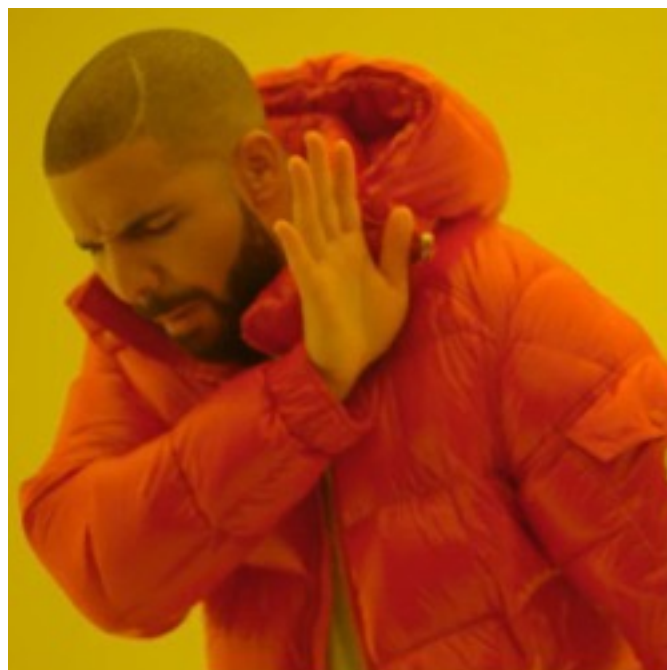


# Calculatoare Numerice (2)

- Cursul 9 –

ILP & Superscalar

Facultatea de Automatică și Calculatoare  
Universitatea Politehnica București



Performance  
by  
parallelism



Performance  
by  
pipelining

# Din episoadele anterioare

---

- Sistemele moderne de memorie paginată pun la dispoziție:
  - Translație, Protecție, Memorie Virtuală.
- Informațiile legate de translație și protecție stocate în tabele de pagini, ținute în memoria principală
- Informațiile legate de translație și protecție caching în “translation-lookaside buffer” (TLB) pentru a permite translatarea + verificarea protecției într-un singur ciclu de ceas, în majoritatea cazurilor
- Memoria virtuală interacționează cu design-ul memoriei cache



# Complex Pipelining: Motivație

---

Banda de asamblare devine complexă atunci când avem nevoie de performanță mărită pentru:

- Unități floating-point cu latență mare sau care sunt doar parțial în b.a.
- Sisteme de memorie cu timp variabil de acces
- Unități aritmetice și de memorie multiple



# Floating-Point Unit (FPU)

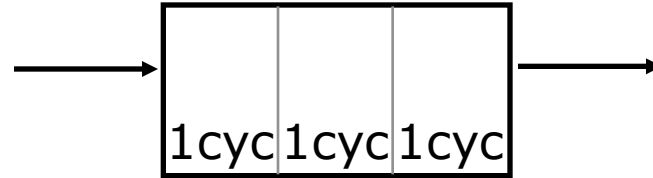
---

- Mult mai mult hardware decât o unitate pe întregi
  - Single-cycle FPU nu este o idee fezabilă
- E comun să avem mai multe FPU-uri
- E comun să avem diferite tipuri de FPU-uri: Fadd, Fmul, Fdiv, ...
- Un FPU poate să fie în b.a., parțial în b.a. sau fără b.a.
- Pentru a folosi concurent mai multe FPU-uri, trebuie să avem nevoie de o tabelă de registre dedicată (FPR) care să aibă mai multe porturi de citire și de scriere

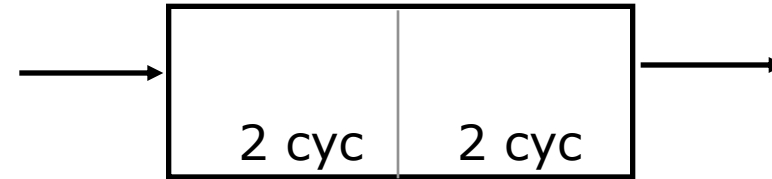


# Caracteristici pentru unitățile funcționale

*fully  
pipelined*



*partially  
pipelined*



Unitățile funcționale au registre interne dedicate b.a.

- ⇒ operanzii sunt memorați atunci când o instrucțiune intră în unitatea funcțională
- ⇒ instrucțiunile următoare pot rescrie tabela de registre în timpul operațiilor cu latență mare

# Floating-Point ISA

---

- Interacțiunea dintre căile de date pentru întregi și floating-point sunt determinate de ISA
- RISC-V ISA
  - tabele de registre separate pentru instrucțiuni FP și cu întregi
    - Singura interacțiune este printr-un set de instrucțiuni tip move/convert (unele ISA nu permit nici măcar asta)
  - load/store separate pentru FPR și GPR, dar ambele folosesc GPR pentru calculul adreselor

# Sisteme de memorie reale

---

Abordări comune pentru îmbunătățirea performanței memoriei:

- Cache-uri - single cycle, mai puțin în cazul unui miss
  - stall
- Memorie pe bancuri – accese multiple la memorie
  - conflicte pe bancuri
- Operații în două faze (separă cererea la memorie de răspuns), majoritatea fără pregătiri inițiale
  - răspunsuri out-of-order

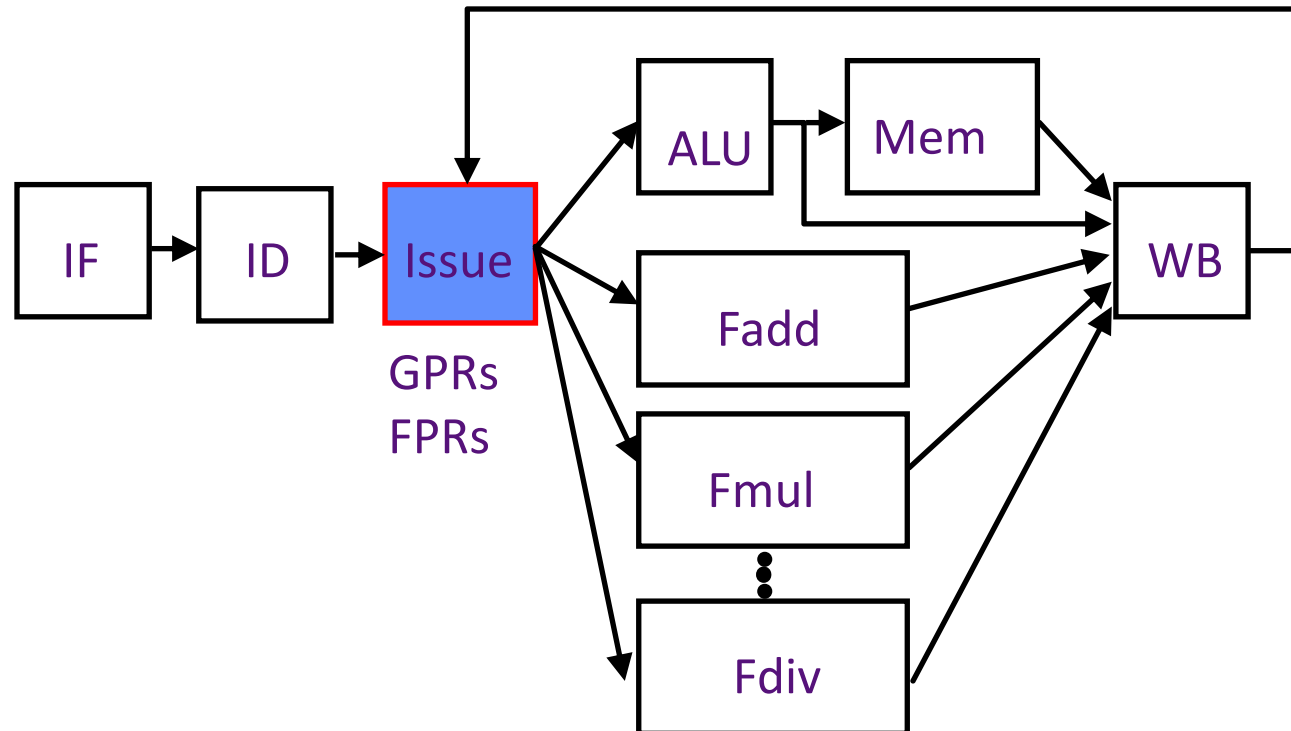
Latența acceselor la memoria principală este de obicei mult mai mare de un ciclu, și de cele mai multe ori, total imprevizibilă

