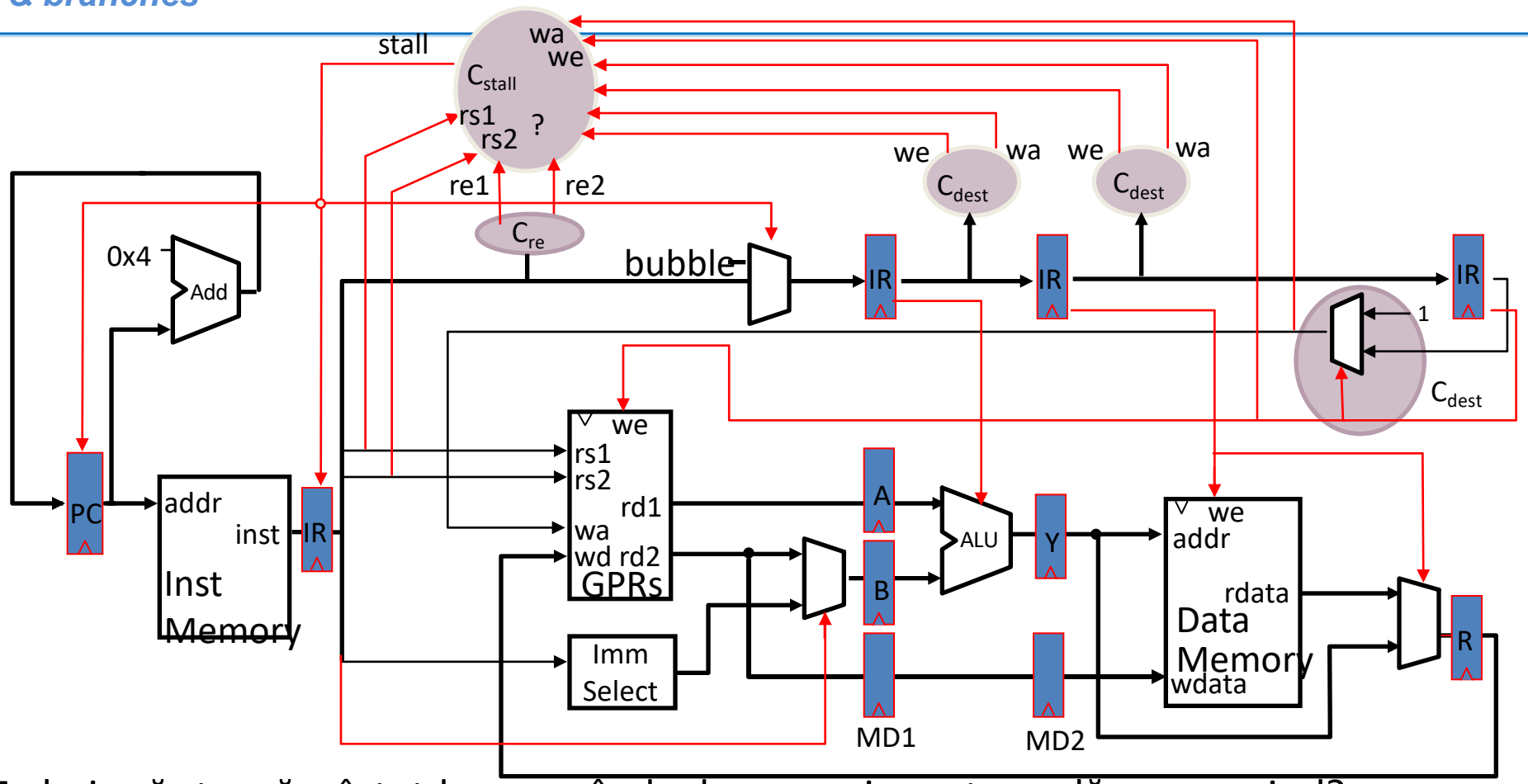
Logica de control pentru interlocking



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

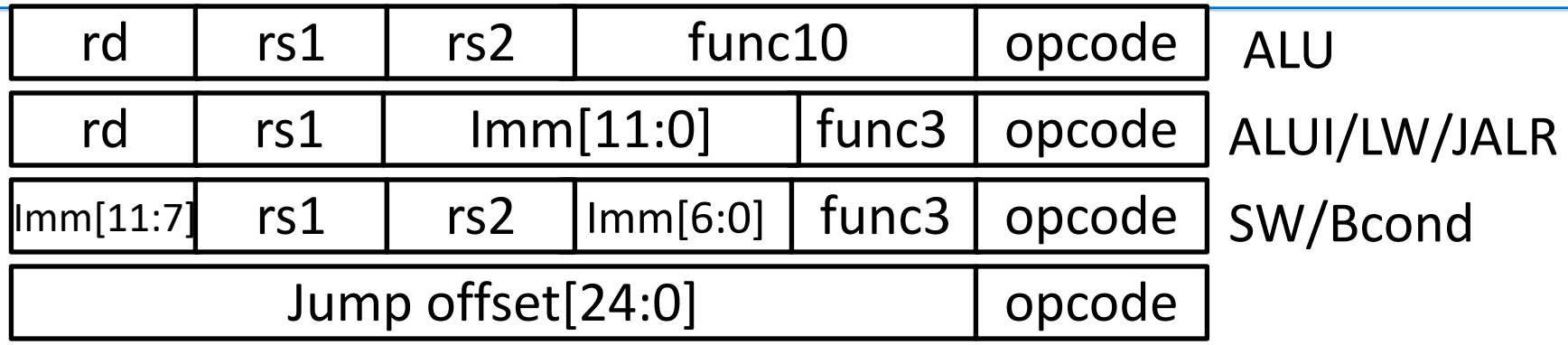


Trebuie să stagnăm întotdeauna când adresa unui rs este egală cu a unui rd?