*Găsirea unei soluții este o problemă centrală pentru arhitectura calculatoarelor*



# Probleme în controlul b.a. complexe

- Conflicte structurale în etapa de execuție dacă un FPU sau o unitate de memorie nu este în b.a. și durează mai mult de un ciclu
- Conflicte structurale la write-back datorate latențelor variabile ale diferitelor unități funcționale
- Hazarde la un write out-of-order datorate latențelor diferitelor unități funcționale
- Cum să rezolvăm excepțiile?

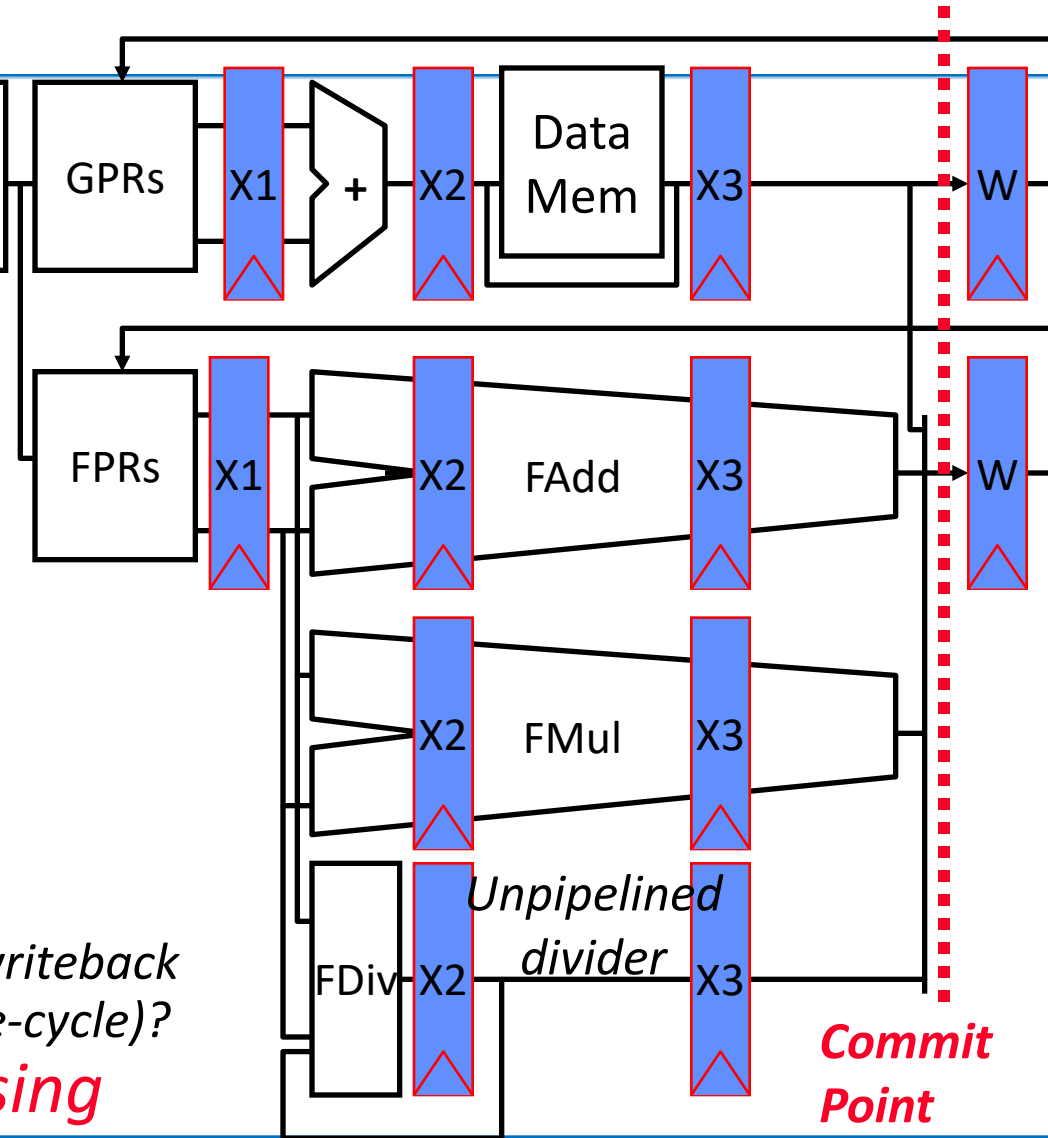


# Complex In-Order Pipeline

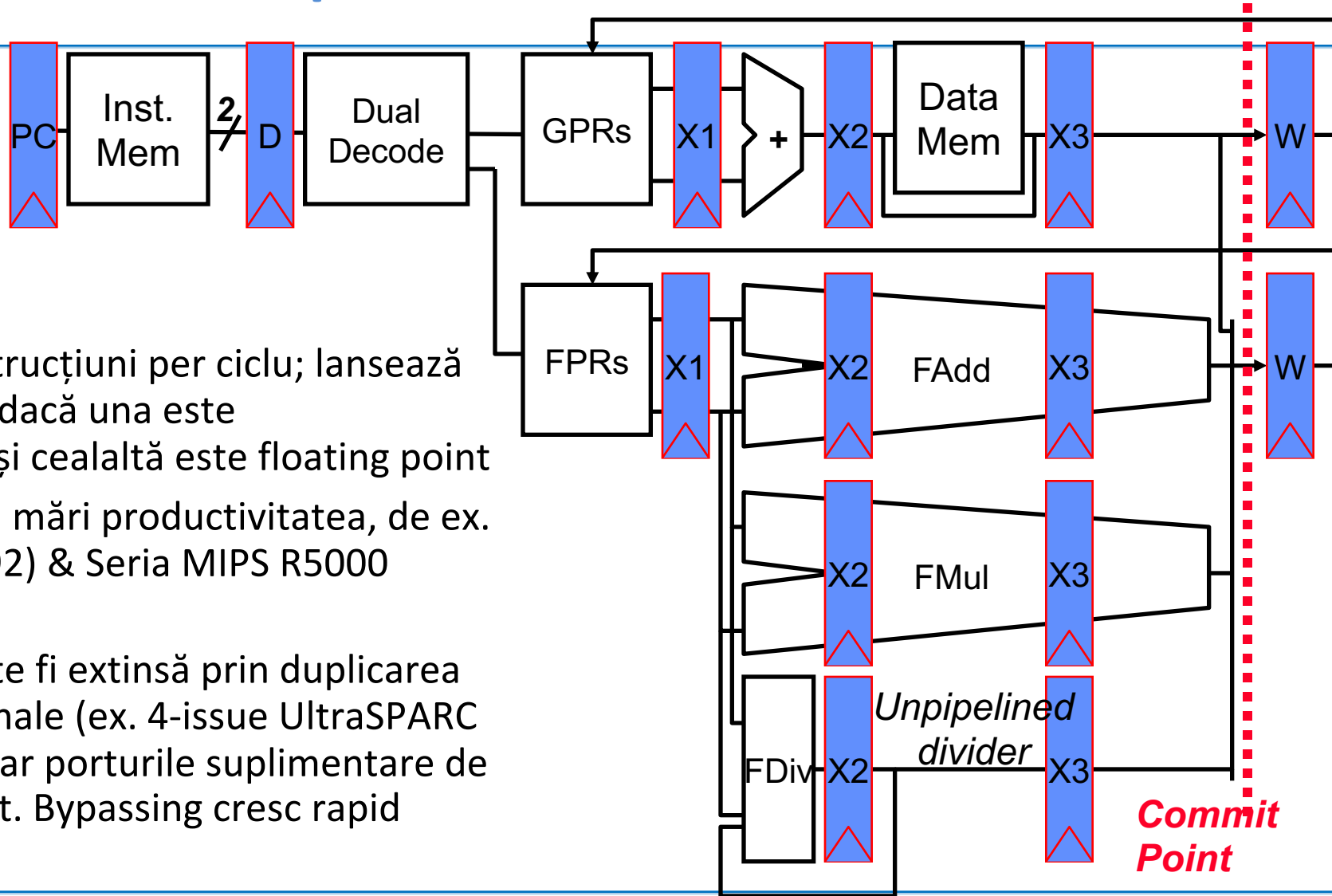
- Întârzie writeback a.î. toate operațiile au aceeași latență la etapa W
  - Porturile de Write nu sunt niciodată supra-aglomerate (one inst. in & one inst. out la fiecare ciclu)
  - Întârzie b.a. la op. cu latență mare ex.: divide, cache miss
  - Tratează excepțiile in-order la commit point

*Cum putem să prevenim ca latența mare la writeback să afecteze operațiile simple pe întregi (single-cycle)?*

**Bypassing**



# In-Order Superscalar Pipeline



- Fetch la două instrucțiuni per ciclu; lansează ambele simultan dacă una este integer/memory și cealaltă este floating point
- Mod eficient de a mări productivitatea, de ex. Alpha 21064 (1992) & Seria MIPS R5000 (1996)