nu orice instrucțiune scrie un registru -> we

nu orice instrucțiune citește un registru -> re

Registrele sursă și destinație



		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	-
B*cond*	rs1,rs2	rs1, rs2	-
	true: PC <- PC + imm		
	false: 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	-> X1
else	-> rd

we = *Case* opcode

ALU, ALUi, LW,JALR	->(ws != 0)
JAL	->on
...	->off

C_{re}

re1 = *Case* opcode

ALU, ALUi, LW, SW, Bcond, JALR	->on
J, JAL	->off

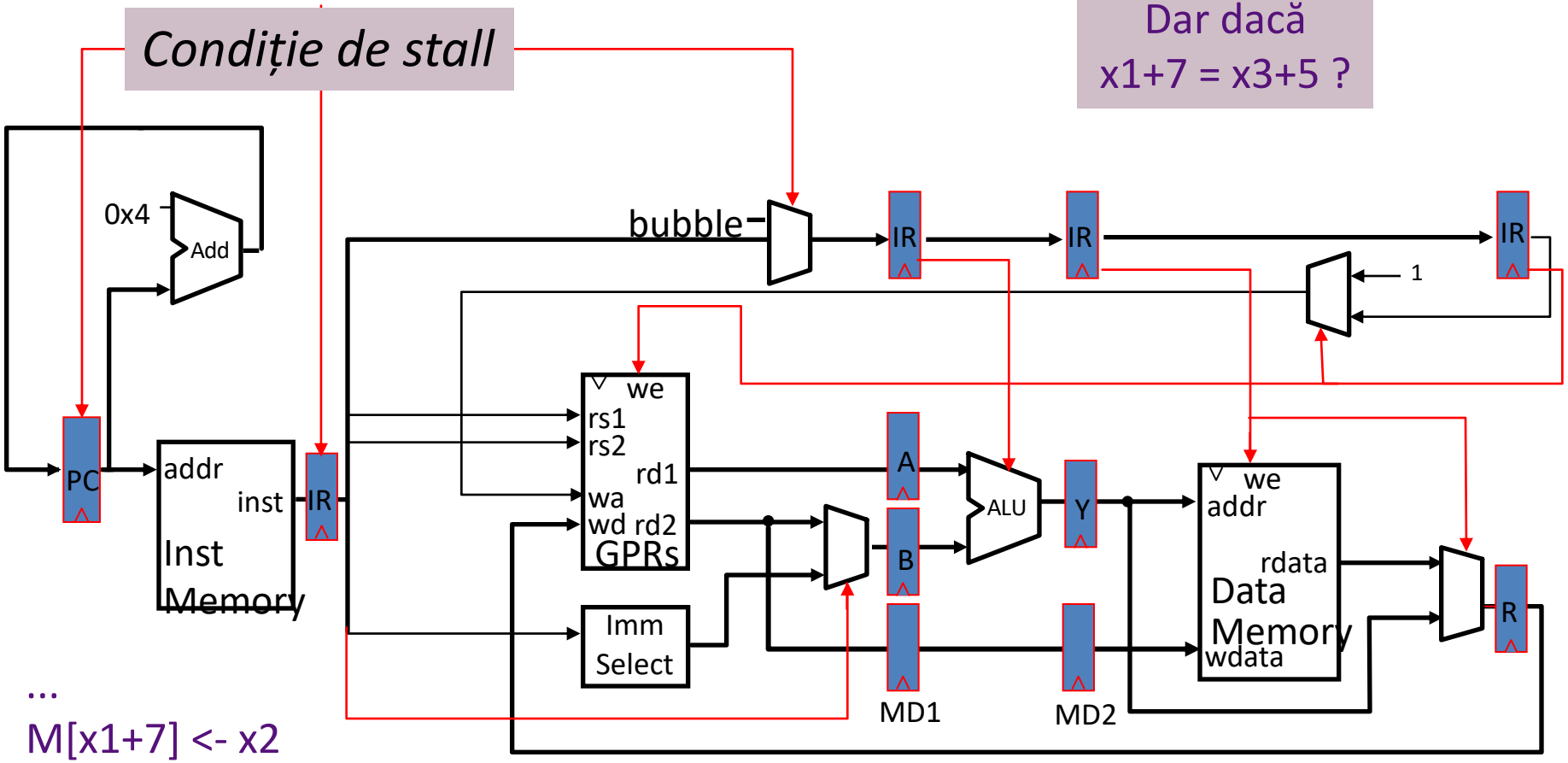
re2 = *Case* opcode

ALU, SW,Bcond	->on
...	->off

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

This is not the full story !

Hazarduri determinate de instrucțiuni Load & Store



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

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

Hazarduri Load & Store

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

x1+7 = x3+5 -> *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.

Soluționarea hazardurilor de date(2)

Strategia 2:

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

Bypassing

timp	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) x1 ← x0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) x4 ← x1 + 17		IF_2	ID_2	ID_2	ID_2	ID_2	EX_2	MA_2	WB_2
(I_3)			IF_3	IF_3	IF_3	IF_3	ID_3	EX_3	MA_3
(I_4)			*stalled stages*				IF_4	ID_4	EX_4
(I_5)								IF_5	ID_5