- Aceeași idee poate fi extinsă prin duplicarea unităților funcționale (ex. 4-issue UltraSPARC & Alpha 21164) dar porturile suplimentare de GPR și costurile pt. Bypassing cresc rapid

# Tipuri de hazarde de date

Luați, de exemplu, următorul tip de operație

$$r_k \rightarrow r_i \text{ op } r_j$$

Dependențe de date

$$\begin{array}{l} r_3 \rightarrow r_1 \text{ op } r_2 \\ r_5 \rightarrow r_3 \text{ op } r_4 \end{array}$$

Read-after-Write  
(RAW) hazard

Anti-dependență

$$\begin{array}{l} r_3 \rightarrow r_1 \text{ op } r_2 \\ r_1 \rightarrow r_4 \text{ op } r_5 \end{array}$$

Write-after-Read  
(WAR) hazard

Dependență de ieșiri

$$\begin{array}{l} r_3 \rightarrow r_1 \text{ op } r_2 \\ r_3 \rightarrow r_6 \text{ op } r_7 \end{array}$$

Write-after-Write  
(WAW) hazard



## Dependențe de registre vs. memorie

---

Hazardele de date datorate operanzilor din registre pot fi depistate în etapa de decode, dar hazardele datorate operanzilor din memorie pot fi determinate numai după calculul adresei efective

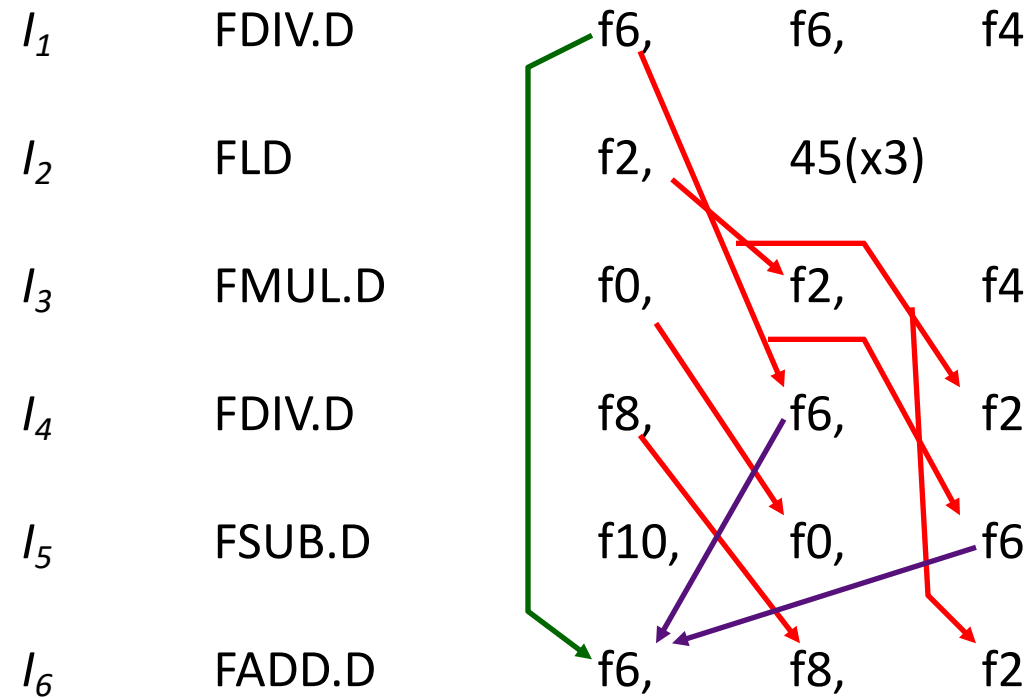
Store:         **$M[r1 + disp1] \leftarrow r2$**

Load:         **$r3 \leftarrow M[r4 + disp2]$**

Unde apare aici hazardul?