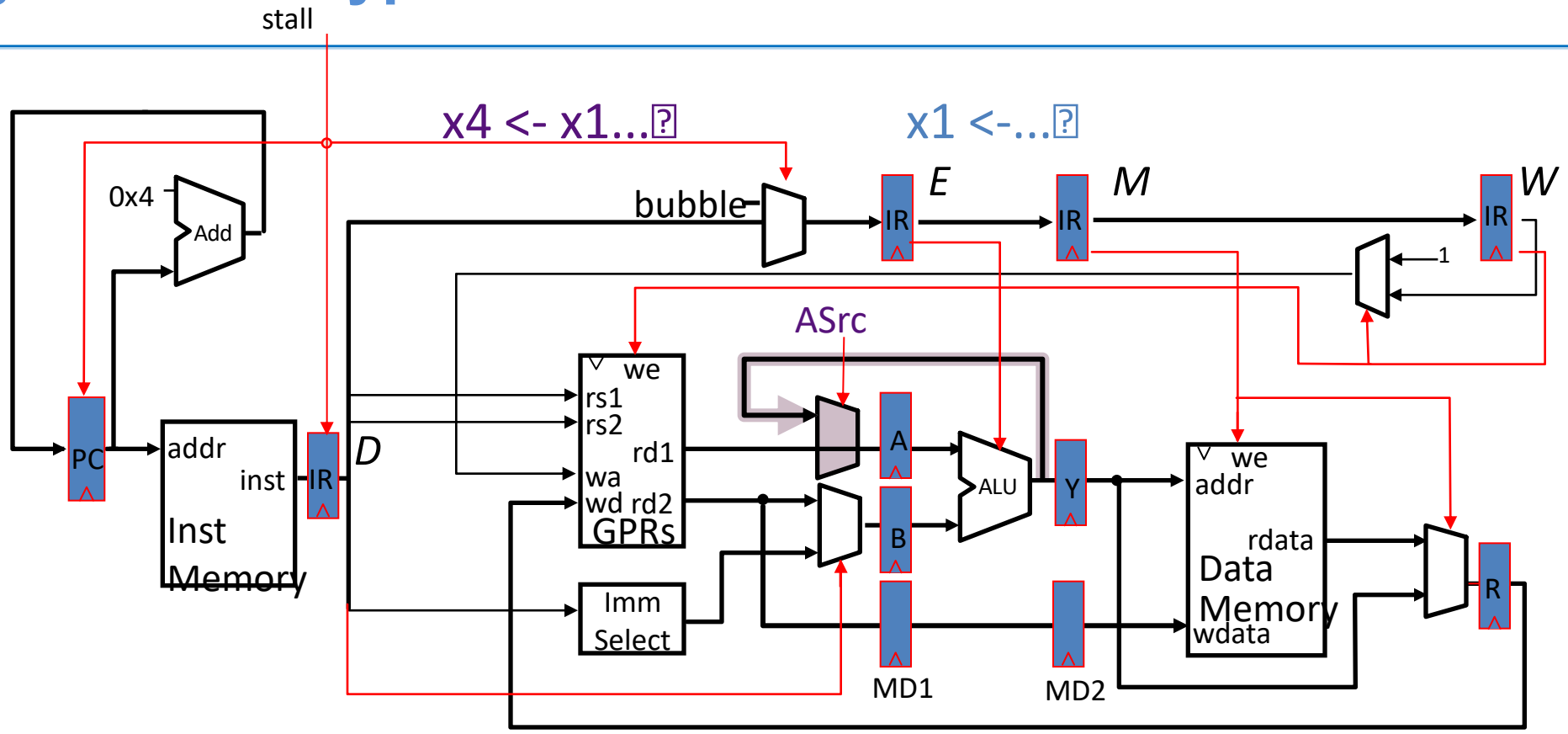
Fiecare *stall sau kill* introduce o bulă în banda de asamblare

=> *CPI > 1*

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

timp	t0	t1	t2	t3	t4	t5	t6	t7	. . . .	
(I_1) x1 ← x0 + 10		IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) x4 ← x1 + 17			IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3)				IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4)					IF_4	ID_4	EX_4	MA_4	WB_4	
(I_5)						IF_5	ID_5	EX_5	MA_5	WB_5

Adăugarea unui bypass

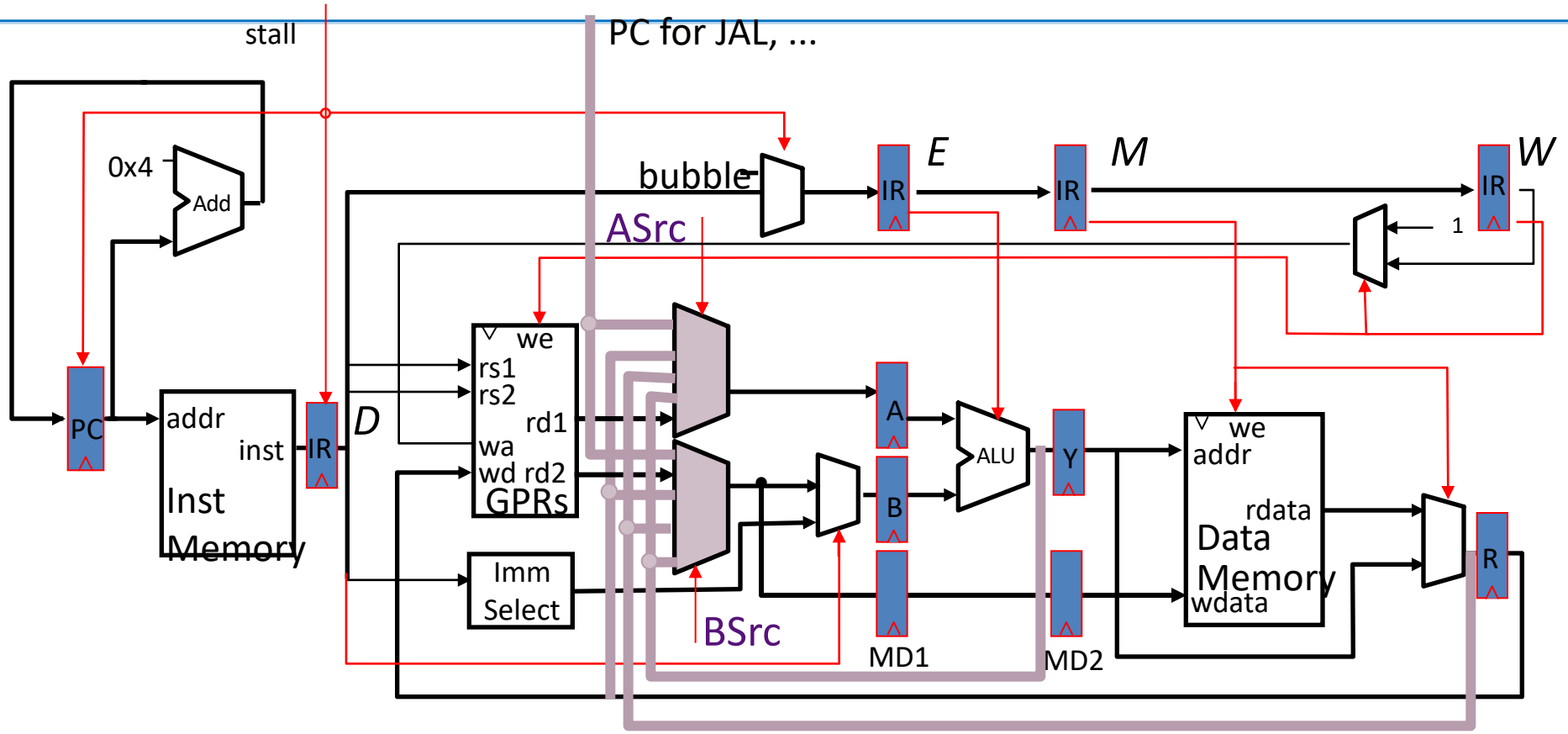


Când ajută acest bypass?

...

(I_1)	x1 <- x0 + 10	x1 <- M[x0 + 10]	JAL 500
(I_2)	x4 <- x1 + 17	x4 <- x1 + 17	x4 <- x1 + 17
	da	*nu*	*nu*

Căi de date complet separate



Mai este nevoie de un semnal de stall ?

$$stall = (rs1_D = ws_E).(opcode_E = LW_E).(ws_E != 0).re1_D + (rs2_D = ws_E).(opcode_E = LW_E).(ws_E != 0).re2_D$$

Soluționarea hazardurilor de date (3)

Strategia 3: Speculează legat de dependență!

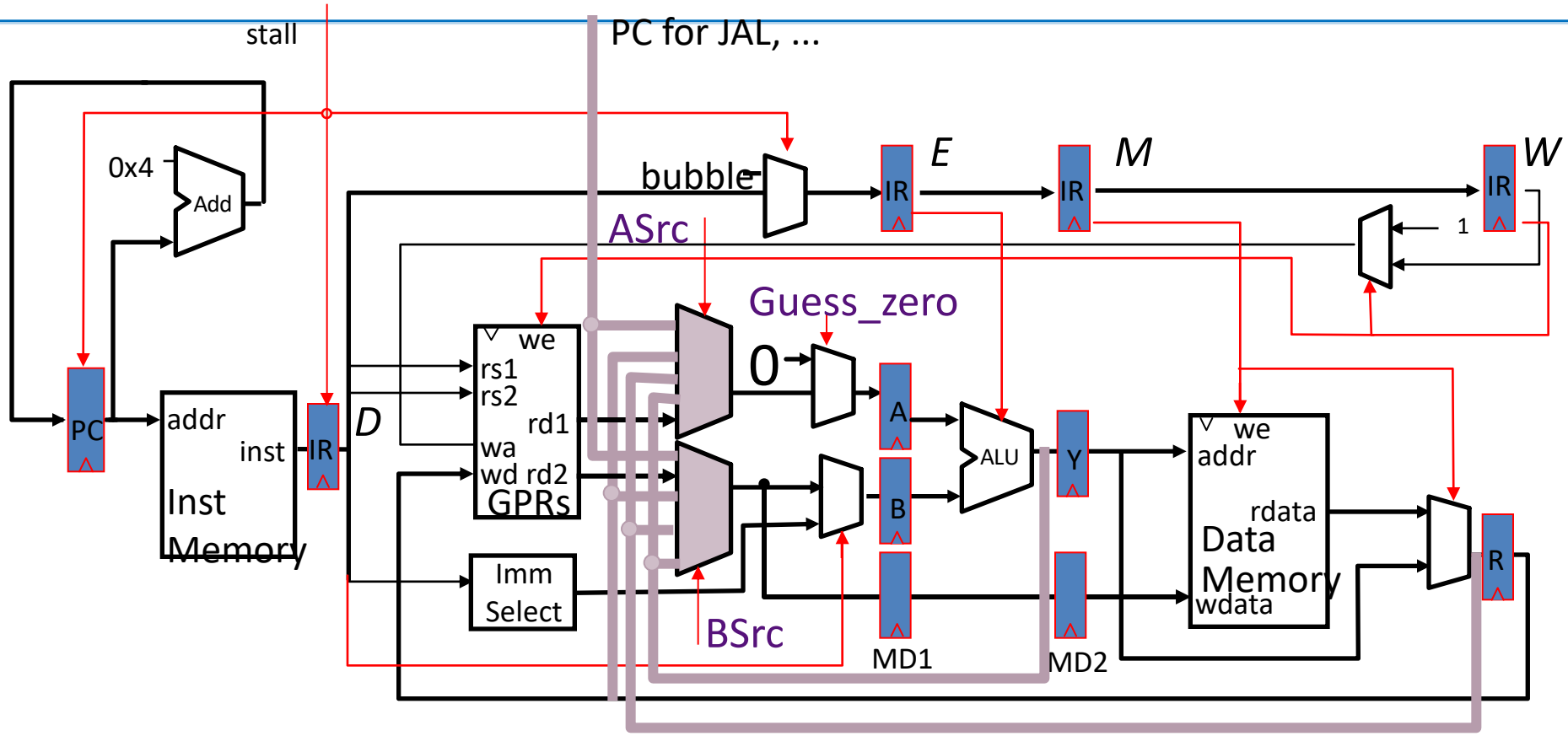
Două cazuri:

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



$$Guess\_zero = (rs1_D = ws_E).(opcode_E = LW_E).(ws_E != 0).re1_D$$

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?

Hazarduri de control

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)

Bule în calculul PC

timp

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .		
(I_1) x1 ← x0 + 10	IF_1	ID_1	EX_1	MA_1	WB_1						
(I_2) x3 ← x2 + 17		IF_2	IF_2	ID_2	EX_2	MA_2	WB_2				
(I_3)				IF_3	IF_3	ID_3	EX_3	MA_3	WB_3		
(I_4)						IF_4	IF_4	ID_4	EX_4	MA_4	WB_4