Oare  **$(r1 + disp1) = (r4 + disp2)$**  ?

# Hazarde de date, un exemplu

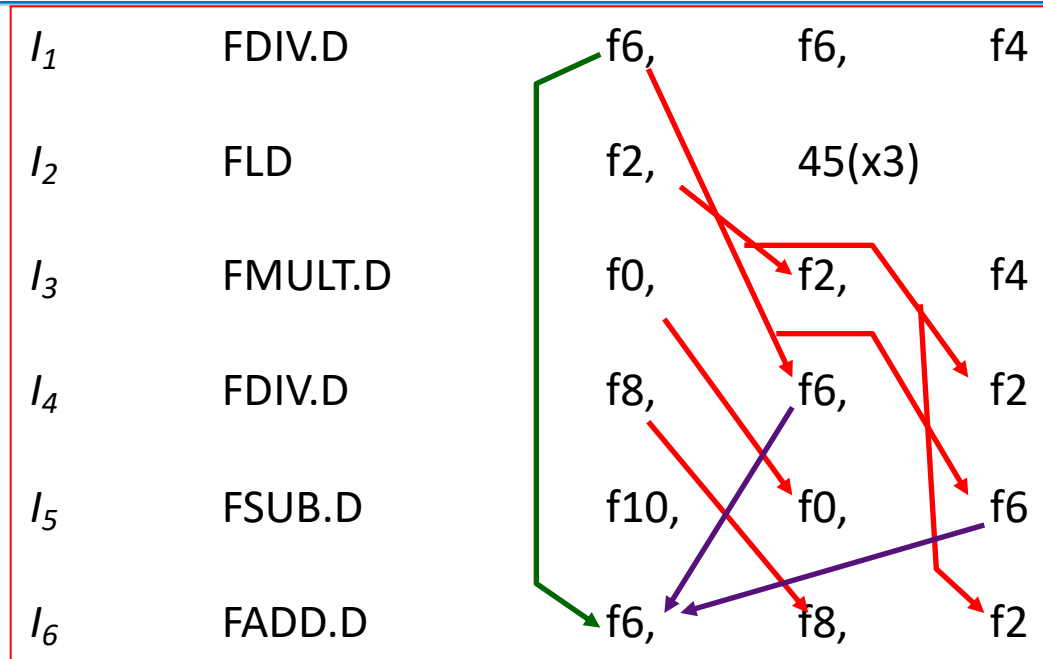


*RAW Hazards*

*WAR Hazards*

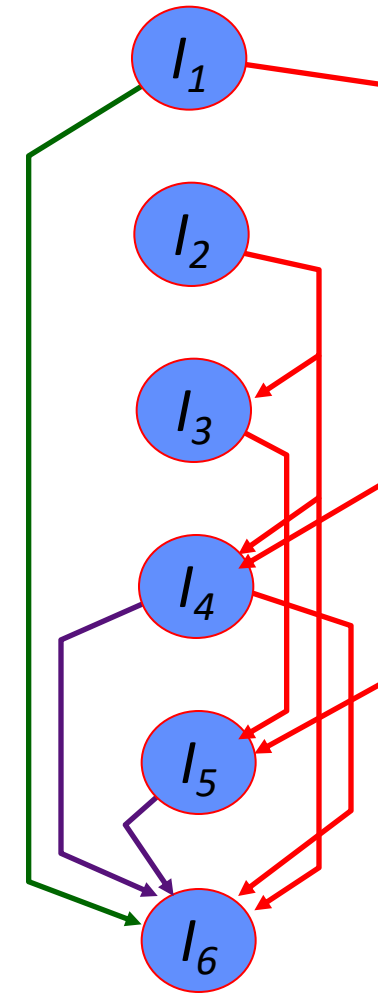
*WAW Hazards*

# Instruction Scheduling



Ordonări valide:

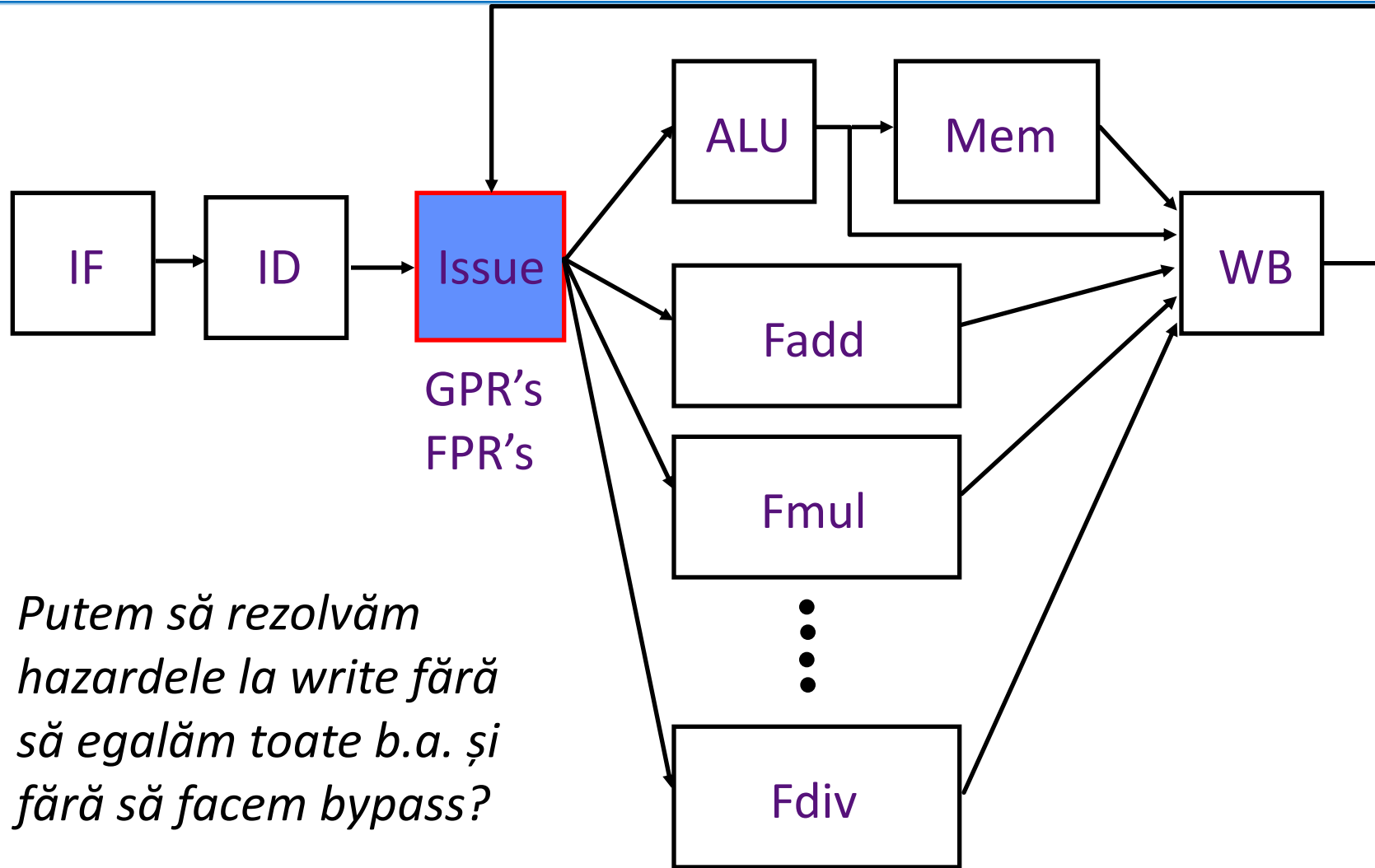
<i>in-order</i>	$I_1$	$I_2$	$I_3$	$I_4$	$I_5$	$I_6$
<i>out-of-order</i>	$I_2$	$I_1$	$I_3$	$I_4$	$I_5$	$I_6$
<i>out-of-order</i>	$I_1$	$I_2$	$I_3$	$I_5$	$I_4$	$I_6$