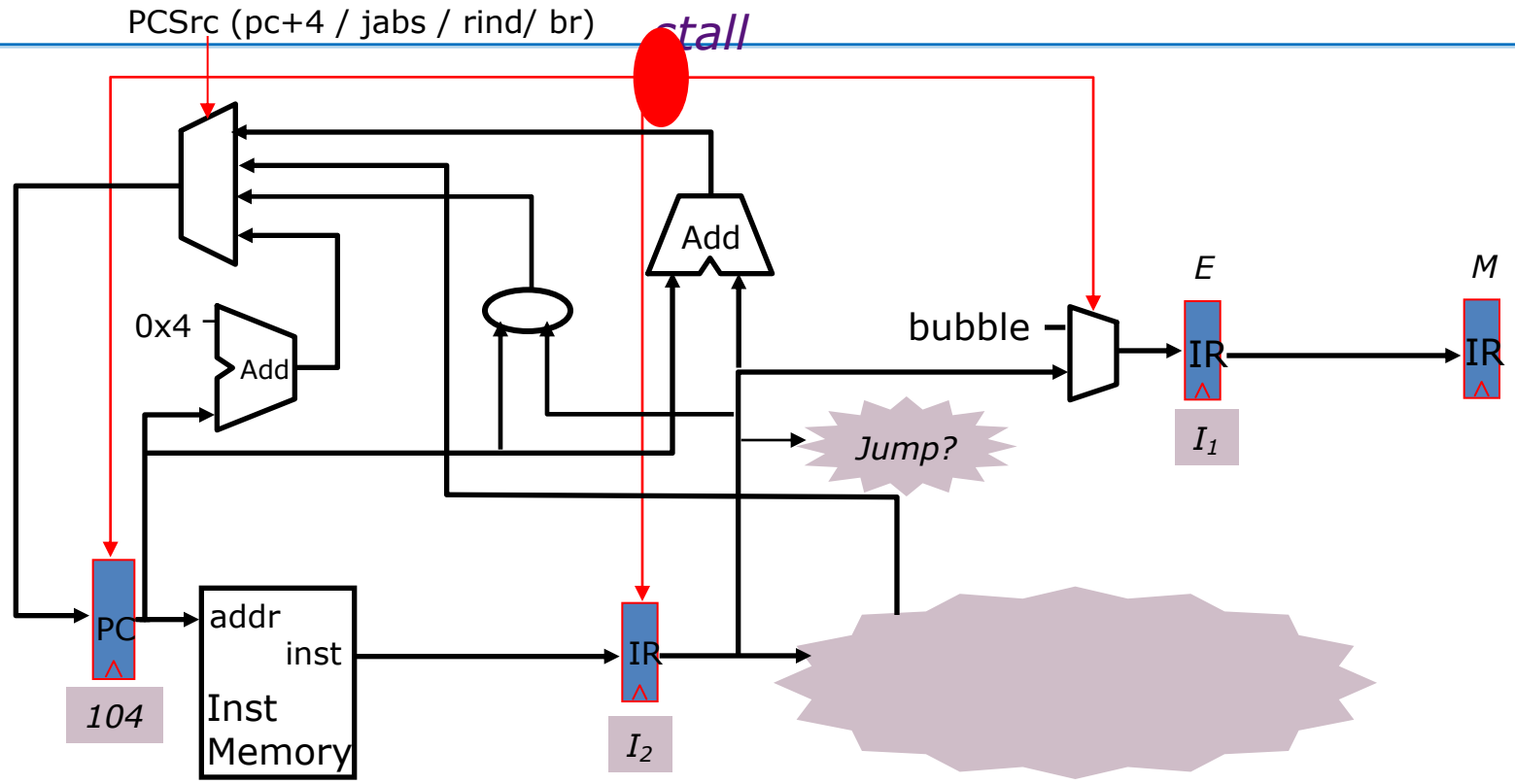
time

Resource Usage

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .		
IF	I_1	-	I_2	-	I_3	-	I_4				
ID		I_1	-	I_2	-	I_3	-	I_4			
EX			I_1	-	I_2	-	I_3	-	I_4		
MA				I_1	-	I_2	-	I_3	-	I_4	
WB					I_1	-	I_2	-	I_3	-	I_4

- ⇒ *pipeline bubble*

Speculăm că următoarea adresă e PC+4



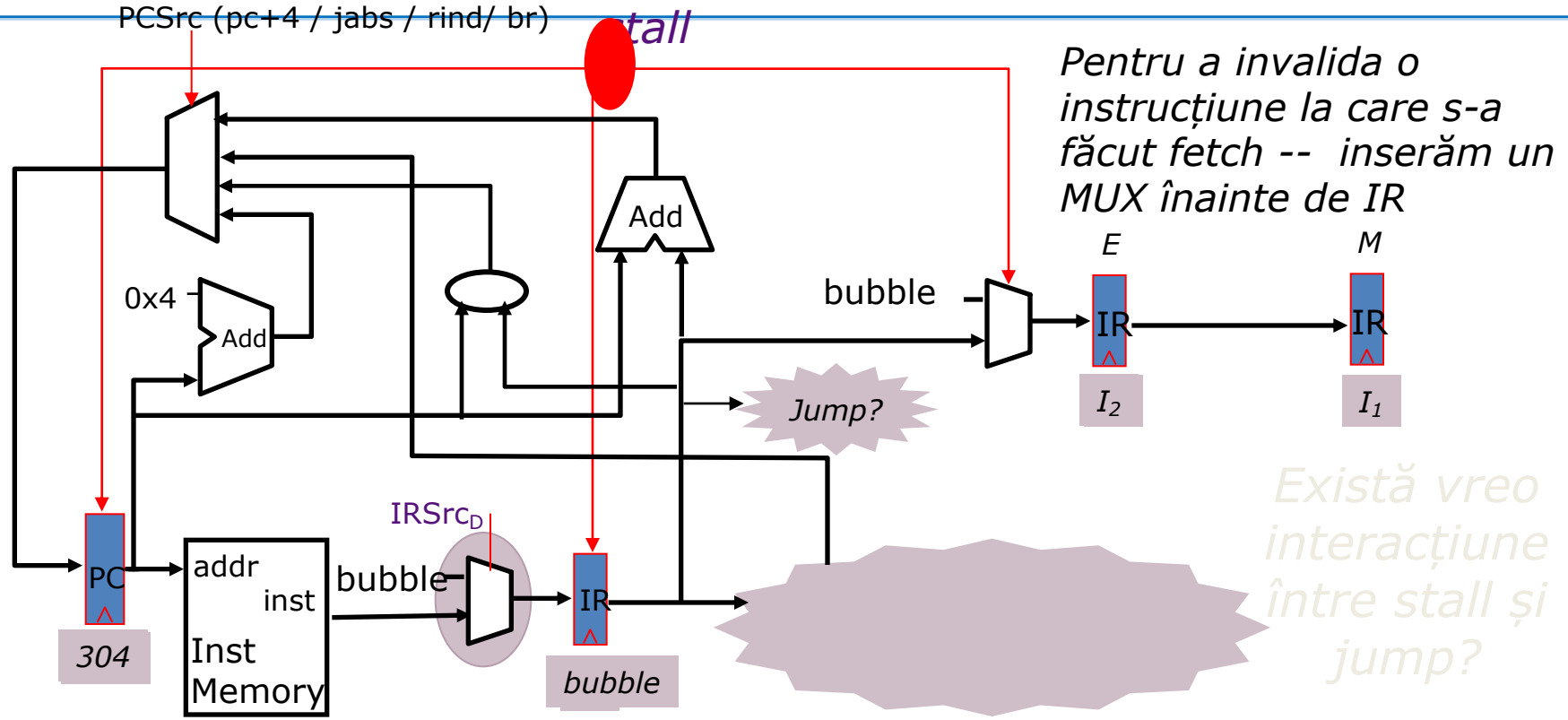
I_1	096	ADD
I_2	100	J 304
I_3	~~104~~	~~ADD~~
I_4	304	ADD