# B.A. Complexe



*Putem să rezolvăm  
hazardele la write fără  
să egalăm toate b.a. și  
fără să facem bypass?*

# Când este sigur să lansăm o instrucțiune?

---

Presupunem că o structură de date ține evidența tuturor instrucțiunilor din toate unitățile funcționale

Trebuie făcute următoarele verificări înainte de a lansa în execuție o instrucțiune:

- Este disponibilă unitate funcțională cerută?
- Sunt disponibile datele de intrare? → RAW?
- Este sigur să scrii la destinație? → WAR? → WAW?
- Există vreun conflict structural în etapa de WB?



# Structură de date pentru execuții corecte

Ține evidența tuturor unităților funcționale

<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Dest</i>	<i>Src1</i>	<i>Src2</i>
Int					
Mem					
Add1					
Add2					
Add3					
Mult1					
Mult2					
Div					

*Instrucțiunea i, la lansare în execuție, consultă tabela*

FU available?

check the busy column

RAW?

search the dest column for i's sources

WAR?

search the source columns for i's destination

WAW?

search the dest column for i's destination

*Se adaugă o intrare în tabelă dacă nu se detectează nici un hazard*

*Intrarea este scoasă din tabelă după Write-Back*



# Simplificarea structurii de date

## Presupunem execuție In-order

---

Presupunem că instrucțiunea nu este trimisă în execuție dacă există un hazard RAW sau unit. funcț. este ocupată, și că operanzii sunt memorați intern de către unit. funcț. la primire:

Instrucțiunea trimisă poate să cauzeze:

WAR hazard ?

*NU: Operanzii sunt citați la lansare în execuție*

WAW hazard ?

*DA: Out-of-order completion*

# Simplificăm structura de date ...

---

- Nu avem hazard WAR
  - > nu e nevoie să memorăm src1 și src2
- Nu se lansează în execuție o operație în cazul unui hazard WAW
  - > numele unui registru poate apare cel mult odată în coloana *dest*
- WP[reg#] : vector de biți pentru memorarea regitrelor în care se vor face operații de scriere (Write Pending)
  - Biți setați la lansare și șterși în etapa de WB
  - > Fiecare etapă din b.a. a fiecărei unit. funcț. trebuie să conțină câmpul dest și un flag care să indice dacă acesta este valid “perechea (we, ws)”

# Tabelă de scoruri pentru execuție In-order

---

Busy[FU#] : vector de biți ce indică disponibilitatea FU.

(FU = Int, Add, Mult, Div)

Acești biți sunt hardcodați în FU.

WP[reg#] : vector de biți care să înregistreze dacă scrierile în reg. sunt în așteptare (Write Pending).

Acești biți sunt setați la lansarea în execuție a op. și șterși la Write Back

Unitatea verifică instrucțiunea (opcode dest src1 src2) în tabelă înainte de a o lansa în execuție

FU available?

RAW?

WAR?

WAW?

Busy[FU#]

WP[src1] sau WP[src2]

*nu poate apare*

WP[dest]

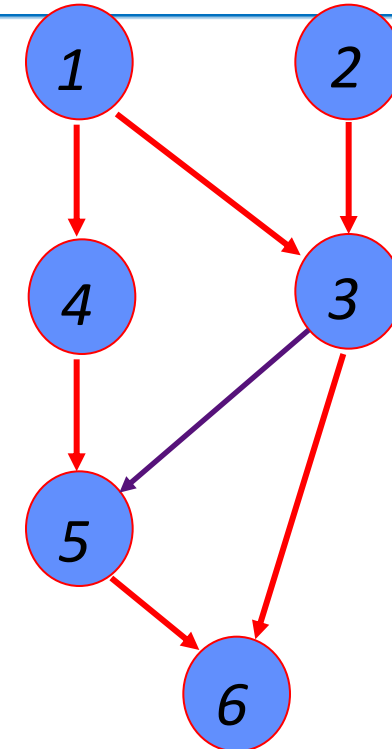
# Dinamica tabelii de scoruri

	Functional Unit Status					Registers Reserved for Writes	
	Int(1)	Add(1)	Mult(3)	Div(4)	WB		
t0	$I_1$			f6			f6
t1	$I_2$ f2			f6			f6, f2
t2					f6	f2	f6, f2 $I_2$
t3	$I_3$		f0			f6	f6, f0
t4			f0			f6	f6, f0 $I_1$
t5	$I_4$			f0 f8			f0, f8
t6					f8	f0	f0, f8 $I_3$
t7	$I_5$	f10				f8	f8, f10
t8						f8 f10	f8, f10 $I_5$
t9						f8	f8 $I_4$
t10	$I_6$	f6					f6
t11						f6	f6 $I_6$

$I_1$	FDIV.D	f6,	f6,	f4
$I_2$	FLD	f2,	45(x3)	
$I_3$	FMULT.D	f0,	f2,	f4
$I_4$	FDIV.D	f8,	f6,	f2
$I_5$	FSUB.D	f10,	f0,	f6
$I_6$	FADD.D	f6,	f8,	f2

# Limitările execuției In-Order: un exemplu

					<i>latency</i>
1	FLD	f2,	34(x2)		1
2	FLD	f4,	45(x3)		<i>long</i>
3	FMULT.D	f6,	f4,	f2	3
4	FSUB.D	f8,	f2,	f2	1
5	FDIV.D	f4,	f2,	f8	4
6	FADD.D	f10,	f6,	f4	1