kill

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	~~104~~	~~ADD~~
I_4	304	ADD

kill

$IRSrc_D$ = *Case* $opcode_D$
- J, JAL ⇒ bubble
- ... ⇒ IM

Diagrame pipeline pentru Jump

timp

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: J 304		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	-	-	-	-		
(I_4) 304: ADD				IF_4	ID_4	EX_4	MA_4	WB_4	

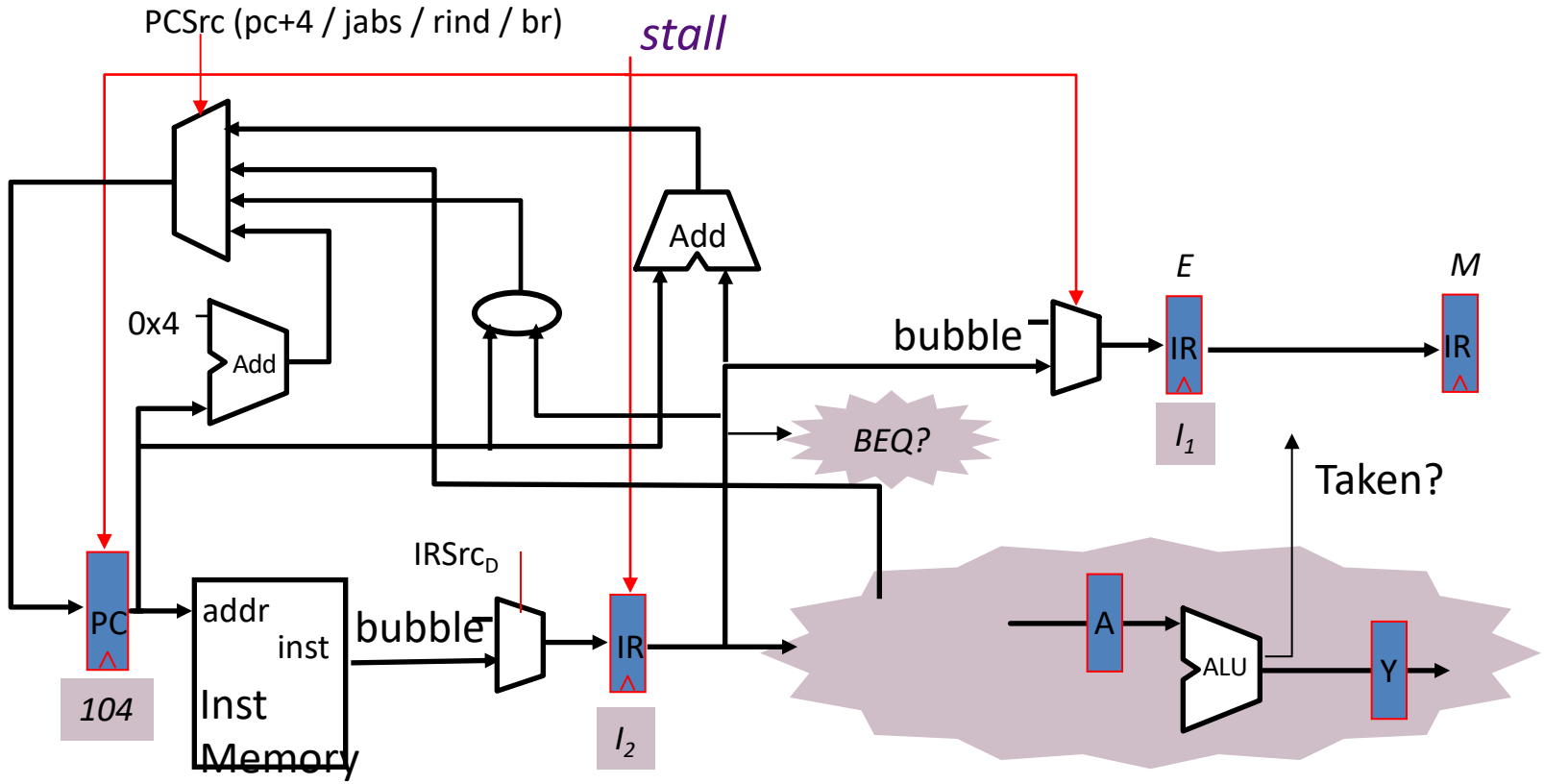
Resource Usage

timp

	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	I_2	I_3	I_4	I_5				
ID		I_1	I_2	-	I_4	I_5			
EX			I_1	I_2	-	I_4	I_5		
MA				I_1	I_2	-	I_4	I_5	
WB					I_1	I_2	-	I_4	I_5