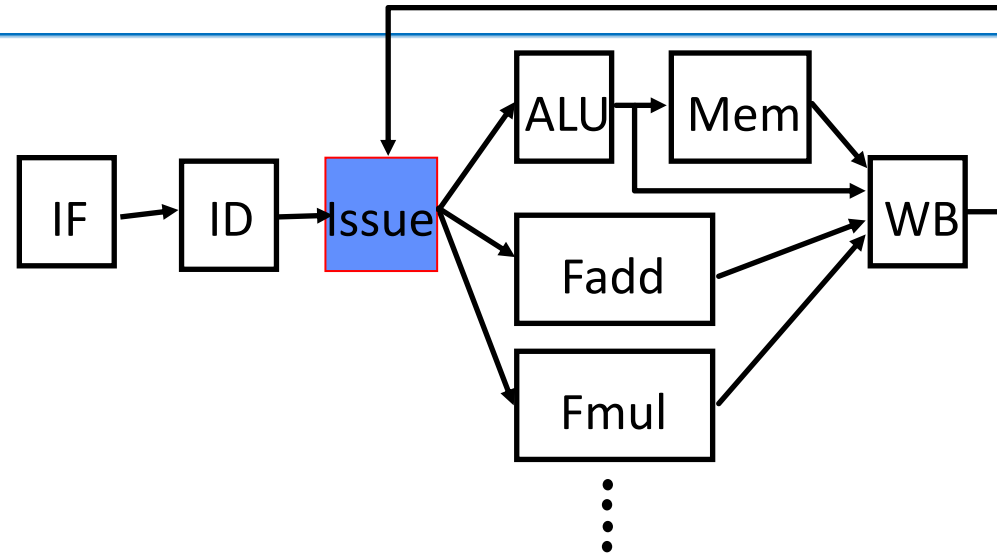


In-order: 1 (2, 1) . . . . . 2 3 4 4 3 5 . . . . 5 6 6

Restricțiile execuției In-order previn ca  
instrucțiunea 4 să fie lansată



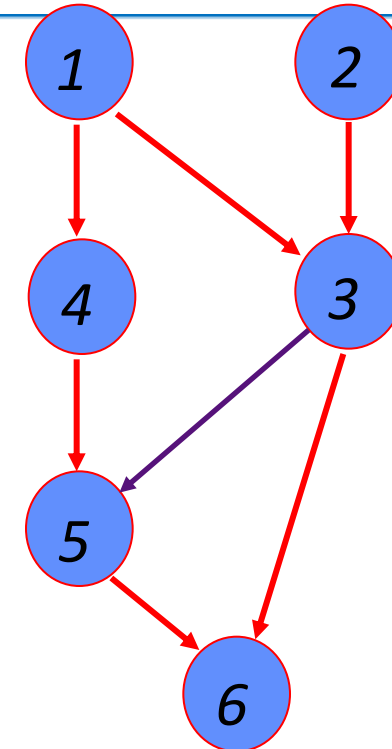
# Execuția Out-of-Order



- Buffer-ul din etapa Issue ține mai multe instrucțiuni care așteaptă lansarea în execuție.
- Decode adaugă instrucțiunea următoare la buffer dacă este loc și dacă instrucțiunea nu cauzează un hazard WAR sau WAW.
  - Notă: WAR este posibil deoarece execuția este out-of-order (WAR nu era posibil pentru in-order cu memoriarea operanzilor în FU)
- Orice instrucțiune din buffer pentru care hazardele RAW sunt rezolvate, poate fi lansată. La write back (WB) se pot introduce în buffer noi instrucțiuni.

# Limitări: In-Order și Out-of-Order

					<i>latency</i>
1	FLD	f2,	34(x2)		1
2	FLD	f4,	45(x3)		<i>long</i>
3	FMULT.D	f6,	f4,	f2	3
4	FSUB.D	f8,	f2,	f2	1
5	FDIV.D	f4,	f2,	f8	4
6	FADD.D	f10,	f6,	f4	1



In-order:            1 (2,1) . . . . . 2 3 4 4 3 5 . . . 5 6 6  
 Out-of-order:      1 (2,1) 4 4 . . . . . 2 3 . . 3 5 . . . 5 6 6

*Execuția Out-of-order nu ne aduce o îmbunătățire semnificativă!*



**NICE TRY**



**BUT NO**

# Câte instrucțiuni pot fi ținute în b.a.?

---

Care dintre proprietățile ISA limitează numărul de instrucțiuni din b.a?

*Numărul de registre*

---

Execuția Out-of-order de sine stătătoare nu aduce nici o îmbunătățire semnificativă!

## Rezolvarea lipsei de registre

---

B.a. Floating Point nu pot fi umplute de cele mai multe ori cu un număr mic de registre.

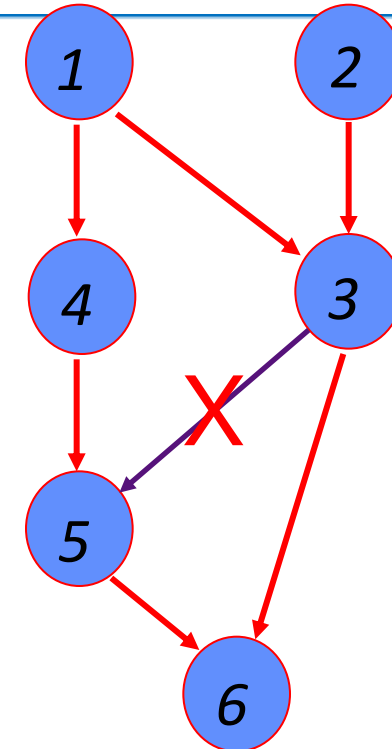
IBM 360 avea doar 4 registre floating-point

*Poate o microarhitectură să folosească mai multe registre decât cele specificate în ISA fără să piardă compatibilitatea cu ISA ?*

Robert Tomasulo de la IBM a sugerat o soluție ingenioasă în 1967 folosind tehnica *register renaming*

# Limitări: In-Order și Out-of-Order

					latency
1	FLD	f2,	34(x2)		1
2	FLD	f4,	45(x3)		long
3	FMULT.D	f6,	f4,	f2	3
4	FSUB.D	f8,	f2,	f2	1
5	FDIV.D	<b>f4'</b> ,	f2,	f8	4
6	FADD.D	f10,	f6,	<b>f4'</b>	1



In-order: 1 (2,1) . . . . . 2 3 4 4 3 5 . . . 5 6 6

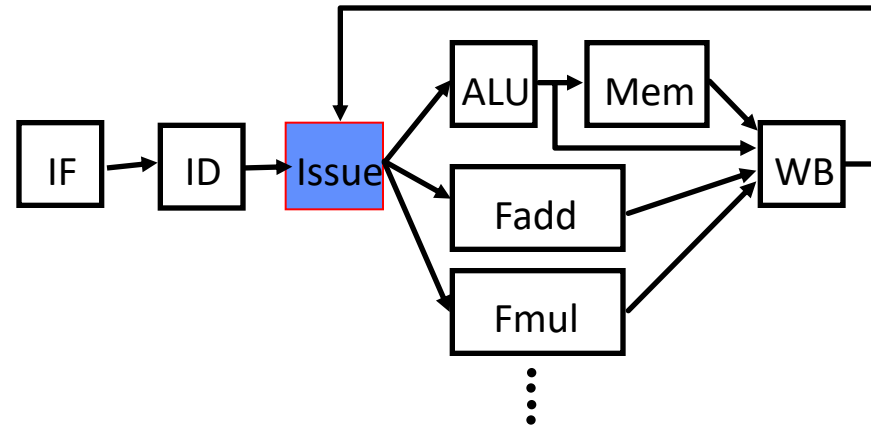
Out-of-order: 1 (2,1) 4 4 5 . . . 2 (3,5) 3 6 6