- $\Rightarrow$ *pipeline bubble*

Implementarea în b.a. a salturilor condiționale

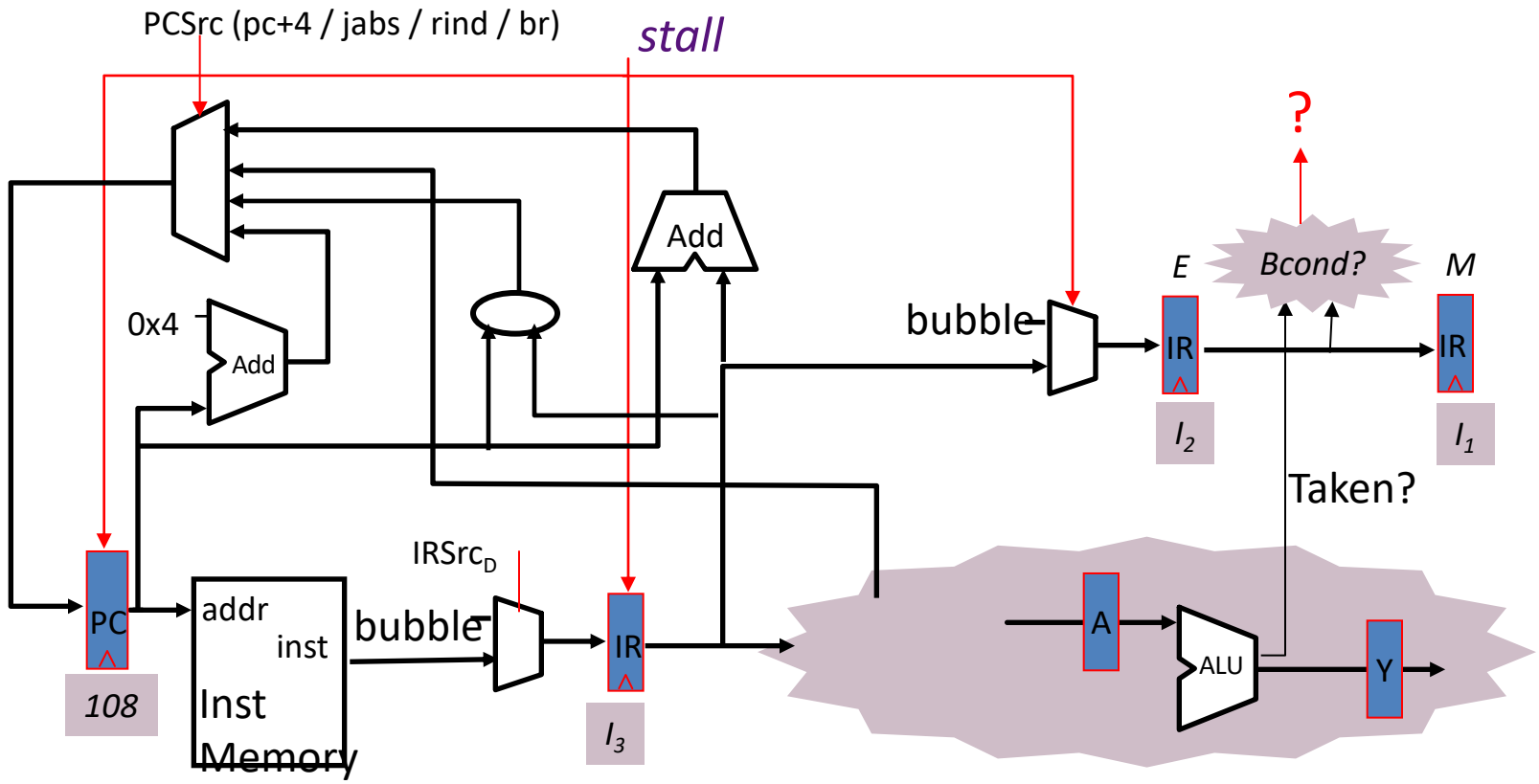


I_1	096	ADD
I_2	100	BEQ x1,x2 +200
I_3	104	ADD
I_4	304	ADD

Condiția de brach nu e cunoscută până la faza de execute

Ce acțiuni putem întreprinde în faza de decode?

Implementarea în b.a. a salturilor condiționale



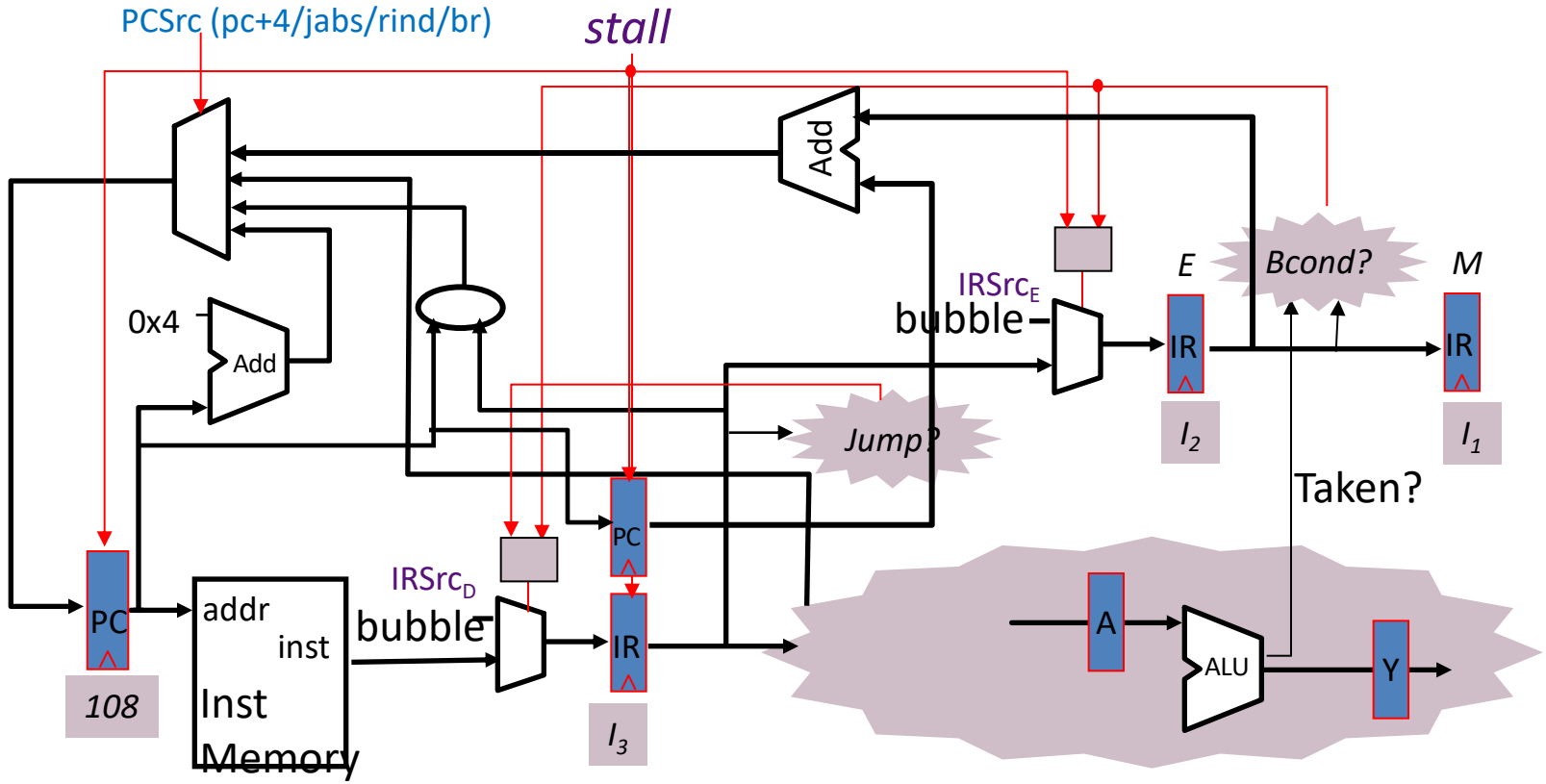
I_1	096	ADD
I_2	100	BEQ x1,x2 +200
I_3	104	ADD
I_4	304	ADD

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



$I_{1:}$	096	ADD	Dacă branch-ul este valid
$I_{2:}$	100	BEQ x1,x2 +200	- Omoară următoarele 2 instrucțiuni
$I_{3:}$	104	ADD	- Instrucțiunea din faza de decode nu este validă
$I_{4:}$	304	ADD	⇒ *semnalul stall nu este valid*

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

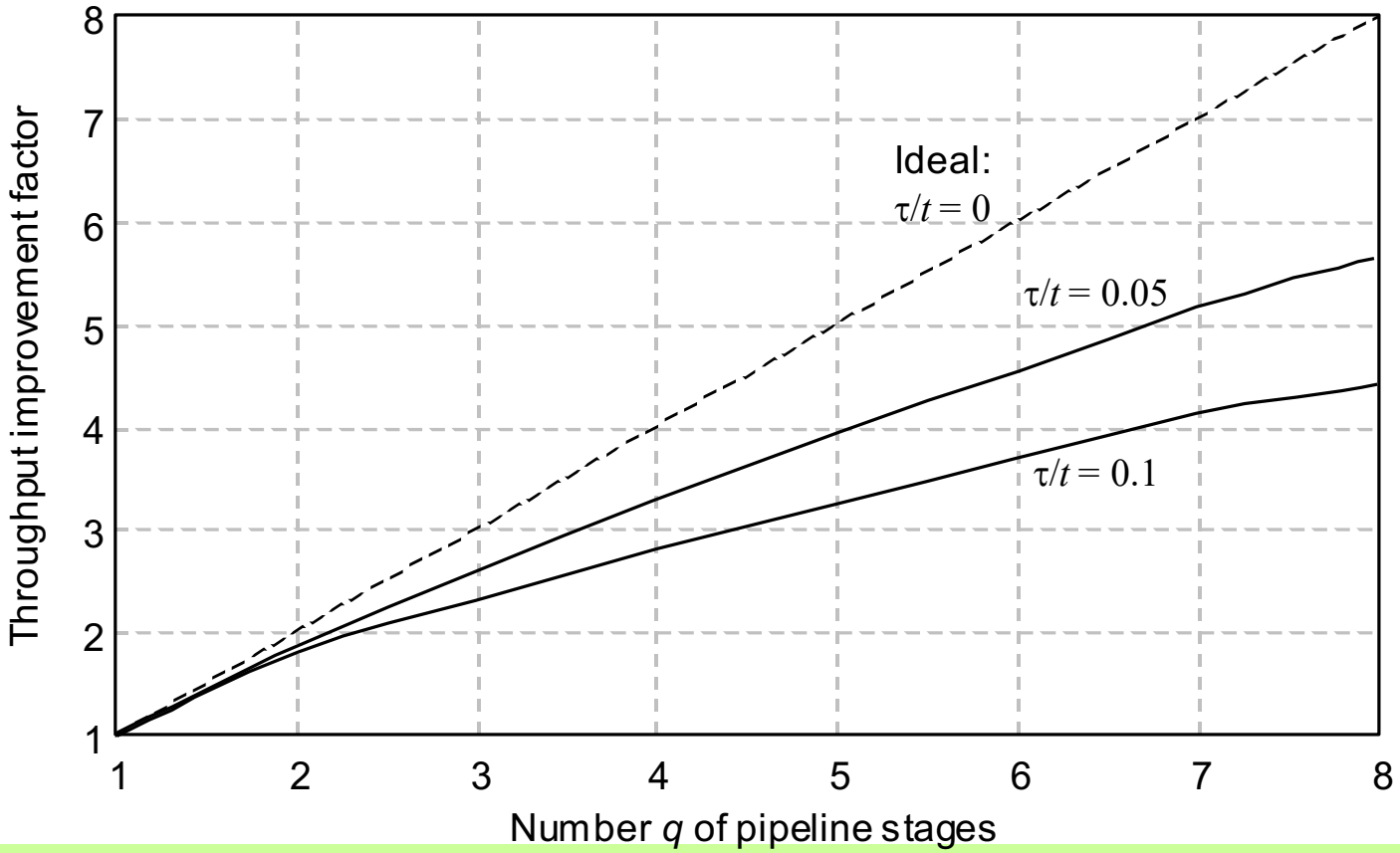
	timp t0	t1	t2	t3	t4	t5	t6	t7	. . . .
(I_1) 096: ADD	IF_1	ID_1	EX_1	MA_1	WB_1				
(I_2) 100: BEQ +200		IF_2	ID_2	EX_2	MA_2	WB_2			
(I_3) 104: ADD			IF_3	ID_3	-	-	-		
(I_4) 108:				IF_4	-	-	-	-	
(I_5) 304: ADD					IF_5	ID_5	EX_5	MA_5	WB_5

Resource Usage

	timp t0	t1	t2	t3	t4	t5	t6	t7	. . . .
IF	I_1	I_2	I_3	I_4	I_5				
ID		I_1	I_2	I_3	-	I_5			
EX			I_1	I_2	-	-	I_5		
MA				I_1	I_2	-	-	I_5	
WB					I_1	I_2	-	-	I_5

\- $\Rightarrow$ *pipeline bubble*

Creșterea de productivitate a unei b.a. cu q etape



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

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.

Productivitatea b.a. afectată de hazarduri

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

Notăm cu β 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: $\beta = 0.25(0.24 + 0.18) = 0.105$

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