*Orice antidependență poate fi eliminată prin redenumire.*

*(renaming != spațiu de stocare suplimentar)*

*Poate fi făcut în hardware? **DA!***

# Register Renaming

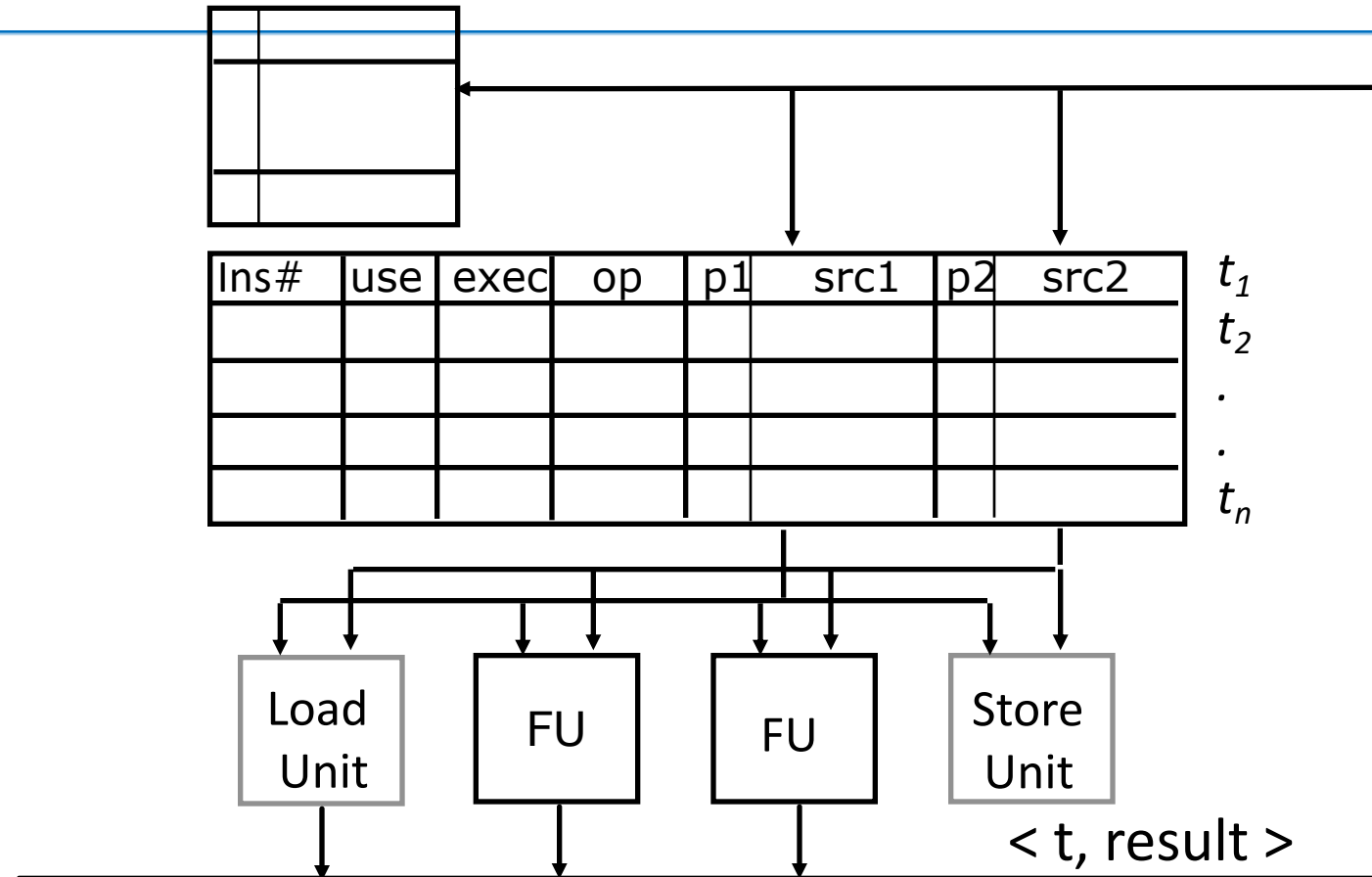


- Etapa Decode face register renaming și adaugă instrucțiunile etapei Issue în instruction ReOrder Buffer (ROB)
  - redenumirea face imposibile hazardele WAR sau WAW
- Orice instrucțiune din ROB al carei hazarde RAW au fost satisfăcute poate fi lansată.
  - Out-of-order or dataflow execution

# Structuri folosite la renaming

*Renaming  
table &  
regfile*

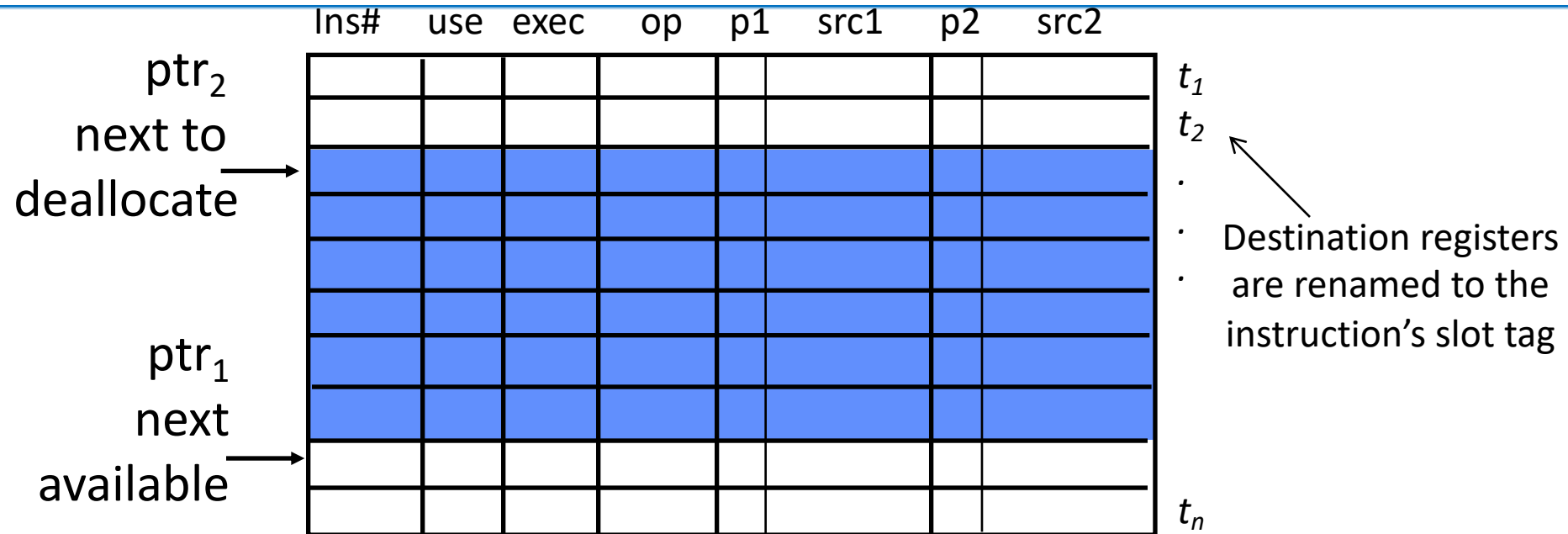
*Reorder  
buffer*



- Instruction template (i.e., tag  $t$ ) este alocat de etapa Decode care asociază acesta cu registrul din tabela de registre
- După execuția instrucțiunii, tag-ul este dealocat



# Management-ul Reorder Buffer



ROB este alocat circular

- Bitul "exec" este setat când instrucțiunea începe execuția
- Când o instrucțiune a terminat execuția, bitul "use" este șters
- ptr<sub>2</sub> este incrementat doar dacă bitul "use" este marcat ca liber

Un slot de instrucțiune este gata de execuție când:

- Conține o instrucțiune validă (bitul "use" e setat)
- Nu a început încă execuția (bitul "exec" este liber)
- Ambii operanzi sunt disponibili (p1 și p2 sunt setați)

# Renaming & Out-of-order Issue

*Un exemplu*

Renaming table

	p	data
f1		
f2		v1
f3		
f4		t5
f5		
f6		t3
f7		
f8		v4

data / t<sub>i</sub>

Reorder buffer

Ins#	use	exec	op	p1	src1	p2	src2
1	0	0	LD				
2	0	0	LD				
3	1	0	MUL	0	v2	1	v1
4	0	0	SUB	1	v1	1	v1
5	1	0	DIV	1	v1	0	t4

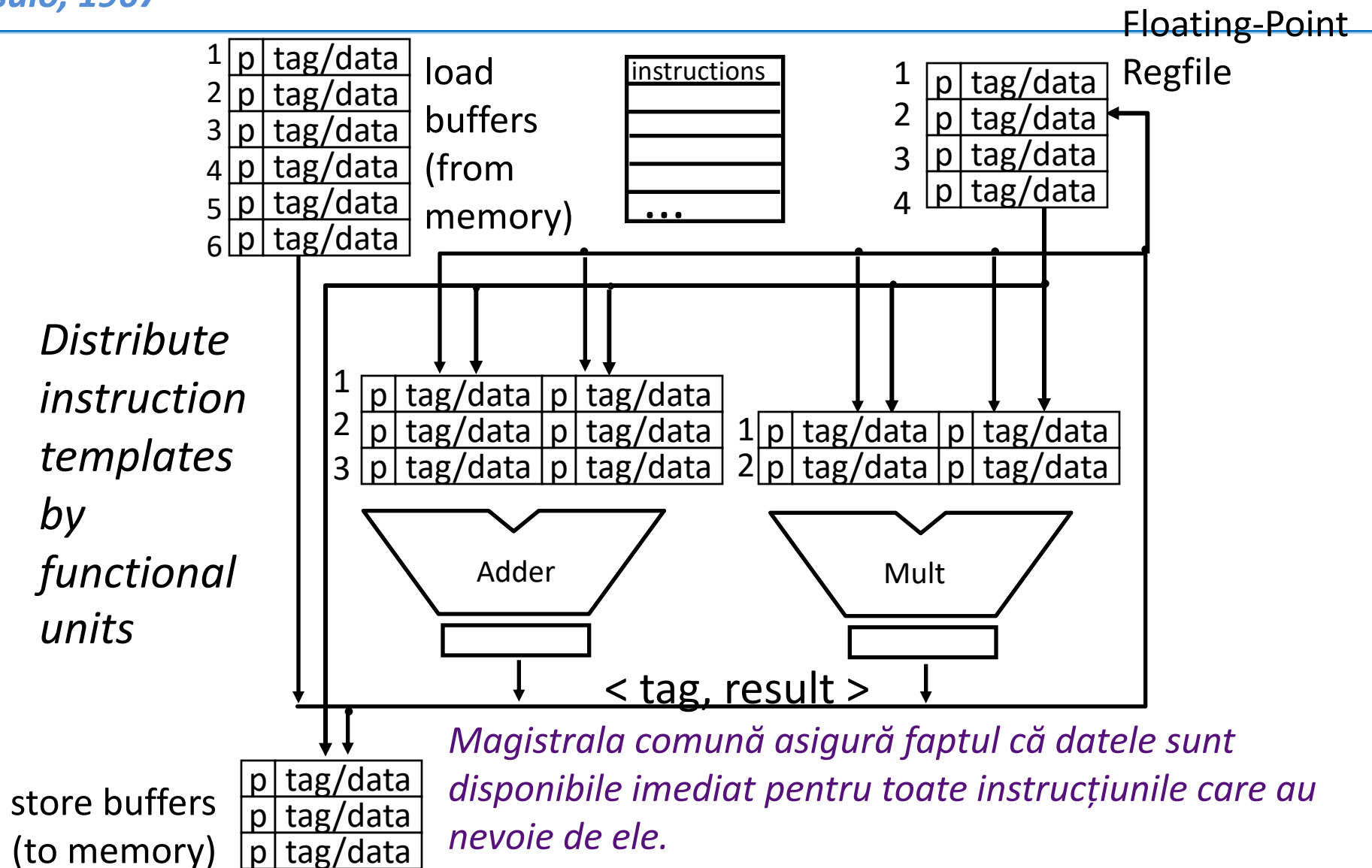
t<sub>1</sub>  
t<sub>2</sub>  
t<sub>3</sub>  
t<sub>4</sub>  
t<sub>5</sub>  
.  
.

1	FLD	f2,	34(x2)
2	FLD	f4,	45(x3)
3	FMULT.D	f6,	f4, f2
4	FSUB.D	f8,	f2, f2
5	FDIV.D	f4,	f2, f8
6	FADD.D	f10,	f6, f4

- Când sunt tag-urile înlocuite de date?  
*Oricând o unit. funcț. produce date*
- Când poate fi reutilizat un nume?  
*La terminarea execuției unei instrucțiuni*

# IBM 360/91 Floating-Point Unit

R. M. Tomasulo, 1967



# Eficiență?

---

Register Renaming și Out-of-order execution au fost implementate prima dată în 1969 la IBM 360/91 dar nu au apărut în modelele următoare până în anii 90.

*De ce ?*

*Motive*

- 1.Eficiente doar pentru o clasă foarte mică de programe
- 2.Latența memoriei era o problemă mult mai mare
- 3.Excepțiile nu sunt precise!

Încă o problemă trebuia rezolvată mai întâi

*Transferul controlului*

---



# 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

